CS4303P4 Report

——Mini Thunder: a casual shoot 'em up game

200009419

Introduction

The game design to be a horizontally scrolling flying and shooting game. The player controls a fighter plane to shoot the enemy fighter down as well as parry the incoming crashing and bullet. The difficulties of the stage will increase as the player gains more score from take down enemies. It reflects on faster moving speed for enemies, faster refreshing, higher hp and so on. The player could upgrade the fighter or restore the hp by collecting supply air drops. A concise and functional user interface is implemented for navigation and status indicating.

Overview & Game Context

The game genre is largely inspired by shoot 'em up games, combined with the techniques of side-scrolling game and absorbed some elements from bullet hell games.

A shoot 'em up game is a game in which the protagonist combats a large number of enemies by shooting at them while dodging their fire (B, Matt). These games are usually viewed from a top-down or side-view perspective, and players must use ranged weapons to take action at a distance. The player's avatar is typically a vehicle or spacecraft under constant attack. Thus, the player's goal is to shoot as quickly as possible at anything that moves or threatens them to reach the end of the level with a boss battle (R, Andrew). ***Space Invaders*** is considered the game that set the template for the shoot 'em up genre. It is a fixed shooter in which the player controls a laser cannon by moving it horizontally across the bottom of the screen and firing at descending aliens.

A side-scrolling game or side-scroller is a video game in which the gameplay action is viewed from a side-view camera angle, and as the player's character moves left or right, the screen scrolls with them. These games make use of scrolling computer display technology. The move from single-screen or flip-screen graphics to scrolling graphics, during the golden age of video arcade games and during third-generation consoles, would prove to be a pivotal leap in game design.

The attributes of horizontally scrolling games are largely established by game ***Defender*** developed by Eugene Jarvis in 1981. The game set on the surface of an unnamed planet. The player controls a spaceship flying either to the left or right. A joystick controls the ship's elevation, and five buttons control its horizontal direction and weapons. The surface will change dynamically according to the movement of the player.

The design of the game is influenced by game ***Raiden***, a 1990 vertically scrolling shooter arcade video game developed by Seibu Kaihatsu. In each stage, the player manoeuvres the Fighting Thunder craft, engaging various enemies and avoiding their attacks. After completing the eighth and final stage, the player returns to the first stage with the difficulty increased. It provides variety of elements for a shooting game, dazzling bullet, crazy barrage, collectable items,

different forms of enemies, and so on. A plenty of game elements in *Raiden* are applied to the designed game which will be demonstrated in design section.

Design

Title

The title “Mini Thunder” derived from the title of the game *Raiden*— a Japanese word literally means “Thunder” in English. The title suggests some relations between these two games and “Mini” suggests that the game is designed to be concise.

Genre

The game designed to be a horizontally scrolling flying and shooting game. The main feature including a horizontally scrolling stage, constantly rendered enemies, fast-paced combat, collectible items. The game has a common impression of a "shmup" game: side-view perspective, an overwhelming number of enemies and bullets, ranged weapons, and an avatar of aircraft. Moreover, the game is designed to be a casual game: simple game play that is easy to understand; a concise interface designed to simplify the operation; short game sessions that enables the users to play the game in a short time period.

Characters & Items

There are three characters in the game—Player Fighter, Enemy Fighter, and Enemy Flying Fortress. Player Fighter is controllable by the player. It has three attributions: HP, damage, and maneuverability. HP decides how many damages the player can take or how many mistakes can be made before the game over. Damage determines how many bullets fired could destroy an enemy aircraft: a larger damage number means less bullets to take down an enemy aircraft. Maneuverability determines how quick the aircraft could move to the location of the cursor under easing function (will be discussed in detail in implementation section).
The enemy fighter will be generated in random position on the right of the stage. It cannot fire, but could crash into the player fighter. It has two attribution: speed and HP. Speed controls how fast it moves across the stage while HP determines how many bullets it can sustain. The enemy flying fortress has more HP and slower speed than fighter, and it can fire bullets at the player according to the player location.

Player Fighter

Enemy Fight

Enemy Flying Fortress

There are three collectable items in the game: airdrop box for HP recovery (the red one), damage upgrading (the blue one), and maneuverability upgrade (the green one). A random airdrop will be added to the game in certain frames. The player could move the plane to collect the box for

upgrading or HP restoration.

