Even Semester (2024)

---


**BINUS UNIVERSITY**

**BINUS INTERNATIONAL**

# Object Oriented Programming

# Final Project Report

**Student Information:**

**Surname: Jonathan** **Given Name:** Kevin Jonathan **Student ID:** 2702342823

**Course Code:** COMP6699001 **Course Name:** Object Oriented Programming

**Class** **:** L2BC **Lecturer** **:** Jude Joseph Lamug Martinez, MCS

**Type of Assignment:** Final Project Report

**Submission Pattern**

**Due Date** **:** 10 June 2024 **Submission Date** **:** 10 June 2024

The assignment should meet the below requirements.

1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer's instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.
4. Compiled pages are firmly stapled.
5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

**Plagiarism/Cheating**

BiNus International seriously regards all forms of plagiarism, cheating, and collusion as academic offenses which may result in severe penalties, including loss/drop of marks, course/class discontinuity, and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

**Declaration of Originality**

By signing this assignment, I understand, accept, and consent to BiNus International terms and policy on plagiarism. Herewith I declare that the work contained in this assignment is my own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

Signature of Student:

Kevin Jonathan

# A. Background

The project guidelines encouraged me to step out of my comfort zone and explore beyond our class lessons. I wanted to try to make something that I always wanted to make ever since semester one of university and that something is a rhythm game. I took inspiration from the game called guitar hero and osu mania and I tried to recreate my own version of the game, but only consisting of one song to play.

# B. Project Specification

## 1. Game Name

- Rectangle Jam

## 2. Game Concept

- "Rectangle Jam" is a rhythm based game inspired by "osu!mania" and "Guitar Hero", where players need to hit notes in time with the music. Players earn higher scores by maintaining a high combo streak and hitting notes with precise timing.
- The game features two types of hit objects or beats that players need to hit on time: notes and sliders. Notes are single, discrete beats that players must hit when they reach the designated hit area near the bottom of the screen. Sliders, on the other hand, are elongated beats that players must hold for a certain duration as they slide across the hit area.

## 3. Game Flow Summary

- Notes and sliders will descend from the top of the panel to the bottom of the panel with consistent speed, but different timings to complement and match with the song that was chosen. The player needs to hit these notes and hold the sliders as they align with the hit area, earning points based on timing accuracy.

## 4. Game Objective

- The main objective of the game "Rectangle Jam" is to achieve the highest score possible by accurately hitting notes and sliders in sync with the music. Players can achieve a higher score by keeping their combo streak which will act as their score multiplier and greatly boost their scores. The players overall need to have a good accuracy and combo streak if they want to achieve the highest score possible at this game.

## 5. Game Display

- The game's graphics have a simple style, primarily created using the Graphics2D class in Java. This class is used to draw the hit area, notes, sliders, score, combo, and the total accuracy.

## 6. Game Mechanics

- The game panel contains four vertical tracks, each corresponding to specific keys on the keyboard (D,F,J,K). When the program starts, notes and sliders descend from the top of the screen to the bottom, where players must press the specific keys in sync with the music to get a high score.

## 7. Game Physics

- Notes and sliders descend from the top of the screen to the hit area at a constant speed. The speed is measured in pixels per second, adjusted dynamically to sync with the music. Notes and sliders have a specific hit window based on timing accuracy. Perfect is when it is hit very near the designated hit area. When a certain key is pressed, it will check the designated track it was assigned to, to see if there exist any notes or sliders in that track. If there is a note or slider, it will check on the current height of the first note or slider that spawned and compare it with some preassigned values to see whether it's a "perfect hit","good hit","okay hit", or a miss. There will be visual and audio feedback which indicate the accuracy of the hit.

## 8. Game Input

- D Key
  - **Pressed:** Changes the rectangle color, plays a sound, updates score and combo for the first lane, and marks the key is pressed.
  - **Released:** Resets the rectangle color, updates score for sliders, and marks the key as released.
- F Key
  - **Pressed:** Changes the rectangle color, plays a sound, updates score and combo for the second lane, and marks the key is pressed.
  - **Released:** Resets the rectangle color, updates score for sliders, and marks the key as released.
- J Key
  - **Pressed:** Changes the rectangle color, plays a sound, updates score and combo for the third lane, and marks the key is pressed.
  - **Released:** Resets the rectangle color, updates score for sliders, and marks the key as released.
- K Key
  - Pressed: Changes the rectangle color, plays a sound, updates score and combo for the fourth lane, and marks the key is pressed. Released: Resets the rectangle color, updates score for sliders, and marks the key as released.
- Escape Key - Toggles the game's pause state.
- Left Mouse Button - For interacting with the buttons in the game’s pause or end state. To resume, restart, or to quit the program.

