CAB202 Assignment 2 - Report (40%)

Race To Zombie Mountain - Teensy

Student: James Allen

Due: 3rd June 2018

Unit: CAB202 - C Programming (Semester 1)

Executive Summary

The game 'Race To Zombie Mountain' consists of a top-down scrolling view, depicting a race car attempting to weave around obstacles on the windy road, scenery outside the road, and rogue fuel depots. When the game is started, a splash display screen is shown, allowing players to read all of the information without suffering any penalties on their final score. After pressing either the left or right button, the screen is cleared and the game begins. Once the game has begun, a dash display is shown at the top of the screen, faithfully representing the condition of the car, fuel left, score and current speed. To pause the game, the users can simply hold the down joystick, with this allowing the player to view the distance travelled and time when the game was paused. Once this button is released, the player will continue from where they were.

To move the car to any horizontal location within the bounds of the screen, players can use the left and right joystick. Similarly, to increase or decrease the speed, the up and down joystick can be used. To mimic a real life scenario, the car suffers extreme frictions whilst travelling off-road and is limited to a speed of 3, however whilst on the road it can travel at a maximum speed of 10. Along with this, the accelerator and brake function in a far more realistic way. In the case that a player hits any of the moving objects, a life is lost, with the player incurring a penalty to their final score.

As the car is travelling a long way to 'Zombie Mountain', the car is required to refill using the fuel depots. After stopping within 2 pixels of a fuel depot, the player's fuel is gradually refilled over 3 seconds, with the player being relocated to a safe location if the tank is completely filled. If the player is successful in all of these ways, then the finish line will be reached and the player will have successfully escaped the zombies.

Along with this, the game now has the added option of saving the game states when the left button is pushed whilst the down joystick is held, and then for these game states to be loaded back onto the teensy when a player feels they are running low on lives! This bi-directional serial communication gives the game an added complexity that certainly makes it more fun to play.

The full source code for the game can be found under the submission in AMS. All features and function line numbers correspond to this.

Contents

1 Part A - Port Basic Game (10%)

1.1 Splash Screen

1.1.1 Description of Feature

The most simple, yet important component of the game "Race To Zombie Mountain" involves a splash screen to be displayed, with the creator's name, student number and game title being shown. In the game generated throughout this assignment, the splash screen is immediately displayed after providing power to the teensy, and remains on the screen until either the left or right button is pressed (SW2 or SW3). This ensures players have an unlimited amount of time to read the information, however suffer no consequences in their game-time score.

1.1.2 Global Variables and Functions Used

The implementation of the function **'splash_display' (lines 372-382)** uses no global or local variables. This is because the display purely implements the function 'draw_string' from the provided 'graphics' library. In order to ensure the splash display remains on the screen until either the left or right button is pressed, a simple while loop was used. This loop checks whether either of these buttons has been pressed, and only when one of them has been pressed does it allow the screen to be cleared.

1.1.3 Test Plan

Setup Process: In order to test this feature, it is a simple matter of plugging in the teensy to a power supply such as a computer. This will immediately turn the teensy on and present the splash display for the user.

Expected Outcome: After supplying the micro controller with power, the expected outcome is that a splash screen will immediately appear, with this remaining on the screen for as long as the user likes. When the player eventually decides to play the game and selects either SW2 or SW3 (buttons on the RHS of the TeensyPewPew), this screen should immediately be cleared, with the game starting straight away. Along with this, the game time should only begin on the press of this button. This ensures the user faces no consequences for reading the information presented.

Actual Outcome: After testing this feature, it was proven to work as expected. Before selecting either SW2 or SW3, this splash display containing the name, student number and game instructions remains on the screen, with this only being cleared after either of these buttons were pressed. After selecting either of these buttons, the game immediately starts and the player begins the race. Along with this, it was proven that the game time only starts once either of these buttons is pressed. By leaving the splash display on the screen for a long period of time such as 5 to 10 seconds (anytime will work though), and then pressing the left or right button directly followed by holding down the center joystick (pause control), will successfully prove that this time always starts from 0.

1.2 Dashboard

1.2.1 Description of Feature

In order to produce a visually appealing game with a dashboard easily distinguished from the rest of the game, it was firstly important to generate a clear boundary line separating the dashboard from the game. After drawing this line, 3 of the key game states were presented, with these being updated every 10ms in the 'process' function. In the dashboard of the game, 'D', refers to the damage of the car, 'S' corresponds to the speed, and 'F' for the fuel remaining.

1.2.2 Global Variables and Functions Used

In order to successfully represent each of the constantly updating game states, 3 global variables were required in the function **'dashboard' (lines 569-5745)**. The first of these global variables was **'damage'**, with this being decreased by 1 integer every time a pixel level collision occurred. The next global variable was **'speed'**, with this being incremented or decreased at different rates throughout the game. The final global variable used in this function was **'fuel**, with this relying on the speed of the car to proportionally decrease. The reason these variables were declared as global variables was due to their use in a wide range of other functions. This ensured they were always updated for the entire game. These global variables were then simply called as a formatted string, to ensure they would allow for updates as the variables changed.

1.2.3 Test Plan and Results

Setup Process: To test the dashboard display, the game must begin by pressing either SW2 or SW3 to enter. After entering the game, the dashboard can be clearly observed, with this remaining unchanged until the car begins moving. By using the left button to increase the speed, the player should then notice the speed incrementally increasing (or decreasing if the right joystick switch is pressed), along with the fuel dropping and any lives being lost if collisions occur.

Expected Outcome: After this splash display is cleared, the dashboard will immediately be shown at the top of the screen, with 'D' corresponding to 'Damage', 'F' corresponding to 'Fuel Left' and 'S' corresponding to 'Speed'. All of these will initially be set to 10, 100 and 0 respectively, with these remaining like this until the car begins moving. After the car begins moving, the speed should be observed to incrementally increase when the left button is pressed, or decrease when the right button is pressed. Along with this, if the car collides with any scenery or obstacles, the player should observe the 'D' score to decrease by 1 for every collision. From the second the car begins moving, the player should also be able to see the fuel proportionally decreasing with respect to the speed of the vehicle. When the fuel reaches 0, this should stop decreasing and the game should finish. This ensures the fuel is never negative, as this would be unrealistic of a real world model.

