ENEE4901 Project II Report: Space Shooting

Team Members: Huafeng Shi (hs2917), Mingyang Zheng (mz2594),
Lingqing Xu (lx2201), Xiaoyang Liu (xl2427)

Section I. Space Shooting: A Brief Description

Space shooting games are a kind of action games, which test the player's speed and reaction time. The purpose of a shooter game is to shoot opponents and proceed through missions without the player's character being killed or dying.

We implement an architecture consists of a server and multiple users, and allow users to compete in the space shooting game. More specifically, we implement a console game of space shooting and it scores the user's performance, and the server is in charge of user management and overall ranking. Under this architecture, users can compete with others on

We implement a shooter game on the Android platform, and allow users to compete with each other. Furthermore, this server-based multiple user structure also support collaborative gaming.

Section II. Development and Test Environment

Our development environment is as follows:

build:gradle:2.3.0

buildToolsVersion "25.0.2"

support:appcompat-v7:25.3.1

Test environmentis as follows:

Real Phone: Huawei Honor 5X; System: EMUI 4.0; Android version: 6.0.1 Android Virtual Device (AVD): Nexus S Android 6.0 API 23 Platform

Section III. Overview

Our system consists of five major modules: Login (MainActivity), Sign Up (register Activity), Start (GameActivity), MyHistoryActivity, and Ranking History.

- For the GameActivity, it includes four attributes, namely, Aircraft, Animation, bullet, and Enemy.
- For the MyHistoryActivity, it includes three attributes, namely, username, score, and date.
- For the RankingHistory, it includes three attributes, namely, usernames, score, and date.

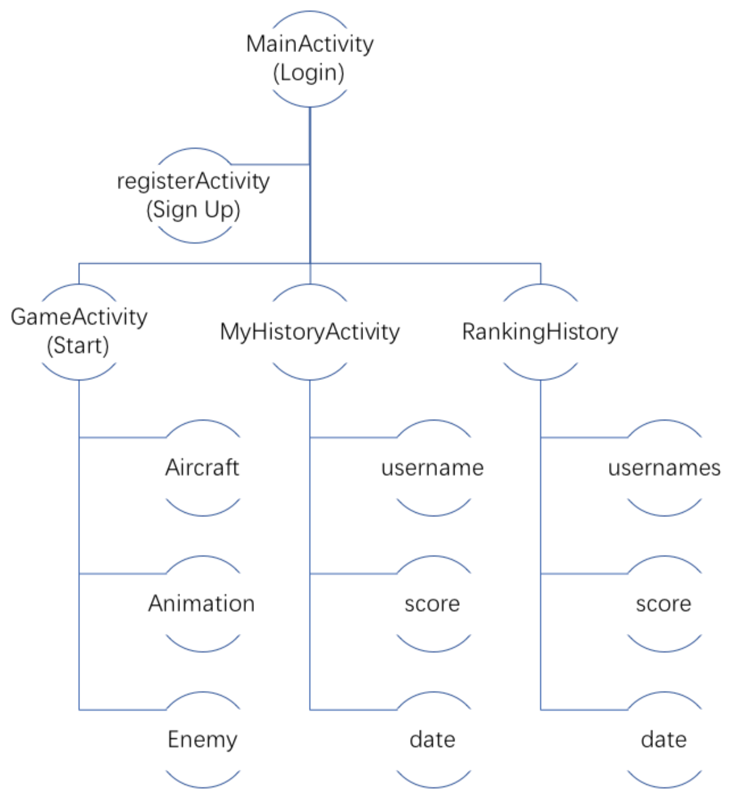


Fig. 1 Overall Framework.

The Login module consists of two major parts: Sign Up, and Sign In.

- For the Sign Up, it invokes the register activity. Note that two cases may happen after execution, namely, registration success, and registration failure with a reason.

- For the Sign In, it includes three actions, namely, get_username_password, post_to_server, and save_to_shared_preference. .Similarly to the register activity, two cases may happen after post_to_sever, namely, sign_in success, and sign_in failure with a reason. Moreover, save_to_shared_preference allows for further use.

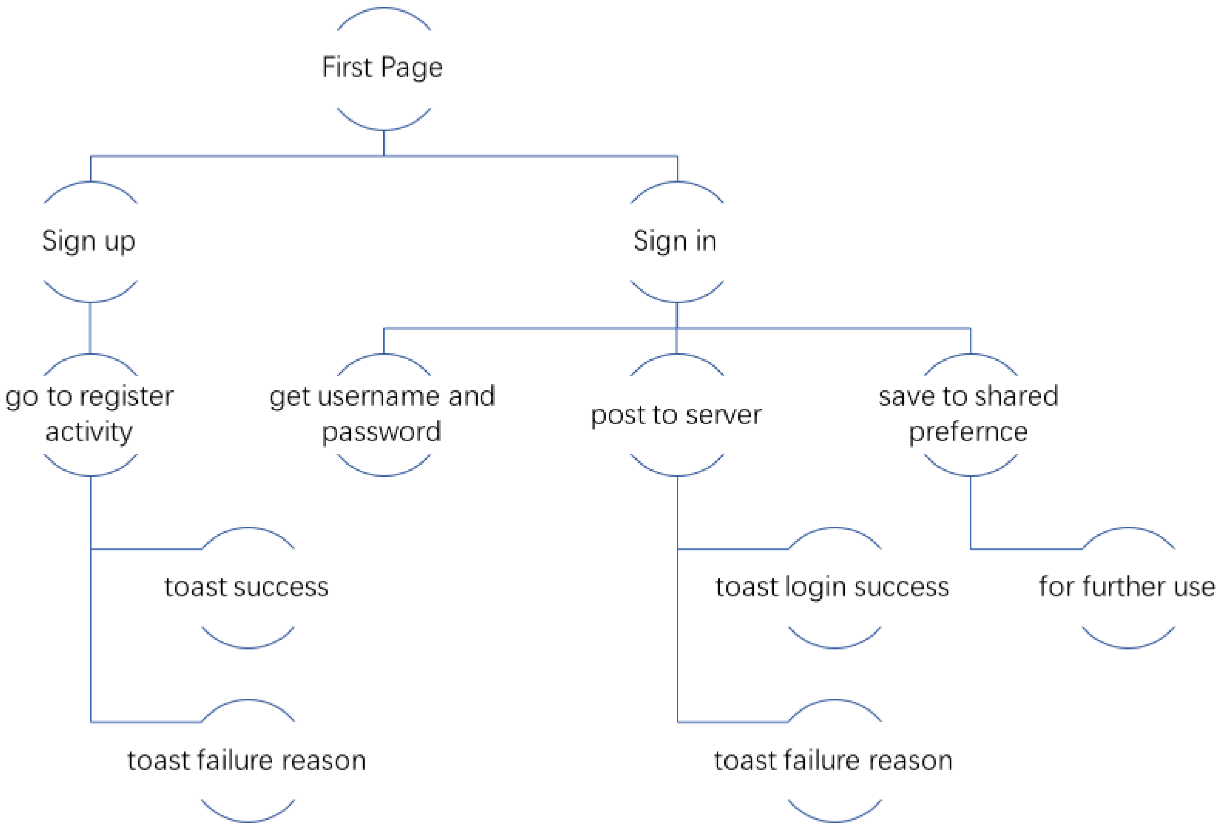
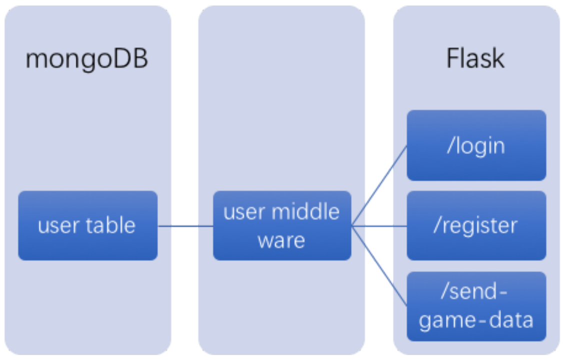


Fig. 2 The flowchat of Sign-up and Sign-in.



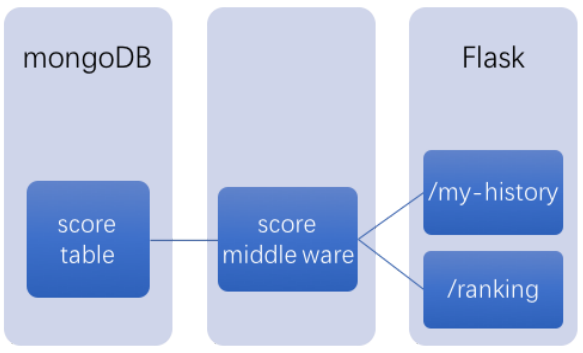


Fig. 3 The flowchat of the server: login, register, and game-data.

Fig. 4 The flowchat of the server: individual history and ranking

Section IV. Modules: Detailed Description

Subseciton A. Sever's Functions: Detailed Description

The server is hosted on AWS EC2. It communicates with app by http request and response, which follows the REST api convention.

It contains five interfaces: login, register, send game score, get recent game record and global ranking. App post or get to corresponding path to send or retrieve data. Data will be saved to User collection and Game collection in MongoDB.

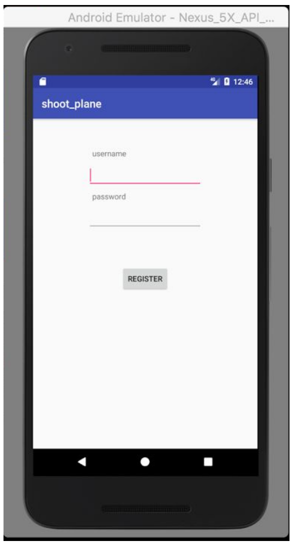
The http request is first handled by Flask, a python microframework. If app post some data, flask will resolve the data and warp it to python dict for future use. Then the middleware will try to save that data to corresponding data table in MongoDB. The success operation or error reason will be packeted to JSON format and then send back to app by flask. If the app want to look up some resources, for example, the recent game history. The middleware will search the desired resources in mongoDB and then return them in JSON format to the app.

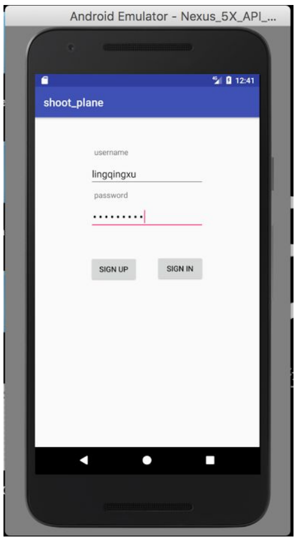
For stability of server, the unit test cases is created for the server by Python unittest package. Every deployment after modification the code of server will first run test cases to ensure every function is well behavioried.