## 9. Game Output

- Notes and sliders - will spawn at the top of the panel and starts descending in the assigned track
- Track - Multiple vertical tracks where notes and sliders descend. It is made up of lines drawn with the ‘Graphics2d’ class.
- Hit area - Consist of four different colored rectangles which have the keybind text on them. They are located at the bottom of the panel.
- Player’s score
- Player’s combo
- Player’s accuracy
- Perfect hit image (assets/hit30.png)
- Good hit image (assets/hit10.png)

- Okay hit image (assets/hit5.png)
- Miss image (assets/miss.png)
- Pause menu
- End menu
- Sway to my beat in cosmos song (assets/swayToMyBeatInCosmos.wav)
- Hit sound effect (hitsound.wav)
- Combo break sound effect (assets/combobreak.wav)

## 10. Game Libraries/Modules

- Java Swing - Provides the framework for the game's user interface, including the main game window, panels, and buttons. Classes like JFrame and JPanel are used to create and manage the game's GUI components.
- Graphics2D - Used to draw most elements of the game, including the vertical tracks, notes, and sliders. The Graphics2D class handles drawing operations, from rendering the player's interface to animating the notes and sliders.
- Java Sound API: Part of the Java Standard Edition, used for playing sound effects and background music.
- Java AWT Event Handling - This module facilitates the management of user interactions within the game.

## 11. Game Files

- GameFrame.java - contains the main file to run the game
- GamePanel.java - This class represents the main gameplay area where most of the game's action takes place. It contains the logic for updating and rendering game elements, such as notes, sliders, and player interactions.
- GameDrawer.java - This class is responsible for drawing various graphical elements of the game onto the game panel.
- BeatInfo.java - This class stores the info for the beats such as the timing it will spawn, the lane it will spawn in, and what type of beat it is (notes / sliders).
- Music.java - This class handles the loading and playback of music tracks within the game. It provides functionality for playing background music, as well as managing volume levels and playback controls.
- Beat.java - This class is an interface which contains the methods for the two types of beats which are notes and sliders.
- Notes.java - This class represents individual notes that appear on the game panel for players to interact with. It manages the appearance, movement, and behavior of notes during gameplay.
- Sliders.java - This class represents the sliders that appear on the game panel. It manages the appearance, movement, and behavior of notes during gameplay.

## 12. Game Images

- Images were taken from a skin called “Aristia(Edit)” from the game “Osu” and were slightly modified. (https://skins.osuck.net/skins/485?v=0)
- Perfect Hit image

- Good Hit image

- Okay Hit image

- Miss Hit image

## 13. Game Sounds

- Combo break sound effect

  (https://www.mediafire.com/file/kwvidor6h6su9lf/raiden_shogun.osk/file)

- Hit sound effect

  (https://osu.ppy.sh/beatmapsets/2178331#osu/4600500)

