CAB202 - Microprocessors and Digital Systems

Assignment 2

Pedro Alves (n9424342)

June 3, 2018

Executive Summary

The objective of this report is to give an overview on the process that went into creating *Zombie Race*. Each section contains information on the implementation of a feature in the game. Multiple tests were conducted were appropriate in order to prove that the required features were correctly implemented.

I have completed Part A and B of the assessment brief and the majority of Part C. The full source code for the game and accompanying server have been submitted via AMS (submission reference number: edb7ce0b-e5ff-4229-8f3d-b76a9ad26a9a).

Contents

Instructions

1. Attempt to drive as far as possible without running out of fuel or HP. You win by crossing the finish line
2. HP is lost by hitting obstacles
3. Immediate death if a fuel station is hit
4. To refuel, hold break while immediately next to a fuel station. The car will be pitted automatically if travelling under the pit limit (speed less than 3) with the breaks held

General

Function	Input	Key
Decrease contrast	Scroll right up	Pot1
Increase constrast	Scroll right down	Pot1

Splash Screen

Function	Input	Key
Play game	Left button	SW2
Play gmae	Right button	SW3

Game

Function	Input	Key
Move left	Joystick left	SW1
Move right	Joystick right	SW1
Pause	Joystick center	SW1
Accelerate	Button right	SW3
Decelerate	Button left	SW2
Limit speed		
Increase Limit	Left scroll up	Pot0
Decrease Limit	Left scroll down	Pot0

Game Paused

Function	Input	Key
Unpause	Joystick center	SW1
Save game	Joystick up	SW1
Load game	Joystick down	SW1

Game Over screen

Function	Input	Key
Play again	Button right	SW3
Splash screen	Button left	SW2
Load game	Joystick down	SW1

Program Overview

Zombie Race is a top-down racing game where the player attempts to drive as far as possible without running out of fuel or colliding with an obstacle. The implementation of this game has been split into several stages that are explained in the later sections.

The basic architecture of the program is that of a state machine. The states for this program are the different screens which the user can see and each provides different functionalities that will be further explored in their specific sections.

```
// Lines 136 - 140
enum GameScreens {
  START_SCREEN,
  GAME_SCREEN,
  GAMEOVER_SCREEN,
} game_screen;
```

After initial setup is complete, the program enters an infinite loop that runs at a rate of about 60 times per second. Inside the loop are two functions, *update()* and *draw()*.

```
// Lines 382 - 411
void update(void)
```

This function calls the specific update function for the current state the game is in. It will also handle input from *Pot0* that controls the current contrast level of the LCD screen and will perform some operations to allow other functions to use edge detection of the teensy's inputs (ie. only update when clicked).

```
// Lines 416 - 411
void draw(void)
```

Will call the draw function of the current state the program is in. Draw function do not change any variables and serve only to write to the LCD through the use of a buffer or directly. Direct draw calls to the LCD will be further explored in the section *Direct screen update*.

```
// Lines 1470 - 1502
void teensy_setup(void)
```

Performs all preliminary calls to setup registers and variables that will be used throughout the program. Will set the clock speed to 8 MHz and the initial LCD contrast to default. The other calls that occur in this functions will be explored in the later sections as they become relevant.

Testing Procedures

Due to difficulties with capturing the state of the game directly from the screen of the Teensy, a USB bi-directional communication was set-up between the Teensy and a server. The current value of variables can then be sent as messages to the server in order to assist with debugging, testing and saving/loading.

In order to run the server, enter the command through the Cygwin terminal: *./server /dev/ttyS2* where *ttyS2* is the device name of the Teensy. If this doesn't match, use the command *ls /dev* to find the name for the Teensy currently connected.

The functions used to achieve USB communications will be explored in the section *Bidirectional serial communication and access to file system.*

Testing was accomplished by sending the current state of selected variables via the function *usb_send_message()*. Multiple states were then compared to verify if the actual outcome matches the expected outcome.

Figure 1 shows the result in the server program when the following command is run

```
usb_send_message(DEBUG, 4, buf, 100, "Timestep: %.3f\nCondition: %d\nFuel: %d\nDistance:
  %d\n%d\n", elapsed_time(game_timer_counter), condition, fuel, distance, 0);
```

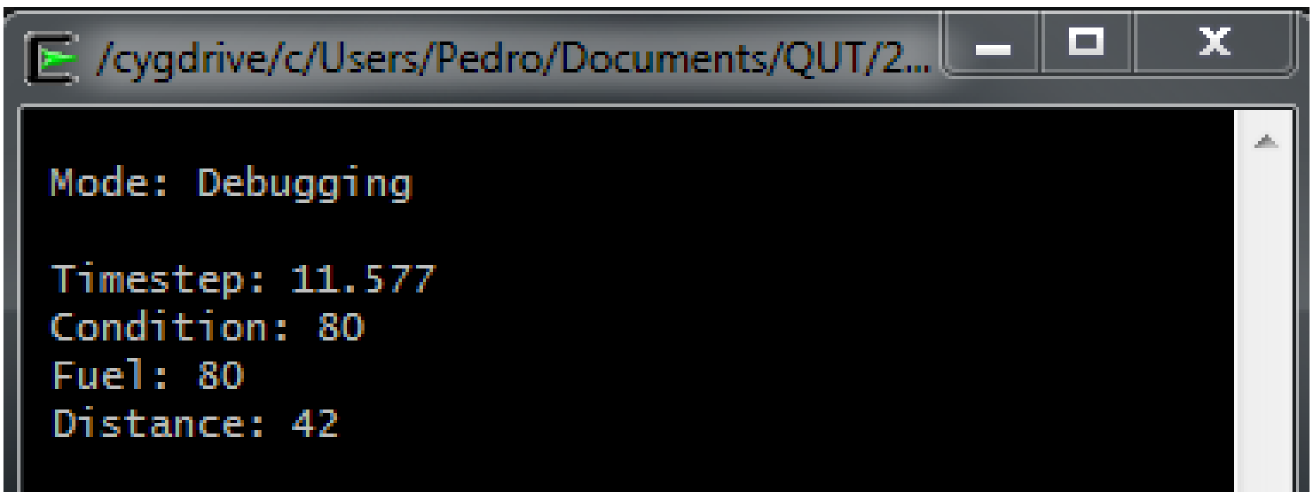

Figure 1: Screenshot of the server program when a debug command is sent with the current variable states

To improve readability of the test plan in the later sections, the information received from the server will be scribed into a table format. Code that was used to perform the tests can be seen in the function *game_screen_draw()* - Lines: 590 - 627.