Subseciton B: Sign In (LogIn), Sign Up (Registration)

Name and Password: R.id.username, R.id.password, Http Post

In registration, If the username has been registered before, the server will return a hint: duplicate username. If successfully sign up the game, there will be a toast says: go back to sign in. In login activity, if what the user entered do not match the data of database table in server, it will return a sign in failure because of wrong username and password. If successfully sign in the game, there will be a toast says: Enjoy the adventure in the activity that displays the start button to play the game, the My History button to check the history of scores of this user, and the Ranking button to check the rank of this specific user in all users who have played this game by the app we developed before.





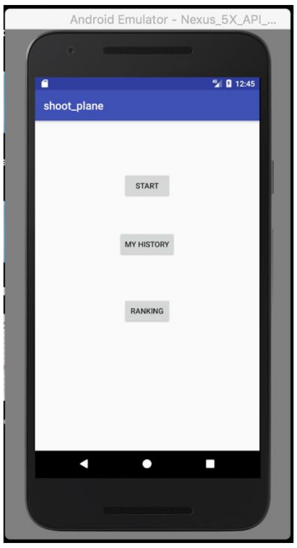
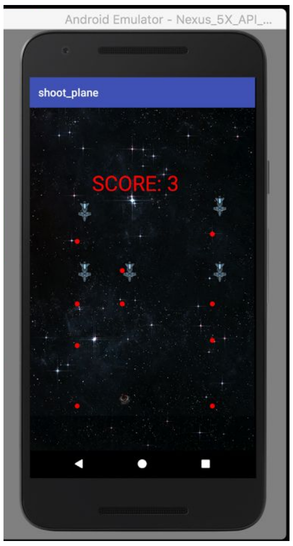
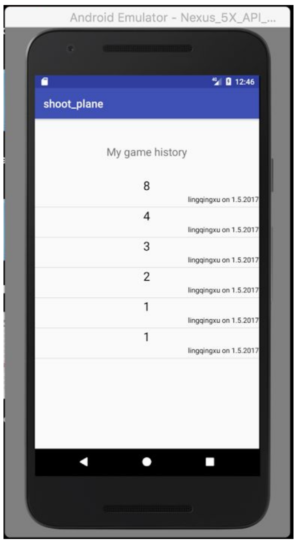
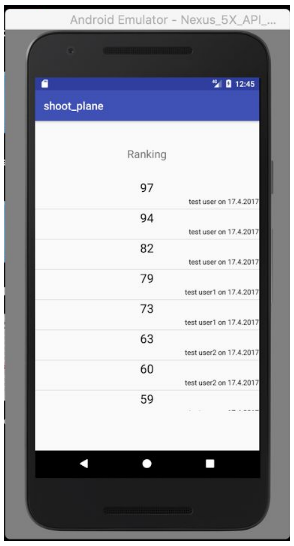


Fig. 5 Register page **Fig. 6 Sign in page** **Fig. 7 Start page**





```
My game history
8
lingqingxu on 1.5.2017
4
lingqingxu on 1.5.2017
3
lingqingxu on 1.5.2017
2
lingqingxu on 1.5.2017
1
lingqingxu on 1.5.2017
1
lingqingxu on 1.5.2017
```



```
Ranking
97
test user on 17.4.2017
94
test user on 17.4.2017
82
test user on 17.4.2017
79
test user1 on 17.4.2017
73
test user1 on 17.4.2017
63
test user2 on 17.4.2017
60
test user2 on 17.4.2017
59
```

Fig. 8 Game page **Fig. 9 My Game history page** **Fig. 10 Ranking page**

Subseciton C. Space Shooting Game

1. Design Overview

We use the procedural way to design our App, which means writing Java code to create and manipulate all the user interface objects. The reason we chose this way is that the

View is changing all the time, which is easier to manipulate using code that xml files. The View is realized by Class GameView using canvas, which draw 40 times per second to make to picture look like animation.

The logic of the game is: After the game begins, user's aircraft continues shooting towards enemies, and enemies shoot towards user's aircraft every 3 seconds. The enemies will explore if colliding with user's bullet. Similarly, if user's aircraft collides with any enemy or enemy's bullet, the game is over. Then the score will be shown on the screen to indicate how many enemies has the user eliminated.

2. Classes Overview

The game development follows the Object Oriented Design, namely, everything is regarded as a class. Below lists the main classes that are used in the game design and their main function.

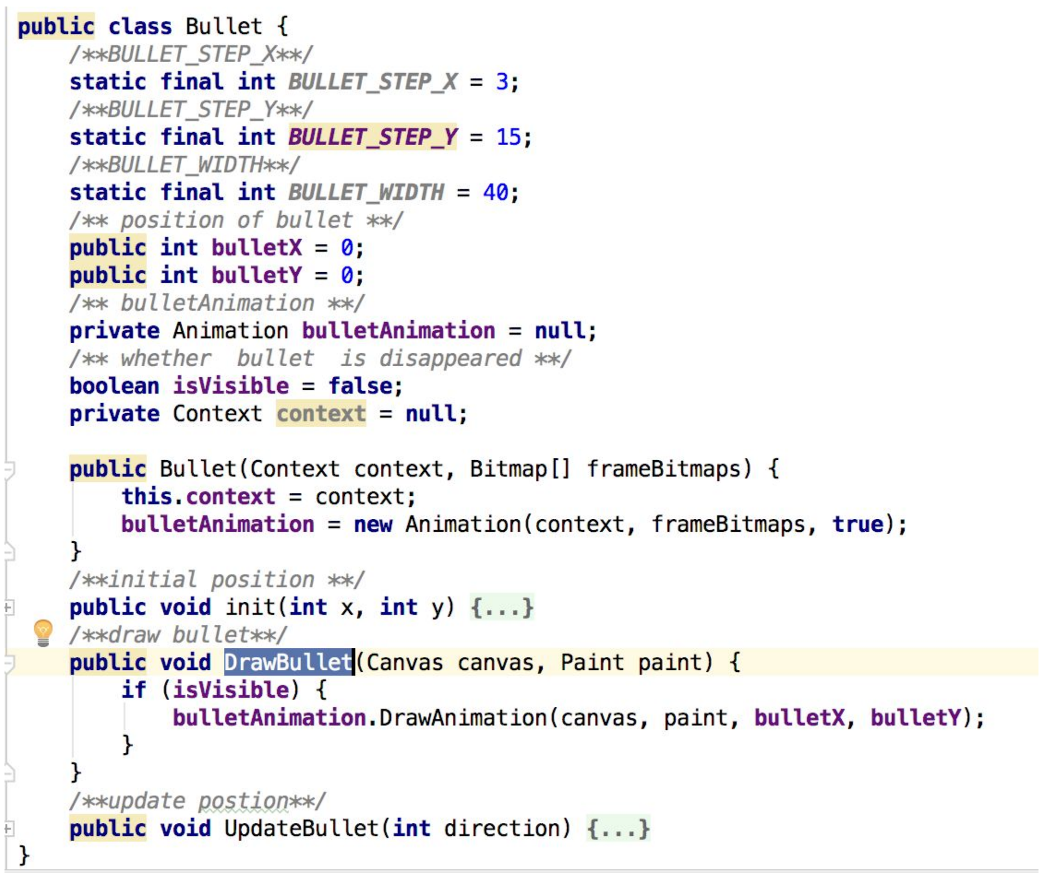
(1) **Class Bullet:**

This class includes the attributes and methods a bullet has. In the game both the enemies and out aircraft can shoot, but their bullets features are different, which is described in their own Bullet Class. Like figure 11.

The Bullet Class mainly records the coordinate and moving step of the bullet, and its animation when moving forward. If the bullet collides with an enemy or is out of boundary of screen, it should not be seen any more, so we use a boolean variable "isVisible" to record its state. Every time before the canvas refresh the animation, the coordinate is updated using the method UpdateBullet(). The DrawBullet() method is responsible for drawing the bullet in the right position after updating.

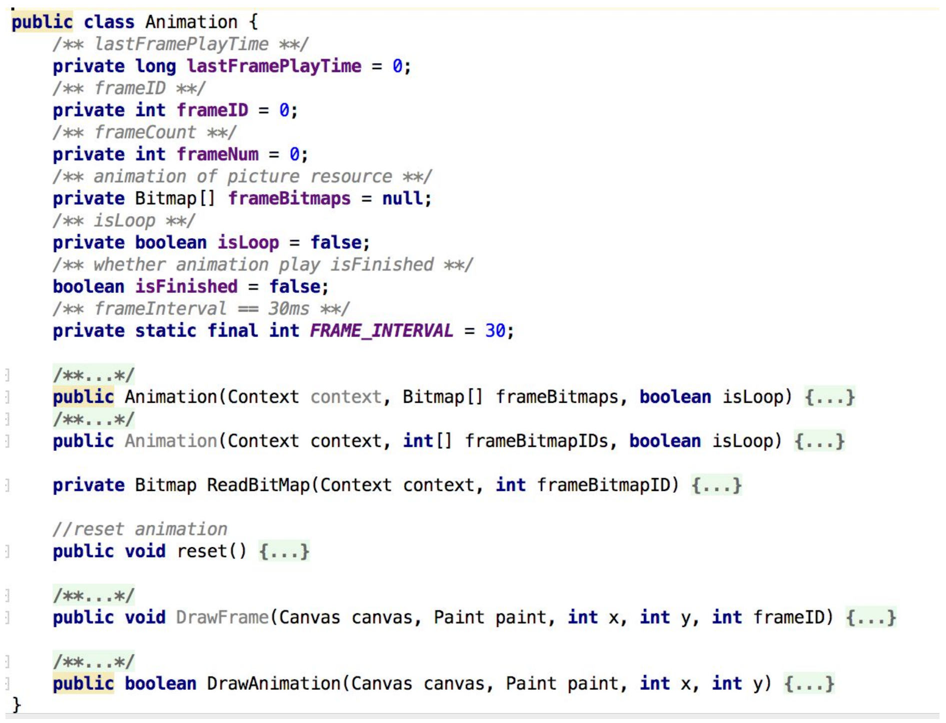
(2) Class Animation

As the name indicates, the Class Animation controls the animation of bullet, user's aircraft, and enemies. It not only records how many frames an animation has and their responding bitmaps, but also the animation play states, such as interval, play time, loop, finish state. The method DrawAnimation (Canvas canvas, Paint paint, int x, int y) realizes draw the animation in order. Like figure 12.



```java
public class Bullet {
    /**BULLET_STEP_X**/
    static final int BULLET_STEP_X = 3;
    /**BULLET_STEP_Y**/
    static final int BULLET_STEP_Y = 15;
    /**BULLET_WIDTH**/
    static final int BULLET_WIDTH = 40;
    /** position of bullet **/
    public int bulletX = 0;
    public int bulletY = 0;
    /** bulletAnimation **/
    private Animation bulletAnimation = null;
    /** whether  bullet  is disappeared **/
    boolean isVisible = false;
    private Context context = null;

    public Bullet(Context context, Bitmap[] frameBitmaps) {
        this.context = context;
        bulletAnimation = new Animation(context, frameBitmaps, true);
    }
    /**initial position **/
    public void init(int x, int y) {...}
    /**draw bullet**/
    public void DrawBullet(Canvas canvas, Paint paint) {
        if (isVisible) {
            bulletAnimation.DrawAnimation(canvas, paint, bulletX, bulletY);
        }
    }
    /**update postion**/
    public void UpdateBullet(int direction) {...}
}
```