Actual Outcome: After thoroughly testing the dashboard display, all features were shown to operate successfully. The fuel, damage and speed initially started at the desired numbers, with the fuel decreasing as the speed increased, and all lives lost causing the damage to decrease by 1 every time. Along with this, the game was tested until the fuel decreased to 0. After reaching 0, the game immediately finished in the desired way expected, with no negative fuel values ever

being displayed. Therefore, the dashboard has been proven to work as expected, and the user can expect this to be exactly the same every time.

1.3 Paused View

1.3.1 Description of Feature

In order to enhance the game, 'Race To Zombie Mountain' now has the added ability of being paused. For the user to successfully pause the game, the down joystick must be held, with this displaying the current distance travelled, along with the time from when the game was paused. After releasing this button, the game resumes as it left off, except with the speed set back to 0 to allow the player to re-orientate themselves with where they were in the game and drive back off towards zombie mountain.

1.3.2 Global Variables and Functions Used

There were a number of global variables required in the function **'pause_screen' (lines 861-884)** for this to be successfully implemented. The first of these global variables was **'distance'**, with this being used in a formatted string to represent the total distance travelled in proportion to the speed of the vehicle. The other global variable required was **'time'**. This was firstly initialised in the **process** function, with this equaling the current game time ('time = current_time()'). After setting this variable equal to the game time, the function **'pause_screen'** was called. In order to ensure that the game time was in fact paused, the timer needed to be paused in such a way that the **time** variable would stop incrementing until the screen was taken off pause mode. The way this was implemented was by setting the timer 3 control register A (TCCR3A) and B (TCCR3B) to 0. Whenever this down joystick was then pressed meant that the timer would cease counting, allowing the **time** global variable to simply display a static value calculated from the most recent update.

1.3.3 Test Plan and Results

Setup Process: In order to test this feature, the splash display should be cleared to ensure the game has started. After starting the game, the down joystick can be held down at any point during the game. This will ensure the pause screen is activated, and only once the down joystick is released will the game resume again.

Expected Outcome: After holding down the down joystick, the screen should display the message 'Game Paused!', along with 2 formatted strings below. The first of these should display the distance travelled at the point at which the pause control was activated. Along with this, the next line should display the time when the game was paused. Neither of these values will change at all whilst this is being held down, however if the button is released and then held down again it can be noted that a slightly higher time will now be displayed.

Actual Outcome: The actual outcome is exactly what was expected, with the center joystick successfully resulting in the game to be paused. The time is also clearly paused, along with a distance value being shown. To absolutely guarantee that the pause mode works, a stop watch was used to check. This verified that the pause mode definitely works as expected.

1.4 Race car, horizontal movement (non-collision)

1.4.1 Description of Feature

To control the movement of the race car, the left and right joystick controls were set to move the car to the left or right, provided the car had a speed greater than zero (naturally a car cannot turn if it is stationary). This section of the game has been advanced upon later in the report, so for further information refer to Part B.

1.4.2 Global Variables and Functions Used

There were 3 global variables required in the function **'race_car_movement' (lines 792-801)**. The first of these was the variable **'speed'**. The reason this was used as a global variable was due to it being required when updating the dashboard display, and controlling the movement of the obstacles, scenery and fuel depot. The other global variables used was simply the car array variable **'car_options[vehicle]'**. This was used as a global variable simply as it is constantly being updated throughout the entire program when the up joystick is held down and the potentiometer is adjusted. Along with this, an if statement was used to ensure that no lateral movement was permitted when the speed was equal to zero, and if statements were also used to ensure it would never drive off the edge.

1.4.3 Test Plan and Results

Setup Process: To setup the game to allow for the testing of this movement, the game must have started. After beginning the game, the user must firstly press the left button to begin moving, with the left and right joystick switch then being used for the steering.

Expected Outcome: When the game begins, the car should initially begin with a speed of 0, meaning the player cannot move the car no matter how much they press the left and right joystick. After pressing the left button for speed, the player should then easily be able to control the car. Along with this, it is expected that the car will never run off either side of the road. By pressing the left switch, the car should be seen to veer off to the left hand side of the screen, however never exit the LCD display. Similarly, when the right joystick switch is held down, the car should veer towards the right hand side of the screen and also never leave the screen.

Actual Outcome: After throughly testing each of these features, the game has proven to work as desired. The car does in fact never leave the screen, and along with this no lateral movement is present when the speed is equal to 0.

1.5 Acceleration and speed

1.5.1 Description of Feature

In order to create the illusion of driving forward, it was of great importance that the obstacles, scenery and fuel depot moved at certain speeds. By using the left and right buttons, the speed could be adjusted anywhere from 0 up to 10, and along with this automatic breaking was implemented that slowed the speed down when the accelerator was released (as discussed in Part B). If the speed was 0, no further deceleration could be applied, and similarly if the speed was 10, no

further acceleration could be applied.

At the beginning of the game, the velocity began at 0, meaning the only command to alter the speed would be the left button. After increasing this speed to anywhere above 3 and then driving completely off the road, this speed was then set back down to 3 instead of the maximum road speed of 10, however still giving players the option to decrease this speed anywhere from 3 down to 0, and increase from 0 up to 3 with the same step sizes. This speed can easily be seen in the dash display, which further helps players to understand why they cannot go any faster or slower. As the extended game for this feature has been implemented, this has been further explored below.

1.5.2 Global Variables and Functions Used

In order to execute this element of the game, it was important to use the global variable **'speed'** in the function **'set_speed' (lines 462-542)**. This is because when the speed is changed, it also governs the step size of the obstacles, scenery and fuel tank. To ensure the speed is always set to a maximum of 3 off the road, and 10 on the road, if statements were used that checked the position of the car. This has also been much further advanced upon in Part B.