Airdrop-maneuverability

Airdrop-HP

Airdrop-Damage

Rules & Control

To control the player fighter, the player could simply move the mouse cursor on the screen and the plane will follow the motion of the cursor. To fire bullets, the player could press and hold the mouse left button. A visual game guide is given as figure1.

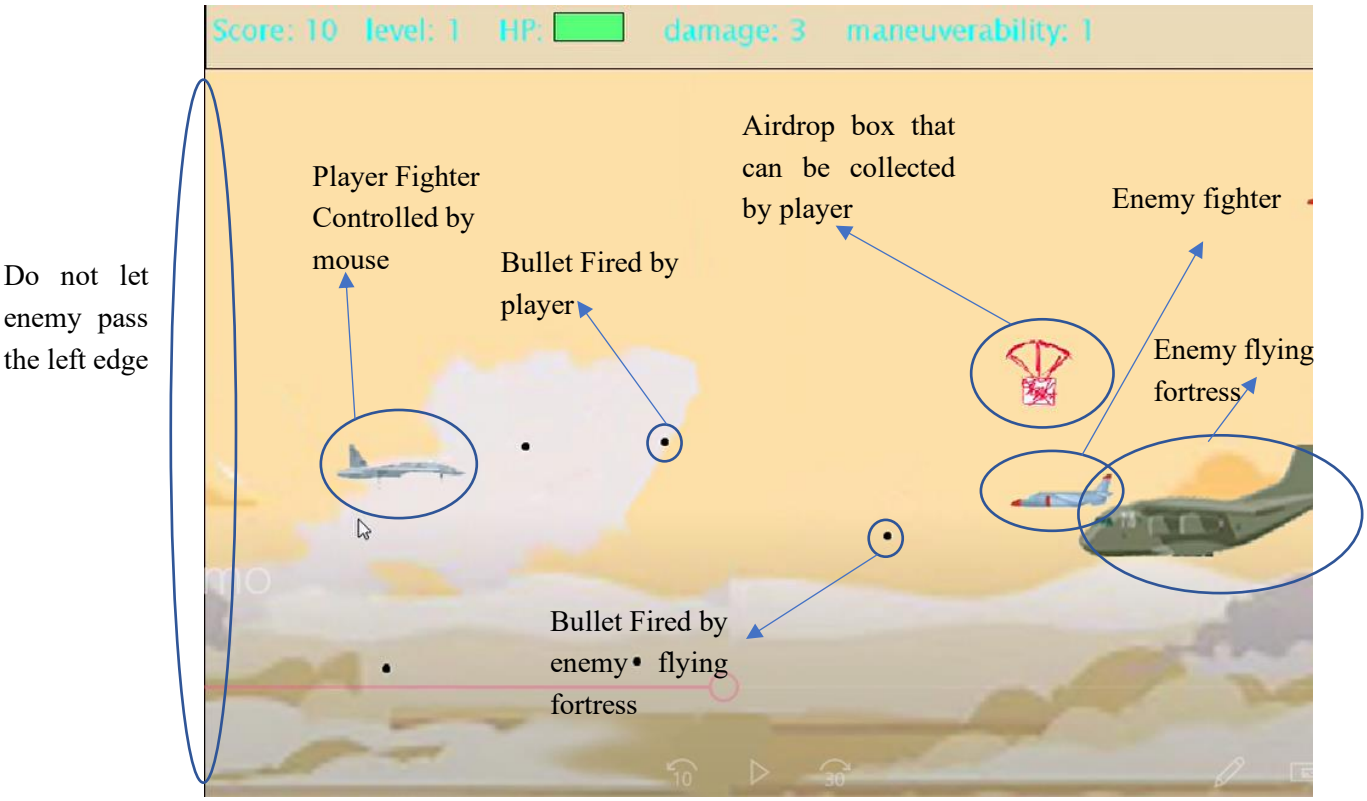

Figure1: Game Screenshot

The rules for the game are straightforward. There is an HP bar on the top panel indicating the current HP left. Once the player fighter crashed into an enemy aircraft, hit by an enemy bullet, or let the enemy aircraft pass the left edge of the stage, the HP will drop and reflected on the HP bar. The initial value of HP is 10, which means the player could sustain above cases ten times before the game ends. The enemy HP will drop equals to the value of the damage each time it hit by a player bullet and the enemy aircraft will be destroyed if the HP is below zero. A crashing will destroy the enemy instantly no matter what value of the HP is. Once the player destroys an enemy aircraft, the game score will be incremented by 1.

