Air-Raid Project

Jakob Stiens, Yongmao Luo, Zhaomeng Wang, Tristan Saidi

Contents

1. Introduction

We plan to design Air-Raid, a variation of the Atari game “River Raid”. The premise of the game is that you control a plane flying above a river. The river will be inbounds and flying over the land will be out of bounds and cause the plane to crash. There will also be obstacles on the river (such as battleships and enemy airships) that cause the plane to crash if it runs into them. To defend itself, the plane will be able to shoot and destroy obstacles on the river. The plane will also have a fuel gauge that will slowly decrease over time - if this gauge hits zero, the plane will also crash. There will be fuel sprites that will appear on the river which the plane can pick up to regenerate fuel. For every obstacle shot the score will increase. The goal of the game is to get as high of a score as possible before crashing.

2. Game Logic

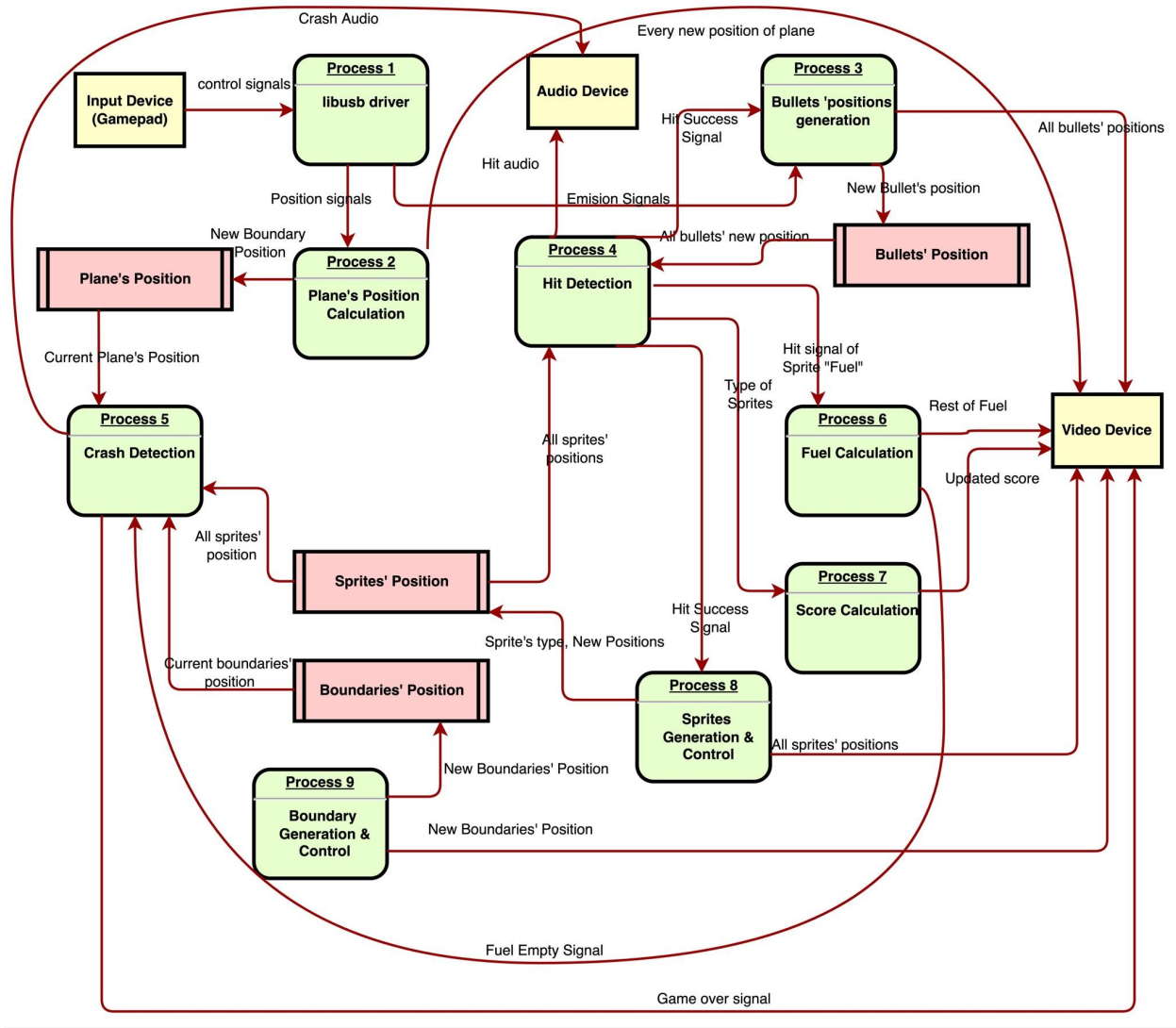


Figure 2-1 Data Flow Diagram of the Air-Raid project

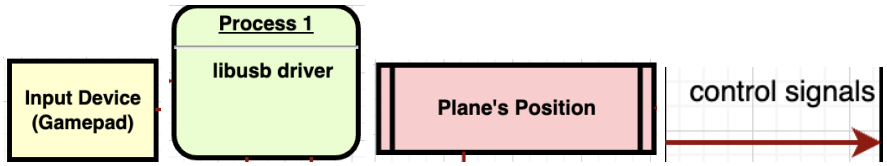


Figure 2-2 From left to right: External Entity, Process, Data Store, Data Flow

Process 1: Control Input & Driver

Users can control the horizontal position of the plane with a joystick. Users can also shoot bullets by pressing a button. We will use a USB device as the control device, so all the control signals will be transmitted through digital signals based on the “libusb” library in Linux. These USB devices will send

control signals to the drivers with interrupt signals, so we can make use of the “libusb_interrupt_transfer” function to receive signals in our own software system.

Process 2: Plane’s Position Calculation

After receiving the joystick input from the user, we need to calculate the plane’s position. The basic idea is that the signal from the joystick will be some digital number where the number gets larger as the joystick is pushed closer to the maximum joystick angle. We will consider this number as the speed the plane should change direction using some unit time (e.g. 0.01s) to derive the actual deviation of the horizontal position of the plane. The vertical position of the plane will not change at all during flying since all the background and sprites will change their vertical position down on the screen. During the process, this module should continuously send the position of the plane to the video device so they can display the plane on the screen.