1.5.3 Test Plan and Results

Setup Process: To setup this component of the game, the player must simply clear the splash display to enter the game. After entering the game, the left button can be used to accelerate and therefore test these features.

Expected Outcome: The expected outcome of this is that the speed of the car begins at 0, and will never exceed a speed of 3 whilst travelling off the road. Along with this, it should never travel faster than 10 when on the road, and when the brake is applied off the road, the speed should still decrease down to 0.

Actual Outcome: After testing this basic feature, the game was proven to work as expected. Even after holding down the accelerator button off the road, the speed was shown to never exceed 3, however would successfully decrease down to 0.

1.6 Scenery and obstacles

1.6.1 Description of Feature

In the game created, there are 3 images for the scenery and 3 for the obstacles. The items of scenery consist of a cactus, dog and bomb, with these only ever being on the left or right sides of the road. The obstacles however only appear on the road and consist of a zombie and 2 identical trees.

1.6.2 Global Variables and Functions Used

To successfully implement the scenery and obstacles, functions were used to initially display them, and for their movement. The initial setup functions required were **'setup_obstacles' (lines 689-719)** and **'setup_scenery' (lines 650-686)**, which utilised the global sprite arrays **'obstacle_array[3]'** and **'scenery_array[3]'** respectively. The reason these were required as global

variables was due to their use in other functions such as **'safe_position'**. After creating each sprite and then drawing them in a similar manner, it was of great importance to implement a check for overlap during the display.

To check whether any of the scenery or obstacles overlapped required a detailed while loop after creating them. This check firstly assumed overlapping to be true through the use of the local boolean variable 'overlapping', with each sprite then being checked for overlap. If the overlap was true, this sprite would then be moved to a new random x location. Although it seemed that this would then be enough to ensure overlap, it was found that after moving to a new location, these then may end up overlapping once again. The code therefore rechecks against all sprites to finally ensure that there is no possibility for overlap (lines 663-681 and lines 703-719). A similar method for checking for overlap was then implemented once the objects and scenery began moving and scrolled back onto the screen, that also checked every case to ensure no overlap ever occurred.

The other functions required were **'obstacle_movement' (lines 761-789)** and **'scenery_movement' (lines 727-758)**, which used the global variable **'speed'** which was dependent on the user inputs from the left and right buttons. After assigning each speed, the vertical step size value (except divided by 4 to make the game playable) was updated to ensure the speed of the obstacles and scenery accurately reflect the dash display. After scrolling to the bottom of the screen, an if statement was used to relocate the sprite back to the top of the screen. From there the function **'pick_side' (lines 638-647)** was required which used a local array 'entry_side' to either choose the left or right side to enter back in on. The smooth scroll on effect also used the function **'draw_blank' (lines 586-593)** as this purely created a blank box under the dash display, allowing the sprites to enter from underneath without being seen. This created the illusion of smoothly scrolling on and off.

1.6.3 Test Plan and Results

Setup Process: In order to setup the scenery and obstacles, the game must simply begin by pressing the left or right button. After pressing these buttons, the scenery and obstacles will appear. After holding down the left button, this will then increase the speed, allowing players to visually observe both the movement and positioning of the scenery and obstacles.

Expected Outcome: The expected outcome for this component of the game is that the scenery will be initially displayed outside of the road, with the obstacles being positioned on the road. After accelerating the car, the scenery and obstacles should remain in the same x position, however increase their y position towards the bottom of the screen. After exiting the screen, the scenery should then enter back onto the screen at new random positions on either the left or right of the road, with the obstacles entering at some random x position on the road.

Actual Outcome: After thoroughly testing this component of the game, it was proven that the scenery and obstacles were initially displayed in their desired locations. Along with this, after observing the smooth scroll on effect, these sprites were then randomised into new positions, however still either off the road for scenery, or on the road for the obstacles.

1.7 Fuel depot

1.7.1 Description of Feature

Although it is possible to complete the game by simply losing lives in order to refuel the race car, using a fuel depot is the most effective way as this does not result in a penalty score being added to the final score at the end. In order to randomise the nature of when the fuel depots would occur, a probability method was used. If the distance travelled was 50m and a random number less than 50 out of 100 was picked, this ensured that there was only a 50% chance of receiving a depot. If the distance travelled was 100m and a random number less than 100 out of 100 picked, this naturally guaranteed a fuel depot at this stage. Along with this, if the fuel depot is hit, this ends the game as all of the lives are lost. As discussed in Part B, more realistic refueling of the car has also been added to allow a gradual fuel refill over 3 seconds.

1.7.2 Global Variables and Functions Used

In order to achieve this feature, a few simple global variables were required in order for the function **'fuel_movement' (lines 808-830)** to work. The first of these was simply the **'speed'**, which was used so that the fuel depots would move at a speed relative to everything else on the screen. The only other global variables required were **'low_chance'**, and **'definite_chance'**. The reason these were created as global variables was due to them being generated in the **'setup'** function. It was necessary to set these as constant unchanging values at the beginning of the game, otherwise the values would always be updating every 10ms in 'process'. By having these as constant values, this meant that when the if loop checked whether the fuel was less than or equal to the desired values and whether the values were less than the set probability numbers, then it would either move the fuel depot or it would not. This ensured no staggered movement of the fuel depot and allowed it to still appear randomly for each game.

1.7.3 Test Plan and Results

Setup Process: In order to setup this component of the game, the car should begin moving. This is all that is required for the player to encounter a rogue fuel depot.

Expected Outcome: The expected outcome is that the player may see a fuel depot after driving for 50m, however on some games should not see one at this point. No matter whether one is presented after 50m, there will always be one that scrolls into view after 100m. This ensures that the player can definitely make it the entire way to zombie mountain without dying due to no fuel. Along with this, it is expected that when the car is moving, the fuel depots also move at the same speed, and when the car stops moving then so does the fuel depot. This ensures that the game is as life like as possible.

