Computer Organization and Assembly Languages

Final Project –Tower Defense on Game Boy Advance

THANK TO

J VIJN AND TONC

FAVONIA

DAVID SCOOT PAUL PREECE NINJA KIWI

B95902034 陳筱雯
B95902049 陳耀男
B95902106 溫在宇

BACKGROUND

『Tower Defense』 is the game which player should defend something. There are many player in the different kind of 『tower defense』. With the development of the technology, there are more and more 『Tower defense』 Flash-based browser game of the tower defense and that can play through the internet by browser and let the game know well.

Take recent example:

『Desktop Tower Defense』 is a Flash-based browser game of the tower defense genre created by first-time game designer Paul Preece in March of 2007. In the span of a few months, the game had been played over 15.7 million times as of July 2007.The game was among one of Webware 100's top ten entertainment web applications of 2007.

The Classic of 『Tower Defense』:

1. Enemy appears and goes to some goal on the map
2. Attack the enemy by different method.

After considering, we try to implement our 『tower defense』 game in the GBA.

IDEAL

Just like other『Tower defense』game, our 『Tower defense』 game's basic prime is simple. The game is played on the map which contains waste, lake, river, and mountain at the center of the map. The player should build some tower to kill the monster. The tower can upgrade by spending some money and contain different technology tree, the player can choose the part that he think it more well. And kill the monster before they are able to reach their objective.

The player has some life points. When the monster reaches some goal it will decrease the remaining life point. When the life point is equal 0, the game is over.

HOW TO PLAY

Direction key	Move Frame on the map
A	Construct the tower which been chosen.
B	Call next monsters wave. (Need on no tower's tiles)
L	Not need in this Game
R	Look next menu
START	Select option on menu
SELECT	Pause when play

Table1. Key

When staying on a tower, you can press R to choose whether to upgrade (speed/range) or not, or even choose to sell it.

You can only construct a tower on the map without road and tower.
Every monster have different grades, a bullet will decrease one grade of it.
Monster which has higher grade will move faster than those who has lower grade.

TOWER TYPE

Common Tower: 
Attack feature: attack single monster
Attack Speed: Common
Attack Range: Common

Range Tower: 
Attack feature: attack monster in the range of the bullet
Attack Speed: Slow
Attack Range: Small

Slow Tower: 
Attack feature: no damage, but the speed of the monster which is attacked will decrease
Attack Speed: Common
Attack Range: Common

IMPLEMENTATION OVERVIEW

After deciding our goal that is to build a tower defense game, we request that we will not implement the program by HAM function or TONC function during the period of our implementation

We separate this game's implementation into two parts. One part is implemented by C++, another part is implemented by assembly. For the C++ part, we use a lot of classes like Game, Menu, Tower, Tower_type, Monster and Bullet to implement the game.
As we known,『Tower defense』is composed by towers, bullets, and monsters. To handle these three components, we have "main" to handle the whole game, and "game" to handle the pace of the game.
All of these operations are supported by function call which is implemented by assembly.

As soon as we open the GBA file, "main" will initialize the setting and be ready to be played.

IMPLEMENTATION - BACKGROUND DISPLAY CONTROL

GBA has 5 different display mode including bitmap mode and tile map mode. We choose to use mode 0 which has 4 tiles backgrounds using 256 colors palette. Each pixel is represented by 8 bits. So a 8x8 tile will use 64 bits. The total tile that can be place in to the VRAM memory is 1024.

Each background can have different priority and can be switch on or off separately. This feature let us easily switch from play screen to menu screen. And can display some text message over the play screen.

The following table listed those tables we used and what is display on it.

Background	**Usage**
0	Display the road which the monster can walk on it.
1	The true background.
2	Display the state panel include tower state, memory, life left etc.
3	Anything that cannot be placed in the OAM memory. Such as menu the blue cross that indicate the range of tower and etc.

Table2. Background

IMPLEMENTATION - OBJECT DISPLAY CONTROL

We have three type of object: tower, monster and bullet. Because there are so many different objects we have implement an OAM memory management system call that have two functions. One for allocate memory and one will free the memory that will not be used anymore. By this way we can place more object on the screen at one time. And can prevent potential memory leak.

The reason that we do not use OAM to display the state panel is that we already have too many objects need to display.

IMPLEMENTATION - OBJECT DISPLAY CONTROL API

To let our C++ program access OAM more easily. We have implemented a set of function in assembly that supports almost every feature the GBA provide. The complete list of function is listed below:

```
void init_sprite(int number, int size, int tile_no)
void set_sprite_pic(int number, int size, int tile_no)
void set_sprite_priority(int number, int priority)
void set_sprite_pos(int number, int x, int y)
void set_sprite_affine(int number, int enable, int affine_matrix_no)
void set_sprite_alpha(int number, int enable)
void set_affine_matrix(int number, int pa, int pb, int pc, int pd);
void enable_sprite(int number, bool enable)

int load_sprite_img(void *ptr, int half_words)

int allocate(int size, int tile_no);
void deallocate(int number);
```

To use this API we first use load_sprite_img() to load picture into VRAM. This function will return a number that point to the memory address that place in VRAM. Loading the picture will not display it on the screen. To display it you need to call allocate which will call init_sprite and set an free OAM slot to this picture. Function allocate() also return a number indicate the OAM slot that will be needed by set_sprite_* functions.