Process 3: Bullets’ Position Generation

Bullets are the thing emitted from the plane when the user pushes the shoot button. When receiving such a signal, we should generate a new bullet from the current plane’s position, and let it move automatically towards the top of the screen. Also, when receiving the hit signal from the Hit Detection module, it means that there are some bullets hitting some sprites. We should then set the positions of those bullets and sprites out of the screen so that they disappear. This module should continuously send the position of all the bullets to the video device so they can display the bullets on the screen.

Process 4: Hit Detection

When shooting bullets, we need to determine if the bullets hit any sprites. We are going to implement this part by comparing the positions of the bullets relative to the sprites. This means that the module should read the position information stored for both the sprites and the bullets. If sprites are hit, this module will inform the Bullets’ Position Generation module and Sprites’ Generation & Control module so they can change the positions of corresponding objects. Also, it will send the play audio signal to the audio device to initiate the sound effect for the explosion .

Process 5: Crash Detection

In this game, the plane is easy to crash. If the plane hits the boundaries of the river, enemy sprites, or runs out of fuel, the plane crashes. Thus, every iteration we need to compare the plane's horizontal position with the current row’s boundaries and all sprites’ positions to see if the player has crashed or not. Meanwhile, the Fuel Calculation Module will send out a signal if the fuel has run out. If the plane crashes, the software will send a signal to hardware so that the video can display the “Game Over” signal. Also, it will send the audio signals to the audio device to generate the sound effect of crashing.

Process 6: Fuel Calculation:

When the player collides with the fuel tank sprite, the player will receive additional fuel as a reward. However, if the player shoots the fuel tank sprite, it will not gain any additional fuel. The current remaining fuel amount will be shown at the bottom of the game area, so the software will need to update the fuel amount given to the video device each round.