Actual Outcome: After testing the fuel depot, it was proven to appear only 50% of the time, and to always appear at the 100m mark. Along with this, the car speed was increased and decreased when this came into view, with this proving that the fuel depot moves at a relative speed to the scenery, obstacles and car.

1.8 Fuel

1.8.1 Description of Feature

In order to create a more realistic feature, a constantly decreasing fuel tank was implemented. This rate of fuel consumption was proportional to the speed of the car, with no fuel ever being used if the car was stationary. Along with this, the fuel depot should allow the car to successfully refill, however if the car touches the depot whilst refueling the game will be over.

1.8.2 Global Variables and Functions Used

To generate an accurate and proportional relationship for the fuel left in the tank, the global variable **'speed'** was once again necessary, along with the global variable **'fuel'**, in the function **'fuel_use' (lines 450-459)**. This was then used to appropriately update the fuel values in the function **'dashboard' (lines 569-574)**. As the fuel is always calculated in relation to the speed, this ensured there was always a proportional relationship.

Along with the fuel level dropping at a proportional rate, it is necessary to refill this in order to make it the entire distance to 'Zombie Mountain'. Currently the race car would only be able to make it a distance of 100m on a tank of fuel, which leaves the player to either lose lives to refill their fuel (allowing a zombie mechanic to refill their car as they are relocated to a safe position), or to use the fuel depot. This has been implemented in the function **'refuel' (lines 833-858)**, with players needing to position their car within 2 pixels of it as it scrolls down the screen, and then stop the car as soon as it is in line. This game has been designed so the player can refuel the car from either side of the road, as most fuel depots have a fuel pump on both sides. After stopping next to this and waiting for 3 seconds to elapse, the car is repositioned in a safe position through the use of the function **'safe_position' (lines 1134-1150)**, ready for the player to drive off again.

1.8.3 Test Plan and Results

Setup Process: In order to test this feature, players must clear the splash display so the game can begin. After beginning the game, the accelerator can then be held down to initialise movement.

Expected Outcome: In this game, it is expected that as the accelerator is held down, the fuel will begin to proportionally decrease. In addition to this, it should be noted that when the speed drops to 0, the fuel will stop decreasing, with this remaining at a constant value until the car begins moving again.

Along with the fuel being used in relation to the speed of the car, the fuel depot should accurately represent a real life scenario. This fuel depot should randomly enter the screen in the same way as the scenery or obstacles, with it travelling at the exact same speed. After the player has noted that the fuel depot is on the screen, the car should be positioned within 2 pixels either to the right or left of the fuel tank until this is next to the car. Once it is next to the car, the speed should be decreased down to 0, and players should visually observed the fuel level increasing. This feature has also been further advanced upon in Part B.

Actual Outcome: After testing this feature, all elements were proven to successfully work.

1.9 Distance travelled

1.9.1 Description of Feature

To ensure the car ends up at 'Zombie Mountain', it is of great importance that the distance increased proportionally in relation to the speed. When the car is travelling at 0m/s, this distance will never change, however as the speed is increased, the distance will increase as well. As the race car begins to approach 'Zombie Mountain', a mountain scrolls smoothly onto the screen to provide players with the knowledge that they are almost there. After the car then passes this line, the game is over and the player can restart the game by pressing down on the joystick.

1.9.2 Global Variables and Functions Used

To ensure the global variable **'distance'** in the function **'distance_travelled' (lines 436-447)** began at an initial 0m, this variable was firstly set as 0. After this, whenever the global variable **'speed'** was altered through the use of the left and right buttons in the function **'race_car_movement' (lines 792-801)**, this would therefore increase the distance travelled. This can be clearly seen on line 445, and it is obvious why at a speed of 0, the distance will not change (the formula essentially states distance = distance + 0 if the race car is not moving). The reason that distance was declared as a global variable was due to it being needed in a number of other functions, with the most important of these being **'dashboard' (lines 469-474)**. This ensured that the distance would also be faithfully represented by the dashboard display at all times throughout the game.

In order to create the finish line effect, the sprite **'finish_line'** was necessary in the function **'setup_finish_line5' (lines 1152-1155)**. This function was used to initialise this sprite behind the dash, ensuring it would be hidden from view until the car reached the required distance of 140m. The reason for needing this as a sprite was to ensure the finish line always moved at a speed relative to everything else on the screen. To ensure this speed was always proportional to the current speed, the global variable **'speed'** was once again used in the function **'finish_line_move'**. Through the use of if statemements, this function was used so the mountain sprite would only be moved at this relative speed when a certain distance was reached. This ensured that on the arrival at zombie mountain, the game would be won and the car character controlling the car would be free from the zombies.

1.9.3 Test Plan and Results

Setup Process: To test not only the distance feature, but also zombie mountain, the game should commence, with the player pressing the left button to accelerate. This will therefore alter both the speed and hence distance that the car has travelled. By ensuring that the fuel tank does not empty (either refuel the car or collide with a scenery or obstacle character), zombie mountain will be reached.

Expected Outcome: The expected outcome for this is that the distance accumulates proportionally to the speed of everything on the screen. In order to check that this is the case, after beginning the game and before increasing the speed from 0, the down joystick can be pressed to enable pause mode. This shows the distance, travelled, with the expected distance at this point being 0m. After this, the left button can be pressed to increase the speed. This should then increase the distance, which can again be checked in pause mode. If the speed is then increased

to 10, the player should notice a significant increase in their distance travelled.

To check the finish line (zombie mountain), players will need to be careful of their fuel, and also dodge all of the scenery and obstacles so they do not lose more than 5 lives. If all of this criteria is met, the mountain sprite will appear, and the user will have successfully completed the game. This should immediately bring up a 'You Won!' message, along with the score, distance and completion time.