Picture Use In Game

Since small memory and few colors can be used, we have a difficult time finding the background picture. It is also difficult to find suitable pictures for other objects ex: tower, monster.
So we use A~Y to represent the monsters, and finally find some simple pictures for the towers (if it is not simple the tower is not good looking due to DPI problem), and draw a very simple picture for the bullets.

Background image use in game

Map And Background Generator

For extension in the further, the road which monster walks on should be changed easily. We obtain this goal by creating a map file define below:

Background describe file *.bg File Format
First line contain two integer X Y indicate the map size.
For the next Y line there will be X character describe the map.
Each character can be '0', '1', 'S' or 'E', which indicate wall, road, start point or end point, respectively.

An example is given at Appendix A.
Program bg2s:
This program will translate *.bg to a tile map that can be read by the GBA. The tile map will be place at background 0.

Program mm:
This program can translate *.bg to *.h which contain the array which show where the monster should go at the moment T.

```
#ifndef TEST1
#define TEST1
static int START_X = 0, START_Y = 2;
static int walk_x[83] =
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1
,1,1,1,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1};
static int walk_y[83] =
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,-1,-1,-1,
-1,-1,-1,-1,0,0,0,0,0,0,0,0,0,0,0,0,0};
#endif
```

walk_x[t] and walk_y[t] array means what direct the monster should walk at moment 8*T.

BIOS CALL PROBLEM

At first we want to use atan2() to calculate the angle of the tower. But we find that the GBA emulator we use do not have a BIOS file. And we cannot find a legal copy on the web. So we decide that we should use a picture which need not be rotate instead.

SOUND HANDLE

After investigating Sound part in GBA, it has four channels and two direct sound channels.

Sound Channel 1 produces square wave variable duty cycle, frequency sweep and envelope function.
Sound Channel 2 is identical to channel 1 but without the frequency sweep function.
Sound Channel 3 acts as a 4-bit Digital-To-Analog Converts that repeatedly plays a pattern of samples. And we can sore sixty-four 4-bits sample and separate into two banks.

Sound Channel 4 produces Pseudo-Noise with an envelope function.
And Direct Sound A and Direct Sound B are the two 8-bit digital-to-analog converts part of the Game boy Advance Sound System. The samples are stored in consecutive addresses. These addresses act as a First-In-First-Out.

Every Sound Channel is needed to set 2 to 4 registers.

ENCOUNTER PROBLEM

Display Handle:
I. Because We use mode 0 which is a tile mode. If the background image is too complicated it may not have enough memory space to save it. To handle this problem we have to change the background image we first draw manually. By copy the same tile many times we finally fit the image into memory.
II. At first we want to use affine sprite which can be rotate by a rotation matrix automatically. But we find that the object picture we use is too small and after rotation it will be out of shape.
III. The

After trying to set registers of every Sound Channel, because our interrupt process by assembly is not work well, we cannot play sound by direct sound in interrupt mode.

And Sound Channel 4 produces Pseudo-Noise exactly produce noise. So, we also cannot use Sound Channel 4.

Therefore, we fail in playing sound by a lot of pattern that is stored by us. Finally, we play sound by composing of many single notes. And playing it at the begin of the game and the end of the game.

We search the famous music of the Mario through internet. And transform music to notes and implement function to handle music.

The definition of the function:

```
void ini();
void set_volume(int right,int left);
void ini_ch1();
void play_ch1(int freq);
void len_ch1(int len);
```

Above functions are all implemented by assembly.

`ini():`

Initial all sound channels to let them be ready for playing.

`set_volume(int right, int left):`

Set the volume of all sound channel

Right is for right volume

Left is for left volume

`ini_ch1():`

Initial sound channel 1 to let it be ready for playing

`len_ch1(int len):  //len means length  0 <= len <= 0x003F`

Set the length of the sound channel 1' sound

The sound length is a 6 bit value obtained from the following formula:

Sound length = (64-register value)*(1/256) seconds.

`play_ch1(int freq): //freq means frequency  0 <= freq <= 0x07FF`

Play sound channel 1 with frequency

The exactly frequency which is played:

(2^17)/(2048 - freq)

Set for easy to use:

```
#define RATE(note, oct) ( 2048-(notes[note]>>(4+(oct)))

//The above definition let we can use note and octave to generate the
value which register should be set.

enum{
    CC = 0, CIS, DD, DIS, EE, FF, FIS, GG, GIS, AA, BES, BB
};

//Above let we can use the note more intuitively

const int notes[12] = {
    8013, 7566, 7144, 6742, // C , C#, D , D#
```

```
    6362, 6005, 5666, 5346, // E , F , F#, G
    5048, 4766, 4499, 4246  // G#, A , A#, B
};

//Above let the note can match its value for the define RATE
```

Appendix A Binary Map

```
File: test.bg
-------------
32  20
00000000000000000000000000000000
00000000000000000000000000000000
S11111111111111111000000000000
00000000000000000010000000000000
00000000000000000010000000000000
00111111111111111110000000000000
00100000000000000000000000000000
00100000000000000000000000000000
00100000000000000000000000000000
00100000000000000000000000000000
00100000000000001111111111111E00
00100000000000001000000000000000
00100000000000001000000000000000
00100000000000001000000000000000
00100000000000001000000000000000
00100000000000001000000000000000
00100000000000001000000000000000
00111111111111111000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```

APPENDIX B SCREEN SHOOT