Process 7: Score Calculation:

The score for the game will increment whenever the player hits any sprites. Different types of targets will have different scores, for the enemy airplane which only has 1 HP, the score is 1 as well; while the battleship has 2 HP so its score is 2. The current score will be shown at the bottom of the game area, so everytime when the score is updated, the software will send a new integer to each score bit. Each bit will range from 0 to 9 and will control the sprite depicting that number in the score.

Process 8: Sprites Generation & Control:

There are three kinds of sprites in our game system: enemy airplanes, battleships and fuel tanks. Except for fuel tanks, all the targets should be shot while playing and have different score values. The generation and control of the sprites are done in software. During the process of the game, enemy airplanes and battleships will be generated randomly in the game area, depending on the difficulty level. The software will send all the positions of all sprites to the video device each round to update their location.

Process 9: Boundary Generation & Control:

For our game we are assuming the resolution our game is 640*480. In our game system, the game scenario (a.k.a. game background) is represented by 4 boundary variables. This allows for the river to have up to two branches at a time. There are 2 types of terrain in the game: ground and river. The boundary between the ground and river is represented by pairs of variables which is the horizontal coordinate. For example, variables with values 10, 100, 200 and 400 means: in that pixel row, the pixels located in 0-9, 101-199 and 401-639 are ground and in 10-100 and 200-400 are rivers. Also, variables with value 100, 100, 200 and 200 means: the pixels located in 0-99 and 201-639 are ground and in 100-200 is the river. If the third and fourth boundary variables are both set to 0, there will only be a single river branch. As time goes on, the airplane flies ahead, and the game scenario goes back (scroll down) at a rate of 60 pixel rows per second. The software will send all the variables of new boundaries to the video device each round to update the boundaries.

3. Interfaces & Objects:

Interface between SW and Video Device (explained further in the Hardware Section):

1. 4 registers for boundaries each row (with address 0x2 to 0x5)
2. All positions of different types of sprites
    a. Fuel Tanks (with addresses 0x20 to 0x 5F)
    b. Enemy Airplanes(with addresses 0x60 to 0x 9F)
    c. Battleships (with addresses 0xA0 to 0xDF)
3. All positions of bullets (coordinate x and y, with addresses 0x6 to 0x1F)
4. Positions of the plane (with address 0x0 0x1)