Testing was done by using a srand seed of 100 in order to allow reproduction later of the same tests.

Splash Screen

The splash screen is the first screen shown when the game is started. After the game is over, the player also has a choice to return to the splash screen. It will display the name of the game and name of the author while waiting for the user to choose to continue playing. The user can choose to start the game by pressing the SW2 or SW3 buttons.

Globals

```
// Line 137
GameScreens START_SCREEN;
```

The value *START_SCREEN* from the *GameScreens* enum is associated with the splash screen. For more info on the *GameScreens* enum global, see *Program Overview*.

```
// Line 85
uint8_t controls_states[7];
```

The current states of the SW2 (left) and SW3 (right) buttons (for more info see *Debouncing*). A state of 1 means the button is pressed. The splash screen will change to the game screen as soon as any of these two variables have a value of 1.

Functions

```
// Lines 515 - 526
void start_screen_update(void)
```

The update function associated with the splash screen that will be called every tick of the game loop. Will check if SW2 or SW3 have been pressed and call *change_screen(GAME_SCREEN)* if true.

```
// Lines 531 - 535
void start_screen_draw(void)
```

Will call *draw_string()* to print the name of the game, name of the author and student number to the teensy LCD.

Testing

The following test cases need to pass for this section's tests:

- Game starts when SW2 is pressed
- Game starts when SW3 is pressed

Test Case: Game starts when SW2 or SW3 is pressed

Lines 457 and 471. A dash in the timestep means that digit was changing rapidly in the server's window.

Timestep	button_left_state	button_right_state	game_screen	Test result
0.00-	0	0	1 (START_SCREEN)	Pass
0.006	0	1	2 (GAME_SCREEN)	Pass
0.00-	0	0	1 (START_SCREEN)	Pass
0.004	1	0	2 (GAME_SCREEN)	Pass

Dashboard

A sub-window in the LCD which displays stats about the player car such as the condition, fuel and speed. A border separates the player from the dashboard area and the player's car is unable to physically move past it (for more info on this, refer to the section *Collision*). It will also display the character 'R' to notify the player that the car is currently refuelling.

Globals

```
// Lines 92 - 94
uint8_t condition;
uint8_t fuel;
double speed;
```

The variables that hold information about the car. They are modified by other functions thus the dashboard only reads their current values.

```
// Line 130
bool refuelling;
```

Used to check if the car is currently refuelling. *dashboard_draw()* will check this and draw the character 'R' if true.

```
// Line 30
#define DASHBOARD_BORDER_X   26
```

The right-most x coordinate of the dashboard. The border is drawn at this line

Functions

```
// Lines 658 - 679
void dashboard_draw(void);
```

Called in the draw function of the game screen. Will draw the line separating the dashboard from the rest of the game screen then will call the *draw_string()* and *draw_formatted()* functions in other to display the current game information. If the player is currently refuelling, will display the character 'R' at the bottom.

Testing

Testing for this section is a mixture of server and visual analysis. The current values to be displayed on the dashboard are sent to the server and visual analysis of the Teensy's LCD screen will determine if the test has passed.

Test Case: Values are correctly displayed

Timestep	Condition	Fuel	Speed	Test result
0.102	100	100	10	Pass
10.289	80	78	6	Pass
21.383	40	86	1	Pass

All the tests passing mean that the expected values (the ones in the table and shown in the server) match the values seen in the dashboard.

Paused View

The user can pause the current game by pressing the center joystick command (SW1). This is only possible when the game state is *GAME_SCREEN*. The paused screen will display extra information such as the total distance travelled by the car and the elapsed time since the start of the game. While the rest of the game view will be behind the paused view, the player's car can still be visible in order to facilitate the regain of control when unpaused. The time only increases when the game is being played, therefore the counter associated with the current game time must be paused when the paused view is active.

Globals

```
// Line 95
uint8_t distance;
// Line 103
uint16_t game_timer_counter;
```

The information that is displayed in the Paused View. These variables are modifed in other sections and Paused View only reads their current value.

```
// Line 104
uint8_t game_paused;
```

Determines if the game's Paused View should be active or not. Modified by the centre Joystick.

```
// Line 105
double time_paused;
```

The return value from elapsed_time(game_timer_counter). The value is set when the game is paused to avoid the thrid decimal point of the time value fluctuating widly due to floating point precision.

```
// Line 85
uint8_t controls_states[7];
```

game_paused is only toggled when the middle joystick is clicked and just held. This avoid the game pausing and un-pausing multiple times when the user only intended to pause or unpause once with one click. For more info see *Debouncing*.

Functions

```
// Lines 347 - 350
double elapsed_time(uint16_t timer_counter)
```

Used to get the time since the game started in seconds. Discussed in more detail in *Timer and volatile data*.

```
// Lines 540 - 571
void game_screen_update(void)
```

Checks if the middle joystick (SW1) has been clicked in order to toggle *game_paused*. Will also get the time that the game was paused.

```
// Lines 576 - 653
void game_screen_draw(void)
```

If the game is paused, instead of drawing the game screen as usual, it will draw the car and display the stats required. The dashboard is also drawn in order to display as much information as possible about the game.

