Mudd Adventure

A 3D Raycasting Game

CSEE 4840 Embedded Systems

Final Report

5/15/2014

Mingrui Xu(mx2151)

Wei Cao (wc2467)

Bowen Dang (bd2384)

Shijie Hu (sh3251)

Table of Contents

Introduction

In our project, we designed and implemented a virtual 3D first-perspective shooting adventure game by using Raycasting technique. The player will be placed into to 3D loop maze and fight with the monsters when moving forward to the finish line. The player can move back and forth, left or right in the 3D world. But the only way to win the game is to beat the monsters and arrive at the finish line alive.

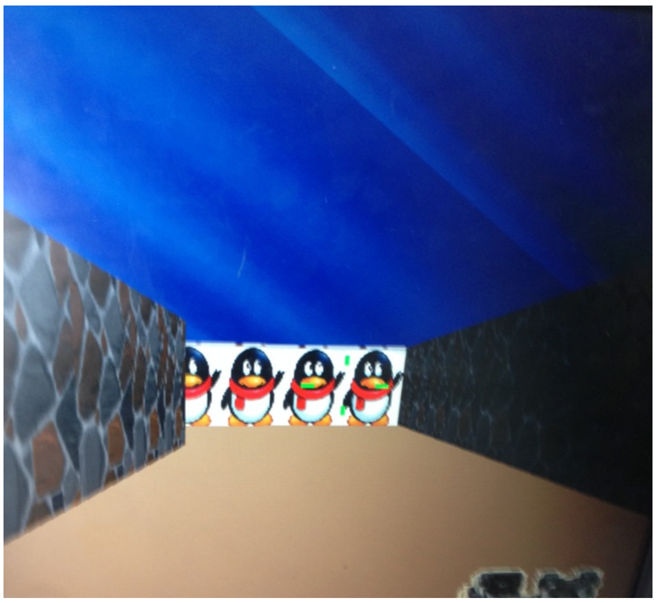
A gamepad will be the game controller as an input. The 3D world will be displayed on the VGA output. Also some sound effects could be sent out from the output speaker during the game.

The project is divided into software and hardware components and the communication between the software side and the hardware side is the key part to realize the game. In the software, we modified the Raycasting and DDA algorithm to keep track and update player's position in the map. The Raycasting algorithm will calculate the wall heights according to the player's position. Software will receive the gamepad inputs from the player and generate the commands to make the update in the game. We also finished wall texture modification and resize in the software.

After the software side finishing the algorithm and calculations, it will pass the data to the hardware. In hardware, we mainly interpret the data to the VGA display on the screen. A FSM including all the game elements is implemented as a very important part of the hardware. Also the sound effects come out from the hardware side.

Even though the game runs good but we think there are still some improvements can be optimized in the future to make it more interesting.

Enjoy the game.



Overview

Here is the overview structure of our project.

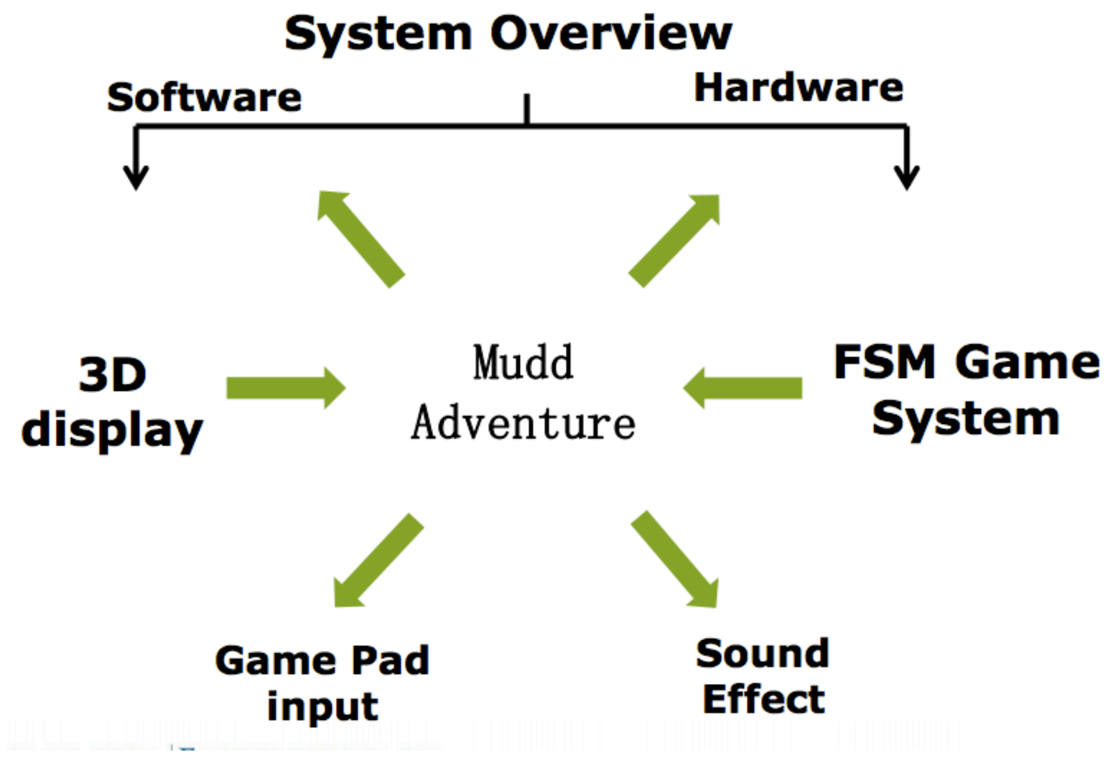
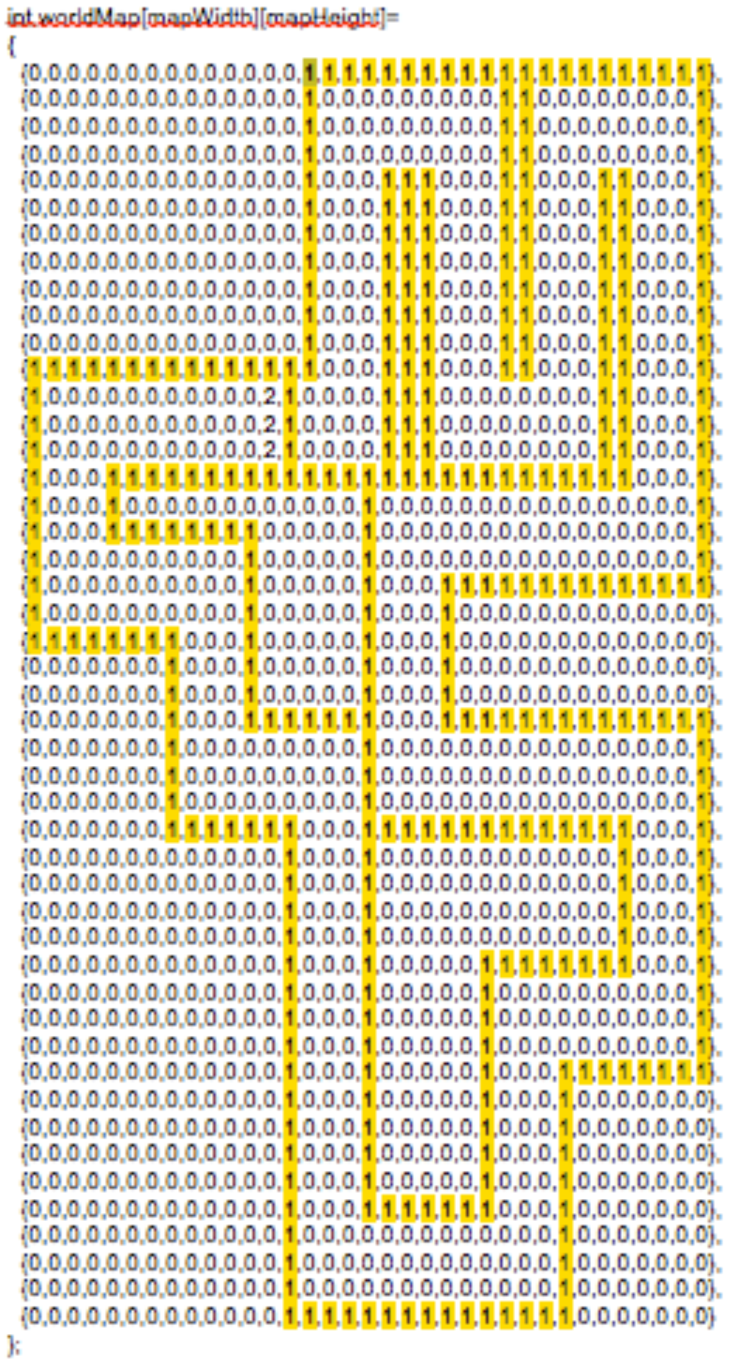


Figure 1. Overview

Software

Game Map

Firstly, we need to define the map of the game at the beginning of the software code.

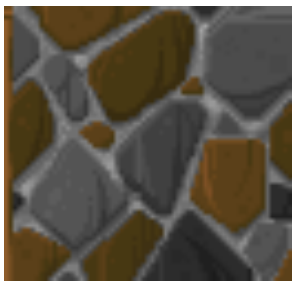


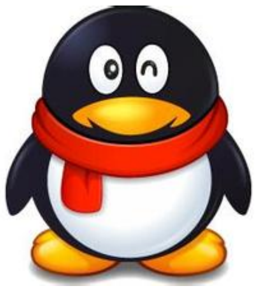
```
int worldMap[mapWidth][mapHeight]=
{
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1},
  {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,1,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,2,1,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,1,1,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,2,1,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,1,1,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,2,1,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,1,1,0,0,0,1},
  {1,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,1},
  {1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
  {1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0},
  {1,1,1,1,1,1,1,1,0,0,0,1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0},
  {0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0},
  {0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0},
  {0,0,0,0,0,0,0,1,0,0,0,1,1,1,1,1,1,1,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
  {0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {0,0,0,0,0,0,0,1,1,1,1,1,1,1,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,1,1,1,1,1,1,1,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,1,1,1,1,1,1,0,0,0,1,0,0,0,0,0,0,0},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0}
};
```

Figure 2. Map

You can see that this map is a loop and consists of “mudd”, this four letters. The player will be placed at the right side of the wall with the label ”2” at the beginning. Then the player will follow the route with the clockwise direction in the 2D map to the finish line with the wall label “2” to finish the adventure game.

Array entries that are 0 represent an empty space where the player can walk through. Array entries larger than 0 represent the wall with some specific texture on it. To make it simple, we just added three textures in the game.


Stone


Penguin

Win

Figure 3. Wall Textures

The texture “Stone” is put on the walls with label “1” which face left and right in the 2D map. The texture “Penguin” is put on the walls with label “1” which face up and down in the 2D map. The texture “Win” is used on the wall of the finish line with label “2”. The reason why we used the textures of a wall stone bricks and a cartoon figure is to test the display performance of our texture generation algorithm and calculation for different kinds of textures. The result is that the wall shows perfectly.

Raycasting Algorithm

Our project is basically a simple 3D shooting adventure game and the Raycasting is the key part of our 3D display rendering technique. This technique is used to create a 3D perspective on the monitor by utilizing a 2D map. So for the software aspect, performing the Raycasting calculation algorithm is the key part. We followed LodeV's Ray Casting tutorial with C++ code as a good start point to understand the algorithm; then we made the modifications to make it generate the data we need in the software.