The current level is dependent on current score, the difficulties of the game will increase according to the level. The difficulties of the game are reflected on enemy HP, enemy speed

and the frequency spawning an enemy. As the level increases, the enemy speed and frequency of spawning will reach a maximum number — to make the game reasonable. However, increments of enemy HP do not have a limitation.

To handle increasing difficulties, the player must collect airdrops to upgrade the damage level and maneuverability level. In every 150 frames, there will be 20% chance to drop a supply box among HP, damage upgrade, and maneuverability upgrade. The player could collect the box by moving the fighter to the position of the box. The red box will restore the player HP to 10. The blue box could increase the damage of player bullets—make it quicker to shoot down enemy aircraft. The green box could increase the swiftness of player fighter—decrease the time delay moving to the position of mouse cursor.

Stage & Goals

The game set on a modern warfare air battle. The player controls a fighter aircraft to destroy incoming enemy air forces. The goal is to gain a higher score by destroying more enemy aircraft and surviving from overwhelming waves of enemy and dense barrage. Instead of creating stages and checkpoints, the game designed to be "infinite" that could be played continuously theoretically.

The game designed to be a casual game. It intended to bring relaxation and fun during the work break. The game control and rules are concise, however, the game can be challenging after level 7. To survive and reach a higher score, the player has to swiftly move the fighter to a correct position in order to destroy an enemy or parry enemy bullet or crash. Additionally, the player needs to neutralize the air threat quickly enough before it reaches the left edge.

The difficulties of the stage will increase as the player score increases. The stage level equals one-tenth of the score, each level increment will decrease the time interval spawning enemy aircraft, increase the enemy speed, and HP. To handle a higher level of battle, the player needs to pick airdrops for upgrading. As the initializing of airdrops relies on a random number generator, luck is an important factor to reach a higher score. This is intended in game design to increase replayability.

User Interface

A concise user interface is designed for navigation. To simplify the operation, buttons are made so that the user could access a page by simply clicking the button without remembering any keyboard input. On main menu (Figure2), the player could start a game by clicking the start button or clicking the control button to watch the page demonstrating the rules (Figure3). When game over, the player will enter game result page showing the score for last game (Figure 4). The player could click the "return to menu" button to go back to main menu.

During the game play, a head-up display is shown on the top of the screen (Figure1). The transparent panel presents the current game information including current score, level, damage, HP bar, and maneuverability.

