Wee Dig Dug: Documentations

Anand Balakrishnan
anandbal@buffalo.edu

Amrit Pal Singh
asingh42@buffalo.edu

May 9, 2017

Contents

1 User Guide

1.1 Introduction

Welcome to **Wee Dig Dug**, a simplified, text-based version of the popular arcade game **Dig Dug** by Namco!. The following project was written in ARM Assembly for the LPC2138 Education Board, with the ARM7TDMI architecture.

1.2 Instructions

You are playing as Mr. Wee Dug, a glorious knight and miner (yes, it is an unconventional combination). You have been recruited by some villagers to kill a few beasts and you get paid depending on what kind of beast you kill.

- Kill a **FYGAR**, a species of vicious dragons, known to take a 100 knights to defeat a single one, and you get 100 points.
- Kill a **POOKA**, often mistaken for a cute, cuddly creature until it tries to eat you up, and you get 50 points.

To top it off, you will be locked up in an abandoned mine (abandoned because of these monsters obviously), and have 120s to kill all monsters, else you die (and don't ask me how you win, this is an arcade game). Best part is, you can mine the sand for 10 points per block, which is one of the perks of being a miner, right?

So, your objective is to earn as many points as you can within the 120 seconds. GOOD LUCK!

1.3 Setup

Before playing the game, please make sure of the following:

1. You are connected to the correct COM port on PuTTy, and at a baud rate of **115200 baud**.
2. Resize the console window to a minimum of 30 rows × 130 columns.
3. Be prepared to enjoy the game.

To start playing, just flash the code onto the LPC2138 board.

1.4 Controls

Keystroke	Action
W	Move UP
S	Move DOWN
A	Move LEFT
D	Move RIGHT
SPACEBAR	SHOOT a bullet
Momentary Push Button	PAUSE Game

1.5 Legend

Symbol	Character	Points
X	FYGARs	100
O	POOKAs	50
∨, ∧, <,>	DUG (YOU)	N/A
#	SAND	10
Z	WALL	N/A
EMPTY BLOCK	AIR	N/A

2 Design Overview

The game is designed based on the **Model-View-Controller (MVC)** architecture. This architecture is a common way of designing applications with a User Interface. In it, each of the following components is responsible for a particular task, and only that component is allowed to perform that task.

Model	This is the part of the framework responsible for maintainin the internal representation of the application. The **Model** in this game holds the location where sand is present, state of each sprite and state variables of the game.
View	The **View** is responsible for rendering the **Model** onto the GUI. It contains routines that display the board, the sprites and the sand on a console screen.
Controller	The **Controller** is responsible for handling user input and triggering changes in the **Model**. It contains the entry point for the game and interrupt handlers.

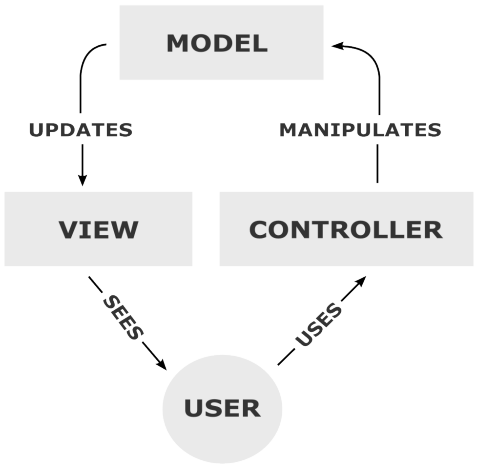

Figure 1: The components of the framework interacting with each other (courtsey: Wikipdia)

3 Controller Design

```
      FILES:  controller.s
WRITTEN BY:  Anand Balakrishnan (anandbal)
```

The **Controller** mainly contains interrupt handlers, and it is also the entry point for the game. In is, we do the following:

- Initialize timer and timer match registers for periodic interrupts.
- Listen for UART0 interrupt, read the keystrokes and perform the corresponding action.
- Listen for External Interrupt Button press and PAUSE the game.

The **Controller** is a relatively small component, responsible mainly for updating the **Model** via subroutines exposed by the **Model**.

```
1 SPRITE
2   DCD X_POS ; Holds X coordinate of the sprite
3   DCD Y_POS ; Holds Y coordinate of the sprite
4   DCD LIVES ; Holds Number of lives the sprite has
5   DCD DIRECTION ; Code for direction the sprite is moving/facing
6   DCD OLD_X_POS ; Previous X coordinate of sprite
7   DCD OLD_Y_POS ; Previous Y coordinate of sprite
8   DCD ORIGINAL_X  ; Original X position (to reset when respawning)
9   DCD ORIGINAL_Y  ; Original Y position (to reset when respawning)
```

Listing 1: Structure for `SPRITE` data

4 Model

FILES: `model.s`, `collisions.s`
WRITTEN BY: `Anand Balakrishnan (anandbal)`

The **Model** maintains the internal representation of the board and triggers **View** updates. It exposed routines that allows the **Controller** to trigger updates on the **Model**.

4.1 Implementation

The **Model** consists of an *"array"* (created using the `FILL` directive) of size 19×15 bytes, each byte representing a grain of sand.

The **Model** also consists of `DCD` tables to hold information of sprites. These tables are structured similar to a `struct`, see **Listing 1**. There are also staticaly defined regions of memory that keep track of the various states the game could possibly be in, for example, `PAUSE`, `GAME_OVER`. The **Model** is also responsible for keeping track of other variables of the game, such as, `LEVEL`, `HIGH_SCORE`, `CURRENT_SCORE` and `TIME`.

4.2 Operations

Operations that are defined by the **Model** are:

- Initialize and reset model.
- Move sprites and update entire model.
- Handle and detect collisions.
- Get if sand exists at given (x,y) coordinate on the board.
- Clear sand at given coordinate (x,y).
- Toggle game states (`BEGIN_GAME`,`PAUSE`, `GAME_OVER`, `RUNNING`).
- Update individual sprites.
- Spawn sprites.

4.2.1 Initialize and Reset Model

4.2.2 Game States and Representation

4.2.3 Update Model and Control Sprites

4.2.4 Collision Detection

5 View

The **View** is responsible for rendering **Model** onto the GUI. It possessed routines that the **Model** uses to trigger updates to the GUI. This implementation was chosen as updates can be triggered as and when the **Model** is updates.

5.1 GUI (*PuTTY* output) Operation

```
                FILE:  gui.s
PRIMARY CONTRIBUTOR:  Anand Balakrishnan (anandbal)
```

The **View** is holds many strings with ANSI escape sequences. Some of these stored strings are used to do the following:

- Change location of cursor.
- Display game stats (time left, current refresh interval, level, high score, current score, etc).
- Print empty board (just walls).

The **View** also holds routines whose primary function is to use these strings and manipulate GUI. It exposes the following subroutines so as to allow the **Model** to trigger updates to GUI.

- `draw_empty_board`
- `populate_board`
- `update_board`
- `clear_sprite`

5.1.1 Draw Empty Board

5.1.2 Populate GUI with Sand and Sprites

5.1.3 Update GUI

5.1.4 Clear a Sprite from GUI