Object-Oriented Implementation

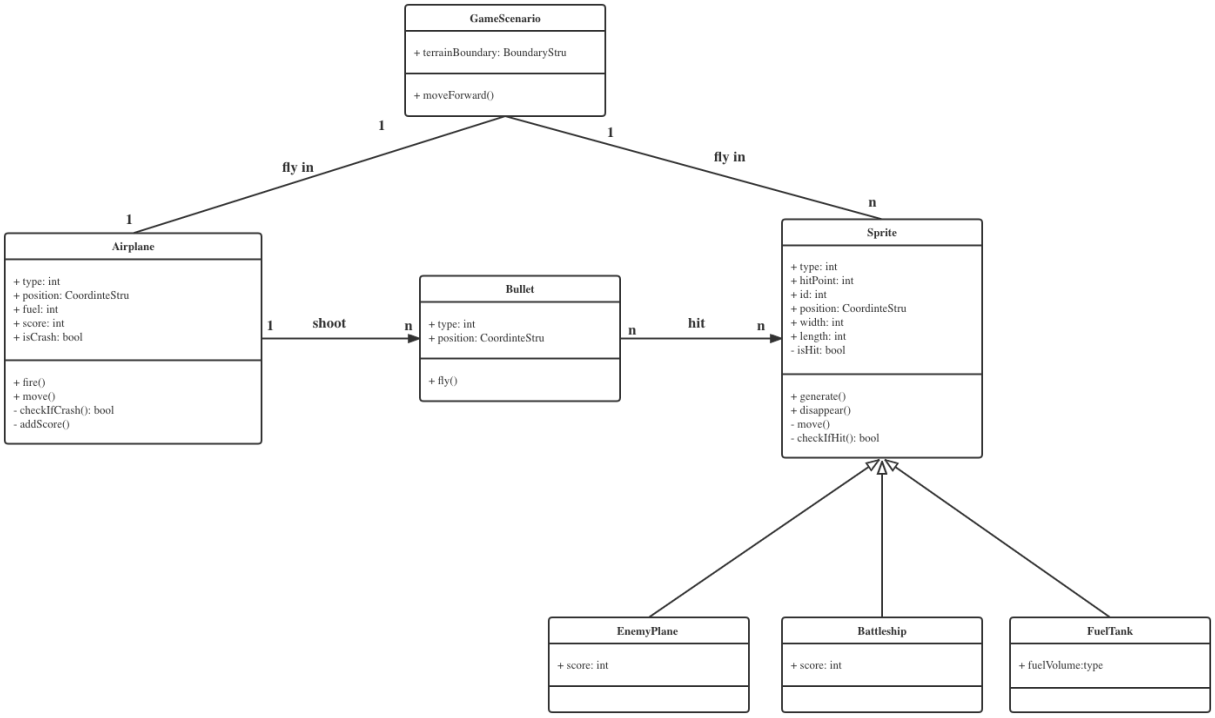


Figure 3-1 UML Class Diagram of River-Raid Project

In this section, we will fully discuss the system by using an object-oriented approach. The UML diagram is shown above in Figure 3-1, which illustrates the detailed design of the system. The whole system can be divided into several classes.

The GameScenairo class contains a main attribute terrainBoundray which represents the boundary of each terrain. The function moveForward() is designed to update the game scenario, making the game area scroll down. The relationship between this class and the Airplane class is an 1-to-1 relation which means there is only one player controlled airplane in the game. The relationship between this class and the Sprite class is a 1-to-n relationship, which means there are more than one sprite in the game.

The Airplane class is one of the most important classes, it is responsible for the player controlled airplane through the whole game. The attribute Type shows which type of the object it is, for example type 0 means it is the Airplane, type 1 means it is the Bullet and so on. The attribute Position is the coordinates of the current position of the plane in the game area. The attribute Fuel is the current fuel amount of the plane. The attribute Score is the current score the player gained in the game. The attribute isCrash is used to notify if the plane has crashed or not. The function fire() is triggered when the player wants to shoot the bullet. The function move() is used to control the airplane movement (left or right). checkIfCrash() is used to check the airplane to see if it crashed or not, and gives a bool return value. addScore() is called when the player fires a shot and hits the target, it will add the hitted target score to the current score. The airplane can shoot many bullets, so the relationship between Airplane and Bullet is a 1-to-n relationship.