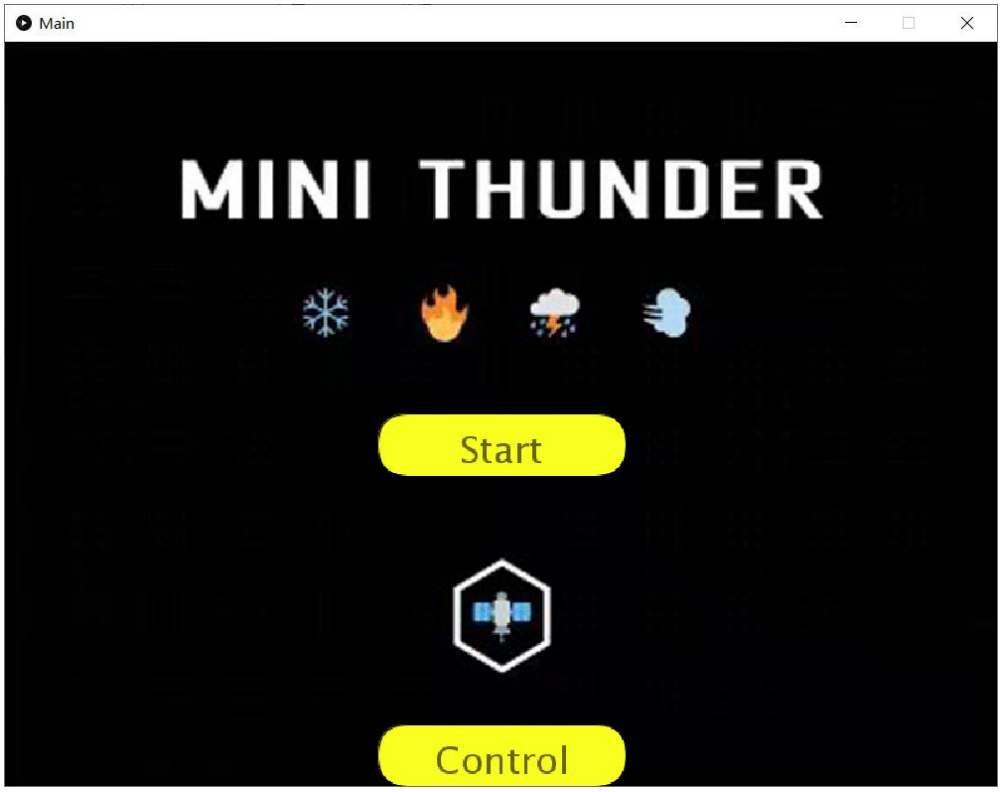

Figure2: Main menu

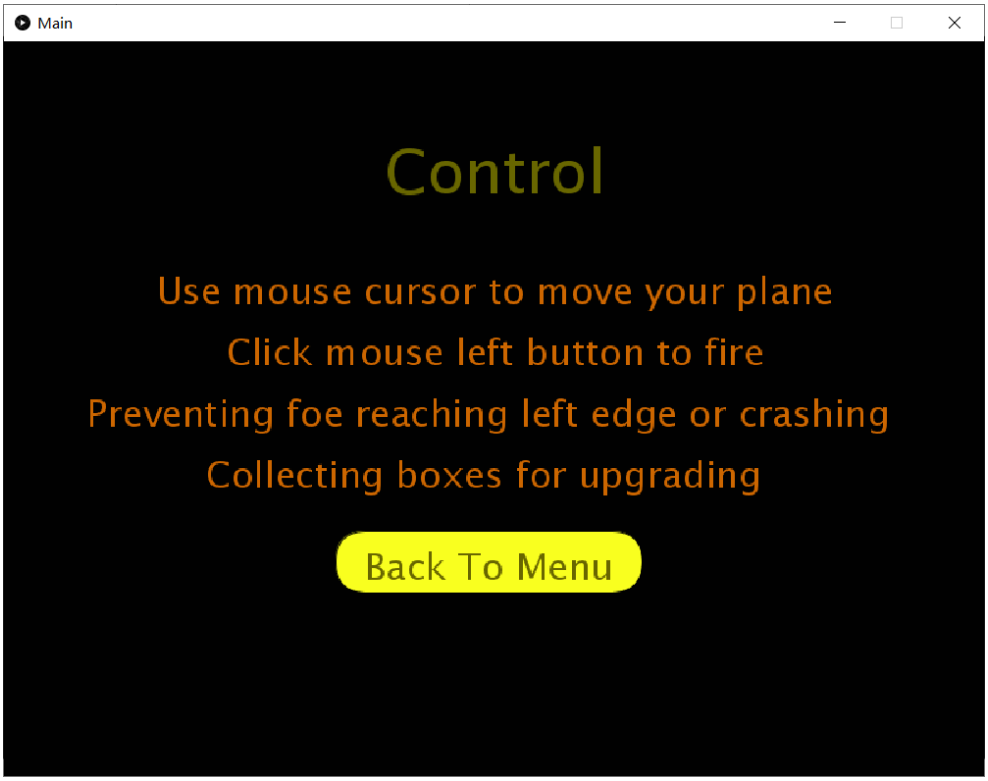

Figure3: Control introduction page

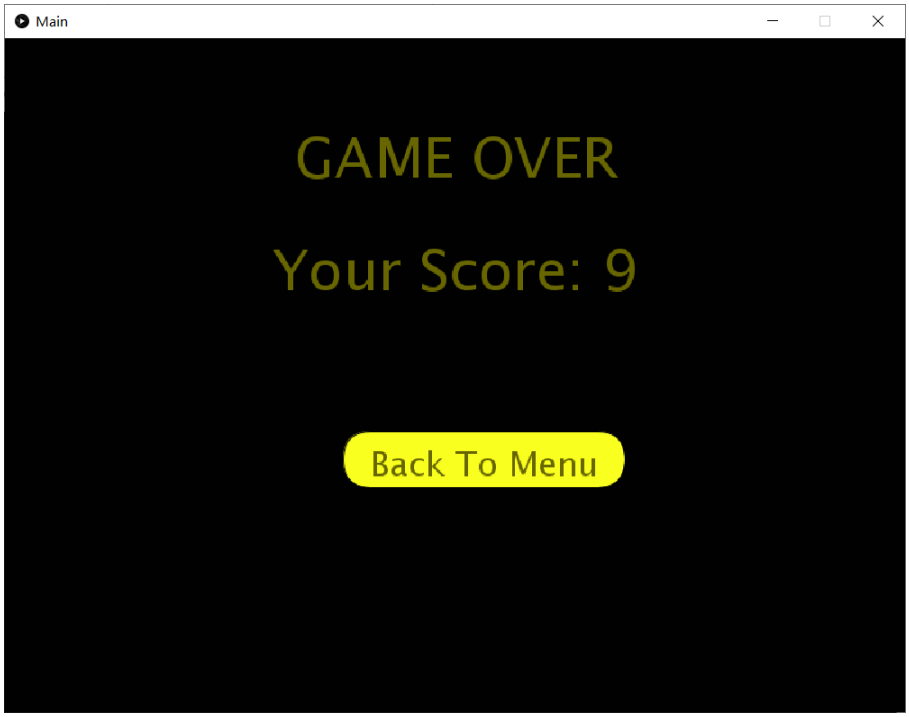

Figure4: Game result page

Implementation

Game Engine

To improve reusability and make the code more structured, before build the game, a game engine is developed to provide basic functions and components for the game. It intended to enhance the clarity, and quality of the code. The game engine follows the design of Model-View-Controller pattern. The game comprises game objects and UI components which are treated by separate controllers.

Game objects assign attributes and behaviours for items shown on the stage e.g., player fighter, bullet. Each of these objects have a RigidBody to perform physical motion such as move and collide. A RigidBody contains the information for physics transitions, a collide box, and a texture for visualising. Stage acts as a container to hold RigidBody, it controls the drawing of texture and handles collisions. The GameController determines the process of the game play: initialising of game objects, difficulty of game play, data updating, and garbage collection.

Canvas is an independent container for visualising that acts as a container to hold UI components. It will not interact with game objects directly, which provides low coupling for the programme. It is designed to display UI in a structured manner, and ensures the scene can be switched for different states without interruption e.g., display a component does not belong to current scene. State controller manage the switching of game states, it interacts with the button handler when a button event is performed.