Fig. 11 Bullet attributes and methods

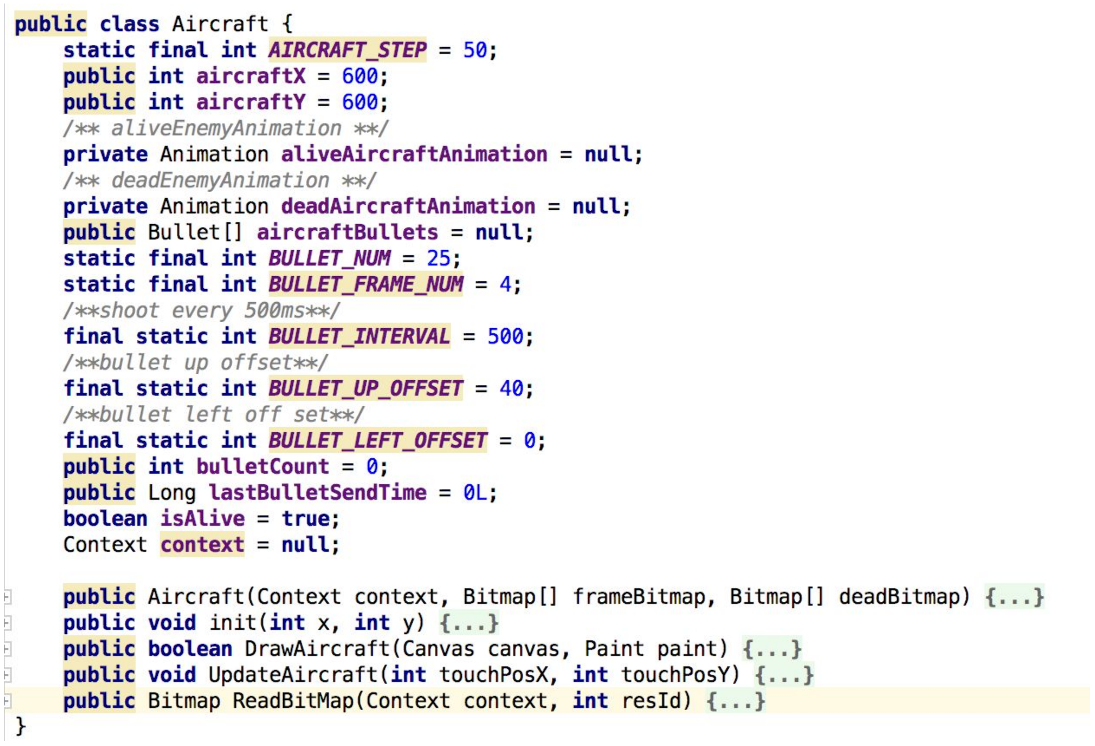


```java
public class Animation {
    /** lastFramePlayTime **/
    private long lastFramePlayTime = 0;
    /** frameID **/
    private int frameID = 0;
    /** frameCount **/
    private int frameNum = 0;
    /** animation of picture resource **/
    private Bitmap[] frameBitmaps = null;
    /** isLoop **/
    private boolean isLoop = false;
    /** whether animation play isFinished **/
    boolean isFinished = false;
    /** frameInterval == 30ms **/
    private static final int FRAME_INTERVAL = 30;

    /**...*/
    public Animation(Context context, Bitmap[] frameBitmaps, boolean isLoop) {...}
    /**...*/
    public Animation(Context context, int[] frameBitmapIDs, boolean isLoop) {...}

    private Bitmap ReadBitMap(Context context, int frameBitmapID) {...}

    //reset animation
    public void reset() {...}

    /**...*/
    public void DrawFrame(Canvas canvas, Paint paint, int x, int y, int frameID) {...}

    /**...*/
    public boolean DrawAnimation(Canvas canvas, Paint paint, int x, int y) {...}
}
```

Fig. 12 Bullet Animations

(3) Class Aircraft and Class Enemy

The class Aircraft and Class Enemy are similar, representing the behavior of the user and enemy respectively. The attributes of the classes include the coordinate of the aircraft/enemy, character of shooting (shooting interval, bullet number, bullet offset, bullet array) and their living state (isAlive or not). They have three methods: init() to initialize the position and state; UpdateAircraft (int touchPosX, int touchPosY) to change the position of aircraft when touching; DrawAircraft (Canvas canvas, Paint paint) to draw it on the canvas.

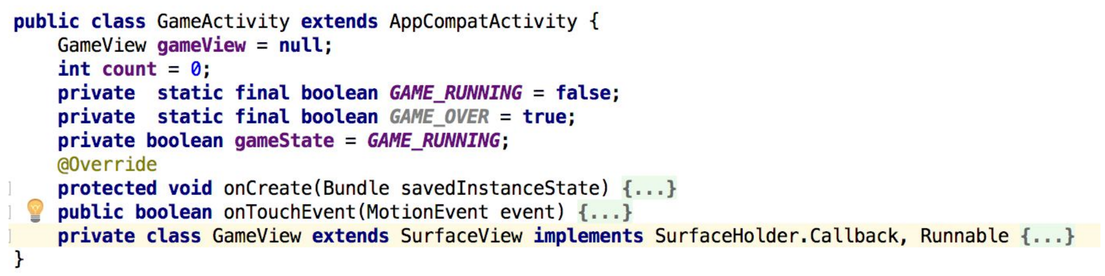


```java
public class Aircraft {
    static final int AIRCRAFT_STEP = 50;
    public int aircraftX = 600;
    public int aircraftY = 600;
    /** aliveEnemyAnimation **/
    private Animation aliveAircraftAnimation = null;
    /** deadEnemyAnimation **/
    private Animation deadAircraftAnimation = null;
    public Bullet[] aircraftBullets = null;
    static final int BULLET_NUM = 25;
    static final int BULLET_FRAME_NUM = 4;
    /**shoot every 500ms**/
    final static int BULLET_INTERVAL = 500;
    /**bullet up offset**/
    final static int BULLET_UP_OFFSET = 40;
    /**bullet left off set**/
    final static int BULLET_LEFT_OFFSET = 0;
    public int bulletCount = 0;
    public Long lastBulletSendTime = 0L;
    boolean isAlive = true;
    Context context = null;

    public Aircraft(Context context, Bitmap[] frameBitmap, Bitmap[] deadBitmap) {...}
    public void init(int x, int y) {...}
    public boolean DrawAircraft(Canvas canvas, Paint paint) {...}
    public void UpdateAircraft(int touchPosX, int touchPosY) {...}
    public Bitmap ReadBitMap(Context context, int resId) {...}
}
```

Fig. 13 Aircraft & Enemy Behaviors

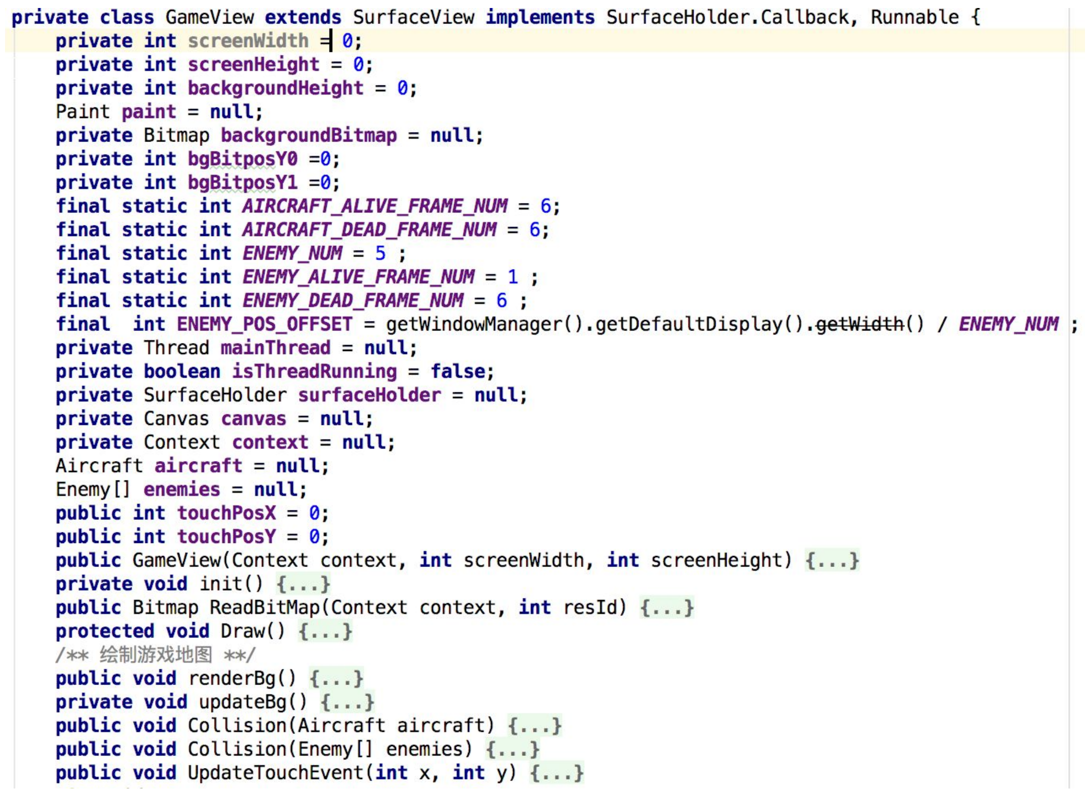
(4) GameActivity

The GameActiviy controls all the operations and states in the game. As mentioned above, we use the procedural way to design our App, so we create an inner class GameView to be responsible for the UI. The GameView class starts a new thread to refresh the animation every 25 ms. After initializing, it first check the state flag "isThreadRunning". If the thread is running, it updates all the instances' state before drawing them on the canvas. If user's aircraft is dead, the thread will stop, and draws the score on the screen.



```java
public class GameActivity extends AppCompatActivity {
    GameView gameView = null;
    int count = 0;
    private  static final boolean GAME_RUNNING = false;
    private  static final boolean GAME_OVER = true;
    private boolean gameState = GAME_RUNNING;
    @Override
    protected void onCreate(Bundle savedInstanceState) {...}
    public boolean onTouchEvent(MotionEvent event) {...}
    private class GameView extends SurfaceView implements SurfaceHolder.Callback, Runnable {...}
}
```