Testing

The tests for this section will also incorporate a mix of debugging and visual analysis. For every test plan, the data sent to the server will be verified with the data being displayed in Paused View to make sure they match. The following test plans were developed to ensure full functionality of this section:

- Passage of time is correct. Game time does not increase when paused
- Distance covered is relative to the speed
- Multiple pause-unpause-pause commands

Test Case: Passage of time is correct. Game time does not increase when paused

Game time	Expected game time	Comments	Test result
1.092	-	SW1 is pressed to pause the game	-
1.092	1.092	No inputs are given, verify time doesn't change after 5 seconds	Pass
6.714	˜6	Game is run for 5 seconds then paused, verify time changes	Pass

Test Case: Distance covered is relative to the speed

The max speed is adjusted by using Pot0 to improve accuracy (only need to hold accelerator instead of trying to maintain correct speed). The game is restarted after every attempt so the time step can show the amount of time travelled at that speed.

Timestep	Distance	Expected distance	Speed	Test result
0.140	0	0	-	Pass
5.325	28	-	10	Pass
5.212	20	<28	6	Pass
5.906	11	<20	3	Pass
5.525	0	0	0	Pass

Test Case: Multiple pause-unpause-pause commands

The center joystick button is pressed every 5 seconds in order to verify that the time moves or stays constant depending on the state of the game.

Timestep	Expected timestep	Paused	Test result
4.653	5	No	Pass
4.653	4.653	Yes	Pass
10.265	10	No	Pass
10.265	10.265	Yes	Pass

Horizontal Movement

When the user moves the Joystick to the left or right, the car will also move in that direction. The car is unable to move when the speed is zero or if by doing so, it would go out of bounds to the right of the screen or move into the dashboard area. More functionality is added in *More realistic steering*.

Globals

```
// Line 85
uint8_t controls_states[7];
```

If any of the states are true, the car will move left or right. For more info see *Debouncing*

Functions

```
// Lines
void game_screen_step(void);
```

Checks if the joystick has moved to the left or right and calls *player_car_move()*. As *game_screen_step()* is only called if the speed is not zero, this already makes sure the non-zero condition is met.

```
// Lines 685 - 739
bool in_bounds(double x, double y);
```

Will check if the car will go out of bounds by moving off the game screen to the right or into the dashboard area to the left.

```
// Lines 852 - 867
void player_car_move(int dx);
```

If the car will be in bounds, moves it one pixel in the specified direction.

Testing

A more comprehensive testing plan will be done in the section *More realistic steering*. Collision with other objects and offroad speed limits were disabled in order to conduct testing as required by the brief.

Test Case: Movement is accurate depending on speed and inputs

For every new entry in the table, the game was restarted in order for the time step to display the amount of seconds that the input and speed were affecting the horizontal movement of the car. After verifying that the car doesn't move when speed is 0, the left and right joystick are pressed to check if the car stays in bounds even after it reaches the edge of the playing area.

Timestep	Player x coord	Expected x coord	stick_left_state	stick_right_state	Speed	Test result
0.228	53	53 (Middle of road)	0	0	10	Pass
6.476	53	53	1	0	0	Pass
5.239	53	53	0	1	0	Pass
11.241	27	27 (dashboard+1)	1	0	10	Pass
10.944	79	79 (LCD_Y - 1)	0	1	10	Pass

Acceleration and Speed

The user can choose to accelerate the car by pressing the left button (SW2) or decelerate by pressing the right button (SW3) with more functionality to this being added in the section *Accelerator and brake.* The speed is limited to a minimum of 0 and a maximum of 10 on the road and 3 off-road. This speed limiter can be modified by relative position of Pot0, more on this feature is covered in section *ADC*. The current speed value is displayed to the user and correctly reflected in the dashboard.

Globals

```
// Line 85
uint8_t controls_states[7];
```

The left button will accelerate the car and the right button will decelerate. For more info see *Debouncing*

```
// Lines 44 - 46
#define SPEED_MIN           0
#define SPEED_MAX           10
#define SPEED_OFFROAD_MAX   3
```

Will set the limits for the speed.

```
\Lines 42 - 43
#define SPEED_THRESH        8
#define SPEED_FACTOR        8
```

Used to calculate the increase in speed with every loop iteration.

```
// Line 29
#define TIMER1_FREQ     7812
```

The frequency TIMER1 is set to.

```
// Line 94
double speed;
```

The current speed value of the car.

```
// Line 98
double speed_counter;
```

Used to decide when to update the game logic.

Functions

```
// Lines 1470 - 1502
void teensy_setup(void)
```

Will set Timer1 to CTC mode in a way that ISR(TIMER1_COMPA_vect) is called 60 times a second. Section *Timers and volatile data* goes more indepth on the values for the registers.

```
// Lines 1609 - 1614
ISR(TIMER1_COMPA_vect)
```

If the game isn't paused and the program state is in the game screen, it will increase the speed counter by a rate of speed/SPEED_FACTOR.

```
// Lines 540 - 571
void game_screen_update(void);
```

If the speed_counter is greater than SPEED_THRESH, the game logic will be updated by one tick.

```
// Lines 685 - 739
void game_screen_step(void)
```

Updates all game logijc by one tick. Since it's called by *game_screen_update()* when the *speed_counter* is greater than *SPEED_THRESH*, every function or variable called from this function will be affected by how fast the car is moving.

```
// Lines 872 - 926
void player_speed_input(void);
```

Will increase or decrease the speed depending if the accelerator or brake is pressed. *Accelerator and brake* will go more in depth regarding this function. This function will ensure the speed is constrained by the limits set.

```
// Lines 1296 - 1306
bool offroad(Sprite sprite)
```

Checks if the sprite sent by the input is outside the boundaries of the road (both to the left or to the right of the road).

Testing

The majority of testing regarding the speed and acceleration will occur in the sections *Accelerator and brake* (combinations of accelerate/decelerate with car in different speeds) and *ADC* (obey limits).

Test Case: Speed stays in the limits set

The right button (SW3) is pressed for 5 seconds in order to keep accelerating the car until the maximum speed. The left button (SW2) is pressed for 5 seconds to brake the car to make sure the speed is limited to minimum of 0.