The basic idea of Raycasting technique is as follows: the map is a 2D square grid, and each square can either be 0 which means the place is vacate and the player is free to walk through, or a positive value which means the square is occupied by a wall.

For each x-coordinate value of the monitor screen, a ray will be sent out starting at the player location and with a direction that depends on both the player's looking direction, and the x-coordinate of the screen. Then, let this ray move forward on the 2D map, until it hits a map square that is a wall. When it hits a wall, the distance between the hit point and the player position will be calculated, and use this distance value to calculate how high this wall has to be drawn on the screen to make the 3D vision. The wall is higher when it is closer. If it is far, it should be relatively small. These are all 2D calculations.

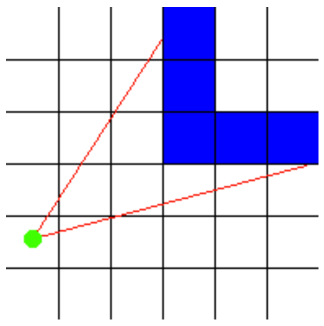


Figure 4. Raycasting laser measure

The image above shows the overview of the process. The green dot is the location of the player. Two red lines are rays. Here, both rays hit the walls so their lengths will be saved for the 3D rendering calculation.

In our game, the screen width is 480 which means that for each time, we need to calculate the wall height 480 times to get data for each column of the screen. Then we need to pass this data set with 480 binary number to the hardware part through Avalon bus. Each column data in the data set is assigned an address from 0 to 479. Then we can retrieve data of each column by calling its address at the hardware and then make the process for the game and VGA display.

Taking the player position as the input, the whole Raycasting software code performs a loop to update the data every time according to the clock rate. When the Raycasting loop is done each time, the time duration of the current and previous frame can be calculated as well as the FPS value. The ARM Cortex A9 processor we use here can deliver devices capable of over 1GHz clock frequency. So the performance should be great. So actually we can just set the frame time as a constant value to determine the speed of moving or rotating and the FPS calculation will be unnecessary.

Data Delivery

After calculations for each column, a 62-bit data number will be generated. Here is table showing the data types we deliver from the software to hardware.

Data Name	Bit Number	Bit Allocation
drawStart	10 bits	9:0
texX	10 bits	19:10
color	2 bits	21:20
scenario	4bits	25:22
control	3bits	28:26
gamereset	3bits	31:29
texY	14bits	45:32
coff	16bits	61:46

Figure 5. Data Delivery Table

Here are the functions of each data:

- **drawStart**: This is the data number calculated by the raycasting algorithm to determine the wall height of each column. Because the wall is symmetrical subject to the half-height horizontal line of the screen, we just need this to draw the whole wall rather than using an extra drawEnd signal.
- **texX**: This is the x index of the texture column on the wall. This data will be used in hardware as the address of texture ROM to assign texture pixels on the column.
- **texY**: This is the y index of the texture column. With texX and texY, we can determine how every pixel of the column looks like on the screen.

- **coff**: This is the result from the texture resize calculation. It will be used in the hardware to make the texture column larger or smaller.
- **color**: This determines the label or the direction of the wall. In the hardware, it will be used to determine which texture from the three above will put on the column.
- **scenario**: In the software, we generate some random numbers and use some condition sentences to determine what kind of sprites will show on the screen. In the hardware, when the scenario meets the requirement of a specific sprite's condition, that sprite will show on the screen. There are three sprites in the game and they will be introduced later.
- **control**: This signal will control the gun fire effect display and gun fire sound.
- **gamereset**: This is used to reset the game.

Gamepad Control

We use a gamepad to control the game. The gamepad protocol is similar to the keyboard from the lab2. The difference is the value for each key on the gamepad is different to the original keyboard. The gamepad controller receives data through PS2 serial interface. It was modified so that while receiving a data token from the gamepad, it stores data information to a register. Then the data go to the software code to performance each game operation.

In the game, the direction keys or direction moving stick are used to make the motion of moving forward, backward, left, right. When forward and backward keys are pressed, the player position in the map will be updated by 0.02 length unit each time in the 2D map. When the left and right keys are pressed, the view plane of the player will be updated. We programmed four command keys with label "1", "2", "3", "4" on the gamepad. When key "1" is pressed, player can pick up the bullets or health pack if they show up on the screen. When key ”2” is pressed, player can reload the bullets. When key "3" is pressed, player can heal himself if he has extra health pack. Key "4" is used to fire the gun.

If the player is dead or wants to play from the beginning at any time, a reset command can be realized by pressing "mode" then pressing key "1".

When the software gets the motion command from the gamepad, the new distance calculations will be activated in the code after locating the new player location on the map. And the new dataset will be send to the corresponding SRAM part. Then new 3D will be showed out via VGA to the screen.

Software to Hardware

Texture Generation

We also add textures to the walls of the maze by generating textures as the same as LodeV's code. Actually we can not only get just the start and end coordinates of the vertical line in the algorithm, which is the integer number tab of the wall, but also more information that is the precisely decimal map position handling by the increment method of the ray-casting algorithm. That means we can obtain the exactly x-coordinate of the pixel of the wall between 0 and 127. Meanwhile, we should also keep in mind that whether we hit an x or a y wall. In the ray-casting algorithm this is given by the output of "color" (As is shown in the appendix of mudd.c). The total textures we use in the map are three, as are shown above. They are used as X-wall, Y-wall, and final-win logo, respectively. Our pictures are 128*128 pixels with each pixel of 24 bits for RGB color display. What we mainly focused on in the texture part is how to lay it out in the hardware. The x-coordinate that has been discussed before is calculated by the ray-casting algorithm as a function of current position in the map. The y-coordinate is represented by two points we call drawstart and drawend. Roughly speaking, we draw the map line-by-line in the VGA raster. In order to show the 3D effect, we use interpolation between start and end to scale the size of the picture. The sky and floor are drawn from the top of the screen to the drawstart, and from drawend to the bottom of the screen, respectively.

Figure 6 shows how to lay it out in hardware. We use two 32-bit data in software to transform y-coordinate and x-coordinate with scaled coefficient to hardware. In hardware, it is receipted by two RAMs and combined to store in one 64-bit logic variable. Then we use function (1) to calculate the pixel address in the texture ROM. Notice that in the actual program, the texX*128 is calculated in the software, as well as the original coefficient, as is shown in equation (2). Doing these operations in software improve time performance of the whole system very significantly. We will show it later in the time analysis section. The coefficient is scaled by multiplying 16384 is to avoid the appearance of decimal number for the resample algorithm and keep high resolution of the interpolation process.

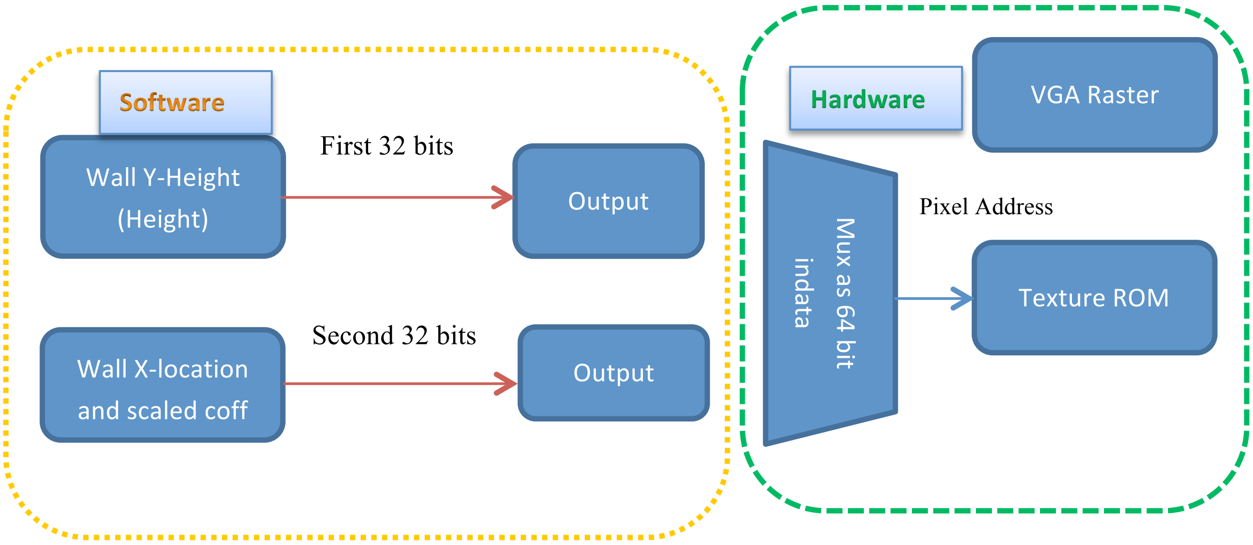


Figure 6

addr_wall = (texX*128+((hcount-drawstart*2)*coefficient)/16384) (1)

coefficient = (128/(drawend-drawstart))*16384 (2)

In sum, the texture generation module maps a 128*128 pixel texture to a wall from the current location of the player. It receives the index of the x-y coordinate of the wall as an input from the software to the hardware. Its output is the address of the color data, and the actual data will be fetched from the texture ROM. We use three ROMs to pre-load the three textures with 24 bits of color data for each pixel. Since we use two different RAMs to receive x-coordinate and y-coordinate respectively, it keeps two combinational logic running in parallel and we lower the clock frequency by half and combine them together before calculating address. This is also a way to improve the image performance by eliminating noise. The interactive RGA_raster and texture part is shown as follows in Figure 5.

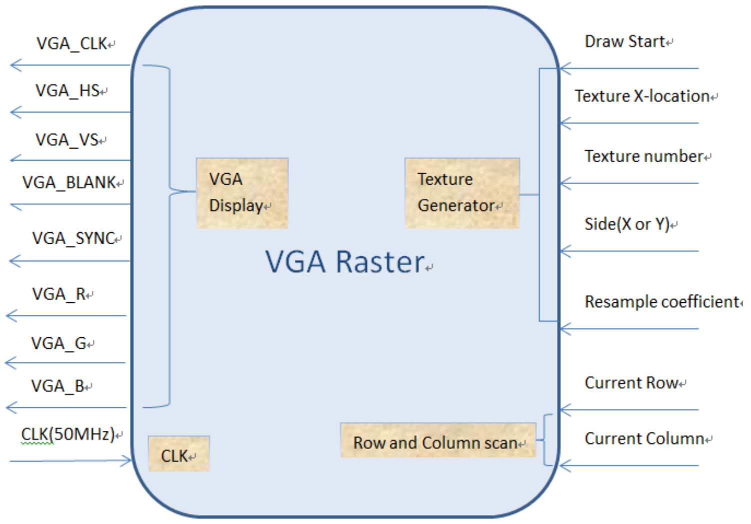


Figure 7. VGA Raster

Hardware

Hardware Architecture

Hardware architecture is showed in above figure.