Fig. 14 Operations & States in game



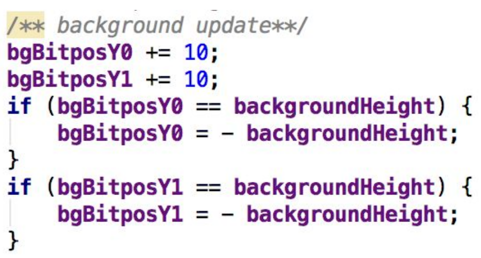
```java
private class GameView extends SurfaceView implements SurfaceHolder.Callback, Runnable {
    private int screenWidth = 0;
    private int screenHeight = 0;
    private int backgroundHeight = 0;
    Paint paint = null;
    private Bitmap backgroundBitmap = null;
    private int bgBitposY0 =0;
    private int bgBitposY1 =0;
    final static int AIRCRAFT_ALIVE_FRAME_NUM = 6;
    final static int AIRCRAFT_DEAD_FRAME_NUM = 6;
    final static int ENEMY_NUM = 5 ;
    final static int ENEMY_ALIVE_FRAME_NUM = 1 ;
    final static int ENEMY_DEAD_FRAME_NUM = 6 ;
    final  int ENEMY_POS_OFFSET = getWindowManager().getDefaultDisplay().~~getWidth~~() / ENEMY_NUM ;
    private Thread mainThread = null;
    private boolean isThreadRunning = false;
    private SurfaceHolder surfaceHolder = null;
    private Canvas canvas = null;
    private Context context = null;
    Aircraft aircraft = null;
    Enemy[] enemies = null;
    public int touchPosX = 0;
    public int touchPosY = 0;
    public GameView(Context context, int screenWidth, int screenHeight) {...}
    private void init() {...}
    public Bitmap ReadBitMap(Context context, int resId) {...}
    protected void Draw() {...}
    /** 绘制游戏地图 **/
    public void renderBg() {...}
    private void updateBg() {...}
    public void Collision(Aircraft aircraft) {...}
    public void Collision(Enemy[] enemies) {...}
    public void UpdateTouchEvent(int x, int y) {...}
```

Fig. 15 New thread to refresh animation

3. Detail Realization

(1) Background Scroll

The background picture has been scrolling back to the player, making an illusion of their own control of the aircraft in the forward flight. In our game, two map images in the screen behind the alternate rolling, this will give players a forward move illusion.

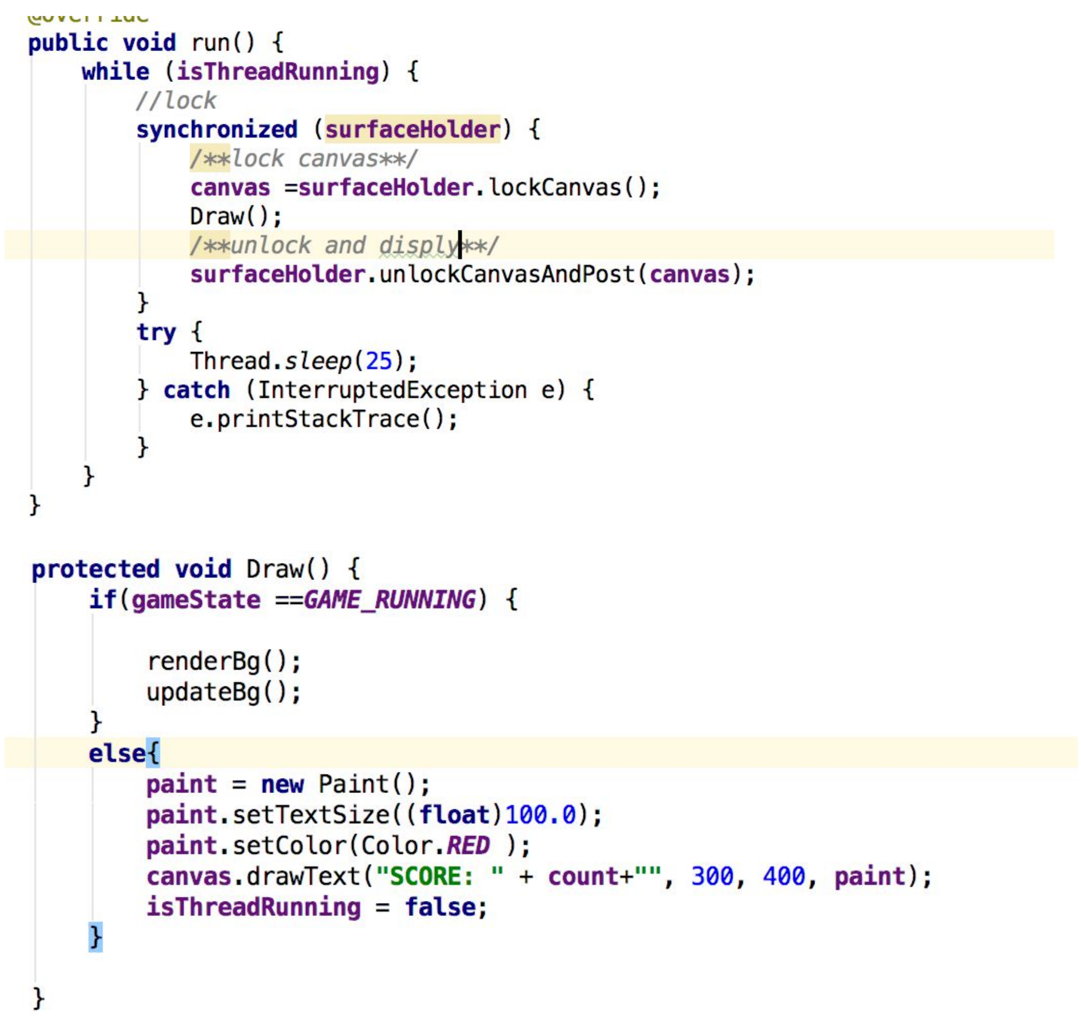


```
/** background update**/
bgBitposY0 += 10;
bgBitposY1 += 10;
if (bgBitposY0 == backgroundHeight) {
    bgBitposY0 = - backgroundHeight;
}
if (bgBitposY1 == backgroundHeight) {
    bgBitposY1 = - backgroundHeight;
}
```

Fig. 16 Background

(2) Animation Refresh

The animation refresh is realized in a standalone thread, which sleep 25 ms in every loop. In every loop, the state(alive/ dead), position(coordinate), corresponding bitmaps are prepared and updated, then they are drawn on the canvas.

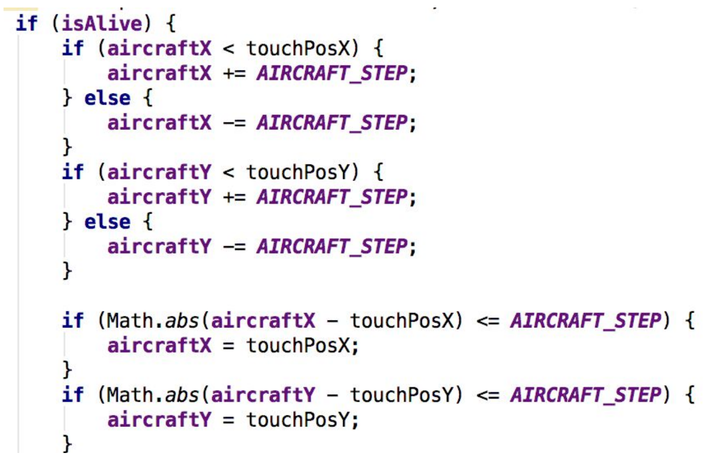


```
public void run() {
    while (isThreadRunning) {
        //lock
        synchronized (surfaceHolder) {
            /**lock canvas**/
            canvas =surfaceHolder.lockCanvas();
            Draw();
            /**unlock and disply**/
            surfaceHolder.unlockCanvasAndPost(canvas);
        }
        try {
            Thread.sleep(25);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

protected void Draw() {
    if(gameState ==GAME_RUNNING) {

        renderBg();
        updateBg();
    }
    else{
        paint = new Paint();
        paint.setTextSize((float)100.0);
        paint.setColor(Color.RED );
        canvas.drawText("SCORE: " + count+"", 300, 400, paint);
        isThreadRunning = false;
    }

}
```

Fig. 17 Animation Refresh

(3) Aircraft Moving

After touching the screen at any point, the program can get the current point of the X, Y coordinates. Calculate the distance between the current X, Y coordinate points and the target X, Y points centered on the X Y coordinates of the current aircraft. Because the aircraft can not directly fly to the target point from the current coordinates, so we add the current coordinates of X, Y to one step every loop. Here we need to consider when the aircraft is moving but the user stops to touch the screen. if you stop touching the plane will stop in place rather than move to the target point until the player touches the new screen X, Y coordinates.