Timestep	Speed	Expected speed	button_left_state	button_right_state	Offroad	Test pass
0.230	10	10	0	1	0	Pass
5.346	10	10	0	1	0	Pass
1.348	3	3	0	1	1	Pass
6.374	3	3	0	1	1	Pass
2.047	0	0	1	0	0	Pass
7.117	0	0	1	0	0	Pass

Scenery and obstacles

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

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

The amount of scenery that can spawn in the game is based on the dimensions of the screen. This ensures that the game is fair for all screen sizes and that at least 5 obstacles are in view at any one time.

Globals

```
// Line
#define NUM_TERRAIN           10
```

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

```
// Line 70
Sprite terrain[NUM_TERRAIN];
```

An array that holds all sprites representing terrain.

```
// Line 76
#define NUM_HAZARD            2
```

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

```
// Line 124
Sprite hazard[NUM_HAZARD];
```

An array that holds all sprites representing road hazards.

```
// Lines 66 - 67
#define TERRAIN     0
#define HAZARD      1
```

Used to define the different types of obstacles.

```
// Lines 70 - 73
#define NUM_TERRAIN_TYPES     2
#define TERRAIN_TREE          0
#define TERRAIN_SIGN          1
```

Used to differentiate between the different types of terrain.

```
// Lines 46-48
#define NUM_HAZARD_TYPES      2
#define HAZARD_TRIANGLE       0
#define HAZARD_SPIKE          1
```

Used to differentiate between the different types of road hazards.

```
// Lines 143-150
Sprite terrain_image[NUM_TERRAIN_TYPES];
Sprite hazard_image[NUM_HAZARD_TYPES];
```

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

```
// Line 82
#define HAZARD_SPAWN_CHANCE 15
```

The percentage chance that a hazard that is out of bounds will spawn again (15 is quite high as this is called every game step tick).

Functions

```
// Lines 978 - 983
void terrain_image_setup(void);
```

Setup the sprites in *terrain_image* that will be used as a reference for each terrain type.

```
// Lines 1095 - 1100
void hazard_image_setup(void);
```

Setup the sprites in *hazard_image* that will be used as a reference for each hazard type.

```
// Lines 988 - 1007
void terrain_setup(void);
```

Iterates through the *terrain* array and calls *terrain_reset()* in order to create a type of terrain for each index. Not all terrain may appear on the screen at once (some may be spawned above the playing area because there is no more space).

```
// Lines 1013 - 1076
void terrain_reset(int index, int y_bot);
```

Moves the terrain so that it's bottom y-coordinate is at y_bot. The type of terrain and it's x-coordinate will also be randomised. Nothing will happen if it collides with another obstacle which means *terrain_step()* will call this again next game tick for the same index (as it'll still be scrolling further below the screen).

```
// Lines 1082 - 1090
void terrain_step(void);
```

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

```
// Lines 1105 - 1124
void hazard_setup(void);
```

Creates hazard sprites and fill the hazard array with them. Will then call *hazard_reset()* in order to put them on the screen.

```
// Lines 1130 - 1168
void hazard_reset(int index, int y_bot);
```

Moves the hazard corresponding to the index to the game world so that its bottom y coordinate matches y_bot. The type of hazard and its x-coordinate will also be randomised. Nothing will happen if it collides with another obstacle which means *hazard_step()* will call this again next game tick. The hazard and terrain setup, update and reset functions are similar but need to be separated due to different arrays being used and both having different limitations on where they can be spawned.

```
// Lines 1173 - 1185
void hazard_step(void);
```

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

```
// Lines 576 - 653
void game_screen_draw(void);
```

Will draw all of the obstacles to the game screen.

Testing

The second terrain in the terrain array was used to verify all test plans.

Test Case: Terrain scrolls

Timestep	Terrain y	Expected y	Speed	Test result
1.742	0	0	10	Pass
2.896	26	>0	10	Pass
3.871	48	>26	10	Pass
3.350	0	0	5	Pass
5.688	27	>0	5	Pass
7.575	46	>27	5	Pass
2.649	21	-	0	Pass
8.165	21	21	0	Pass

This test also shows that terrain scrolls relative to the speed of the car. At a speed of 10, it took the terrain about 2 seconds to reach the bottom of the screen. At a speed of 5, it took the terrain 4 seconds.

Fuel Depot

The Fuel Depot is a type of obstacle that refuels the player's car if it parks next to it, the refuel rate has been added in the section *Fuel level increases gradually* . To smooth out the gameplay, the player will automatically park the car if they're travelling at a speed of 2 or less and are pressing the brakes. A collision with the fuel depot will immediately end the game for the player, regardless of the car's condition.

Globals

```
// Lines 62 - 63
#define FUEL_STATION_MIN      140
#define FUEL_STAION_MAX       180
```

The minimum and maximum distance from the top of the screen that the fuel station can spawn. Note that these numbers aren't the distance values shown in the Paused View but are in how many game steps it would take to the fuel station to appear at the top of the screen.

```
// Line 128
Sprite fuel_station;
```

The sprite which represents the fuel depot.

```
// Line 129
int fuel_station_counter;
```

Counts down the game steps until the fuel station can spawn on top of the screen.

Functions

```
// Lines 1191 - 1231
void fuel_station_reset(void)
```

Will spawn the fuel station at the top of the screen. This will force the next road section (see *Curved road*) to be a straight section. This functions randomly chooses which side of the road to spawn and will also remove any terrain in the way.

```
// Lines 1237 - 1252
void fuel_station_step(void)
```

Will step the sprite for the fuel depot by moving it downwards one pixel and then check if it went out of bounds. If it has, it will reset the *fuel_station_counter* to a value between the limits set by the constants.

```
// Lines 745 - 784
void game_screen_setup(void)
```

Will set the *fuel_station_counter* to a value between the limits set by the constants.

```
// Lines 576 - 653
void game_screen_draw(void)
```

Will draw the fuel staion to the playing area.

Testing

Although the fuel station is closely linked to the other obstacles and should move in the same way, a testing plan was made to verify.