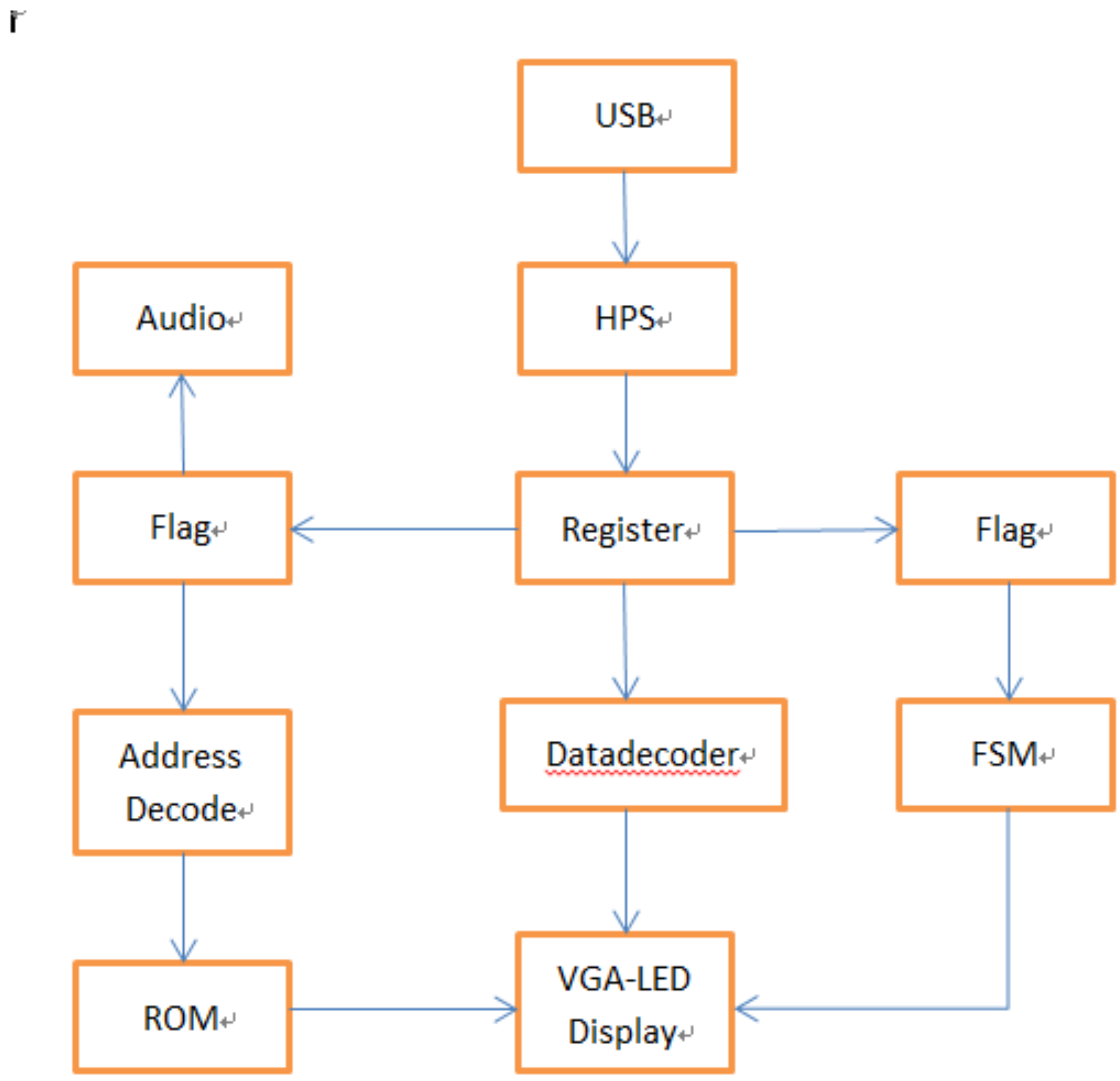


Figure 8. Hardware Structure

Joystick communicate with HPS according to USB protocol, HPS reads joystick's key value and encode them into control flag. Together with 3D display data generate by ray casting algorithm, these flags are written to avalon bus.

For FPGA part, the 32 bits writedata signal reads data in and write them in a register RAM.

Register read 2 32-bits words in, stored them in 2 RAM and send 64-bits word out. Read address of the RAM is vertical line counter because every word given by software contains information of a line on the screen. Then these words are sent to module data_decoder.

Data_decoder read 64-bits words in and decode them into different flags: shoot, fire, control, inscreen, senario, color, game reset. Shoot flag is one of joystick's key value which represent behavior "shoot" in game. Control and game reset flag represent other keys of the joystick which represent reset, pick up, refill etc. These control flags are sent to a finite state machine .

Another kind of flag is to represent the position and condition of spirits on. inScreen is to decide whether the current point is on the wall; color is to tell which wall the current point hit at.These flags make pasting spirits more easier.

Data decoder also has the function of address decode. Horizontal counter and vertical counter of VGA changes every clock cycle. When we need to draw a picture on screen, data decoder visit a ROM which stored the picture and read the specific value of the address. This address is the function of clock and other flags such as inScreen and color. The color value stored in ROM is then send to VGA-LED screen and showed.

Some addresses are easy to calculate such as boss spirit and sky, while some spirits' addresses, such as wall texture, are hard to calculate. For these texture are reshaped so they require sampling. The address generate method for texture is special discussed in other parts. Following figures are the spirits we used in our project.

Figure 9. Gun, Game Sprites and Sky

Apart from paste pictures on screen, we also need digital displayed on screen. So we first implement a 7-segment digital display module which initialize a 7 segment digital of arbitrary size and position. Second we implement a digital decoder module to transfer numbers into 7 segments code. Then we write a module to transfer any binaries into decimals. With these three module, we can show any digital on any part of screen.

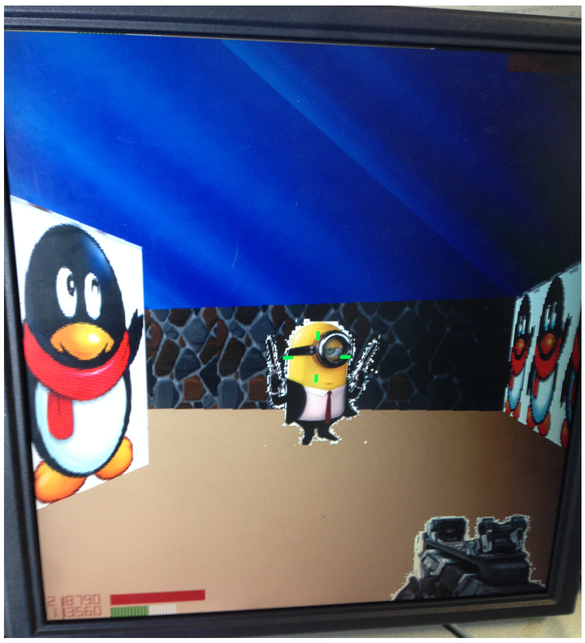


Figure 10. Game Environment

As showed above, the digitals are displayed at the left corner of the screen. The digitals show player's blood volume and bullets' amount. These numbers are generated form finite state machine and change accordingly.

Motion Effect

We add some motion effect to our game. When shoot the boss, there will be fire flashing at muzzle, and riflescope will converge. These effects are controlled by a flag direct coming from software and update every clock cycle.

Memory

Memory module is used a lot in design. All pictures are transferred to memory initial file and stored in ROM. ROM module has three ports, clock, read address and q. Datas are stored in memory in sequential. When given an address, ROM will send a corresponding data at the postage of clock.

RAM is another important module in memory. Apart from read port which is the same as ROMs, RAM also have write port. Given an address and a data, RAM will read the data and stored it to the corresponding space.

In our design, we use two port RAM as register and buffer. These two port RAM shorten critical length and simplify our synchronous design.

VGA_LED screen

VGA_LED screen is the most important peripheral. VGA works under 50MHz clock, and has initial resolution 1280* 480. Every cycle horizontal counter pluses 1 and at the end of each line and vertical counter plus 1. Every point assigned a color value and displayed on screen.

Timing Design

The reference clock frequency available for us is 50MHz. VGA-LED screen is driven by 50MHZ and its initial resolution is 1280*640. However, the initial design can only run at 7.5MHz and the spirits has many unexpected noise points even for a static spirits.

First we add a register to store the data. The circuit and restricted clock became 10MHz. Then we find division in address decoder is time-wasting. So we move this part to software and clock restriction moves to about 25MHz. Now the critical path is still the division path. However division is hard to exclude from design so we lower the resolution of the screen form 1280*480 to 640* 480. Under this resolution, the address changes every 2 clock cycle, which is enough for division and other calculation. After these change, the spirits are free from noise.

FSM

After designing the framework of the hardware, we further improve our game to make it a First-Personal Shooting Game(FPS). To avoiding timing problem, we use a FSM to build our game system.

Before we building our system, we first come up with an overall design. Actions of the player are listed in the following, each refer to state in the FSM.

1. Move

We can imagine that when the player is moving in the maze(which is already realized by the game pad), there are chances that he got treasures or meet with enemies. The chances of different depend on a random value, which is generated in the software. It then flow through the IO-bus into the hardware and trigger the event in the FSM.

2. Event

There are two possible events in our game design: encountering enemies, then fight with it; and finding treasure, pick it up. There are two kinds of treasure boxes in the game, caisson and first-aid-bag. They are valuable because the player is really easy to die. When each event is triggered, the game will soon enter another state to make the game process.

3. Fighting

When we encounter a monster, we have to fight with it or die. Because of the limit of memory , we just make it in the form of round system, to avoid any timing problem. The time period of each round is short and it can simulate a real time system. In the player's round we can make three possible actions: shoot, heal and reload. In our game, player is equipped with up to 3 clips and 3 first-aid-bags. Every time the player is out of bullet, or badly hurt, he could take a round to use these items. After the player finished his round, the boss hits the player and does an amount of damage. The player can't move forward until the monster's hp falls to zero and is defeated.

4. Healing & Reloading

The initial hp of the player is 90, which is shown on the left bottom of the screen. When the hp of the player has fallen below zero, he will definitely fail. Here we use a single button to control healing action. It is the same as reloading. The amount of ammunition will decrease very fast in the fight and time to use the caisson is important.

5. Lose & Win

As long as the player defeated all the bosses and reached the terminal point, or is killed by a monster, the game is over. This state needs only special efficacy, no operation of the player.

6. Reset

An important part of the game system is restart function. In our game design, we use a button to reset the game. When reset is triggered, everything in the game is initialized. The player goes back to the starting point with full hp and clip.

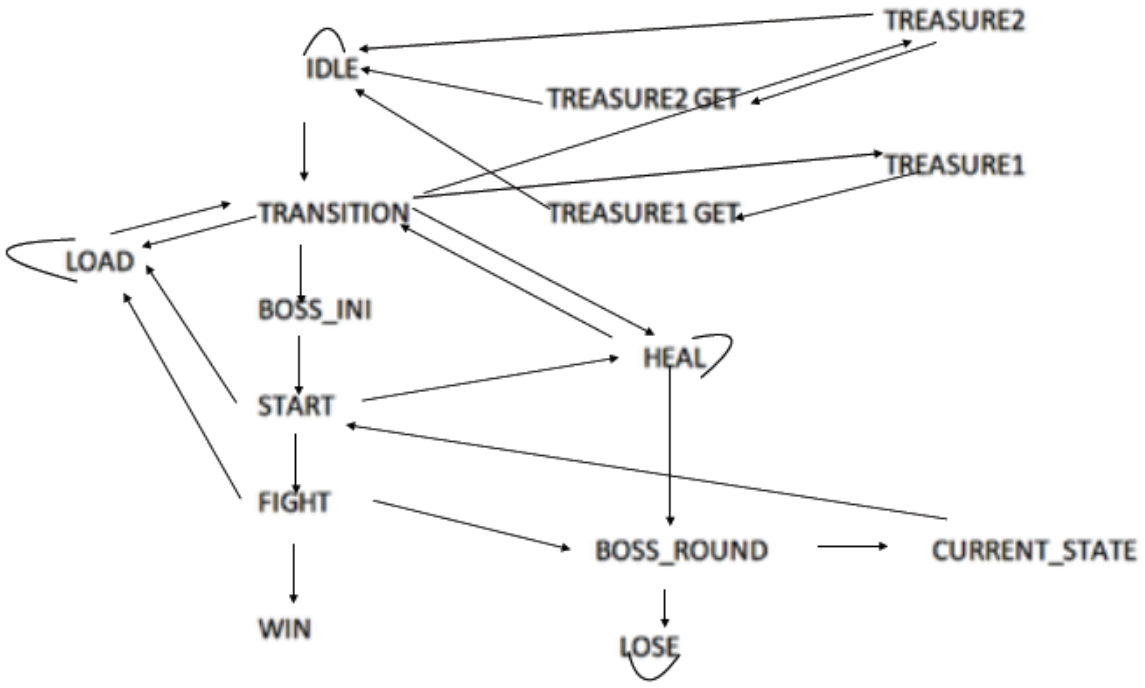


Figure 11. Game FSM 1st Verson

After determining all the basic parts of the game system, we now draw sketch of our FSM of our game system, as shown below:

In our FSM, we need 6 input signals, including control signals from the game pad, and the random variable deciding the event from the software. To further complete our game system, we consider some possible scenarios.