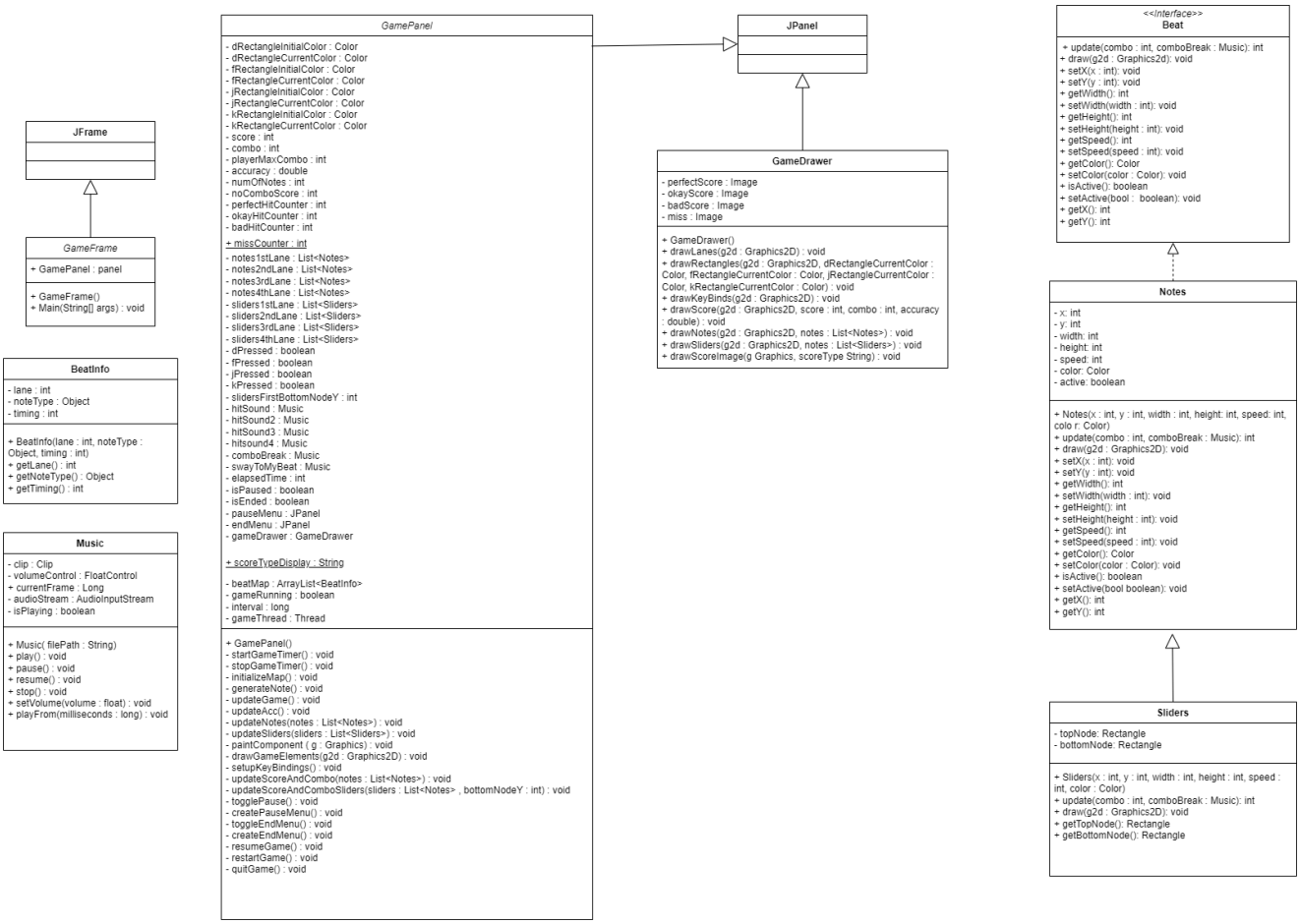
- Sway To My Beat in Cosmos / Main game song

  (https://youtu.be/QbPtrnmGlZ8?si=dGo16IQMdKkBznf1)