Test Case: Fuel station scrolls

Timestep	Depot y	Expected y	Speed	Test result
8.115	6	˜0	10	Pass
9.352	34	>6	10	Pass
9.916	47	>34	10	Pass
12.087	0	˜0	5	Pass
14.400	27	>0	5	Pass
16.059	47	>27	5	Pass
11.414	20	-	0	Pass
23.195	20	20	0	Pass

This test also shows that the fuel depot scrolls relative to the speed of the car. At a speed of 10, it took the depot about 2 seconds to reach the bottom of the screen. At a speed of 5, it took the depot 4 seconds.

Fuel

The car start with a fuel tank which decreases as it moves. The faster it moves, the faster the fuel is depleted. After parking next to a fuel depot, the fuel tank is refilled to max, the rate at which the fuel is increased is covered in the section *Fuel level increases gradually*. The fuel tank can also be refilled to max after a collision with an obstacle.

Globals

```
// Line 93
double fuel;
```

The current amount of fuel available to the player.When it reaches 0, the game is over.

```
// Line 130
bool refuelling;
```

Represents whether the car is currently refuelling.

```
// Line 60
#define FUEL_FACTOR        3
```

Affects the rate at which the fuel decreases.

```
// Line 97
uint8_t distance_counter;
```

When the distance counter is greater than the *FUEL_FACTOR*, fuel is decreased by one and distance increased by one. As the distance counter is incremented every time the game is stepped, it means the rate that fuel depletes is affected by the speed of the car.

Functions

```
// Lines 1257 - 1268
void check_refuel();
```

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

```
// Lines 1273 - 1291
void refuel();
```

Called every time the game steps. If the car isn't already refuelling, call *check_refuel()*. Otherwise it will make sure the car's speed has remained at 0 and the brake is pressed.Every time this function is called, the fuel is increased by a rate described in the section *Fuel level increases gradually*.

```
// Lines 685 - 739
void game_screen_step(void)
```

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

Testing

The car refuelling will be tested in the section *Fuel level increases gradually*. To ensure all other functionality meets the criteria, the following test plan was performed.

Test Case: Fuel depletes proportionally to the speed

Timestep	Fuel	Expected fuel	Speed	Test result
0.043	100	100	10	Pass
5.040	72	<100	10	Pass
10.288	48	<72	10	Pass
0.179	100	100	5	Pass
5.390	85	>72	5	Pass
10.055	71	>48	5	Pass
0.431	100	100	0	Pass
10.313	100	100	0	Pass

Distance Travelled

Globals

```
// Line 97
int distance_counter;
```

Incremented every time the game steps. Distance is increased by one every time the counter is larger than *FUEL_FACTOR*.

```
// Line 95
uint8_t distance;
```

Represents the units of distance travelled by the car since the game started.

```
// Line 96
uint8_t finish_line;
```

The distance that the player has to complete before they win the game. Setup in *game_screen_setup(void)*. Currently hardcoded to 250 (no reason to make another global).

Functions

```
// Lines 685 - 739
void game_screen_step(void)
```

Will increment *distance_counter* and will check if the distance can be increased as well. Will decrease *finish_line* every time the distance is updated and when it reaches 0, the player has won.

Testing

Testing to see if the player has won is done in the section *Game Over Dialogue*.

Test Case: Distance updates proportionally to the speed

Timestep	Distance	Expected distance	Speed	Test result
0.047	0	0	10	Pass
5.218	29	>0	10	Pass
10.444	59	>29	10	Pass
0.060	0	0	5	Pass
5.145	14	<29	5	Pass
10.251	29	<59	5	Pass
0.257	0	0	0	Pass
10.186	0	0	0	Pass

Collision

Collision uses simple bounding box detection - this functionality is expanded in the section *Pixel-level collision detection*. The car is reset and the condition reduced if the car hits an obstacle head on. The car's horizontal movement is stopped if it tries to move into an obstacle.

Globals

```
// Line 92
uint8_t condition;
```

The only global added by this section. Collision detection makes use mostly of globals already implemented when the scenery is created.

Functions

```
// Lines 1311 - 1338
bool check_collision(Sprite sprite);
```

Iterates through every obstacle in the game and checks if the sprite passed to this function collides with any of therm.

```
// Lines 1343 - 1356
bool check_sprite_collided(Sprite sprite1, Sprite sprite2)
```

Checks if the two sprites passed to this function collide with each other.

```
// Lines 685 - 739
void game_screen_step(void)
```

Will check if the player has collided with any object every time the car moves. Will also check if the car has collided with a fuel depot and throw the game over dialogue if it has.

```
// Lines 1390 - 1416
void handle_collision(void)
```

When it's found that the player has collided with an object that is not the fuel depot, this function will reset the location of the player and clear any hazards on the way while also reducing the car's condition and changing to the game over screen if it reaches 0.

```
// Lines 852 - 867
void player_car_move(int dx)
```

Won't allow the player to move if it means colliding sideways with an obstacle

Testing

In order to ensure all functionality has been implemented, the following test plans were devised:

- Car does not move when doing so would result in a sideways collision
- Car collides head on with an obstacle

Car does not move when doing so would result in a sideways collisions

The two x values under car/obstacle x refer to leftmost x value in the object's bounding box and the rightmost x value in the object's bounding box (x + width) respectively. If the first x of the car is less than the second x of the obstacle, then there should be a sideways collision due to the car wanting to move left. If the second x of the car is larger than the first x of the obstacle, then there should also be a collision due to the car wanting to move right. If it is equal, then the car and the object are side by side.

Timestep	Car x	Object x	stick_left_state	stick_right_state	Collision	Expected collision	Test result
0.473	53 - 57	55 - 60	0	0	0	0	Pass
7.534	51 - 55	55 - 60	0	1	0	0	Pass
7.871	51 - 55	55 - 60	0	1	0	0	Pass
3.161	62 - 66	55 - 60	0	0	0	0	Pass
7.292	60 - 64	55 - 60	1	0	0	0	Pass
7.689	60 - 64	55 - 60	1	0	0	0	Pass

This test involved moving to a path next to an object, then holding the left or right joystick continuously in order to test if the car remained in the same position in order to avoid clipping into an obstacle.