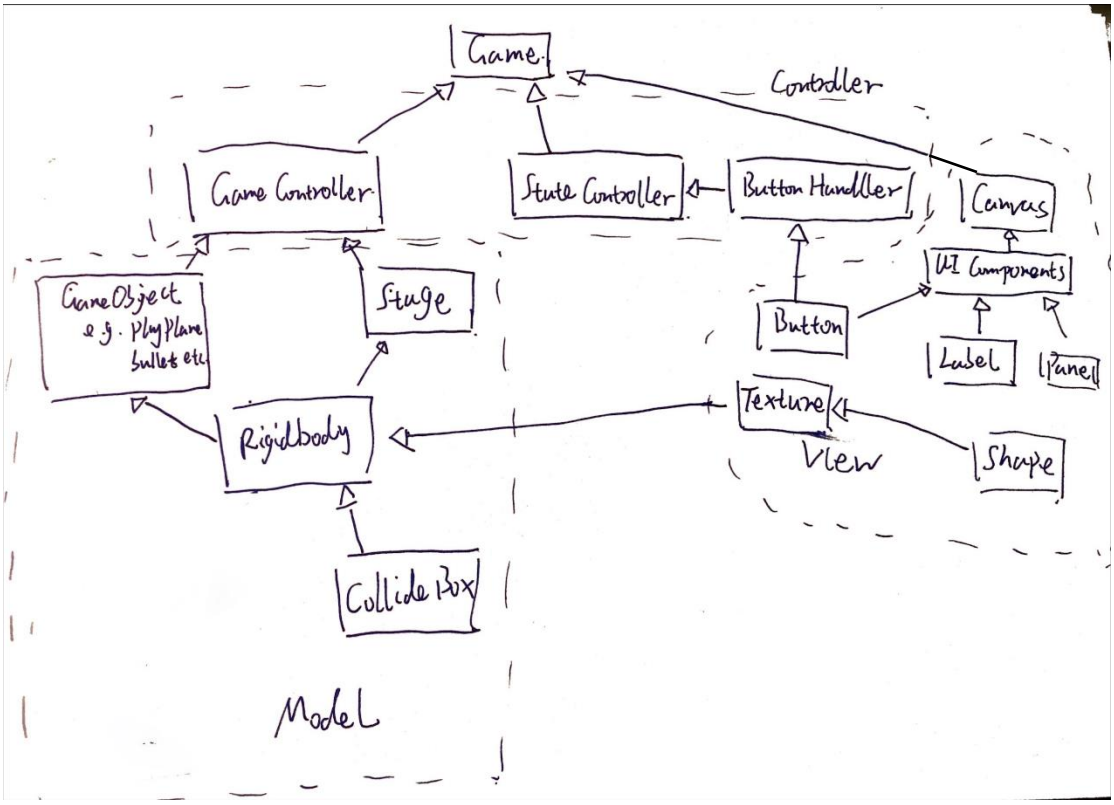

Figure5: Architecture of Game Engine

Physics

Scrolling

To make the background scrolling thus to create a view that the plane is moving forward, the background is a combination of two similar images. The background has a fixed speed moving towards left, once it reaches its right edge, it will be recovered to its initial position (because two images are similar, the player cannot tell where they actually are). Hence, creating an illusion that the player aircraft is constantly moving forward.

Motion

RigidBody class provides common physics behaviour: speed, acceleration, gravity, rotation, and collision. Speed is represented by position changes in the 2D space during a fixed time interval. Acceleration stands for speed changes within a fixed time instead. In processing, each draw call is treated as a time unit, for physics status are updated in each draw call.

In the game play, player fighter will “chase” the mouse cursor to create an easing effect. The following code enables the effect by assigning dynamic speed according to the position of aircraft and mouse position. When the distance between the aircraft and the mouse cursor is large, the speed will be large, vice versa. Moreover, the multiplier can be modified, and thus could control the overall maneuverability for the aircraft: a larger multiplier means a shorter moving time and more swiftness.

```
pf.r.setVelocity(new PVector( x: maneuverability * maneuverLevel * (mouseX - posX),  y: maneuverability * maneuverLevel * (mouseY - posY)));
```

Code1: Player motion

Collision

Collision is specified by collide box and handled by Stage class. It uses a simple algorithm to detect collision: once the distance between centres of the two collide boxes is smaller than the sum of widths or hights in both X and Y direction, a collision is detected. Notably, the width or hights stand for the length from the centre point to the edge for convenience.

To handle collision, the container will traverse all of the collide box in each draw call. It will identify the rigidbody responsible for the collision and call OnCollision method taking the opposite one as a parameter. Hence, each game object could identify which object it collides into.

```
public void handleCollision() {
    for (RigidBody r : container) {
        for (ColideBox c : colliderContainer) {
            if (c.getR() != r) {
                for (ColideBox ColliderForR : r.getColideBox()) {
                    if (Math.abs(ColliderForR.getX() - c.getX()) <= Math.abs(ColliderForR.getWidth() + c.getWidth())
                            &&Math.abs(ColliderForR.getY() - c.getY()) <= Math.abs(ColliderForR.getHeight() + c.getHeight())) {
                        r.onCollision(c.getR());
                        System.out.println("collidion: "+r.getTag());
                        ColliderForR.setColision(true);
                    }
                }
            }
        }
    }
}
```

Code2: Collision handler

Game Logic

Game logic is handled by GameController class. It controls the generation, transition, and destruction of game objects. Meanwhile, it modifies the stage to ensure objects are correctly displayed on the screen. A game object has a series of “intrinsic” attributions that are independent of the controller. For instance, when a particular collision is detected, a Bullet will destroy itself—removing the rigidbody from the stage. It is implemented by using Anonymous Inner Class to assign customised behaviour for collision events. The following code shows that, for Bullet fired by player, once a collision is detected, and the tag of the object it collides into equals “foe”, then it will explode.

```
r=new RigidBody(t){
    @Override
    public void onCollision(RigidBody r) {
        if(r.getTag().equals("foe")){
        explode();}
    }
};
```