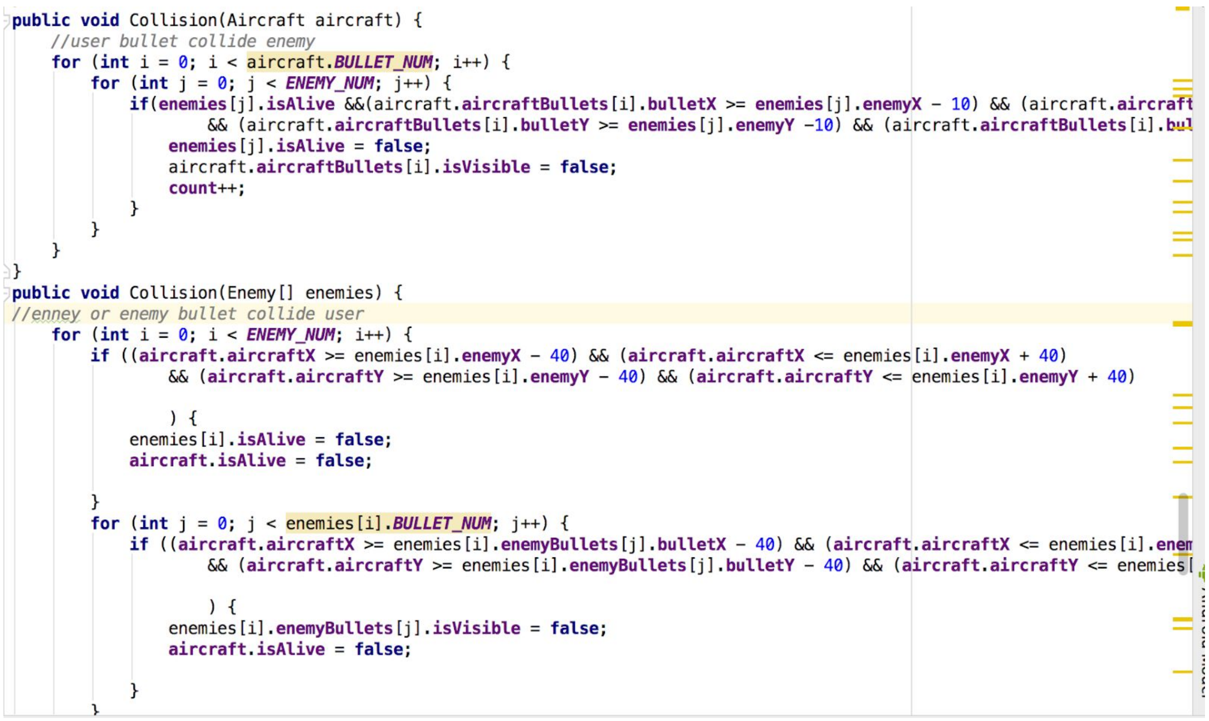


```
if (isAlive) {
    if (aircraftX < touchPosX) {
        aircraftX += AIRCRAFT_STEP;
    } else {
        aircraftX -= AIRCRAFT_STEP;
    }
    if (aircraftY < touchPosY) {
        aircraftY += AIRCRAFT_STEP;
    } else {
        aircraftY -= AIRCRAFT_STEP;
    }

    if (Math.abs(aircraftX - touchPosX) <= AIRCRAFT_STEP) {
        aircraftX = touchPosX;
    }
    if (Math.abs(aircraftY - touchPosY) <= AIRCRAFT_STEP) {
        aircraftY = touchPosY;
    }
```

Fig. 18 Aircraft Moving

(4) Collision

As the number of bullets is quite a lot, each bullet needs to use an object to record the current bullet X, Y coordinates and drawing in the screen area, each plane is also an object recording its X, Y coordinates with the drawing area in the screen. In this way, dealing with the collision is actually the collision of a rectangular area of each bullet with each enemy's rectangular area. By traversing the bullet object and the enemy object we can calculate the result of the collision, and change the state of the aircraft.

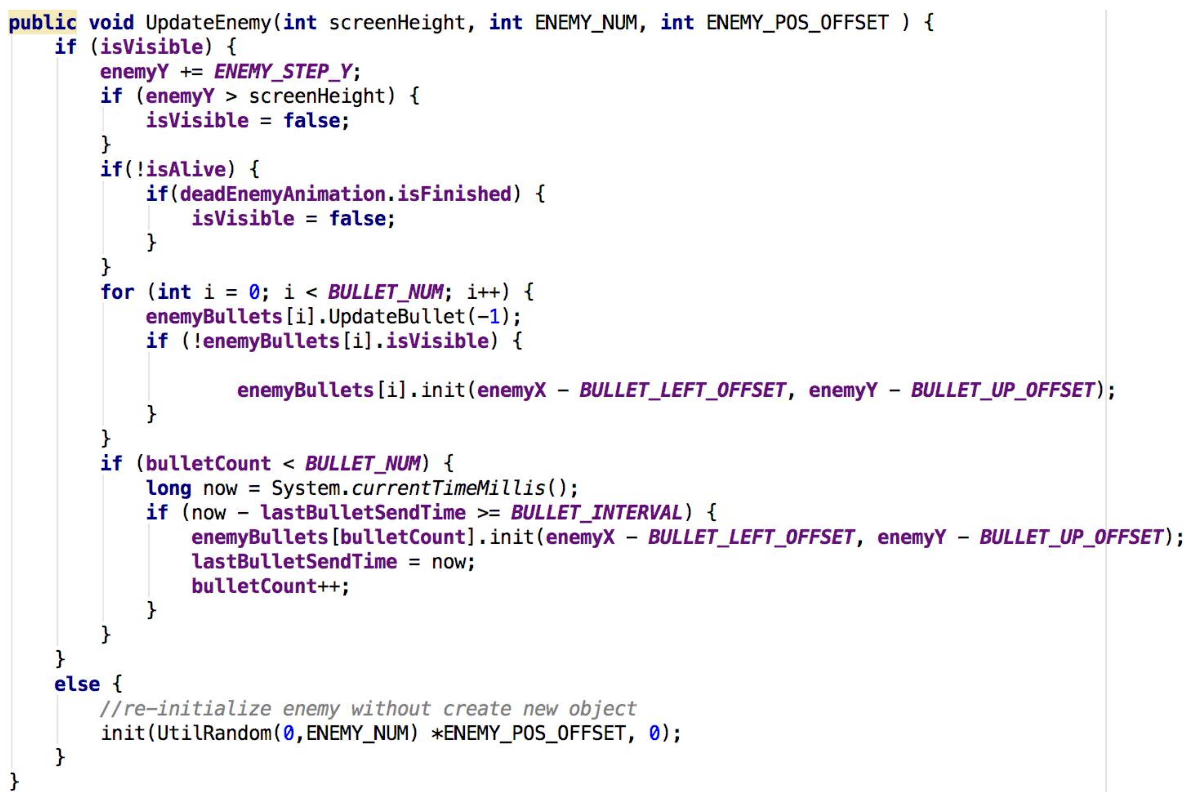


```
public void Collision(Aircraft aircraft) {
    //user bullet collide enemy
    for (int i = 0; i < aircraft.BULLET_NUM; i++) {
        for (int j = 0; j < ENEMY_NUM; j++) {
            if(enemies[j].isAlive &&(aircraft.aircraftBullets[i].bulletX >= enemies[j].enemyX - 10) && (aircraft.aircraft
                    && (aircraft.aircraftBullets[i].bulletY >= enemies[j].enemyY -10) && (aircraft.aircraftBullets[i].bul
                enemies[j].isAlive = false;
                aircraft.aircraftBullets[i].isVisible = false;
                count++;
            }
        }
    }
}
public void Collision(Enemy[] enemies) {
//enney or enemy bullet collide user
    for (int i = 0; i < ENEMY_NUM; i++) {
        if ((aircraft.aircraftX >= enemies[i].enemyX - 40) && (aircraft.aircraftX <= enemies[i].enemyX + 40)
                && (aircraft.aircraftY >= enemies[i].enemyY - 40) && (aircraft.aircraftY <= enemies[i].enemyY + 40)

                ) {
            enemies[i].isAlive = false;
            aircraft.isAlive = false;

        }
        for (int j = 0; j < enemies[i].BULLET_NUM; j++) {
            if ((aircraft.aircraftX >= enemies[i].enemyBullets[j].bulletX - 40) && (aircraft.aircraftX <= enemies[i].enem
                    && (aircraft.aircraftY >= enemies[i].enemyBullets[j].bulletY - 40) && (aircraft.aircraftY <= enemies[

                    ) {
                enemies[i].enemyBullets[j].isVisible = false;
                aircraft.isAlive = false;

            }
        }
```

Fig. 19 Collision

(5) Memory Leaks Preventing

If in accordance with the above idea directly, we will create bullet objects and enemy objects frequently, which will cause serious problems such as memory leaks. As the number of bullets that need to be drawn on the screen and the number of enemy planes is certainly limited, we can initialize the fixed bullet object and the enemy object only to update these objects and their logic. For example, in the current game screen I need 5 aircraft, the code I will only allocate five enemy objects, respectively, to detect if the bullet hit these objects or the bitmap goes down the bottom of the screen. If so, we can reset the property, so that the aircraft reappeared in the top of the battlefield. In this way, the player will feel endless enemies, but we don’t have to create new objects every time.

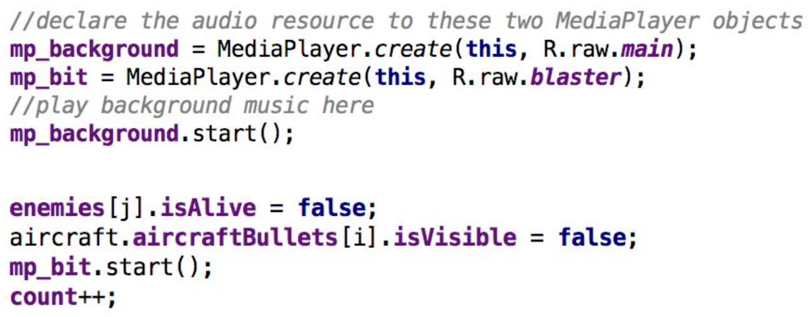


```java
public void UpdateEnemy(int screenHeight, int ENEMY_NUM, int ENEMY_POS_OFFSET ) {
    if (isVisible) {
        enemyY += ENEMY_STEP_Y;
        if (enemyY > screenHeight) {
            isVisible = false;
        }
        if(!isAlive) {
            if(deadEnemyAnimation.isFinished) {
                isVisible = false;
            }
        }
        for (int i = 0; i < BULLET_NUM; i++) {
            enemyBullets[i].UpdateBullet(-1);
            if (!enemyBullets[i].isVisible) {

                    enemyBullets[i].init(enemyX - BULLET_LEFT_OFFSET, enemyY - BULLET_UP_OFFSET);
            }
        }
        if (bulletCount < BULLET_NUM) {
            long now = System.currentTimeMillis();
            if (now - lastBulletSendTime >= BULLET_INTERVAL) {
                enemyBullets[bulletCount].init(enemyX - BULLET_LEFT_OFFSET, enemyY - BULLET_UP_OFFSET);
                lastBulletSendTime = now;
                bulletCount++;
            }
        }
    }
    else {
        //re-initialize enemy without create new object
        init(UtilRandom(0,ENEMY_NUM) *ENEMY_POS_OFFSET, 0);
    }
}
```

Fig. 20 Memory Leaks Preventing

(6) Game Music

To make the game more enjoyable, we add background music and special effects music. The special effect music is triggered when an enemy is eleminated.



```java
//declare the audio resource to these two MediaPlayer objects
mp_background = MediaPlayer.create(this, R.raw.main);
mp_bit = MediaPlayer.create(this, R.raw.blaster);
//play background music here
mp_background.start();

enemies[j].isAlive = false;
aircraft.aircraftBullets[i].isVisible = false;
mp_bit.start();
count++;
```

Fig. 21 Game Music Implementation

Subseciton D. Individual History

For an individual user, the history information includes two types: recent activity, and history statistics.