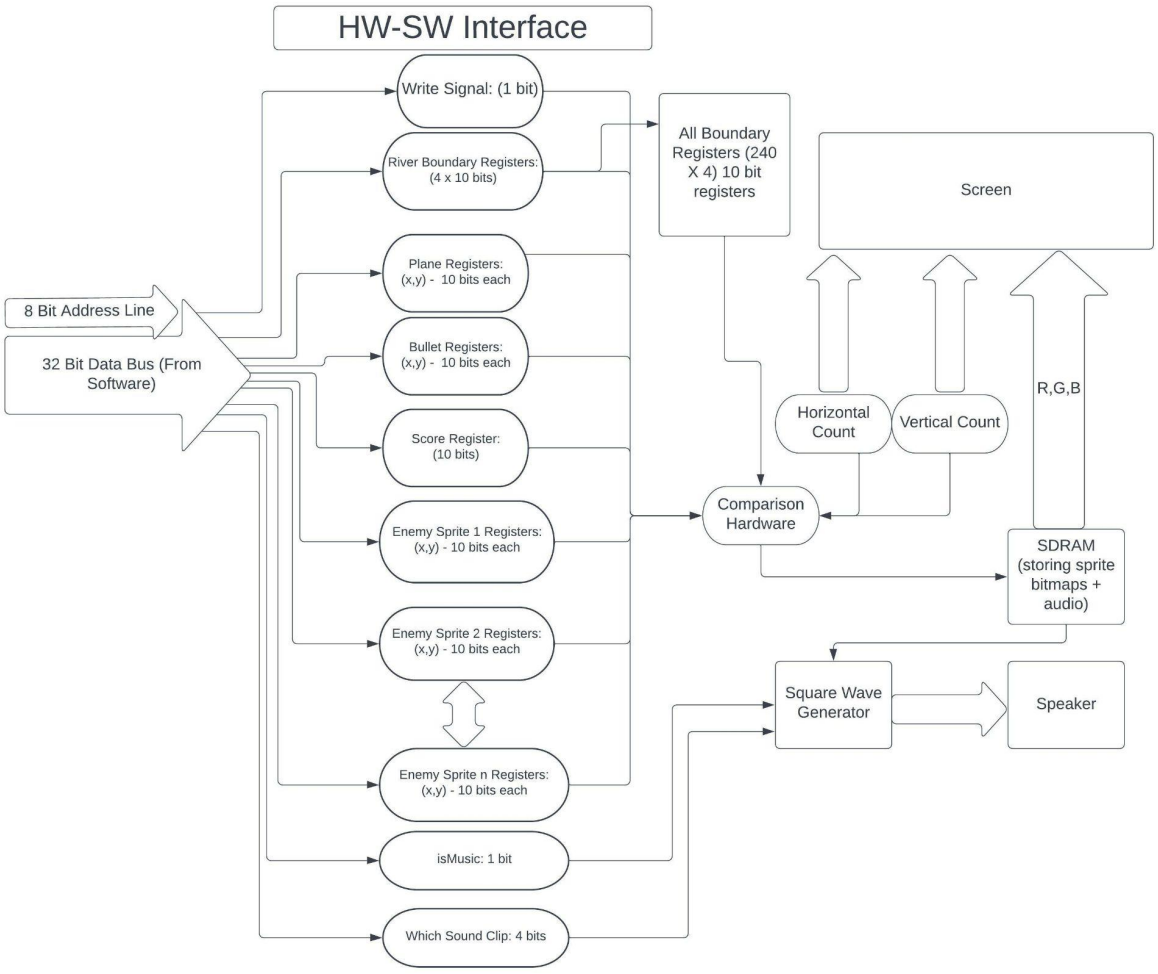
The Bullet class implements the bullet the airplane shot in the game. The attribute Type shows which type of the object it is, Bullet has type value of 1. The attribute Position is the coordinates of the current position of the bullet in the game area. The function fly() is used to set the position of the bullet, it provides the track of the bullet, at any given time the position can be obtained by using fly().

The Sprite class is a parent class which is inherited by 3 child classes. The attributes type show what object it is, type 2 means it is the EnemyPlane; type 3 means the Battleship and type 4 means the FuelTank. The attribute hitPoint shows how many hits it needs to be destroyed and get its corresponding rewarding score. Because there are more than one of each kind of sprite, we need attribute ID to identify each individual object. The usage of attribute position is the same with the others class. The attribute width and length represents the width and length of the pixel of each object. The attribute isHit is used to show if the object is hitted by the bullet or not. The function generate() and disappear() is used to generate the object randomly within the predefined proper region or make it vanish when the plane fly passes and the game scenario scrolls down. The function move() is the same as the counterpart function in airplane class. And the function checkIfHit() is used to check if the object is hitted or not and gives the bool return value.

In those three child classes, EnemyPlane, Battleship and FuelTank, they all have the attributes and functions in the parent class Sprite. Besides, the EnemyPlane and Battleship have attributes of score which means the rewarding score it has when destroyed by the player. The class FuelTank has the attribute of fuelVolume which is the amount of fuel that should be added if the plane hits the Fuel Tank sprite.

4. Hardware

a. Hardware Diagrams



The hardware will handle displaying information from software on the VGA monitor, and playing audio; this will be accomplished by stages of cascaded hardware and interconnections. To start, we will have a 32 bit data bus line for which the software can write to the hardware. This 32 bit bus will be wired to registers that control the positions of sprites and other control signals needed by the hardware (for instance, whether music is playing or where the rivers boundary is of the most recent row). The register that software intends to write to will be indicated by an 8 bit address line, with an extra signal for write-enable (as per Lab 3).