Actual Outcome: After thoroughly testing this feature, all requirements were shown to accurately operate. When the speed was 0, no distance would ever increase, and if the speed was increased, this distance would increase at a proportional rate. Along with this, zombie mountain was shown to operate very successfully, with this remaining hidden right until the end of the game. After it would begin to move down the screen, the speed was also altered and this sprite suitably changed its speed to ensure it was always moving at a proportional speed to the screen.

1.10 Collision

1.10.1 Description of Feature

Another vital feature was the ability to lose lives as a result of hitting anything moving on the screen. The game is created in such a way that if it hits any of the obstacles or scenery, 1 life is lost, and if the race car hits the fuel depot then the game is over. This has been advanced upon in part C, with the successful implementation of pixel level collision detection being used. Refer to this for global variables and the test plan.

1.11 Game Over Dialogue

1.11.1 Description of Feature

One of the most crucial elements of the game was the ability to play again after the game finishes. After either losing all of the possible lives and dying, or victoriously finishing the game, the screen is cleared and a display screen appears. Along with this, it also displays the distance travelled, total time and their final score. This score takes into account a penalty if they have lost lives, with it being increased by however many lives were lost. After pressing the down joystick, the game is restarted, with all game states being reset as if the game was beginning from the very start again.

1.11.2 Global Variables and Functions Used

A number of global variables were required in the 2 functions used. The first function **'exit_screen' (lines 1166-1184)** was implemented to only be called whenever the global boolean variable **'game_over'** was true. This was set to true in elements of the functions 'dash_display', 'race_car_movement and 'fuel_tank_movement' due to the game over conditions being met (car reached finish, had no lives left or collided with fuel tank). After this condition had been met, 'exit_screen' could then be called, which immediately implemented the function 'clear_screen' to ensure only the display messages were on the screen. After determining through an if statement whether the game was either lost or won, the player would be presented with the appropriate messages from this function. The global variables **'distance'**, **'final_time'** and **'score'** were then necessary in providing the player with any relevant information in these messages. Similar to the dash display, the function

'draw_formatted', was used in order to include a variable in each string.

The function **'exit_prompts' (lines 1187-1206)** was then called, which required the global boolean variable **'game_over'** (initially set to false) and **'new_game'** (initially set to true), along with all of the global game states. In order for this function to successfully operate, the **'main' (lines)** function had to be altered for it to operate successfully. The first alteration was to include a while loop around the 'game_over' while loop (lines 275-282). This while loop was set to only operate when 'new_game' was true. After the 'exit_prompts' function checked through an if else-if statement for whether the down joystick had been pressed to indicate a new game, setting 'game_over' as false for yes, or leaving 'new_game' as false for no. If 'game_over' was to false for yes, this leaves 'new_game' as true and resets 'game_over' back to false, meaning a new game begins in the same way as it initially started. The only other element required was to reset all of the game state global variables back to their initial values.

1.11.3 Test Plan and Results

Setup Process: To setup this feature, the player has a number of options. The first of these is to have 5 collisions, another is to hit a fuel tank, another is to run out of fuel, or the other option is to try and make it to zombie mountain.

Expected Outcome: After applying any of these methods and ensuring the game states drop to 0, the play should be presented with a screen saying they have either lost (in all of the cases mentioned above except for making it to zombie mountain) or that they have lost. After being presented with this data, the time should have stopped, along with the distance travelled. After leaving this on the screen for as long as the user likes, the down joystick can then be pressed to restart the game. On the restarting of a new game, the initial game state conditions should be reset to what they were when the game started the first time, and the time should have restarted again. To ensure that the timer and distance have both been reset back to 0, the down joystick can be held to display the pause mode. This allows the player to visually observe that the timer and distance have definitely been reset.

Actual Outcome: After testing this feature by following the steps above, it was proven to restart the game, along with providing the user with all of the initial game states. This increases the functionality of the game and makes it far more user friendly.

2 Part B Extend Game (10%)

2.1 Curved Road

2.1.1 Description of Feature

In order to increase the functionality of the game, a realistic curved road was created. This road would ensure that if the car remained in a fixed location, that it would occasionally be off the road, whilst sometimes being on the road. To ensure the road worked as expected, it had to be designed

in such a way that the road speed would increase in proportion to everything on the screen. Along with this, the curves are clearly smooth flowing lines, with the position being obviously changed.

2.1.2 Global Variables and Functions Used

The global variables used in the function **'road_setup' (lines 596-601)** was simply the array **'x_road[LCD_Y]'**. The reason this was used as a global array was due to its use in other function such as 'setup_scenery', 'setup_obstacles', 'scenery_movement' and 'obstacle_movement'. This function was purely used to set up an initial x position for the road, with this being then altered in the function **'curved_road'**. This function required a number of crucial global variables, with the most important of these being **'road_move'**. The reason this is such an important global variable is due to it being required when saving and loading data from the text file to the teensy. This purely gives an accurate reading on how much the road should be altered in relation to the speed, and for it to be flowing downwards on the screen (hence the negative). The reason this variable is so important is because when this is changed, then so are the **'x_road'** positions. This is because these are used in the multiple sin curves as a scaling factor which changes the frequency appropriately. This therefore means that the curve will appear to be moving in relation to the speed, with the road following more random paths.

The way this road was then implemented was through the use of the function **'draw_pixel'** from the 'graphics' library. By using a for loop to place each element of the array at an incrementally increasing vertical position ensured that the curved shape would be created. This array was simply used for either side of the road, with this then being a highly visual game feature for players.

2.1.3 Test Plan and Results

Setup Process: The setup process for this feature is very simple, with road simply being displayed as soon as the game begins. When the left button is then pressed to increase the speed the road will begin to move.

Expected Outcome: The expected outcome is for the road to initially remain stationary when the speed is equal to 0. This is because the frequency component of the sine curve will remain constant, therefore causing no noticeable change. After the player begins to accelerate the vehicle, they should then notice that the road begins to weave around in a flowing like manner. Along with this, the car can be positioned to one side of the road, and players should notice that the car will naturally come on and off the road due to the curvy nature of the road. Along with this, the road should also speed up and down in proportion to the speed of everything on the screen. This will therefore ensure that the road is as realistic looking as possible.