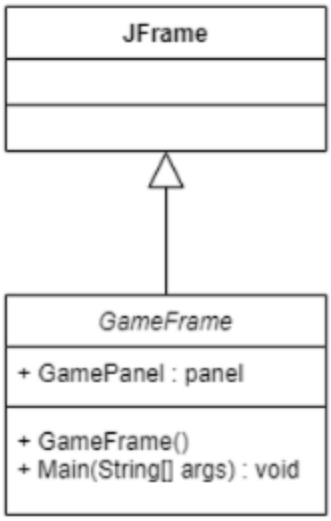
### 14. Game Fonts

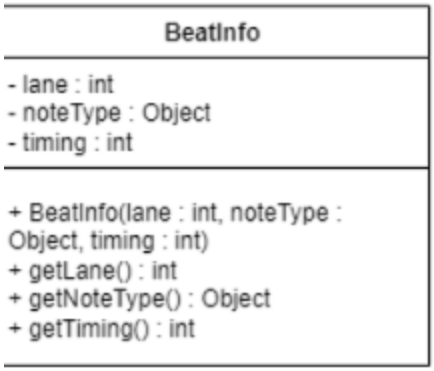
- Arial Font

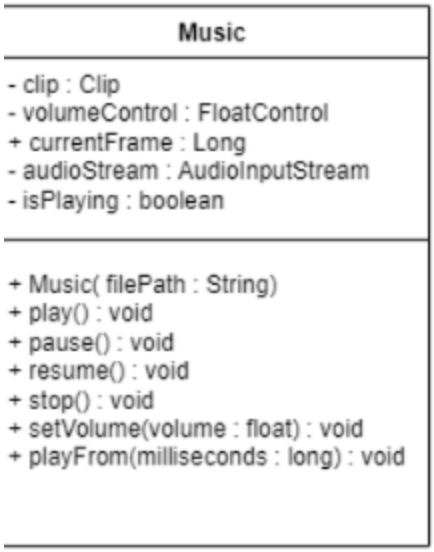
# C. Class Diagram

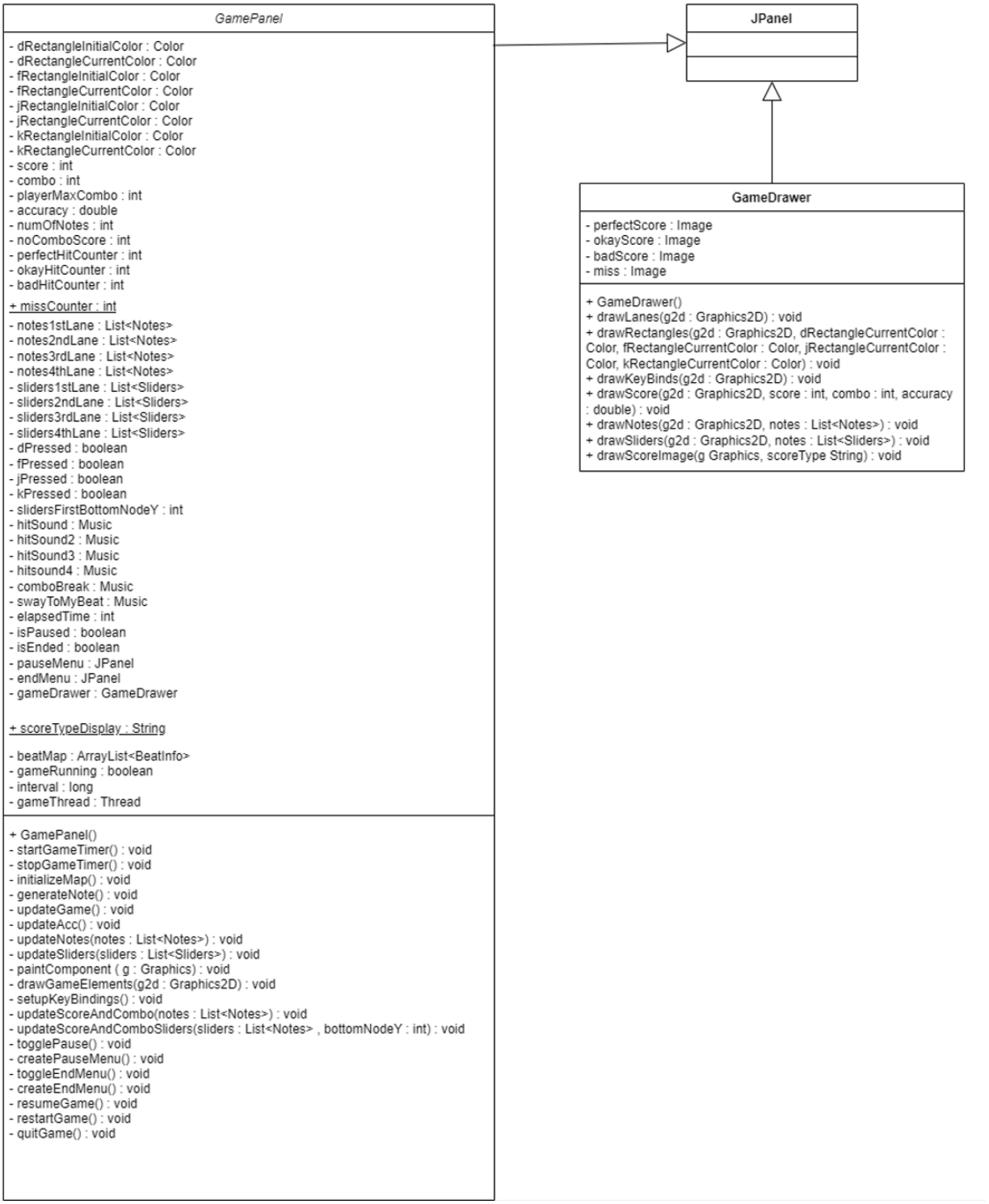


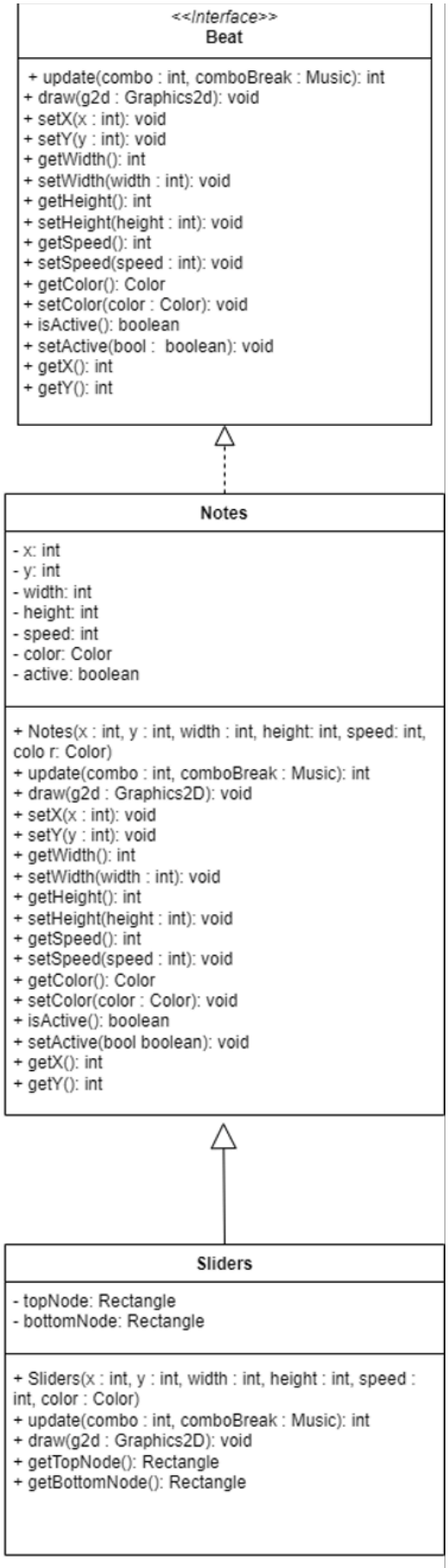

More zoomed in class diagram:


JFrame
GameFrame
+ GamePanel : panel
+ GameFrame()
+ Main(String[] args) : void


BeatInfo
- lane : int
- noteType : Object
- timing : int
+ BeatInfo(lane : int, noteType : Object, timing : int)
+ getLane() : int
+ getNoteType() : Object
+ getTiming() : int


Music
- clip : Clip
- volumeControl : FloatControl
+ currentFrame : Long
- audioStream : AudioInputStream
- isPlaying : boolean
+ Music( filePath : String)
+ play() : void
+ pause() : void
+ resume() : void
+ stop() : void
+ setVolume(volume : float) : void
+ playFrom(milliseconds : long) : void