Car collides head on with an obstacle

The car's x coordinate is matched with that of an object so that if it continues moving straight, a collision will occur. Testing was then done to verify that when a car did collide, its condition was updated and the player's location was reset.

Timestep	Car x	Car y	Object y	Object type	Condition	Expected condition	Test result
13.178	51-55	41	14	Hazard	100	100	Pass
18.767	47-51	41	39	Hazard	80	80	Pass
20.862	36 - 40	41	33	Terrain	100	100	Pass
25.534	53 - 57	41	41	Terrain	80	80	Pass
12.813	45 - 49	41	20	Fuel Depot	100	100	Pass
16.185	-	-	-	Fule Depot	Game over	Game over	Pass

Game Over Dialogue

The game over dialogue is given when the player has either won the game by travelling far enough or lost by crashing into an obstacle or running out of fuel. In this screen, the user can choose to go back to the splash screen with the left button (SW2), start a game immediately with the right button (SW3) or load a game from the server with the center joystick buton (SW1).
Due to the program being a state machine, the game can be easily restarted by changing to the desired state.

Globals

```
// Line 99
bool game_over_loss;
```

Represents if the game was over either through a loss (true) or win (false).

Functions

```
// Lines 790 - 809
void gameover_screen_update(void)
```

Will poll for input to check if state should be changed to splash, game or load.

```
// Lines 814 - 825
void gameover_screen_draw(void)
```

Will draw a prompt depending if the user has won or lost. Will also print commands that the user can press in order to change the state. Will display the distance and elapsed time as well.

```
// Lines 495 - 510
void change_screen(int new_screen)
```

Changes the current state of the game to one define in the *GameScreens* enum. If the state is changing to the *GAME_SCREEN*, will also call a setup function first in order to reset all of the game variabels.

Testing

Test plan: Game over displays when game is wont or lost and can differentiate between the two

The game was played twice to complete this test. In the first playthrough, the game was finished by travelling long enough and the game over screen was verified to make sure that it displays game won. In the second playthrough, the car was sent straight into a fuel station in order to check if the game over screen said that we had lost the game. This test also verifies that the distance and elapsed time (timestep) are written correctly in the LCD display.

Timestep	Distance	Game won	Expected game won	Test passed
62.577	250	Yes	Yes	Passed
11.478	53	No	No	Passed

Curved Road

The road in *Zombie Race* takes a smoothly varying but unpredictable path. The road is built in sections that determine the x-positions of new road pieces that spawn at the top of the screen. Each section is randomly assigned a smoothness factor, a direction and a length. This makes the road hard to predict, the smoothness factor (*ROAD_CURVE*) affects the sharpness in the current curve applied to the road. Once a section has reached its assigned length, a new section is created.

Globals

```
// Lines 49 - 51
#define ROAD_LEFT          0
#define ROAD_RIGHT         1
#define ROAD_STRAIGHT      2
```

Determines the direction of movement for the current section of road being created.

```
// Line 53 - 54
#define ROAD_CURVE_MIN     1
#define ROAD_CURVE_MAX     3
```

How smooth the road turns. A larger value means the road takes longer to change it's x value.

```
// Line 56 - 57
#define ROAD_SECTION_MIN   15
#define ROAD_SECTION_MAX   35
```

How many steps a road can take before the direction of movement is change (the movement can change to be in the same direction).

```
// Line 112
uint8_t road[LCD_Y];
```

The lx coordinates of the left side of the road packed into an array the size of the height of the LCD.

```
// Line 113
uint8_t road_width = 16;
```

The width of the road. This is used in conjunction with the road array in order to draw the right side of the road or detection if an object is offroad.

```
// Line 114
uint8_t road_counter;
```

Counts how many steps the road has taken. After it passes the value of *road_curve*, the road will move one pixel in the current direction it's travelling.

```
// Line 115
uint8_t road_curve;
```

The current curve value of the road. Will be between the limits (inclusive) set by the *ROAD_CURVE* constants.

```
// Line 116
uint8_t road_direction;
```

The current direction of travel of the road.

```
// Line 117
uint8_t road_section_length;
```

How many steps the road has taken with the current section. It is given a value set by the limits *ROAD_SECTION* when a new section is created and is decremented every time the road steps. Once it reaches zero, a new road section is created.

Functions

```
// Lines 932 - 973
void road_step(void)
```

Called every time the game steps. Will shift the *road* array to the right one place, effectively stepping the road down by one pixel. Depending on the properties of the current road section, it will then decide where to place the new x-coordinate in *road[0]*.

Testing

The smoothness of the road was achieved through trial and error by modifying the values of *ROAD_CURVE*.The following test plan was conducted to verify that the road curves significantly enough that the player will go offroad if they keep going straight. This was done by getting the x-coordinates of the road and the x-coordinates of the player and making sure they don't overlap.
This testing plan ocurred throughout one game session to show that the road keeps moving throughout the whole game.

Timestep	Car x	Road x	Car offroad	Expected offroad	Test result
0.615	53 - 57	46 - 62	0	0	Pass
9.954	53 - 57	30 - 46	1	1	Pass
26.358	53 - 57	31 - 47	1	1	Pass
52.416	53 - 57	49 - 65	0	0	Pass
60.502	53-57	66 - 82	1	1	Pass

Accelerator and Brake

This section expands upon *Acceleration and Speed* so it uses the globals and functions already defined there. When the accelerator (SW2) or brake (SW3) are pressed, the speed is modified by a rate that satifies the conditions outlined in the test plan.

Functions

```
// Lines 872 - 926
void player_speed_input(void)
```

Called every iteration of the game loop. It will read the state of SW2 and SW3 and modify the speed variable by a rate appropriate to the condition (outlined in the test plan). Will always make sure the speed is kept to its bounds whether offroad or on the road. This function will verify that when the accelerator is pressed, the brake will have no effect on the state of the car and vice-versa.

Testing

Test Plan: Braking - 10 to 0 in two seconds

The left button (SW2) was held to brake the car and make sure it is slows down at the required rate. The time was then taken when the car is travelling at a speed of 10 and then taken again as soon as the car hits a speed of 0.