Actual Outcome: After testing all aspects of this feature, it was proven to work exactly as expected. The road most definitely weaves to the left and right in a random manner, with the car being guaranteed to leave the road if it continues without any lateral movement. It was also shown to increase and decrease at a proportional speed to everything on the screen.

2.2 Accelerator and Brake

2.2.1 Description of Feature

Although the simple accelerator and brake were implemented before this, it was decided that this element of the design brief would allow the game to function even more realistically. The left and right buttons still function as the accelerator and brake respectively, however the game now has added features which make it more fun and harder to play. This means that when neither the accelerator nor the brake is pressed, the car naturally slows down, and along with this the car takes longer to accelerate and brake when it is off the road. These new features have been detailed further below.

2.2.2 Global Variables and Functions Used

There were a number of global variables used in the function **'set_speed' (lines 462-542)**, with the most important of these being **'speed'**. This is of course because the speed must be controlled by all of these set time conditions. In order to control the time element, an interrupt service routine of timer 1 was implemented. By using the volatile integer **'speed_overflow'**, this effectively allowed the time of each of these speed increases and decreases to be set. In all of the if statements relating to each speed condition, the modulus operator was used. This essentially allowed any amount of time to be found, and after meeting these conditions, **'speed_overflow'** could be set back to 0. This successfully ensured that all time constraints were successfully met.

2.2.3 Test Plan and Results

Setup Process: To test this feature, the game simply must have been started. The player can then simply sit back and watch as the speed slowly increases up to 1, however also has the ability to use the left or right buttons to control this speed. This will allow all features to be accurately tested.

Expected Outcome: The player should immediately notice that whenever they are on the road, the speed will increase to 10 in approximately 5 seconds when the accelerator is pressed, and will increase up to 3 over a period of 5 seconds off the road. Along with this, when neither the break nor accelerator is pressed, the car will decelerate back down to a cruising speed of 1 in approximately 3 seconds whilst on the road, and also in the same amount of time off the road. In addition to this, when the car is on the road and the speed is at 0, this will slowly increase up to 1 over 2 seconds, and similarly when the car is off the road this will increase up to 1 over a period of 3 seconds. The final expectation is that when the brake is pressed, the car will decelerate from its maximum speed down to 0 in approximately 2 seconds.

Actual Outcome: After extensive testing, the maths surrounding the timer overflow was proven to be correct, with all of the above features being successfully met. A stopwatch was also used to further prove that all of these times worked as intended. With the game now including many more speed options, this definitely helps to improve the challenge of the game.

2.3 More Realistic Steering

2.3.1 Description of Feature

Another feature that was implemented was more realistic steering for players. Previously the lateral movement was kept constant, however in this teensy edition, it seemed more suitable for this to increase in proportion to the speed as well. This simply means that as the car speed increases, so does the speed of the lateral movement. For example, if the car is moving very slowly at a speed of 1, then the car will also be constricted to a slow lateral movement.

2.3.2 Global Variables and Functions Used

The implementation of this realistic steering is in the function **'race_car_movement' (lines 792-805)**, with the global variable **'speed'** being again of great importance. To ensure this lateral movement is proportional to the speed of the objects, **'speed'** was simply added to the initial lateral movement. This ensured that whenever the car would begin to accelerate more, the left and right veering speed of car would also increase.

2.3.3 Test Plan and Results

Setup Process: In order to setup this feature, it is again just a simple matter of beginning the game and pressing down the left button to begin moving. The left and right joystick can then be used to move the car left and right.

Expected Outcome: The expected outcome of this would be for the car to travel much faster left and right whenever the car accelerates more. When the speed is equal to 0, this should result in the car having no lateral movement, however when the speed is just slightly increased, the car should be able to move slowly left and right. The player should then notice a significant difference when the speed of the car is greatly increased. This should mean that the player can move left and right far quicker.

Actual Outcome: After testing this comprehensively, it was proven that the car would most definitely travel faster laterally when the speed of the car was increased. This also remained true to the proportional relationship generated, meaning this had been successfully implemented.

2.4 Fuel Level Increases Gradually

2.4.1 Description of Feature

With the basic game simply allowing the player to receive a full tank after 3 seconds, this extension now allows players to partially refill their car even if they do not wait for the whole 3 seconds until it is full. This allows players to visually see the fuel being topped up, allowing again, a more realistic aspect to the game.

2.4.2 Global Variables and Functions Used

The global variables used for this gradual refueling feature was simply **'fuel'**. This was implemented in the function **'refuel' (lines 833-858)**, with this allowing players to either receive a

full tank of fuel over 3 seconds, or to simply refill their car for as long as they choose and then continue driving off.

2.4.3 Test Plan and Results

Setup Process: In order to setup this feature, the car must begin driving towards zombie mountain. At 2 random points during the game, a fuel tank should appear, with the player having to direct their car within 2 pixels of the depot. This will then allow the feature to be tested.

Expected Outcome: The expected outcome for this would be that the fuel begins to gradually fill up whenever a player parks up beside the depot. Players should then be able to drive off at any point, taking note of their fuel tank now being more full. Along with this, if a player decides to brake next to the fuel depot for 3 seconds, they should receive a full fuel tank by this point (this is assuming their fuel was being filled from 0 to 100).

Actual Outcome: When testing this game feature, the actual outcome was exactly as expected. The fuel refills over a period of 3 seconds, however the gradual refill means that a player does not need to waste valuable playing time if they only require a small amount of fuel. This feature further helps to increase the functionality of the game, and is a major improvement over the original fueling method.

3 Part C Demonstrate Mastery (20%)

3.1 Use ADC

3.1.1 Description of Feature

In order to increase the functionality of the game, the left potentiometer was utilised to allow players to switch their race car during the game. The reason for using a potentiometer instead of a button was due to it being very accessible for the player, easily allowing them to choose their desired car. It also seemed like an appropriate choice seeing as a toggle type system was essentially required to choose between the vehicles.