A. Recent Activity

At its most basic it can be used to time and record the scores for any single game. The attributes includes the reaction speed, the fly distance, the number of obstacles. For the moments, the score for each game is by simply weighted sum of those attributes.

B. History Statistics

The aim is to record scores and stats for each individual game. It can also be used to record user-defined stats for every player with the option of outputting the data to a file.

This module visually shows a statistic histogram to the player, and excite him to player another new game so that he can update his scores.

Subseciton E. System History

For the whole system, the history information also includes two types: recent activity, and history statistics.

A. Recent Activity

We record the most recent several shooting game's in a list style. This information give a direct implication of the system's current states. Usually, active players check this information a lot during his play.

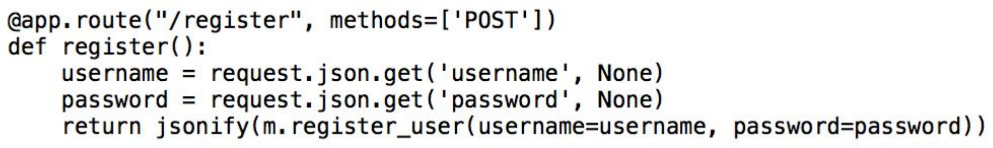
B. History Statistics

This is a statistics made from all activities happen in the system from its launch. Most importantly, we provide an overall ranking for both all users and all games.

Section V. Interface Description and Implementation

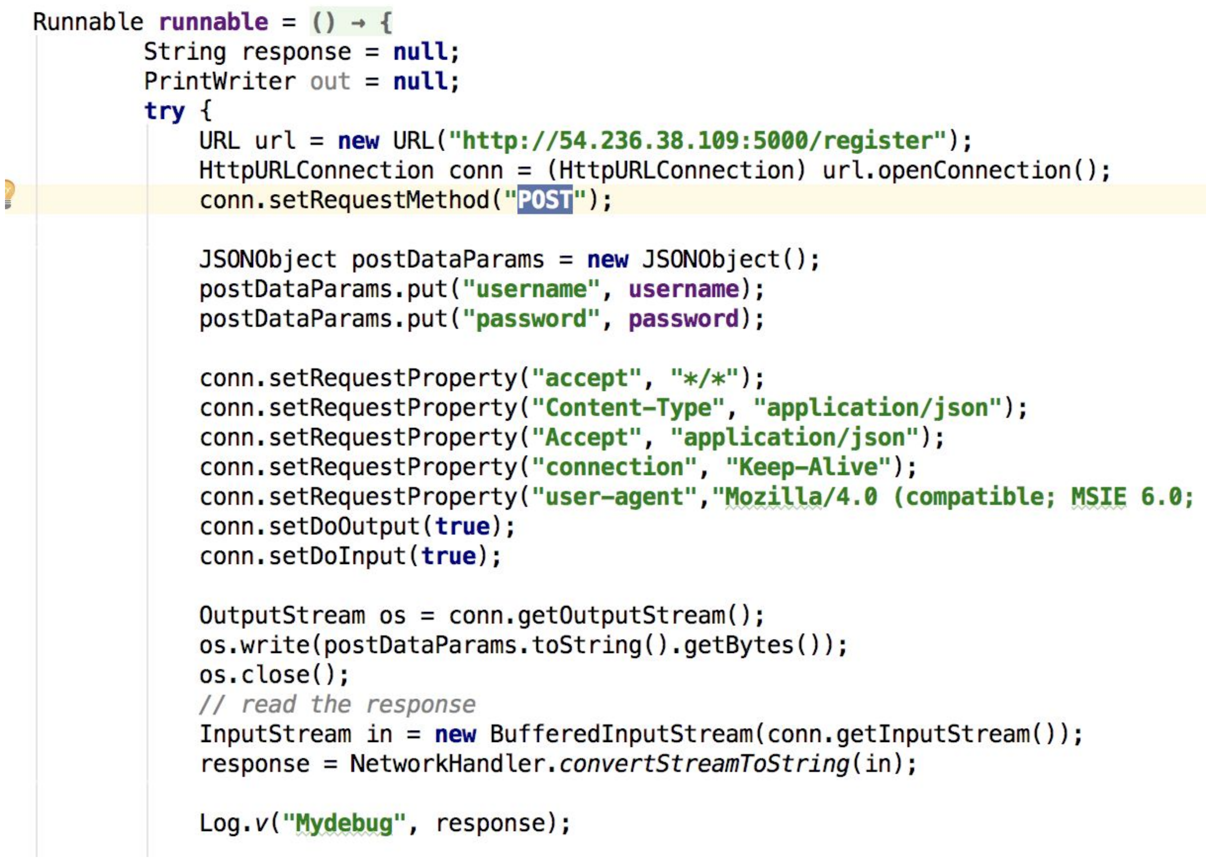
Subseciton A. Registration: Sign Up

The POST method is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line. Here, the address of our server is http://54.236.38.109:5000/. The code of server parts are written by python:



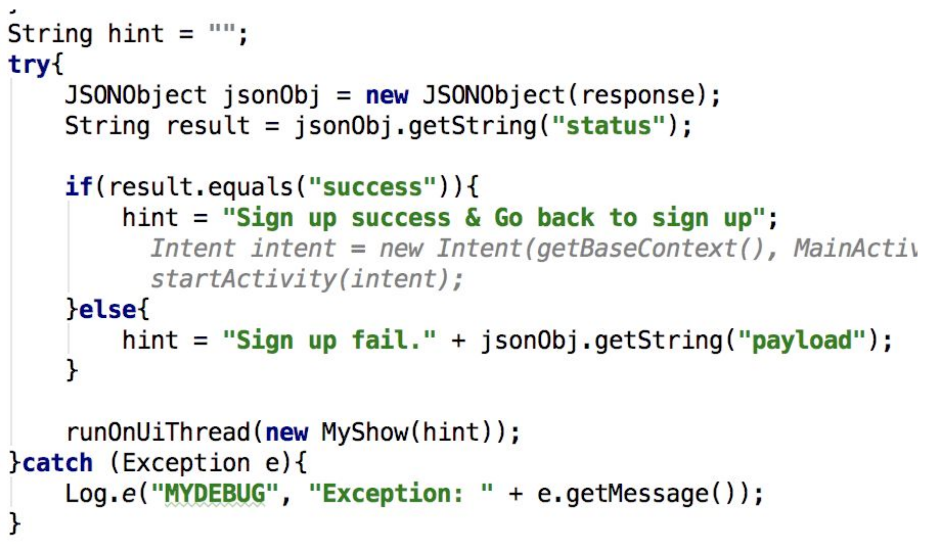
```
@app.route("/register", methods=['POST'])
def register():
    username = request.json.get('username', None)
    password = request.json.get('password', None)
    return jsonify(m.register_user(username=username, password=password))
```

Fig. 22 Register method



```
Runnable runnable = () → {
        String response = null;
        PrintWriter out = null;
        try {
            URL url = new URL("http://54.236.38.109:5000/register");
            HttpURLConnection conn = (HttpURLConnection) url.openConnection();
            conn.setRequestMethod("POST");

            JSONObject postDataParams = new JSONObject();
            postDataParams.put("username", username);
            postDataParams.put("password", password);

            conn.setRequestProperty("accept", "*/*");
            conn.setRequestProperty("Content-Type", "application/json");
            conn.setRequestProperty("Accept", "application/json");
            conn.setRequestProperty("connection", "Keep-Alive");
            conn.setRequestProperty("user-agent","Mozilla/4.0 (compatible; MSIE 6.0;
            conn.setDoOutput(true);
            conn.setDoInput(true);

            OutputStream os = conn.getOutputStream();
            os.write(postDataParams.toString().getBytes());
            os.close();
            // read the response
            InputStream in = new BufferedInputStream(conn.getInputStream());
            response = NetworkHandler.convertStreamToString(in);

            Log.v("Mydebug", response);
```

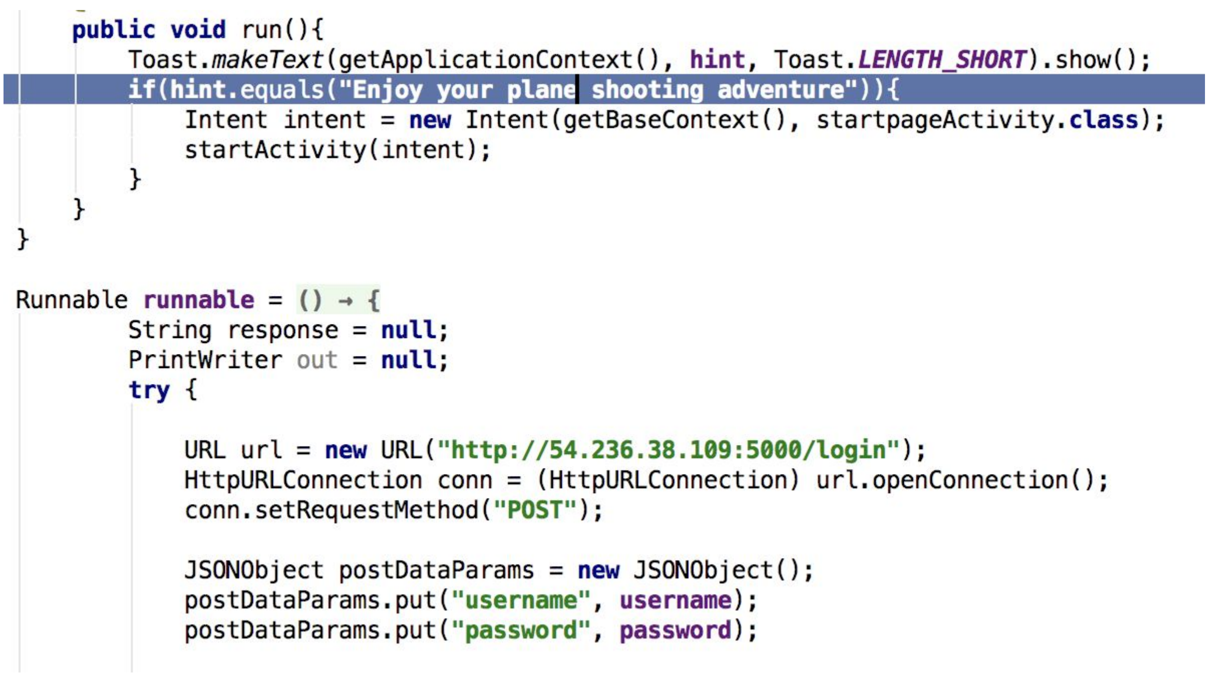
Fig. 23 Post registrition data



```
String hint = "";
try{
    JSONObject jsonObj = new JSONObject(response);
    String result = jsonObj.getString("status");

    if(result.equals("success")){
        hint = "Sign up success & Go back to sign up";
          Intent intent = new Intent(getBaseContext(), MainActiv
          startActivity(intent);
    }else{
        hint = "Sign up fail." + jsonObj.getString("payload");
    }

    runOnUiThread(new MyShow(hint));
}catch (Exception e){
    Log.e("MYDEBUG", "Exception: " + e.getMessage());
}
```