Timestep	Expected timestep	Speed	Test Pass
0.233	0	10	Pass
2.050	2	0	Pass

Test Plan: Accelerating - 1 to 10/3 in five seconds

The right button (SW3) was held to keep the car accelerating. A snapshot was taken when the car is at a speed of 0 and then again as soon as the car hits a speed of 10 or 3. The maximum speed will depend if the car is on the road or off the road.

Timestep	Expected timestep	Speed	Offroad	Test result
0.263	0	0	0	Pass
5.291	5	10	0	Pass
2.338	2	0	1	Pass
7.053	7	3	1	Pass

Test Plan: Idle - 10/3 to 1 in three seconds

No buttons were pressed and the car was left to decelerate on it's own. First snapshop was teken when the car had a maximum speed (dependent if it was offroad or not), and then the second snapshot was taken as soon as the speed hit 1.

Timestep	Expected timestep	Speed	Offroad	Test result
0.092	0	10	0	Pass
3.308	3	1	0	Pass
0.873	0	3	1	Pass
3.743	3	1	1	Pass

Test Plan: Idle - 0 to 1 in 2/3 seconds

The brake was held at the start to make the speed equal to 0, a snapshot was then take. The brake was then released and no buttons were pressed. A second snapshot was then when the speed of the car hit 1.

Timestep	Expected timestep	Speed	Offroad	Test result
0.389	0	0	0	Pass
2.355	2	1	0	Pass
1.081	1	0	1	Pass
4.246	4	1	1	Pass

More realistic steering

This section expands upon *Horizontal Movement* so it uses the globals and functions already defined there. The rate at which the car moves horizontally is affected by the speed. Thus, the faster the car is moving forward, the faster it can swerve left or right.

Functions

```
// Lines 852 - 867
void player_car_move(int dx)
```

This moves the car one pixel to the left or right depending on the value of dx. It will also check if the car stays in bounds when moving. There is no need to make the x coordinate of the car change by a horizontal value because this function is called from *game_step()* (see *Acceleration and Speed*), therefore this function is already affected by the speed of the car.

Testing

To verify that the horizontal movement is affected by the speed of the car, the car was moved at a constant speed from its initial position until it reaches the left border of the playing area where the game was pause and the current game information sent to the server. To enable the speed to stay constant, the offroad speed limit and collisions with objects were disabled temporarily.

Timestep	Expected timestep	Car x	Speed	Test passed
0.093	0	53	10	Pass
2.305	-	27	10	Pass
0.140	0	53	5	Pass
4.364	>2.305	27	5	Pass
0.139	0	53	2	Pass
7.900	>4.364	27	2	Pass

.

Fuel level increases gradually

This section expands upon *Fuel* and *Fuel Depot* so it uses the globals and functions already defined there. Instead of having to wait for 3 seconds before the car is refuelled to max, the fuel levels now increase at a rate that would go from 0 to 100 in three seconds.

Functions

```
// Lines 1273 - 1291
void refuel(void)
```

Refuel is called every tick of the game loop. It will check if the player meets the criteria to start or continue refuelling (brake pressed and speed of 0) and then will increase the fuel amount by a rate that would make the fuel tank go from 1 to 100 in three seconds.

Testing

This test verifies that the fuel increases at a rate that would allow a fuel tank to go from 1 to 100 in three seconds. The fuel variable will be checked to make sure it is displayed correctly in the dashboard in real time as the car refuels and that partial top-ups would be allowed if the car stops refuelling halfway through.

Timestep	Fuel	Expected timestep	Test result
21.557	10	-	n/a
22.961	63	<24.557	Pass
24.234	100	24.557	Pass

ADC

This program uses the ADC and the two potentionmeters to affect the game. By modifying Pot0, the maximum speed is limited to a value that is proportional from the distance of the potentionmeter from its central position. When the potentionmeter is at the central position, the maximum speed will have a value of 5. The higher the value of the potentionmeter, the higher the speed limit - however, the global limit of 3 on offroads will still be enforced.
Pot1 controls the contrast levels. This is mostly a helper function to allow different users to change the screen contrast to one they feel comfortable with.

Globals

```
// Line 26
#define LCD_MAX_CONTRAST    0x7F
```

Controls the maximum contrast that the potentionmeter can set.

Functions

```
// Lines 1504 - 1507
void adc_init(void)
```

Enables the ADC and sets the prescaler to the maximum value possible (128) as there's no need for a high frequency.

```
// Lines 1509 - 1524
uint16_t adc_read(uint8_t channel)
```

Set the voltage reference selection to be AV_{CC} with external capacitor on AREF pin. Select the analog input channel based on the *channel* variable and also set the differential gain to 1. Will then begin conversion then return the digital value for the channel (channel 1 = Pot1 and 0 = Pot0).

```
// Lines 382 - 411
void update(void)
```

```
// Lines 872 - 926
void player_speed_input(void)
```

Testing

This testing plan ensures that the maximum speed set is proportional to the distance of the potentiometer from its central position. It will also verify that the offroad speed limit is still enforced even if a maximum value is set on Pot0.

Pot0	Speed Limit	Offroad
0	1	0
130	2	0
500	5	0
800	8	0
1023	10	0
0	1	1
500	2	1
1023	3	1

De-bounce all switches

All controls (except for the potentionmeters) go through a debouncing process in order to ensure that commands sent to the Teensy are as accurate as possible.

Globals

```
// Lines 154 - 162
enum Controls {
  MOVE_LEFT = 0,
  MOVE_RIGHT = 1,
  ACCEL = 2,
  DECEL = 3,
  PAUSE = 4,
  SAVE_GAME = 5,
  LOAD_GAME = 6
};
```

The different control states that are needed to be kept track of.

```
// Lines 85 - 86
uint8_t controls_states[7];
uint8_t prev_controls_states[7];
```

Each contains the current state of a control input. Ie. *controls_states[MOVE_LEFT]* will return 1 or 0 depending on whether the move left command (in this case joystick left - SW1) is pressed. The previous states are also save so that functionality of only accepting input on the rising or falling edge.

```
// Lines 89
uint8_t controls_history[7] = {0};
```

Holds the readings history of each input associated with each control.