What happens when there is no enemy? In fact the player still should be able to shoot, or to heal himself. Furthermore, we want our weapon charged automatically.

After adding all the essential input signals, we finished our FSM design. As shown in the graph, we add some redundant states to our design. Our aim is to split one state with many in-state steps and calculations in two to meet the timing requirement.

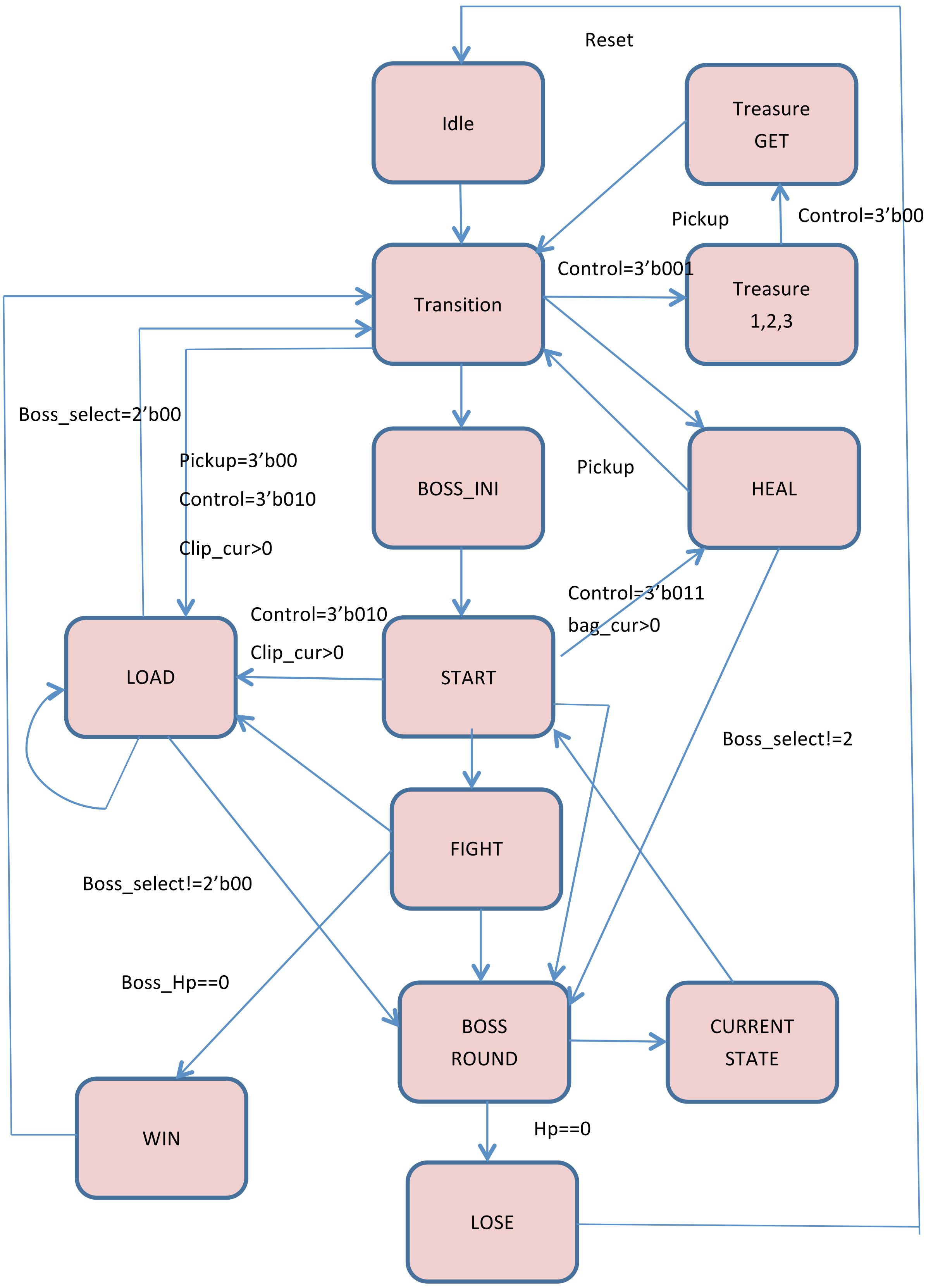


Figure 12. Game FSM 2nd Version

There are some problems we met during the design. The greatest problem we met is the synchronous of software input and state machine. At first our random variable changes so fast that it can't be applied to the state machine. To solve this problem, we use the moving forward keycode as the trigger event of the random variable, and it is also the input of the FSM. The two events (pushing button and encountering) are proved to work well with each other.

Another problem with our design is the synchronous of the state machine and the display. At first our state changes too fast, which affects its output which controls the display. Most of the time the control signal changes before a complete picture is shown on the screen. Our solution is to make the trigger evet from positive edge of clock to a combination of vcount and hcount. That ensures that a control signal won't change or be transmitted to the display module before the formal circle of scan.

Besides the system design, we made some improvements to make our whole game look better, like simulating a Muzzle sparks. The detailed information is referred in the hardware architecture part.

Audio

The protocol the audio codec uses for configuration is the Inter-Integrated Circuit (I2C) protocol. This is a two-wire protocol originally designed by Phillips Semiconductor in the 1980s to connect peripherals to the CPU in TV sets. Nowadays it's used to connect low-speed peripherals in all sorts of devices. The two wires in the I2C protocol are labeled SDAT and SCLK for Serial Data and Serial Clock respectively. In I2C, data is sent a bit at a time over the SDAT wire, with the separation between bits determined by clock cycles on the SCLK wire. SDAT and SCLK are also sometimes abbreviated as SDA or SCL.

SSM2603 register is the place where the sound comes out. Also the sound effects are stored in ROM as .mif files. When the fire button is pressed, the sound command is triggered and the fire sound will come out.

Lesson learned

- **Love what you are doing:** The embedded system project is really a tough task. We have to devote lots of time on design, coding, debugging and improving the structure. The board information is very limited so we have to study the coding and structure from scratch. Sometimes, we really felt suffering when some thing goes wrong. Then spending time on debugging is another task. So, we need take lots of efforts on the project and loving it is the key motivation to make it better.
- **Good structure is necessary:** The data and file structure of both software and hardware is the very important part. A bad structure can lead to failure and resource waste. So just have a good plan at the beginning so the structure will be maintained great.
- **Debugging technical:** Debugging is also important and common during the design and testing. In order to find the bugs quickly, we have to use some debugging technical to test the code more efficiently.
- **Timing issue:** We did not pay great attention to the timing issue or clock. The result was we got much noise and errors. So if you expect your project having better performance, please focus more on timing.
- **Teamwork:** To finish a relatively large project efficiently, a good teamwork and collaboration is quite necessary. We divided the tasks to each one fairly and clearly, so overall the progress of our project is smooth.

Responsibilities

Mingrui Xu

Adapting algorithm of software and mapping
Texture Generation and system interconnection between components
Group organization

Wei Cao

Algorithm software design and debug
Audio design and realization
VGA rastering and debugging

Shijie Hu

Hardware design and memory module
Timing optimization
Gamepad realization

Bowen Dang

Gameplay design
FSM design and
Gamepad realization

Reference

1. Lode's Computer Graphics Tutorial, Raycasting
http://lodev.org/cgtutor/raycasting.html

Appendix

Vga_led.sv

```
/*
 * Avalon memory-mapped peripheral for the VGA LED Emulator
 *
 *
 *
 */

// now this module is only about sub-module connection and digital tranfer

module VGA_LED(
                    input logic             clk,
                  input logic            reset,
                  input logic [31:0]    writedata,//control: writedata[7:0]
                  input logic            write,
                  input                      chipselect,
                  input logic [9:0]    address,
                  output logic [7:0] VGA_R, VGA_G, VGA_B,
                  output logic          VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n,
                  output logic          VGA_SYNC_n,
                    output logic        VGA_shoot);



    logic [9:0]    vcount;
    logic [10:0] hcount;
     logic [23:0] color1;
//   logic                   shoot;
     logic [1:0]    wound;

              VGA_LED_Emulator led_emulator(.clk50(clk), .*);
                Data_Decoder data_decoder( .shoot(VGA_shoot),.*);

 logic [6:0] hp;
 logic [5:0] am;
 logic [1:0] clip, bag;
 logic [7:0] hp_t,hp_n,am_t,am_n,clip_t,clip_n,bag_t,bag_n,hex90_t,hex90_n,hex60_t,hex60_n;
```

```
                digital_display digital_am(.number(am),.tens_code(am_t),.nums_code(am_n),.*);

                digital_display digital_hp(.number(hp),.tens_code(hp_t),.nums_code(hp_n),.*);
                digital_display digital_clip(.number(clip),.tens_code(clip_t),.nums_code(clip_n),.*);

                digital_display digital_bag(.number(bag),.tens_code(bag_t),.nums_code(bag_n ),.*);


    digital_display digital_90(.number({8'd90}),.tens_code(hex90_t),.nums_code(hex90_n),.*);
    digital_display digital_60(.number({8'd60}),.tens_code(hex60_t),.nums_code(hex60_n),.*);
endmodule
```

