CAB202 - Microprocessors and Digital Systems

Assignment 2

Pedro Alves (n9424342)

June 2, 2018

Executive Summary

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
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
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
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
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);
```

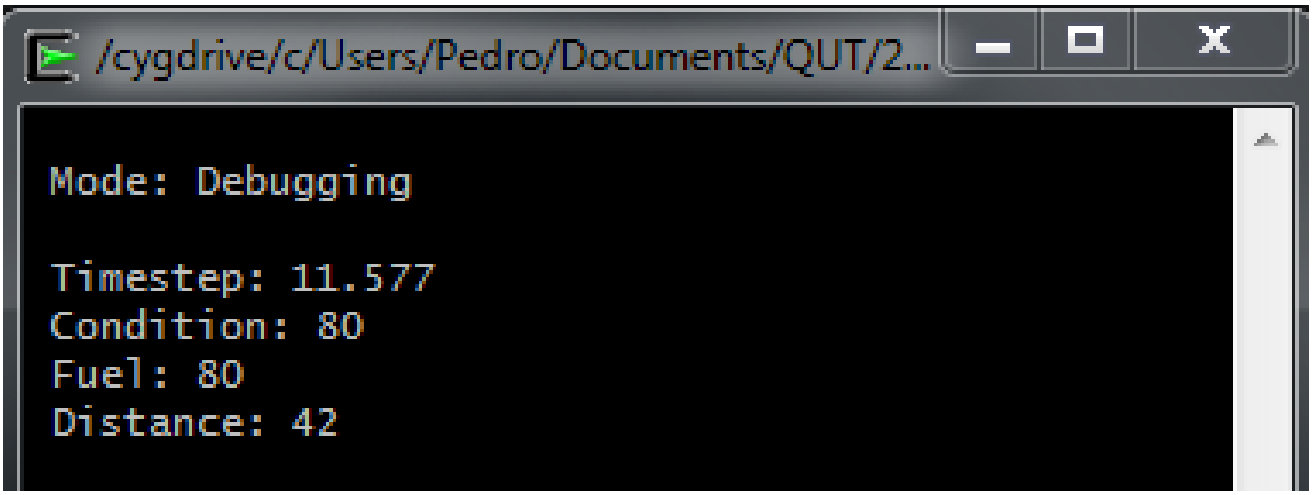

Figure 1: Screenshot of the server program when a debug command is sent with the current variable states

To improve readability of the test plan in the later sections, the information received from the server will be scribed into a table format. Under each teest case, the lines that contain the pieces of code that need to be uncommented to replicate the test are included.

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
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
uint8_t button_left_state;
// Line
uint8_t button_right_state;
```

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
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
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 110 - 112
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
bool refuelling;
```

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

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

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

Functions

```
// Lines
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
uint8_t distance;
// Line
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
uint8_t game_paused;
```

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

```
// Line
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.

```
// Lines
uint8_t stick_centre_state;
uint8_t prev_stick_centre_state;
```

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.

Functions

```
// Lines
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
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
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

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

Conclusion