Along with the left potentiometer being used to control the style of car, the right potentiometer was also utilised. The functionality of this one is to simply control the back light display. As discussed in the PWM section of the report, this is an essential inclusion seeing as the LCD displays are not always the same on each teensy.

3.1.2 Global Variables and Functions Used

3.1.3 Test Plan and Results

Setup Process: In order to test this feature, the player should have advanced on from the splash display with the game being in progress. At any point throughout the duration of the game the up joystick can be held down, clearing the screen and presenting a selection of cars. The player can then adjust the left potentiometer, allowing the selection box to flick between vehicles. Once

the left button is released, whatever car the selection box was left on will be assigned as the new race car for the duration of the game, or until a new car is selected.

The right potentiometer is designed to be very easy to use. By simply adjusting this potentiometer should either increase or decrease the back light on the teensy. This will mean that if players are noticing any kind of lag, this can be adjust so it is far easy to see.

Expected Outcome: After testing all of these features, these operated exactly as epxected. This helps to further increase the functionality and overall usability for the players.

3.2 De-bounce All Switches

3.2.1 Description of Feature

To improve the functionality of the game, all of the switches were de-bounced. The reason that these were de-bounced was to ensure that when the buttons was pressed, this was definitely only recognised as a single press instead of multiple presses.

3.2.2 Global Variables and Functions Used

The de-bouncing of these switches was implemented through the use of an interrupt service routine of Timer 0 (lines 1319-1358). The first step in de-bouncing the switches was to include all of the switches in an array called **'switch_pressed'**. After this, the algorithm for de-bouncing these switches was quite simple, with this allowing each switch to then be recognised as either 'officially open' or 'officially closed'.

3.2.3 Test Plan and Results

This feature is quite hard to test, however it was checked manually when generating the code by placing each of the switches in a formatted string, and then observing the state when each was pressed. It was seen that when the switch was pressed, the 0 was change to a 1, and then back to a 0, proving that the de-bouncing was successfully implemented. The algorithm also makes sense so further proves why this is correct.

3.3 Direct Screen Update

3.3.1 Description of Feature

As it is quite easy to forget about how much fuel remains in the tank, a necessary feature to the game was to create a flashing fuel tank that would begin when the fuel was less than 25%. This means that players are more aware of how much fuel they have left when it begins to get low. The reason it was implemented as a direct screen update was because it seemed like an important aspect for the player to know, even whilst the game is paused. With this being directly written to the LCD display, this means that it will always be on the screen, therefore allowing it to accurate serve its purpose.

3.3.2 Global Variables and Functions Used

There were a number of global variables used for the function **'draw_flash_fuel' (lines 1227-1245)** and **'erase_flash_fuel' (lines 1247-1258)** to be successfully implemented. The main one of these was the bitmap array **'fuel1_original'**. When initially setting up this bitmap to ensure it had the same appearance as a sprite, it was important to use a for loop in the function **'setup_flash_fuel'** to write each bit. After this, the LCD command function was used so the image could be positioned in the correct location. For this function to work in the desired way, the function **'draw_flash_fuel'** was necessary, with this sending the data to the LCD screen, therefore meaning the fuel tank was visible. By using an if statement to check whether the global variable **'fuel'** was less than 25, this would then ensure the fuel tank would be shown on the screen. Although the function **'erase_flash_fuel'** did not need to be implemented for this to meet the needs of the direct screen update, the incorporation of this allowed the update to be even more obvious for players. This function is almost exactly the same as the display function, however this one ensures that nothing is displayed on the screen (LCD_DATA(0)). By then using an if statement, all odd numbers below 25 will result in the fuel tank being erased. This therefore creates the effect of a flashing fuel tank, with this also being proportional to the speed of the vehicle. This means that the faster the car travels, the faster the fuel will begin the flash.

3.3.3 Test Plan and Results

Setup Process: The setup process to test this feature requires the car to have used 75% of its fuel. Once this has been used and 25% remains, the player can observe this direct screen update.

Expected Outcome: The expected outcome of this feature is that the fuel tank at the top of the screen will begin to flash when the fuel is an even number below 25%. This should look similar to all of the sprites, and also flash at a rate proportional to the speed of everything on the screen.

Actual Outcome: The actual outcome is exactly what was expected, with this therefore providing an effective method of notifying the player that their fuel tank is almost empty.

3.4 Timers and Volatile Data

3.4.1 Description of Feature

Timers and volatile data were used throughout the entire game. The first case of this was the 16 bit fast timer (Timer 3) used to generate the game time. Along with this, Timer 1 was also used, with this simply being needed to generate the random seed at the beginning of the game. The reason 2 separate timers were used was because the game timer would always begin from 0 when the splash screen was cleared, therefore meaning the scenery and obstacles were never randomised accurately. Therefore, by using a different timer, this meant that the random seed did not have to start from this point, ensuring that a different random seed was always created. Along with this, Timer 0 was also used for timing the acceleration and speed. This proved to be a far more effective method than simply relying on the process delay time.

3.4.2 Global Variables and Functions Used

All of the functions and variables used for these timers have been outlined earlier in the report, so refer to Part B for the acceleration and speed explanation involving the interrupt service routine, and Part A for the timer.

3.4.3 Test Plan and Results

There are no specific test cases for this, except that the timer functions correctly (which is does as discussed earlier), and that the acceleration and brake times are accurately implemented (this feature has also already been extensively discussed). Therefore, this use of timers and volatile data has most definitely been used in an appropriate way.

3.5 PWM

3.5.1 Description of Feature

When testing the game, it was noticed that if the back light was set to its maximum setting, it often made it very hard to see the screen properly. Along with this, it often added an element of 'lag' (not in terms of the game speed but simply the aesthetics). After significant testing, it was determined that by altering this back light brightness, this issue would always be fixed. For this reason, it seemed suitable that the user should be able to adjust the screen brightness of their screen to suit their specific teensy. This can be done using the right potentiometer, and ensures that all players have a fair opportunity when playing the game.