VGA_LED_Emulator.sv

```
/*
 * Seven-segment LED emulator
 *
 * Stephen A. Edwards, Columbia University
 */

module VGA_LED_Emulator(
 input logic            clk50, reset,
 input logic [7:0]
hp_t,hp_n,am_t,am_n,clip_t,clip_n,bag_t,bag_n,hex90_t,hex90_n,hex60_t,hex60_n,
 input logic [23:0]  color1,
 output logic [9:0] vcount,
 output logic [10:0] hcount,
 output logic [7:0] VGA_R, VGA_G, VGA_B,
 output logic           VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n);

/*
 * 640 X 480 VGA timing for a 50 MHz clock: one pixel every other cycle
 *
 * HCOUNT 1599 0                    1279           1599 0
 *                  _______________                 ________
 * ___________|       Video          |____________|    Video
 *
 *
 * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
 *          _______________________          _____________
 * |____|          VGA_HS             |____|
 */
    // Parameters for hcount
    parameter HACTIVE          = 11'd 1280,
                    HFRONT_PORCH = 11'd 32,
                    HSYNC              = 11'd 192,
                    HBACK_PORCH    = 11'd 96,
                    HTOTAL            = HACTIVE + HFRONT_PORCH + HSYNC + HBACK_PORCH; //
1600

     // Parameters for vcount
     parameter VACTIVE         = 10'd 480,
                    VFRONT_PORCH = 10'd 10,
                    VSYNC              = 10'd 2,
                    VBACK_PORCH    = 10'd 33,
                    VTOTAL            = VACTIVE + VFRONT_PORCH + VSYNC + VBACK_PORCH; // 525

     //logic [10:0]                          hcount; // Horizontal counter
                                                          // Hcount[10:1] indicates pixel column
(0-639)
   logic                          endOfLine;
     logic                          endofc;
     logic                          endofv;

     always_ff @(posedge clk50 or posedge reset)
        if (reset)         begin
                 hcount <= 0; endofc <= 0;
                 end
        else if (endOfLine) begin
                 hcount <= 0; endofc <= 0;
                 end
        else                    begin
                 hcount <= hcount + 11'd 1; endofc <= 1;
                 end

     assign endOfLine = hcount == HTOTAL - 1;

     // Vertical counter
     //logic [9:0]                           vcount;
     logic                          endOfField;

     always_ff @(posedge clk50 or posedge reset)
        if (reset)           begin
                 vcount <= 0; endofv <= 0;
                 end
        else if (endOfLine)
           if (endOfField)    begin
                      vcount <= 0; endofv <= 0;
                  end
         else                  begin
                      vcount <= vcount + 10'd 1; endofv <= 1;
                      end

     assign endOfField = vcount == VTOTAL - 1;

     // Horizontal sync: from 0x520 to 0x5DF (0x57F)
     // 101 0010 0000 to 101 1101 1111
     assign VGA_HS = !( (hcount[10:8] == 3'b101) & !(hcount[7:5] == 3'b111));
     assign VGA_VS = !( vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);

     assign VGA_SYNC_n = 1; // For adding sync to video signals; not used for VGA

     // Horizontal active: 0 to 1279          Vertical active: 0 to 479
     // 101 0000 0000    1280              01 1110 0000    480
     // 110 0011 1111    1599              10 0000 1100    524
     assign VGA_BLANK_n = !( hcount[10] & (hcount[9] | hcount[8]) ) &
                 !( vcount[9] | (vcount[8:5] == 4'b1111) );

     /* VGA_CLK is 25 MHz
      *                  __      __      __
      * clk50       __|     |__|     |__|
      *
      *                  _____           __
      * hcount[0]__|          |_____|
      */
     assign VGA_CLK = hcount[0]; // 25 MHz clock: pixel latched on rising edge



     logic                   inChar; // In any character






       assign inChar = ( vcount> 431 && vcount < 465 && hcount > 1215)||( vcount > 470 &&
vcount < 479 && hcount> 1215);


     logic [2:0]                  charx; // Coordinate within the 8x16 char
     logic [3:0]                  chary;
```

```
  assign charx = vcount[2:0];
  assign chary = hcount[4:1];


  logic horizBar, leftCol, rightCol, topCol, botCol; // Parts of the disp.

assign horizBar = !(charx[2:1] == 2'b11);    // Whensz in any horizontal bar
assign leftCol    = (charx == 3'd0);                   // When in left column
assign rightCol = (charx == 3'd5);                   // When in right column
assign topCol      = !chary[3] & !(chary[2:0] == 3'd7); // Top columns
assign botCol      = (chary >= 4'd6) & (chary <= 4'd12); // Bottom columns

logic [7:0] segment; // True when in each segment
assign segment[0] = horizBar & (chary == 4'd 0);

  assign segment[5] = rightCol & topCol;//mirror

  assign segment[4] = rightCol & botCol;//mirror
assign segment[3] = horizBar & (chary == 4'd 12);

  assign segment[2] = leftCol & botCol;//mirror

  assign segment[1] = leftCol & topCol;//mirror
assign segment[6] = horizBar & (chary == 4'd 6);
assign segment[7] = (charx == 3'd6) & (chary == 4'd14);



logic [2:0] column; // Being displayed
  logic                row;    // SH
assign column = vcount[5:3];
  assign row = hcount[5];


logic [7:0] curSegs;

        assign curSegs =    (column == 3'd3 && row == 0)? bag_n :
                                        (column == 3'd1 && row == 0)? hp_t :
                                           (column == 3'd0 && row == 0)? hp_n :
                                           (column == 3'd3 && row == 1)? clip_n :
                                           (column == 3'd1 && row == 1)? am_t:
                                        (column == 3'd0 && row == 1)? am_n:
```

```
                                                 (column == 3'd7 && row == 0)? hex90_t :
                                                 (column == 3'd6 && row == 0)? hex90_n :
                                                 (column == 3'd7 && row == 1)? hex60_t :
                                                 (column == 3'd6 && row == 1)? hex60_n : hex60_n;

   always_comb begin
   if (reset)
     {VGA_R, VGA_G, VGA_B}    = {8'h20, 8'h20, 8'h20}; // grey
    else

        {VGA_R, VGA_G, VGA_B}    = {color1[23:16],color1[15:8],color1[7:0]};


     if (inChar)
         if (|(curSegs & segment))
            {VGA_R, VGA_G, VGA_B} = {8'hA5, 8'h2A, 8'h2A};
   end
endmodule // VGA_LED_Emulator
```

data_decoder.sv

```
//This module decode input data to color data, flag data, also decide the priority of the pictures.

//input data [21:0] 3D display ,[24 :22] : 000:enermy1 001: enermy2 010: box1 011:box2 1XX:
NOTHING
module Data_Decoder (

             input logic             clk,write,
           input logic            reset, chipselect,
           input logic [31:0]    writedata,
             input logic [9:0]    vcount,
             input logic [10:0] hcount,
             input logic [9:0] address,
           output logic [23:0] color1,
             output logic            shoot,
             output logic [6:0] hp,
             output logic [5:0] am,
             output logic [1:0] clip, bag,
             output logic [1:0]    wound);

   logic [15:0] addr1, addr2, addr3, addr4, addr5, addr6, addr7, addr8, addr9, addr10,addr11;
   logic [23:0] q1;//boss
    logic [23:0] q2;//gun
```

```
  logic [23:0] q3;//fire
  logic [23:0] q4;//Winwin
  logic [23:0] q5;// texture1
  logic [23:0] q6;//sky
  logic [23:0] q7;
  logic [23:0] q8;
  logic [23:0] q9;
  logic [23:0] q10; //texture2
  logic [23:0] q11;


  logic [2:0] control;
  logic [2:0]    pickup;
logic [7:0] boss_hp;

  logic [1:0] hurt, boss_sec;
  logic empty, game_win, game_lose, treasure_clip, treasure_heal;

  logic    count;
  logic [10:0]    htemp;
  assign htemp = 2*hcount[10:1];



          fight fignt_process (.*);

            ROM1 BOSS1 (.clock(count), .q(q1),.address(addr1));//boss1
            ROM2 GUNROM    (.clock(count), .q(q2),.address(addr2));//gun
            ROM3 FIREROM (.clock(count), .q(q3),.address(addr3));//fire
            ROM4 Winwin (.clock(count), .q(q4),.address(addr4));//Winwin
            ROM5 Texture1(.clock(count), .q(q5),.address(addr5));//texture1//minion
            ROM6 Sky      (.clock(count), .q(q6),.address(addr6));//Sky
          // ROM7 FINDBOX    (.clock(count), .q(q7),.address(addr7));//FIND A BOX
            ROM8 BOX1                (.clock(count), .q(q8),.address(addr8));// AID
            ROM9 BOX2        (.clock(count), .q(q9),.address(addr9));// BULLET BOX
            ROM10 Texture2        (.clock(count), .q(q10),.address(addr10));// texture2//brick
            ROM11 GAMEOVER (.clock(count), .q(q11),.address(addr11));




         //ASK A RAM TO STORE DATA
```

```
            logic [63:0] indata;

            logic inScreen,color,win;

            RAM1 register1 (.clock(clk),
                        .wraddress(address),.rdaddress((959-2*vcount)),
                              .data(writedata),.q(indata[63:32]),
                              .wren(write));
            RAM2 register2 (.clock(clk),
                        .wraddress(address),.rdaddress((958-2*vcount)),
                              .data(writedata),.q(indata[31:0]),
                              .wren(write));




            always_ff @(posedge clk )begin
                if (reset) begin
                inScreen = 0;
                color = 0;
                end
                else
                begin
                //shoot <= indata [30];
                inScreen = ((hcount>=(2*(indata[9:0])))&&(hcount<=(2*(640-indata[9:0]))));
               color = (indata[21:20]==2'b01);
                win = (indata[21:20]==2'b11);
                count <= count +1;
                end
            end

                always_ff @(posedge clk) begin

                    shoot <= indata [30];
                    control <= indata[28:26];
            end




            logic fire,gun,boss1,boss2,box1,box2,bhp,mhp,aim;
            assign fire = ( htemp >1015 && htemp < 1115 && vcount > 100 &&vcount < 150 )
&&(!empty)&&(control==4) && count ;
            assign gun    =    htemp > 1043 && htemp < 1279 && vcount > 0 && vcount < 160 ;
            assign boss1 =    (htemp > 551 && htemp < 851 && vcount> 185    && vcount < 285)
&& (boss_sec==1) ;
```

```
        assign boss2 =   (htemp > 551 && htemp < 851 && vcount> 185    && vcount < 285)
&& (boss_sec==2) ;
        assign box1 =    (     htemp > 609 && htemp <689 && vcount >200    && vcount < 252)
&& (treasure_heal==1);
        assign box2 =      (    htemp > 609 && htemp <689 && vcount >200    && vcount < 252)
&& (treasure_clip==1);
        assign bhp = ( htemp > 1 && htemp < 33 && vcount < boss_hp );
        assign mhp = (    vcount > (423-hp) && vcount < 423 && htemp > 1215 && htemp <
1245);
        assign    aim= ((htemp > (579 + shoot *10)    && vcount > 238 && htemp < (599 +
shoot*10) && vcount <242 )||          //left
                      (htemp > (679- shoot*10) && vcount > 238 && htemp < (699-shoot*10)
&& vcount <242)||                     // right
                       (htemp > 635 && vcount > (210 +shoot *10) && htemp < 643 &&
vcount <( 220+ shoot * 10))||             //up
                       (htemp > 635 && vcount > (260 - shoot *10) && htemp < 643 &&
vcount < (270 -shoot *10)));             //down


        assign sky= (htemp < 641 );
        assign ground =( htemp > 639 && htemp < 1279);


       logic [31:0]    temp,addr_wall;


        assign temp={16'h0,indata[61:46]};
        //assign addr_wall
=(indata[45:32]+(((2*hcount[10:1]-{indata[9:0],1'b0})*temp)/16384));
      assign addr_wall =(indata[45:32]+(((htemp-{indata[9:0],1'b0})*temp)/16384));

        always_ff @(posedge count) begin
        if (sky) begin //sky
                //addr6 <= (((479-vcount)*128)/480+hcount/5);
              addr6 <= (vcount *2+ htemp)/ 5;
                color1 <= q6;
        end

        if (ground) begin color1 <= {8'hd2, 8'hb4, 8'h84}; // ground
        end

        if (inScreen && color) begin      //Mikilin
                    addr10 <= addr_wall;
                    color1<=q10;
    end

    else if (inScreen && win) begin      //Mikilin
                    addr4 <= addr_wall;



                    color1<=q4;
    end

    else if(inScreen)begin //minions
                    addr5 <= addr_wall;

                    color1<=q5;
    end

  if ( boss1) begin             //boss1
                            addr1 <= (vcount-186)*300+(htemp-551);
                            if (q1 != {8'hff,8'hff,8'hff})
                            color1<=q1;
    end

    if ( boss2 ) begin             //boss2
                            addr1 <= (vcount-186)*300+(htemp-551);
                            if (q1 != {8'hff,8'hff,8'hff})
                            color1<=q1;
    end

    if ( box1 ) begin             //box1
                            addr8 <= (vcount-201)*78+(htemp-611);
                            if (q8 != {8'hff,8'hff,8'hff})
                            color1<=q8;
    end

    if ( box2) begin             //box2
                            addr9 <= (vcount-201)*78+(htemp-611);
                            if (q9 != {8'hff,8'hff,8'hff})
                            color1<=q9;
    end

    if (fire) begin    //fire
               addr3 <= (vcount-101)*100+(htemp-1005);
                    if ( q3 != {8'hff, 8'hff, 8'hff})
```

```
                                color1 <= q3;
        end

        if ( gun ) begin        //gun
                            addr2 <= (vcount)*240+htemp-1035;
                                if (q2 != {8'hff,8'hff,8'hff})
                                color1<=q2;
        end



        if (aim) begin
            color1<= {8'h00, 8'hff, 8'h00}; //green
        end
        if ( bhp) begin
                                    color1 <= {8'hb2, 8'h22, 8'h22};//red
        end
        if ( mhp ) begin //HP_sh
                               color1 <= {8'hb2, 8'h22, 8'h22}; // Red
        end

        if ( vcount > (423-60)    && vcount < 423 && htemp > 1250 && htemp < 1280 &&
(vcount%3!=0)) begin //POWER_sh
            color1<= {8'hdc, 8'hdc, 8'hdc}; // grey loss
        end

       if ( vcount > (423-am)    && vcount < 423 && htemp > 1250 && htemp < 1280 &&
(vcount%3!=0))    begin //POWER_sh
            color1<= {8'h00, 8'h80, 8'h00}; // green
        end

        if (( vcount > 464 && vcount < 466 && htemp > 1216 && htemp < 1247)||( vcount >
464 && vcount < 466 && htemp > 1248 && htemp < 1277 ))    begin
        color1 <= {8'hA5, 8'h2A, 8'h2A};
        end
        if (game_lose) begin
                  /*            color1 <= {8'h00,8'h00,8'h00};
                                end
                                else if ( (530 < htemp < 726) && (100 < vcount < 266)) begin*/
                                addr11 <= (vcount)/3*65+(htemp/3);
                                color1 <= q11;
                                end
        end
            endmodule
```