Fig. 24 Get registrition status

Subseciton B. LogIn: Sign In

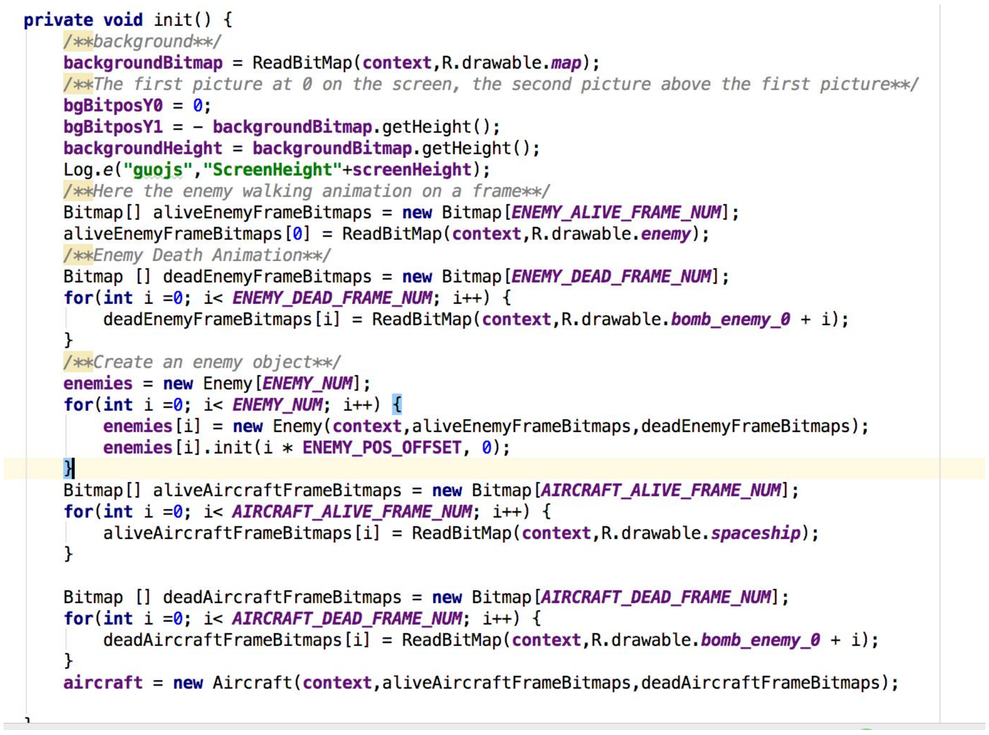


```
    public void run(){
        Toast.makeText(getApplicationContext(), hint, Toast.LENGTH_SHORT).show();
        if(hint.equals("Enjoy your plane shooting adventure")){
            Intent intent = new Intent(getBaseContext(), startpageActivity.class);
            startActivity(intent);
        }
    }
}

Runnable runnable = () → {
        String response = null;
        PrintWriter out = null;
        try {

            URL url = new URL("http://54.236.38.109:5000/login");
            HttpURLConnection conn = (HttpURLConnection) url.openConnection();
            conn.setRequestMethod("POST");

            JSONObject postDataParams = new JSONObject();
            postDataParams.put("username", username);
            postDataParams.put("password", password);
```

Fig. 25 Post sign in data

Subseciton C. Space Shooting Game

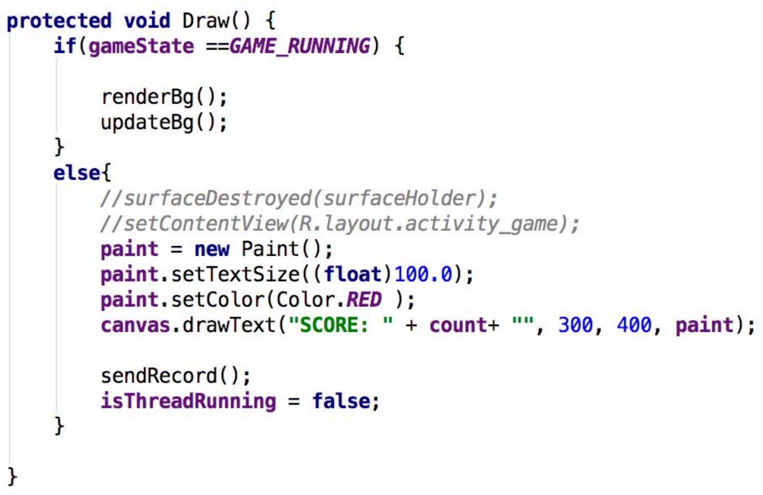
We initialize the state before starting the game.



```
private void init() {
    /**background**/
    backgroundBitmap = ReadBitMap(context,R.drawable.map);
    /**The first picture at 0 on the screen, the second picture above the first picture**/
    bgBitposY0 = 0;
    bgBitposY1 = - backgroundBitmap.getHeight();
    backgroundHeight = backgroundBitmap.getHeight();
    Log.e("guojs","ScreenHeight"+screenHeight);
    /**Here the enemy walking animation on a frame**/
    Bitmap[] aliveEnemyFrameBitmaps = new Bitmap[ENEMY_ALIVE_FRAME_NUM];
    aliveEnemyFrameBitmaps[0] = ReadBitMap(context,R.drawable.enemy);
    /**Enemy Death Animation**/
    Bitmap [] deadEnemyFrameBitmaps = new Bitmap[ENEMY_DEAD_FRAME_NUM];
    for(int i =0; i< ENEMY_DEAD_FRAME_NUM; i++) {
        deadEnemyFrameBitmaps[i] = ReadBitMap(context,R.drawable.bomb_enemy_0 + i);
    }
    /**Create an enemy object**/
    enemies = new Enemy[ENEMY_NUM];
    for(int i =0; i< ENEMY_NUM; i++) {
        enemies[i] = new Enemy(context,aliveEnemyFrameBitmaps,deadEnemyFrameBitmaps);
        enemies[i].init(i * ENEMY_POS_OFFSET, 0);
    }
    Bitmap[] aliveAircraftFrameBitmaps = new Bitmap[AIRCRAFT_ALIVE_FRAME_NUM];
    for(int i =0; i< AIRCRAFT_ALIVE_FRAME_NUM; i++) {
        aliveAircraftFrameBitmaps[i] = ReadBitMap(context,R.drawable.spaceship);
    }

    Bitmap [] deadAircraftFrameBitmaps = new Bitmap[AIRCRAFT_DEAD_FRAME_NUM];
    for(int i =0; i< AIRCRAFT_DEAD_FRAME_NUM; i++) {
        deadAircraftFrameBitmaps[i] = ReadBitMap(context,R.drawable.bomb_enemy_0 + i);
    }
    aircraft = new Aircraft(context,aliveAircraftFrameBitmaps,deadAircraftFrameBitmaps);
```

Fig. 26 Space Shooting Game Initialization

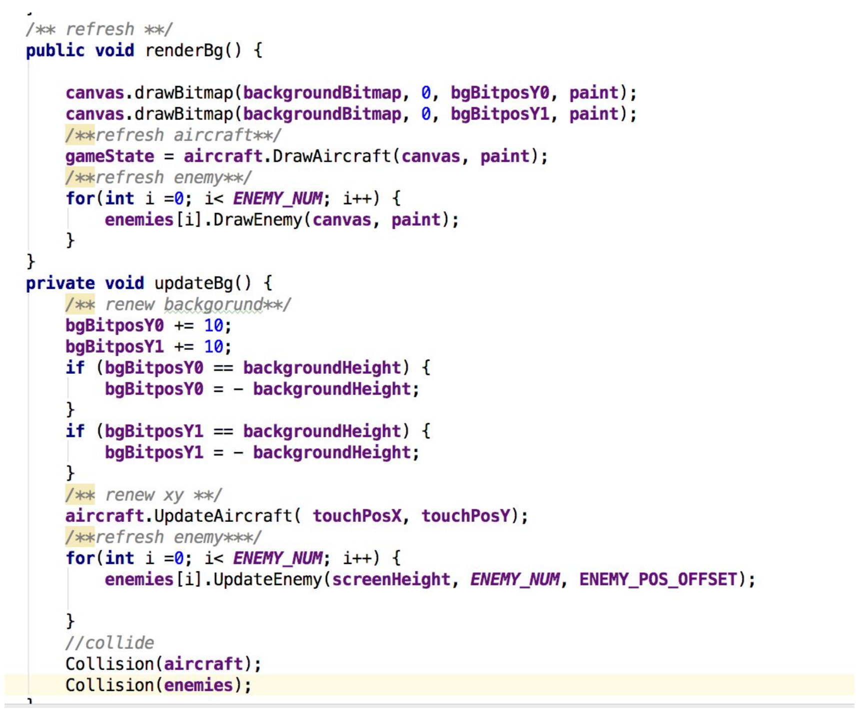
Every time we refresh the canvas, we draw new paint on it.



```
protected void Draw() {
    if(gameState ==GAME_RUNNING) {

        renderBg();
        updateBg();
    }
    else{
        //surfaceDestroyed(surfaceHolder);
        //setContentView(R.layout.activity_game);
        paint = new Paint();
        paint.setTextSize((float)100.0);
        paint.setColor(Color.RED );
        canvas.drawText("SCORE: " + count+ "", 300, 400, paint);

        sendRecord();
        isThreadRunning = false;
    }

}
```