Functions

```
// Lines 1531 - 1607
ISR(TIMER0_OVF_vect)
```

Timer0 is used for the debouncing of the inputs. In this Interrupt Service Routine, the current values of each pin associated with each control will be read and added to their respective histories. If more than 6 straight readings are found, the current state of that control is matched to the reading.

Direct screen update

This program has the ability to draw sprites directly to the LCD without the use of the buffer. Currently, the player's car is drawn this way, bypassing the buffer entirely.
The sprites must be drawn after *show_screen()* is called. This ensures that the buffer does not delete the sprite.

Functions

```
// Lines 458 - 489
void sprite_draw_direct(Sprite sprite)
```

Draws the sprite directly to the LCD, bypassing the buffer. It calculates what bank and range of x-coordinates in the LCD that we want to draw to and will proceed to set the LCD pointer to the first coordinate that the sprite appears (making this function pretty fast) . Then it will trawl accross the byte that spans the bank at each x-coordinate and set the locations that contain a 1 in the bitmap to 1 in the byte. Afterwards, the byte is sent to the LCD through LCD_DATA.

Timers and volatile data

This program currently uses two timers - Timer0 and Timer1. Timer 0 is used primary for debouncing of switches. However, it is also used as the clock of the program so that the speed of the main loop can be controlled and to be able to get the time elapsed since a game began. Timer1 is used to control the scroll rate of the game so that every obstacle's step functions are called at the same time. This ensures they remain in sync.

Globals

```
// Line 29
#define TIMER1_FREQ          7812
```

How many times per second the ISR associated with Timer1 is called.

```
// Line 103 and 166
uint16_t game_timer_counter;
uint16_t loop_counter;
```

Incremented every time Timer0 overflows. They are used to count the time since the program started (loop_counter) or since a game started (game_timer_counter).

Functions

```
// Lines 1531 - 1607
ISR(TIMER0_OVF_vect)
```

Called every time Timer0 overflows. It is used to increment the time counters as well as to debounce the switches.

```
// Lines 1609 - 1614
ISR(TIMER1_COMPA_vect)
```

Called 60 times per second (TIMER1_FREQ/60). Used to incremet *speed_counter* (see *Acceleration and Speed* in order to control sroll speed of the game.

```
// Lines 347 - 350
double elapsed_time(uint16_t timer_counter)
```

Calculates the elapsed time in seconds since the time counter passed to this function started running.

```
// Lines 1470 - 1502
void teensy_setup(void))
```

Sets up Timer0 and Timer1.

Register Values

```
// Lines 1485 - 1487
TCCR0A = 0x00;
  TCCR0B = 1<<CS02;
  TIMSK0 = 1<<TOIE0;
```

Set prescaler to 256 and allows interrupts on overflow.

```
// Lines 1490 - 1492
TCCR1B = 0x0C;
TIMSK1 = 0x02;
OCR1A = TIMER1_FREQ/60;
```

Enable CTC mode with a prescaler of 1024, allows interrups when the timer value matches OCR1A. This ensures ISR(TIMER1_COMPA_vect) is called about 60 times per second.

Program Memory

In order to conserve RAM, the sprite arrays are stored in program memory with the use of the PROGMEM keyword.

Globals

```
  // Lines 169 - 177
  uint8_t car_image[] PROGMEM= {
    0b01100000,
    0b11110000,
    0b01100000,
    0b01100000,
    0b11110000,
};
```

An exmaple of how a sprite array is stored in program memory.

Functions

```
  // Lines 365 - 377
  uint8_t* get_image_from_pgm(Sprite sprite)
```

Since the sprite array is now stored in the device's memory instead of RAM, this functions utilizes a PGM micro to read the pointer to the device's memory. An example of this macro can be seen below.

```
  // Line 370
  img[i] = pgm_read_byte(&(sprite.bitmap[i]));
```

Bidirectional USB communication

A server program has been developed that allows it to act as a companion to the Teensy - for instructions on how to run the server, see *Testing Procedures*.

Globals - Teensy

```
  // Lines
  enum USBCommand {
    SAVE = 1,
    LOAD = 2,
    DEBUG = 3
};
```

The different commands that can be sent alongside a message to the server so it knows how to process that data.

Functions - Teensy

```
  // Lines 1458 - 1465
  void usb_send_message(enum USBCommand command, int line_num ,char * buffer, int
    buffer_size, const char * format, ...)
```

A modified version of *draw_formatted()*, it will send a string to the server for processing. The command and line numbers are sent beforehand by using *usb_serial_putchar()*.

```
  // Lines 1470 - 1502
  void teensy_setup(void)
```

Calls *usb_init()* and waits until communications are established.

```
  // Lines 1421 - 1446
  void game_state_save(void)
```

It will send to the server all game defining variables in order for them to be saved. Each variable will be separated by a newline character.

```
  // Lines 1451 - 1453
  void game_state_load(void)
```

Functions - Server

```
  // Lines 37 - 56 (server.c)
  void process(void)
```

Checks the usb_serial if any new communication has been received from the Teensy. It'll use *fgetc* twice in order to get a code and the number of lines sent. Depending on the code, it will call the save() (1), load() (2) or debug() (3) function.

```
  // Lines 62 - 93 (server.c)
  void save(void)
```

It will decode the data sent so that the variables can be stored in the save file.

```
  // Lines 58 - 60 (server.c)
  void decode(char* data, int index)
```

Will read the lines of data sent and depending on the index, assign one of the game variable the value decoded.

```
  // Lines 95 - 105 (server.c)
  void debug(void)
```

Will display the values of each line in the message to the terminal.

Conclusion

The game *Zombie Race* was implemented in accordance with Part A and B of the brief. The majority of the part C features were implemented but I ran out of time to complete pixel collisions and the loading part of the USB communications.
Improvements consist of finishing Part C as well as more varieties of obstacles. As the LCD allows drawing of pixels, maybe some rudimentary animations (such as Formula sparks) could be implemented.