Code3: Customised OnCollision method

The GameController class focuses on the initializing of game objects. It has a timer to record the current frame number. The idea to prevent using system time is that the frame per second of the game is not a constant value, thus, may cause an unstable update. The code below describes how enemy aircrafts are initialised with time going. In a frame interval, an enemy aircraft will be spawned on a random Y value on the right edge of the screen. The level variable determines the difficulty of the game. At the beginning of game, an enemy fighter will be spawned in every 50 frames. In late game, it will be spawned in every 20 frames. When an enemy aircraft is spawned, it has a 1/10 chance to be a boss—flying fortress.

```
//adding enemy to the stage randomly
int wave = (50 - 2 * level) < 20 ? 20 : (50 - 2 * level);
if (timmer % wave == 0) {
    if(Math.random()<0.1){
        ft = new FoeFighter( x: 800,  y: 100 + (int) (Math.random() * 400),  (3 + level) > 10 ? 10 : (3 + level),  hp: 2 + level / 2,  boss: true);
    }
    else{
    ft = new FoeFighter( x: 800,  y: 100 + (int) (Math.random() * 400),  (3 + level) > 10 ? 10 : (3 + level),  hp: 2 + level / 2);}
    ft.damageTaken=damageLevel;
    foeFighterArr.add(ft);
    stage.addRigidBody(ft.r);
```

Code4: Enemy spawning

A flying fortress can fire bullets targeting the player, the flying direction of the bullets depend on the firing position and current player position. A bullet will be “fired” in every 20 frames on each existing boss. Firstly, it obtains correct angle between the enemy and player in radius by calling Math.atan2 method provided by java. Then, the angle is transformed into degree form, and a bullet is initialised with given degree.

```
//boss can fire at player
if(timmer%20==0){
for (FoeFighter f : foeFighterArr){
    if(f.boss){
        double rad=  Math.atan2(f.r.getY()-pf.r.getY(),pf.r.getX()-f.r.getX());
        int angle= (int)Math.toDegrees(rad);
        System.out.println(angle);
        FoeBullet fb=new FoeBullet(f.r.getX(), y: f.r.getY()+50,angle);
        foeBulletArr.add(fb);
        stage.addRigidBody(fb.r);
    }
}}
```

Code5: Bullet targeting player

User Interface

State

Game state is controlled by StateController. Current state number decide what to be displayed on the screen. The number can be changed due to a mouse clicking or end game and thus leads to another scene.

HUD

HUD is independent of GameController, it acts like a canvas. However, it could obtain updated information from GameController e.g., HP, level and display them on a transparent panel.

Button

When the mouse cursor moved into the area of button, the button will be transferred into “hang” status and highlighted. A button works only under designated game state. If mouse left is pressed, a state transform will be performed according to the state transition number specified in individual Button. As an example, if a button has a state transition number of 1 (game), if it is pressed, the game will start and current state number will become 1. Any customised code block for a button will be called in trigger() method.

```
for(Button b:buttonArr){
    if(b.x<=mouseX&&mouseX<=(b.x+b.width)
    &&b.y<=mouseY&&mouseY<=(b.y+b.height)
    &&stateController.getCurrentState()==b.getCurrState()){
        b.setHang(true);
        if(p.mousePressed&&p.mouseButton==37){
            stateController.setCurrentState(b.stateTrans);
            b.trigger();
        }
    }
    else {
        b.setHang(false);
    }
```

Code6: Hang and Click event for button

Test & Evaluation

Before the application been put into usage, it should pass the unit test, cluster test and system test. The series of test will ensure the system works functionally without any detectable bug.
Then, a self-evaluation process starts, followed by third-party evaluation. For evaluation, the overall achievement of the project is measured by functional point. Functional point can provide an instant and quantised evaluation for each section in the project. The points are weighted according to the significance and general effort needed of each component in the entire system. The table below demonstrated the point each section could achieved.

Control		**Game Logic**		**UI**		**Total**
point	score	point	score	point	score	
5	5	10	9	5	3	17/20

Table1: Self Evaluation

To obtain feedback from the public, questionnaires are designed to collect the general opinion from the users. The users could give a score between 1 and 5 (5 means best, 1 means worst) to measure the achievement and user experience of the programme. In addition, the users could report any bug and shortcoming in the form. All the testers should be treated anonymously with no personal data to be collected.

Question	Score (1-5)
What do you think of the Player Control?	5
What do you think of the Game rule and mechanics?	5
How do you like the user interface design?	4
Is the game easy or hard (1 for easiest)	3
What is your overall impression to the game	5

Table2: Third Party Evaluation

References

B, Matt (1990), *The Complete YS Guide to Shoot 'Em Ups*, Your Sinclair, July, 1990 (issue 55), p. 33

R, Andrew & A, Ernest (2006). *Fundamentals of Game Design*. Prentice Hall.