digital_decode.sv

```
module digital_decode(

input      clk,reset,
input      logic [4:0] digital,
output     logic [7:0] digital_code
);

always_ff @ (posedge clk) begin
if (reset)
   digital_code = 8'b00111111; // 0
else begin
     case (digital)
    0:digital_code <= 8'b00111111; // 0
    1:digital_code <= 8'b00000110; // 1
    2:digital_code <= 8'b01011011; // 2
    3:digital_code <= 8'b01001111; // 3
    4:digital_code <= 8'b01100110; // 4
    5:digital_code <= 8'b01101101; // 5
    6:digital_code <= 8'b01111101; // 6
    7:digital_code <= 8'b00000111; // 7
    8:digital_code <= 8'b01111111; // 8
    9:digital_code <= 8'b01101111; // 9
       endcase
end
end
endmodule
```

digital_display.sv

```
module digital_display(

input logic clk,reset,
input    logic [7:0] number,
output logic [7:0] tens_code,
output logic [7:0] nums_code
);
```

```
logic [4:0] tens;
logic [4:0] nums;

assign tens=number/10;
assign nums=number%10;

digital_decode decode_tens (.digital(tens),.digital_code(tens_code),.*);
digital_decode decode_nums (.digital(nums),.digital_code(nums_code),.*);

endmodule
```

fight.sv

```
module fight(
        input logic clk,reset,count,
        input logic [9:0]    vcount,
        input logic [10:0] hcount,
        input logic [63:0]    indata,
        output logic [7:0] boss_hp,
        output logic [6:0] hp,
        output logic [5:0] am,
        output logic [1:0] hurt, clip, bag, boss_sec,
        output logic empty, game_win, game_lose, treasure_clip, treasure_heal
);


parameter SIZE =    4;
parameter IDLE = 4'b0000,
                          MAKETHROUGH = 4'b0001,
                          FIGHT = 4'b0010,
                          TREASURE1 = 4'b0011,
                          TREASURE2 = 4'b0100,
                          START = 4'b0101,
                          WIN = 4'b0110,
                          BOSS_ROUND = 4'b0111,
                          LOAD = 4'b1000,
                          HEAL = 4'b1001,
                          LOSE = 4'b1010,
                          CURSTAT = 4'b1011,
                          BOSS_INI = 4'b1100,
                          MODE_SELECT = 4'b1101,
                          TREASURE1_GET = 4'b1110,
                          TREASURE2_GET = 4'b1111;
```

```
reg [SIZE-1:0] state;
reg [8:0] boss_hp_cur;
reg [5:0] am_cur;
reg [6:0] hp_cur;
reg [1:0] clip_cur;
reg [1:0] bag_cur;

logic [1:0] boss_select;
logic [4:0] boss_attack;
logic [2:0] pickup;
logic [2:0] control;

assign pickup = indata [24:22];
assign control = indata [28:26];


always_ff @(posedge count && (vcount == 0) && (hcount == 0)) begin
        if (indata[29]) begin
                hp = 7'd90;
                am = 6'd60;
                hp_cur = 7'd90;
                am_cur = 6'd60;
                hurt = 2'b00;
                empty = 1'b0;
                clip_cur = 2'b10;
                clip = 2'b10;
                bag_cur = 2'b10;
                bag = 2'b10;
                game_win = 1'b0;
                game_lose = 1'b0;
                treasure_clip = 1'b0;
                treasure_heal = 1'b0;
                boss_sec = 2'b00;
                boss_select = 2'b00;
                state <= IDLE;
        end else
          case (state)
                IDLE: begin
                        game_win = 1'b0;
                        game_lose = 1'b0;
                        treasure_clip = 1'b0;
                        treasure_heal = 1'b0;
                        boss_sec = 2'b00;
                        boss_select = 2'b00;
                        hp = 7'd90;
                        am = 6'd60;
                        hp_cur = 7'd90;
                        am_cur = 6'd60;
                        clip_cur = 2'b10;
                        clip = 2'b10;
                        bag_cur = 2'b10;
                        bag = 2'b10;
                        state <= MAKETHROUGH;
                end

                MAKETHROUGH: begin

                        if (pickup == 3'b000) begin                     // nothing happened, what if
there is control signal
                    if (control == 3'b001) begin
                        state <= MODE_SELECT;
                    end else if ( control == 3'b010) begin          // Loading the clip
                        if (clip_cur > 0) begin
                            state <= LOAD;
                    end else begin
                        state <= MAKETHROUGH;
                    end
                    end else if ( control == 3'b011) begin          // Healing yourself
                        if (bag_cur > 0) begin
                            state <= HEAL;
                        end else begin
                            state <= MAKETHROUGH;
                        end
                    end else if ( control == 3'b100) begin          // Wasting your AM
                        if ( am_cur > 0 ) begin
                            am = am_cur - 1;
                            am_cur <= am_cur - 1;
                            state <= MAKETHROUGH;
                        end else begin
                            empty = 1'b1;
                            state <= MAKETHROUGH;
                        end
                    end else begin
                        state <= MAKETHROUGH;
                    end
                        end else if (pickup == 3'b001) begin                // boss1 comes!
                    boss_select = 2'b01;
                    boss_sec = 2'b01;
                state <= BOSS_INI;
            end else if (pickup == 3'b010) begin                    // boss2 comes!
                boss_select = 2'b10;
                boss_sec = 2'b10;
                state <= BOSS_INI;
            end else if (pickup == 3'b011) begin                    // get a AmBox
                state <= TREASURE1;
            end else if (pickup == 3'b100) begin                    // get a FirstAidBag
                state <= TREASURE2;
            end else begin
                state <= MAKETHROUGH;
            end
end

TREASURE1: begin
            treasure_clip = 1'b1;
            if ( control == 3'b001 ) begin
                state <= TREASURE1_GET;
            end else begin
                state <= TREASURE1;
            end
end

TREASURE1_GET: begin
            if (clip_cur < 2'b11) begin
                clip = clip_cur + 1;
                clip_cur <= clip_cur + 1;
            end else begin
                clip = clip_cur;
            end
            treasure_clip = 1'b0;
            state <= MAKETHROUGH;
end

TREASURE2: begin
            treasure_heal = 1'b1;
            if ( control == 3'b001 ) begin
                state <= TREASURE2_GET;
            end else begin
                state <= TREASURE2;
            end
end

TREASURE2_GET: begin
```

```
        if (bag_cur < 2'b11) begin
                bag = bag_cur + 1;
                bag_cur <= bag_cur + 1;
        end else begin
                bag = bag_cur;
        end
        treasure_heal = 1'b0;
        state <= MAKETHROUGH;
end

MODE_SELECT: begin
        state <= START;
end

BOSS_INI: begin
        boss_hp = (boss_select == 2'b10)*50 + (boss_select == 2'b01)*100;
        boss_hp_cur = (boss_select == 2'b10)*50 + (boss_select == 2'b01)*100;
        boss_attack = (boss_select == 2'b10)*7 + (boss_select == 2'b01)*3;
        state <= START;
end

START: begin
        if (control == 3'b001) begin                    // mode change
                state <= MODE_SELECT;
        end else if ( control == 3'b010) begin          // loading your clip
                if (clip_cur > 0) begin
                        state <= LOAD;
                end else begin                                  // you have no more AMs
                        state <= BOSS_ROUND;
                end
        end else if ( control == 3'b011) begin          // Emergency Heal
                if (bag_cur > 0) begin
                        state <= HEAL;
                end else begin
                        state <= BOSS_ROUND;
                end
        end else if ( control == 3'b100) begin
                state <= FIGHT;
        end else begin
                state <= START;
        end
end

FIGHT: begin
```

```
        if (boss_hp_cur < 10) begin
                        boss_hp_cur = 0;
                        boss_hp = 0;
                        state <= WIN;
        end else begin
                if (am_cur > 3) begin
                        am = am_cur -2;
                        am_cur <= am_cur - 2;
                        boss_hp_cur <= boss_hp_cur - 3;
                        boss_hp = boss_hp_cur - 3;
                        state <= BOSS_ROUND;
                end else begin
                        am_cur = 0;
                        am = 0;
                        empty = 1;
                        state <= LOAD;
                end
        end

end

BOSS_ROUND: begin
        if (hp_cur < boss_attack) begin
                hp_cur = 0;
                hp = 0;
                state <= LOSE;
        end else begin
                hp = hp_cur - boss_attack;
                hp_cur <= hp_cur - boss_attack;
                state <= CURSTAT;
        end
end

CURSTAT: begin
        //hurt = (hp_cur < 20)*2'b10 + (20 <= hp_cur < 50)*2'b01 + (50<= hp_cur)*2'b00;
        if ( am_cur == 0 && clip_cur != 0) begin
                        state <= LOAD;
        end else begin
                        state <= START;
        end
end

LOAD: begin
        if (control == 3'b000) begin
```

```
                if ( clip_cur > 0 ) begin
                        clip = clip_cur - 1;
                        clip_cur <= clip_cur -1;
                        if (am_cur < 30) begin
                                am = am_cur + 30;
                                am_cur <= am_cur + 30;
                        end else begin
                                am = 60;
                                am_cur <= 60;
                        end
                        empty = 1'b0;
                end
                if (boss_select == 2'b00) begin
                        state <= MAKETHROUGH;
                end else begin
                        state <= BOSS_ROUND;
                end
        end else begin
                state <= LOAD;
        end
end

HEAL: begin
        if (control == 3'b000) begin
                if ( bag_cur > 0 ) begin
                        bag = bag_cur -1;
                        bag_cur <= bag_cur -1;
                        if (hp_cur < 40) begin
                                hp = hp_cur + 50;
                                hp_cur <= hp_cur + 50;
                        end else begin
                                hp = 90;
                                hp_cur = 90;
                        end
                end
                if (boss_select == 2'b00) begin
                        state <= MAKETHROUGH;
                end else begin
                        state <= BOSS_ROUND;
                end
        end else begin
                state <= HEAL;
        end
end
```

```
        WIN: begin
                game_win = 1'b1;
                boss_sec = 2'b00;
                boss_select = 2'b00;
                hp_cur = 7'd90;
                hp = 7'd90;
                am_cur = 6'd60;
                am = 6'd60;
                state <= MAKETHROUGH;
            end

        LOSE: begin
            game_lose = 1'b1;
            state <= LOSE;
        end
        endcase
end



endmodule
```