The hardware will access data from these registers as it writes to the VGA monitor. It will sweep through horizontally and vertically all 640 x 480 pixels; if it gets within range of any sprite (it will check all sprites for each pixel) it will start drawing it. The pixel information for each sprite will be stored in SDRAM and will therefore need to be accessed whenever it is being drawn. The hardware will also draw the map using information sent from software. The map itself is rather simple - just a river overlaid on top

of a green background. We decided to therefore generate the map in software by deciding where the boundaries of the river are for each row, and sending that data over to hardware. These boundaries will be stored in 4 registers per each 2 vertical rows. The hardware will do a simple comparison of the three values with *Horizontal Count* and decide whether the current pixel is blue or green. The resolution of each row will be 2 vertical pixels. This means that the pixel comparison result will be used for that row and the row below it. The boundaries for every row will also remain in memory so that the background can be constantly generated. This will require 240 sets of 4 registers.

Finally, the hardware will transmit preloaded audio information to an external speaker on command from the software. Audio clips will be preloaded into memory, and the software will select which of the clips to play. Upon selection, the hardware will read values from memory (which will represent frequencies to play for certain periods of time) and generate this signal as a square wave.

b. Register Descriptions

Write Signal: 1 bit integer storing whether new data is being written to the registers

River Boundary Left: 10 bit integer storing the left boundary of the river
River Boundary Right: 10 bit integer storing the right boundary of the river
River Boundary Left2: 10 bit integer storing the left boundary of the river
River Boundary Right2: 10 bit integer storing the right boundary of the river
As a note: there will be 240 sets of these 4 registers. This allows each set of 4 registers to control the boundaries for 2 vertical pixels.

Plane Position x: 10 bit integer storing the x position of the plane
Plane Position y: 10 bit integer storing the y position of the plane

Bullet 1 Position x: 10 bit integer storing the x position of the plane's bullet
Bullet 1 Position y: 10 bit integer storing the y position of the plane's bullet
…
Bullet 5 Position x: 10 bit integer storing the x position of the plane's bullet
Bullet 5 Position y: 10 bit integer storing the y position of the plane's bullet

Score Digit 1: 4 bits controlling the value of digit 1 of the score
Score Digit 2: 4 bits controlling the value of digit 2 of the score
Score Digit 3: 4 bits controlling the value of digit 3 of the score
Score Digit 4: 4 bits controlling the value of digit 4 of the score
Score Digit 5: 4 bits controlling the value of digit 5 of the score

Enemy Sprite 1 Position x: 10 bit integer storing the x position
Enemy Sprite 1 Position y: 10 bit integer storing the y position
...
Enemy Sprite 64 Position x: 10 bit integer storing the x position of the plane
Enemy Sprite 64 Position y: 10 bit integer storing the x position of the plane

isMusic Boolean: 1 bit signal to indicate whether music should be played
Which Sound Clip: 2 bit signal to indicate which music clip should be played (shoot, collide, explode)

5. Memory Budget

Graphics Memory

The following graphics are not finalized yet and are subject to change before the final project.

Category	**Size (pixels)**	**# of images**	**Total Size (bits)**
Plane	40*40	1	38,400
Battleship	40*80	2	153,600
Enemy Airplane	40*40	1	38,400
Score Board	100*20	1	48,000
Number	20*20	10	96,000
Fuel Icon	20*15	1	7,200
Bullet	2x10	1	480
Explosion	40*60	1	57600
Fuel	40x40	1	38,400
Game Over	20x80	2	76,800

Total Memory Budget (bits): 554,880

Audio Memory

The following audio sounds are not finalized yet and are subject to change before the final project.

	Shoot	**Collision**	**Explosion**
Time (s)	0.5	0.5	0.5

F (KHz)	8	8	8
memory(bit)	4,000*16	4,000*16	4,000*16

Total Audio Budget (bits): 192,000

6. Milestones

1. Hardware: implement reading and writing from memory, Software: finish half of the submodules and the unit tests
2. Hardware: implement background generation and VGA, Software: finish all of the submodules and the unit tests
3. Hardware: implement sprites, Software: implementation of hardware simulation (testing oriented, should be as easy as possible) and put all modules together
4. Hardware: implement joystick, Software: verifying the correctness of game logic and add more logics if possible
5. Ensure hardware and software work properly
6. Bug Fixing and stretch goals