3.5.2 Global Variables and Functions Used

The functions used to implement this feature were **'set_duty_cycle' (lines 1308-1311)** and **'light_fade' (lines 1313-1316)**. By using the define variable **ADC_MAX'**, allows the threshold limit of the potentiometer to be appropriately adjusted, therefore altering the brightness of the back light whenever this potentiometer is adjusted. As the potentiometer keeps this set value, this also means that when the player finds the ultimate back light for their teensy, this will no longer need to be adjusted at all.

3.5.3 Test Plan and Results

Setup Process: In order to test this feature, it is a simple matter of beginning the game and adjusting the right potentiometer.

Expected Outcome: It should be noticed that when the right potentiometer is adjusted, this alters the back light on the teensy. Plays should adjust this until it suits their device, with this then being set as their back light setting for all other games they play.

Actual Outcome: After following the setup procedure and adjusting the back light, this was proven to work as expected. It also allowed the teensy to be far easier to see, providing evidence that this will be a crucial element in creating a game that works on all devices.

3.6 Pixel-level Collision Detection

3.6.1 Description of Feature

With the regular collision algorithm being too inaccurate for such a tiny screen, pixel level detection was implemented to allow players to have close encounters with scenery. This ensured that no collisions would occur when the car technically hand touched the object, yet had only touched the bounding box of that object.

3.6.2 Global Variables and Functions Used

This algorithm for checking for collisions has been implemented in the function **'sprites_collided' (lines 1105-1127)**. The way this was solved was by firstly noting that all sprites were no longer than 8 bits wide, meaning that the maximum overlap when collided would be 15 bits wide. With this being the case, a 'uint16_t' word was then used to store each single row. After this, only these rows with vertical overlap were considered. This then allows one byte from the right most sprite to be right shifted as shown in line 1120. The bitwise & operator was then used, with this resulting in only the collided pixels being presented. This works due to truth tables, with a 1 and a 0 being a 0 and a 1 and a 1 being a 1 (collision). In conjunction with this and the if statement on lines 118-119, this then means that only the overlapping pixels are detected, and the collisions can be successfully found. This has also been done using only 1 for loop to process large pixel groups, and uses the bitwise & operator to achieve the desired results.

3.6.3 Test Plan and Results

Setup Process: The setup process this for this feature is very simple, with the player simply needing to start the game and begin the race to 'Zombie Mountain'.

Expected Outcome: The expected outcome is that the player will be able to move extremely close to all of the scenery and obstacles, with random invisible collisions no longer being present.

Actual Outcome: After driving the car up to all of the scenery and obstacles, this pixel level detection was proven to work. This allows players to more accurately play the game, and ensures that lives are only lost when these items definitely touch the car.

3.7 Bidirectional serial communication and access to file system

3.7.1 Description of Feature

This feature was one of the most complex components of the game, not only to successfully implement, but also to setup. The basic way this serial communication works is by saving the file, reading from the generated text file, and then sending this decoded information back to the game as global variables. For example, if the damage was initially saved as 3, this will be sent back to the game, with this variable overwriting the current one on the screen when it is loaded.

3.7.2 Global Variables and Functions Used

3.7.3 Test Plan and Results

Setup Process: In order to setup bidirectional serial communication, the first and most important step is to ensure that 'TheServer.c' has been included in the Makefile as 'TheServer.exe'. If this has not been completed, this save and load will not function at all. After ensuring that this is setup correctly, and that the 'make' command in cygwin has successfully compiled these documents, the teensy should be reloaded to ensure everything is set up correctly. After this, the command '/ls /dev' should be typed into the cygwin window. This will provide a list of file directories, but special attention should be payed to the 'tty' ones. After determing which of these should be used (in most cases ttyS3 will be used), the commnand './TheServer /dev/ttyS3' must be entered. This will then setup a screen similar to the way the first game was created. This is the main setup procedure for this component of the game, and will ensure that the serial communication works properly as outline below.

Expected Outcome: With everything being setup and the game having commenced, the user should still see the blank cygwin screen with nothing currently on it. At any point during the game, the player can press the down joystick to pause the game, and whilst holding this down, press the left button. This will result in the file being saved in the text file ' CurrentGameStates'. A message will also be presented on the screen providing the player with evidence that all of the current game states have definitely been saved. This document purely contains a long list of numbers, with each of these either being a single display variable (such as speed, fuel, damage etc), or a set of scenery, obstacle, car, road or fuel coordinates.

With the game states being successfully saved, the next step is for the player to load this data back onto the device! This is an exciting feature as if a player has only 1 life remaining and wants a second chance, this data can be reloaded from when the damage for example was 5 instead of 1. Using this is very similar to how the game states were saved, except this time the right button should be pressed when the pause control is held. This will present a ' Loading...' string on the screen of the teensy, with a subsequent message stating that loading was successful (this will be shown on the cygwin screen).

Actual Outcome: After significant testing, the desired effect was achieved. Whenever the save command is prompted, this successfully generates a new text file, with this then containing all of the relevant game states. Along with this, the load command was also proven to work accurately. This was quite easy to test, as a simple photo can be taken at the time of saving, and this can be compared to the screen at the time of loading. Although being a challenging feature, this really helps to increase the complexity of the game, whilst almost creating a 'hack' for clever players.

Conclusion:

The game 'Race To Zombie Mountain' successfully generates a challenging game, allowing players to try again as many times as they would like for personal bests. This game allows players to not only choose their desired race car, but to dodge and weave from any dangerous objects that may cause damage to their car. Along with this, the game has the very exciting option of saving and loading the previous game states off the teensy, into a text file and back onto the teensy. Although

the game was limited due to the size of the NOKIA5110 screen, this leaves the game with a retro old school feel.