audio_top.sv

```
module audio_top (
    input   OSC_50_B8A,
     input shoot,
    inout   AUD_ADCLRCK,
    input   AUD_ADCDAT,
    inout   AUD_DACLRCK,
    output AUD_DACDAT,
    output AUD_XCK,
    inout   AUD_BCLK,
    output AUD_I2C_SCLK,
    inout   AUD_I2C_SDAT,
    output AUD_MUTE,

    input   [3:0] KEY,
    input   [3:0] SW,
    output [3:0] LED
);

wire reset = !KEY[0];
wire main_clk;
wire audio_clk;

wire [1:0] sample_end;
wire [1:0] sample_req;
wire [15:0] audio_output;
wire [15:0] audio_input;

clock_pll pll (
        .refclk (OSC_50_B8A),
        .rst (reset),
        .outclk_0 (audio_clk),
        .outclk_1 (main_clk)
);

i2c_av_config av_config (
        .clk (main_clk),
        .reset (reset),
        .i2c_sclk (AUD_I2C_SCLK),
        .i2c_sdat (AUD_I2C_SDAT),
        .status (LED)
);

assign AUD_XCK = audio_clk;
assign AUD_MUTE = (SW != 4'b0);

audio_codec ac (
        .clk (audio_clk),
        .reset (reset),
        .sample_end (sample_end),
        .sample_req (sample_req),
        .audio_output (audio_output),
        .audio_input (audio_input),
        .channel_sel (2'b10),

        .AUD_ADCLRCK (AUD_ADCLRCK),
        .AUD_ADCDAT (AUD_ADCDAT),
        .AUD_DACLRCK (AUD_DACLRCK),
        .AUD_DACDAT (AUD_DACDAT),
        .AUD_BCLK (AUD_BCLK)
);

audio_effects ae (
        .clk (audio_clk),
        .sample_end (sample_end[1]),
        .sample_req (sample_req[1]),
        .audio_output (audio_output),
        .audio_input    (audio_input),
        .control (shoot)
);
Endmodule
```

audio_effects.sv

```
module audio_effects (
        input    clk,
        input    sample_end,
        input    sample_req,
        output [15:0] audio_output,
        input    [15:0] audio_input,
        input        control
);

reg [15:0] romdatafire,stepdatastep;
reg [12:0]    index1 = 13'd0;
reg [14:0]    index2 = 15'd0;
reg [15:0] last_sample;
reg [15:0] dat;

assign audio_output = dat;

//parameter SINE          = 0;
//parameter FEEDBACK = 1;

ROM12 SHOOT (.address(index1),.q(romdatafire),.clock(clk));
//ROM13 STEP (.address(index2),.q(romdatastep),.clock(clk));


always @(posedge clk) begin

                if (control==1)//[SINE])
                        //dat <= last_sample;
                                  begin
                        dat <= romdatafire;
                              if (index1 == 13'd8063)
                                index1 <= 13'd0;
                        else
                                index1 <= index1+1'b1;
```

```
            end


                /*    begin
                dat <= romdatastep;
                    if (index2 == 15'd28829)
                      index2 <= 15'd28829;
                else
                      index2 <= index2+1'b1;
            end */

    /*      else    if (control[FEEDBACK])
              begin
                dat <= romdatafire;
                    if (index1 == 13'd8063)
                      index1 <= 13'd8063;
                else
                      index1 <= index1+1'b1;
            end

      */      else
                dat <= 16'd0000;
              end


endmodule
```

mudd.c

```
/*
 * Userspace program that communicates with the led_vga device driver
 * primarily through ioctls
 *
 * Stephen A. Edwards
 * Columbia University

 */


#include <time.h>
#include <math.h>

#include <stdlib.h>
#include <stdio.h>
```

```
#include "vga_led.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include "fbputchar.h"
#include <sys/socket.h>
#include <arpa/inet.h>
#include "usbkeyboard.h"


int vga_led_fd;
struct libusb_device_handle *keyboard;
int endpoint_address;


//place the example code below here:

#define mapWidth 47
#define mapHeight 35
#define texWidth 128
#define texHeight 128
#define w 480
#define h 640

#define random(x) (rand()%x)

int worldMap[mapWidth][mapHeight]=
{
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1},
  {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,1,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,2,1,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,1,1,0,0,0,1},
    {1,0,0,0,0,0,0,0,0,0,0,0,2,1,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,1,1,0,0,0,1},
    {1,0,0,0,0,0,0,0,0,0,0,0,2,1,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,1,1,0,0,0,1},
    {1,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,1},
    {1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
    {1,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
    {1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
    {1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
    {1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {1,1,1,1,1,1,1,1,0,0,0,1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,1,0,0,0,1,1,1,1,1,1,1,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
    {0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
    {0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
    {0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
    {0,0,0,0,0,0,0,1,1,1,1,1,1,1,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,1},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0,1},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,1,1,1,1,1,1,1,1},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,1,1,1,1,1,1,0,0,0,1,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0}
};



void write_screen(unsigned int data1[480],unsigned int data2[480])
{
    vga_led_screen screen;
    int i;
    for (i = 0 ; i < 480 ; i++) {
    screen.column = i;
    screen.data1 = data1[i];
    screen.data2 = data2[i];
  if (ioctl(vga_led_fd, VGA_LED_WRITE_SCREEN, &screen)) {
            perror("ioctl(VGA_LED_WRITE_SCREEN) failed");
            return;
        }
    }
}


int main()
{
    //vga_led_arg_t vla;
    //int i;
    static const char filename[] = "/dev/vga_led";

    //static unsigned char message[8] = { 0x39, 0x6D, 0x79, 0x79,
                             //            0x66, 0x7F, 0x66, 0x3F };

    printf("VGA LED Userspace program started\n");

    if ( (vga_led_fd = open(filename, O_RDWR)) == -1) {
        fprintf(stderr, "could not open %s\n", filename);
        return -1;
    }

  struct usb_keyboard_packet packet;
    int transferred;
    char keystate[12];

  if ( (keyboard = openkeyboard(&endpoint_address)) == NULL ) {
        fprintf(stderr, "Did not find a keyboard\n");
        exit(1);
    }










    printf("initial state: ");

    double posX = 13.5, posY = 15.5;    //x and y start position
    double dirX = -1.0, dirY = 0.0; //initial direction vector
    double planeX = 0.0, planeY = 0.66; //the 2d raycaster version of camera plane



    int counttime=6000;
    int control =0;        // decide whether to shoot
    int wound = 0;
    int delay = 0;         // decide firing time
    int randa = 5;    // whether meets with a boss
    int scenario = 0;      // put into the hardware
    int gamereset = 0;

  while(counttime>0)
    {
        int x;
        unsigned int passdata[480];
        unsigned int passdata1[480];

        for(x = 0; x < w; x++)
        {

            //calculate ray position and direction
            double cameraX = 2.0 * x/480.0-1.0; //x-coordinate in camera space
            double rayPosX = posX;
            double rayPosY = posY;
            double rayDirX = dirX + planeX * cameraX;
            double rayDirY = dirY + planeY * cameraX;
            //which box of the map we're in
            int mapX = (int)rayPosX;
            int mapY = (int)rayPosY;

            //length of ray from current position to next x or y-side
            double sideDistX;
            double sideDistY;

              //length of ray from one x or y-side to next x or y-side
            double deltaDistX = sqrt(1 + (rayDirY * rayDirY) / (rayDirX * rayDirX));
            double deltaDistY = sqrt(1 + (rayDirX * rayDirX) / (rayDirY * rayDirY));

            double perpWallDist;

            //what direction to step in x or y-direction (either +1 or -1)
            int stepX;
            int stepY;
```

```
      int hit = 0; //was there a wall hit?
      int side; //was a NS or a EW wall hit?
      //calculate step and initial sideDist
      if (rayDirX < 0)
      {
        stepX = -1;
        sideDistX = (rayPosX - mapX) * deltaDistX;
      }
      else
      {
        stepX = 1;
        sideDistX = (mapX + 1.0 - rayPosX) * deltaDistX;
      }
      if (rayDirY < 0)
      {
        stepY = -1;
        sideDistY = (rayPosY - mapY) * deltaDistY;
      }
      else
      {
        stepY = 1;
        sideDistY = (mapY + 1.0 - rayPosY) * deltaDistY;
      }
      //perform DDA
// printf("hit\n");

      while (hit == 0)
      {
        //jump to next map square, OR in x-direction, OR in y-direction
        if (sideDistX < sideDistY)
        {
          sideDistX += deltaDistX;
          mapX += stepX;
          side = 0;
        }
        else
        {
          sideDistY += deltaDistY;
          mapY += stepY;
          side = 1;
        }
        //Check if ray has hit a wall
        if (worldMap[mapX][mapY] > 0) hit = 1;
```

```
	}

	//Calculate distance projected on camera direction (oblique distance will give fisheye
effect!)
	if (side == 0)
	perpWallDist = fabs((mapX - rayPosX + (1 - stepX) / 2) / rayDirX);
	else
	perpWallDist = fabs((mapY - rayPosY + (1 - stepY) / 2) / rayDirY);

	//Calculate height of line to draw on screen

	int k=h/perpWallDist;
	int lineHeight = abs(k);

	//calculate lowest and highest pixel to fill in current stripe
	int drawStart = -lineHeight / 2 + h / 2;
	if(drawStart < 0)drawStart = 0;
	int drawEnd = lineHeight / 2 + h / 2;
	if(drawEnd >= h)drawEnd = h - 1;

	int color = 2;

	//give x and y sides different brightness
	if (side == 1) {color = color / 2;}
	if (worldMap[mapX][mapY]==2) {color = 3;}
	//calculate value of wallX
	double wallX; //where exactly the wall was hit
	if (side == 1) wallX = rayPosX + ((mapY - rayPosY + (1 - stepY) / 2) / rayDirY) * rayDirX;
	else		wallX = rayPosY + ((mapX - rayPosX + (1 - stepX) / 2) / rayDirX) * rayDirY;
	wallX -= floor((wallX));

	//x coordinate on the texture
	int texX = (int)(wallX * (double)(texWidth));
	if(side == 0 && rayDirX > 0) texX = texWidth - texX - 1;
	if(side == 1 && rayDirY < 0) texX = texWidth - texX - 1;



	int lengthH = drawEnd - drawStart;
	int coff=0;
	coff=(128*8192)/(lengthH);
	int texY = 0;
		texY = 128*texX;
      passdata[x]=(scenario<<22|gamereset<<29|control<<26|color<<20|texX<<10|drawStart);/
/change SH
```

```
        passdata1[x]=(coff<<14|texY);

}



        write_screen(passdata,passdata1);
//printf("passdata %320d\n", passdata);
        //printf("good\n");
/*
        int c;
        for (c = 0;c<480;c++)
        {
        printf("%d;",passdata[c]);
        }
*/

control = 0;

    libusb_interrupt_transfer(keyboard, endpoint_address, (unsigned char *) &packet,
6,&transferred, 0);


      if ((transferred ==8 && (((packet.keycode[0]!= 0x7f || packet.keycode[1]!= 0x7f) &&
packet.keycode[0]!=0x80) ||            packet.keycode[5]!=0x0f)))

  {


        //printf("%02x %02x %02x %02x %02x %02x %02x %02x %02x\n ",
packet.keycode[0],packet.keycode[1],packet.keycode[2],packet.keycode[3],packet.keycode[4],p
acket.keycode[5],packet.keycode[6],packet.keycode[7]);

                wound = 0;
                gamereset = 0;
                control = 0;
        if((packet.keycode[1] == 0x00))
        {               int locationx1=(int)posX + dirX * 0.02;
                         int locationy1=(int)posY;
                                 int locationx2=(int)posX;
                         int locationy2=(int)posY + dirY * 0.02;
                  if(worldMap[locationx1][locationy1] == 0){
                //posX=posX+((packet.keycode[1] == 0xff)-(packet.keycode[1] ==
0x00)+(packet.keycode[5] == 0x04)-(packet.keycode[5] == 0x00))*0.01;}
```

```
                        if (randa < 10){
                    randa = random(100);
                    if ( randa > 80){
                        scenario = 0x01;}
                    else if ( randa >    70){
                        scenario = 0x02;}
                    else if ( randa > 30){
                        scenario = 0x03;}
                    else if ( randa > 10){
                        scenario = 0x04;}
                    else {scenario = 0x00;}
                }
                else { randa = random(100); scenario = 0x00;}
                printf("%02x\n",scenario);
                posX = posX + dirX*0.02;}
                if(worldMap[locationx2][locationy2] == 0){
                posY = posY+ dirY * 0.02;
                }



            }
            if((packet.keycode[1] == 0xff))
            {
               scenario = 0x00;
                  int locationx1=(int)posX - dirX * 0.02;
                        int locationy1=(int)posY;
                              int locationx2=(int)posX;
                        int locationy2=(int)posY - dirY * 0.02;
                  if(worldMap[locationx1][locationy1] == 0){
                //posX=posX+((packet.keycode[1] == 0xff)-(packet.keycode[1] ==
0x00)+(packet.keycode[5] == 0x04)-(packet.keycode[5] == 0x00))*0.01;}
                posX = posX - dirX*0.02;}
                if(worldMap[locationx2][locationy2] == 0){
                posY = posY - dirY * 0.02;
                }
            }
            if((packet.keycode[0] == 0x00))
            {
               scenario = 0x00;
                //posX=posX+((packet.keycode[0] == 0xff)-(packet.keycode[0] ==
0x00)+(packet.keycode[0] == 0x04)-(packet.keycode[0] == 0x00))*0.01;
                //posX=posX-0.01;

                double oldDirX = dirX;
```

```
        dirX = dirX    - dirY * 0.01;
        dirY = oldDirX * 0.01 + dirY;
        double oldPlaneX = planeX;
        planeX = planeX    - planeY * 0.01;
        planeY = oldPlaneX * 0.01 + planeY;



}


if ((packet.keycode[0] == 0xff))
{
    scenario = 0x00;
    double oldDirX = dirX;
        dirX = dirX * cos(-0.01) - dirY * sin(-0.01);
        dirY = oldDirX * sin(-0.01) + dirY * cos(-0.01);
        double oldPlaneX = planeX;
        planeX = planeX * cos(-0.01) - planeY * sin(-0.01);
        planeY = oldPlaneX * sin(-0.01) + planeY * cos(-0.01);

    //posX=posX+0.01;
}

if ((packet.keycode[5] == 0x8f))
{
    scenario = 0x00;
    control=4;

    wound = 1;

    printf ("would %10d\n",wound);


}

if ((packet.keycode[5] == 0x1f)&&packet.keycode[4]==0x7f)
{
    scenario = 0x00;
    control=1;

    printf ("would %10d\n",wound);

    usleep(100000);
```

```
        }

        if ((packet.keycode[5] == 0x1f)&&packet.keycode[4]==0x80)
        {
              posX = 13.5, posY = 15.5;
            dirX = -1.0, dirY = 0.0;
                planeX = 0.0, planeY = 0.66;
            gamereset = 1;

        }


        if ((packet.keycode[5] == 0x2f))
        {
            scenario = 0x00;
            control=2;

            wound = wound + 1;

            printf ("would %10d\n",wound);

            usleep(100000);

        }

        if ((packet.keycode[5] == 0x4f))
        {
            scenario = 0x00;
            control=3;

            wound = wound + 1;

            printf ("would %10d\n",wound);

            usleep(100000);

        }

    }

        //else {printf("nothing happened\n");}
  //usleep(100000);
```

```
        //posy=posy+((packet.keycode[0] == 0xff)-(packet.keycode[0] == 0x00)+(packet.keycode[5]
== 0x02)-(packet.keycode[5] == 0x06))*0.0025;
        //posX=posX+((packet.keycode[1] == 0xff)-(packet.keycode[1] == 0x00)+(packet.keycode[5]
== 0x04)-(packet.keycode[5] == 0x00))*0.01;



    }
    printf("VGA LED Userspace program terminating\n");
    return 0;
}
```