Fig. 27 Canvas refresh

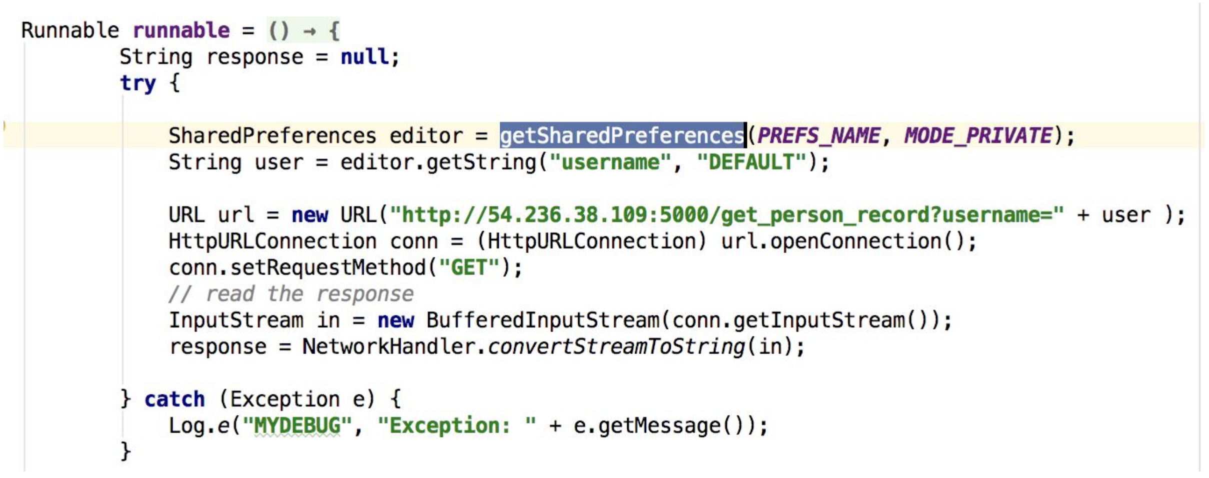
The renderBg() and updateBg() is realized below.



```
/** refresh **/
public void renderBg() {

    canvas.drawBitmap(backgroundBitmap, 0, bgBitposY0, paint);
    canvas.drawBitmap(backgroundBitmap, 0, bgBitposY1, paint);
    /**refresh aircraft**/
    gameState = aircraft.DrawAircraft(canvas, paint);
    /**refresh enemy**/
    for(int i =0; i< ENEMY_NUM; i++) {
        enemies[i].DrawEnemy(canvas, paint);
    }
}
private void updateBg() {
    /** renew backgorund**/
    bgBitposY0 += 10;
    bgBitposY1 += 10;
    if (bgBitposY0 == backgroundHeight) {
        bgBitposY0 = - backgroundHeight;
    }
    if (bgBitposY1 == backgroundHeight) {
        bgBitposY1 = - backgroundHeight;
    }
    /** renew xy **/
    aircraft.UpdateAircraft( touchPosX, touchPosY);
    /**refresh enemy***/
    for(int i =0; i< ENEMY_NUM; i++) {
        enemies[i].UpdateEnemy(screenHeight, ENEMY_NUM, ENEMY_POS_OFFSET);

    }
    //collide
    Collision(aircraft);
    Collision(enemies);
```

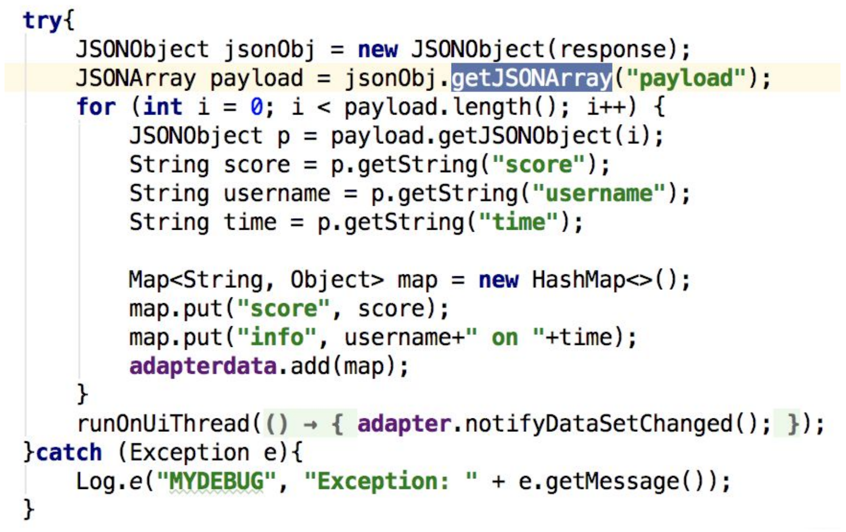
Fig. 28 Render and Updata

Subseciton D. Individual History



```
Runnable runnable = () → {
        String response = null;
        try {

            SharedPreferences editor = getSharedPreferences(PREFS_NAME, MODE_PRIVATE);
            String user = editor.getString("username", "DEFAULT");

            URL url = new URL("http://54.236.38.109:5000/get_person_record?username=" + user );
            HttpURLConnection conn = (HttpURLConnection) url.openConnection();
            conn.setRequestMethod("GET");
            // read the response
            InputStream in = new BufferedInputStream(conn.getInputStream());
            response = NetworkHandler.convertStreamToString(in);

        } catch (Exception e) {
            Log.e("MYDEBUG", "Exception: " + e.getMessage());
        }
```

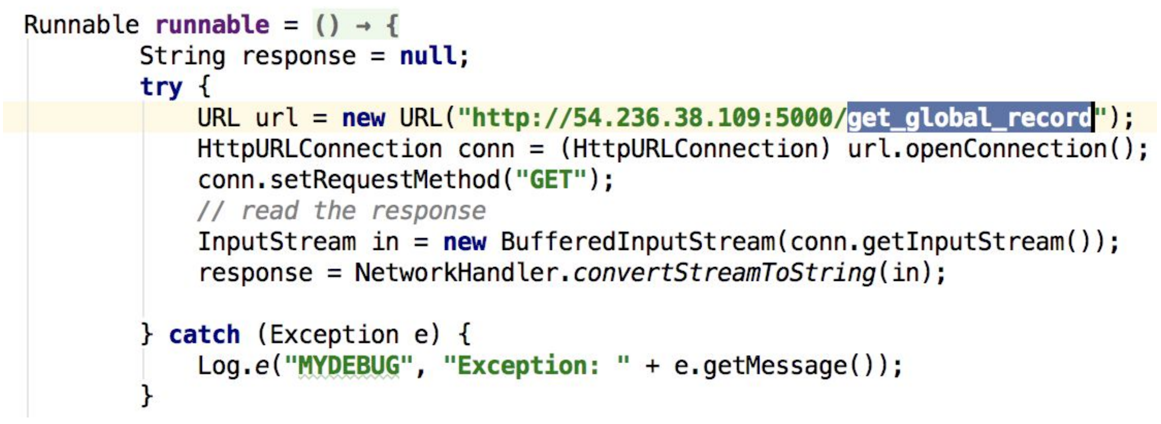
Fig. 29 Individual History-get data



```
try{
    JSONObject jsonObj = new JSONObject(response);
    JSONArray payload = jsonObj.getJSONArray("payload");
    for (int i = 0; i < payload.length(); i++) {
        JSONObject p = payload.getJSONObject(i);
        String score = p.getString("score");
        String username = p.getString("username");
        String time = p.getString("time");

        Map<String, Object> map = new HashMap<>();
        map.put("score", score);
        map.put("info", username+" on "+time);
        adapterdata.add(map);
    }
    runOnUiThread(() → { adapter.notifyDataSetChanged(); });
}catch (Exception e){
    Log.e("MYDEBUG", "Exception: " + e.getMessage());
}
```

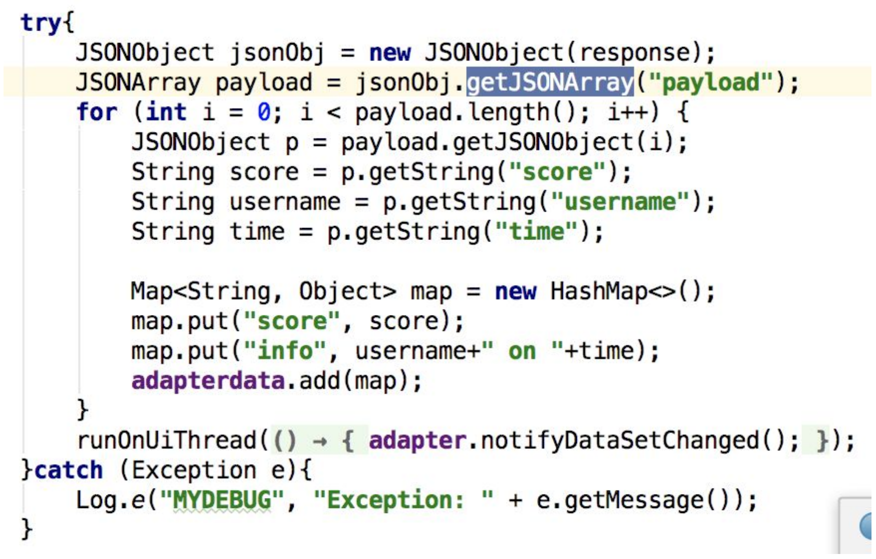
Fig. 30 Individual History-get data

Subseciton E. System History



```
Runnable runnable = () → {
        String response = null;
        try {
            URL url = new URL("http://54.236.38.109:5000/get_global_record");
            HttpURLConnection conn = (HttpURLConnection) url.openConnection();
            conn.setRequestMethod("GET");
            // read the response
            InputStream in = new BufferedInputStream(conn.getInputStream());
            response = NetworkHandler.convertStreamToString(in);

        } catch (Exception e) {
            Log.e("MYDEBUG", "Exception: " + e.getMessage());
        }
```

Fig. 30 System History-get global record


```java
try{
    JSONObject jsonObj = new JSONObject(response);
    JSONArray payload = jsonObj.getJSONArray("payload");
    for (int i = 0; i < payload.length(); i++) {
        JSONObject p = payload.getJSONObject(i);
        String score = p.getString("score");
        String username = p.getString("username");
        String time = p.getString("time");

        Map<String, Object> map = new HashMap<>();
        map.put("score", score);
        map.put("info", username+" on "+time);
        adapterdata.add(map);
    }
    runOnUiThread(() → { adapter.notifyDataSetChanged(); });
}catch (Exception e){
    Log.e("MYDEBUG", "Exception: " + e.getMessage());
}
```