vga_led.c

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "vga_led.h"

#define DRIVER_NAME "vga_led"

/*
 * Information about our device
 */
struct vga_led_dev {
        struct resource res; /* Resource: our registers */
        void __iomem *virtbase; /* Where registers can be accessed in memory */
        //u8 segments[VGA_LED_DIGITS];
                u32 data1[480];
        u32 data2[480];
} dev;

/*
```

```
  * Write segments of a single digit
  * Assumes digit is in range and the device information has been set up
  */



static void write_screen(int x, u32 data1, u32 data2)
{
                iowrite32(data1,dev.virtbase+8*x);
        iowrite32(data2,dev.virtbase+8*x+4);
        dev.data1[x] = data1;
        dev.data2[x] = data2;
}

/*
  * Handle ioctl() calls from userspace:
  * Read or write the segments on single digits.
  * Note extensive error checking of arguments
  */
static long vga_led_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
        vga_led_arg_t vla;
        vga_led_ball ball;
                vga_led_screen screen;

        switch (cmd) {


        case VGA_LED_WRITE_SCREEN:
                if (copy_from_user(&screen, (vga_led_screen *) arg,
                                      sizeof(vga_led_screen)))
                        return -EACCES;
                if (screen.column > 480)
                        return -EINVAL;
                write_screen(screen.column,screen.data1,screen.data2);
                break;



        default:
                return -EINVAL;
        }

        return 0;
```

```
}

/* The operations our device knows how to do */
static const struct file_operations vga_led_fops = {
        .owner          = THIS_MODULE,
        .unlocked_ioctl = vga_led_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice vga_led_misc_device = {
        .minor          = MISC_DYNAMIC_MINOR,
        .name           = DRIVER_NAME,
        .fops           = &vga_led_fops,
};

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init vga_led_probe(struct platform_device *pdev)
{
        int playgo = 2751488;
        static unsigned char welcome_message[VGA_LED_DIGITS] = {
                0x3E, 0x7D, 0x77, 0x08, 0x38, 0x79, 0x5E, 0x00};
        int i, ret;

        /* Register ourselves as a misc device: creates /dev/vga_led */
        ret = misc_register(&vga_led_misc_device);

        /* Get the address of our registers from the device tree */
        ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
        if (ret) {
                ret = -ENOENT;
                goto out_deregister;
        }

        /* Make sure we can use these registers */
        if (request_mem_region(dev.res.start, resource_size(&dev.res),
                               DRIVER_NAME) == NULL) {
                ret = -EBUSY;
                goto out_deregister;
        }

        /* Arrange access to our registers */
	dev.virtbase = of_iomap(pdev->dev.of_node, 0);
	if (dev.virtbase == NULL) {
		ret = -ENOMEM;
		goto out_release_mem_region;
	}

	//write_ball(100,100);

	for (i = 0; i < 480; i++)
		write_screen(i, playgo,playgo);



	return 0;




out_release_mem_region:
	release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
	misc_deregister(&vga_led_misc_device);
	return ret;
}

/* Clean-up code: release resources */
static int vga_led_remove(struct platform_device *pdev)
{
	iounmap(dev.virtbase);
	release_mem_region(dev.res.start, resource_size(&dev.res));
	misc_deregister(&vga_led_misc_device);
	return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id vga_led_of_match[] = {
	{ .compatible = "altr,vga_led" },
	{},
};
MODULE_DEVICE_TABLE(of, vga_led_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver vga_led_driver = {
	.driver	= {
		.name	= DRIVER_NAME,
		.owner	= THIS_MODULE,
		.of_match_table = of_match_ptr(vga_led_of_match),
	},
	.remove	= __exit_p(vga_led_remove),
};

/* Called when the module is loaded: set things up */
static int __init vga_led_init(void)
{
	pr_info(DRIVER_NAME ": init\n");
	return platform_driver_probe(&vga_led_driver, vga_led_probe);
}

/* Called when the module is unloaded: release resources */
static void __exit vga_led_exit(void)
{
	platform_driver_unregister(&vga_led_driver);
	pr_info(DRIVER_NAME ": exit\n");
}

module_init(vga_led_init);
module_exit(vga_led_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Stephen A. Edwards, Columbia University");
MODULE_DESCRIPTION("VGA 7-segment LED Emulator");
```

vga_led.h

```
#ifndef _VGA_LED_H
#define _VGA_LED_H



#include <linux/ioctl.h>

#define VGA_LED_DIGITS 8


typedef struct {
   unsigned int column;
   unsigned int data1;
```

```
    unsigned int data2;

} vga_led_screen;


typedef struct {
    unsigned int cox;
    unsigned int coy;
} vga_led_ball;


typedef struct {
    unsigned char digit;        /* 0, 1, .. , VGA_LED_DIGITS - 1 */
    unsigned char segments; /* LSB is segment a, MSB is decimal point */
} vga_led_arg_t;

#define VGA_LED_MAGIC 'q'

/* ioctls and their arguments */
#define VGA_LED_WRITE_DIGIT _IOW(VGA_LED_MAGIC, 1, vga_led_arg_t *)
#define VGA_LED_READ_DIGIT    _IOWR(VGA_LED_MAGIC, 2, vga_led_arg_t *)
#define VGA_LED_WRITE_BALL    _IOW(VGA_LED_MAGIC, 3, vga_led_arg_t *)
#define VGA_LED_WRITE_SCREEN    _IOW(VGA_LED_MAGIC, 4, vga_led_arg_t *)

#endif
```