GamePanel
- dRectangleInitialColor : Color
- dRectangleCurrentColor : Color
- fRectangleInitialColor : Color
- fRectangleCurrentColor : Color
- jRectangleInitialColor : Color
- jRectangleCurrentColor : Color
- kRectangleInitialColor : Color
- kRectangleCurrentColor : Color
- score : int
- combo : int
- playerMaxCombo : int
- accuracy : double
- numOfNotes : int
- noComboScore : int
- perfectHitCounter : int
- okayHitCounter : int
- badHitCounter : int
+ missCounter : int
- notes1stLane : List<Notes>
- notes2ndLane : List<Notes>
- notes3rdLane : List<Notes>
- notes4thLane : List<Notes>
- sliders1stLane : List<Sliders>
- sliders2ndLane : List<Sliders>
- sliders3rdLane : List<Sliders>
- sliders4thLane : List<Sliders>
- dPressed : boolean
- fPressed : boolean
- jPressed : boolean
- kPressed : boolean
- slidersFirstBottomNodeY : int
- hitSound : Music
- hitSound2 : Music
- hitSound3 : Music
- hitsound4 : Music
- comboBreak : Music
- swayToMyBeat : Music
- elapsedTime : int
- isPaused : boolean
- isEnded : boolean
- pauseMenu : JPanel
- endMenu : JPanel
- gameDrawer : GameDrawer
+ scoreTypeDisplay : String
- beatMap : ArrayList<BeatInfo>
- gameRunning : boolean
- interval : long
- gameThread : Thread
+ GamePanel()
- startGameTimer() : void
- stopGameTimer() : void
- initializeMap() : void
- generateNote() : void
- updateGame() : void
- updateAcc() : void
- updateNotes(notes : List<Notes>) : void
- updateSliders(sliders : List<Sliders>) : void
- paintComponent ( g : Graphics) : void
- drawGameElements(g2d : Graphics2D) : void
- setupKeyBindings() : void
- updateScoreAndCombo(notes : List<Notes>) : void
- updateScoreAndComboSliders(sliders : List<Notes> , bottomNodeY : int) : void
- togglePause() : void
- createPauseMenu() : void
- toggleEndMenu() : void
- createEndMenu() : void
- resumeGame() : void
- restartGame() : void
- quitGame() : void
JPanel
GameDrawer
- perfectScore : Image
- okayScore : Image
- badScore : Image
- miss : Image
+ GameDrawer()
+ drawLanes(g2d : Graphics2D) : void
+ drawRectangles(g2d : Graphics2D, dRectangleCurrentColor : Color, fRectangleCurrentColor : Color, jRectangleCurrentColor : Color, kRectangleCurrentColor : Color) : void
+ drawKeyBinds(g2d : Graphics2D) : void
+ drawScore(g2d : Graphics2D, score : int, combo : int, accuracy : double) : void
+ drawNotes(g2d : Graphics2D, notes : List<Notes>) : void
+ drawSliders(g2d : Graphics2D, notes : List<Sliders>) : void
+ drawScoreImage(g Graphics, scoreType String) : void


<<Interface>>
Beat
+ update(combo : int, comboBreak : Music): int
+ draw(g2d : Graphics2d): void
+ setX(x : int): void
+ setY(y : int): void
+ getWidth(): int
+ setWidth(width : int): void
+ getHeight(): int
+ setHeight(height : int): void
+ getSpeed(): int
+ setSpeed(speed : int): void
+ getColor(): Color
+ setColor(color : Color): void
+ isActive(): boolean
+ setActive(bool : boolean): void
+ getX(): int
+ getY(): int
Notes
- x: int
- y: int
- width: int
- height: int
- speed: int
- color: Color
- active: boolean
+ Notes(x : int, y : int, width : int, height: int, speed: int, colo r: Color)
+ update(combo : int, comboBreak : Music): int
+ draw(g2d : Graphics2D): void
+ setX(x : int): void
+ setY(y : int): void
+ getWidth(): int
+ setWidth(width : int): void
+ getHeight(): int
+ setHeight(height : int): void
+ getSpeed(): int
+ setSpeed(speed : int): void
+ getColor(): Color
+ setColor(color : Color): void
+ isActive(): boolean
+ setActive(bool boolean): void
+ getX(): int
+ getY(): int
Sliders
- topNode: Rectangle
- bottomNode: Rectangle
+ Sliders(x : int, y : int, width : int, height : int, speed : int, color : Color)
+ update(combo : int, comboBreak : Music): int
+ draw(g2d : Graphics2D): void
+ getTopNode(): Rectangle
+ getBottomNode(): Rectangle

# D. Essential Algorithms

1. Setting up the main window

```
public class GameFrame extends JFrame {
    3 usages
    public GamePanel panel;
    1 usage
    GameFrame(){
        panel = new GamePanel();
        panel.requestFocusInWindow();
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        this.setSize( width: 600, height: 800);
        this.setTitle("Rectangle Jam");
        setResizable(false);
        this.add(panel);
        this.pack();
        // sets the window in the middle of the screen
        this.setLocationRelativeTo(null);
        this.setVisible(true);
    }

    public static void main(String[] args) {
        new GameFrame();
    }
}
```

This class is responsible for setting up the main window of my game. It will first create an instance of 'GamePanel' which contains the game's drawing and logic. It will then request focus for the 'GamePanel' panel, ensuring it can receive keyboard input. It sets the size of the window with a width and height of 600 and 800 pixels and it cannot be resizable. The title of the window is "Rectangle Jam". It also adds the 'GamePanel' panel into the window.

2. Drawing the tracks and hit area

```
// Draw the vertical lanes on the screen
1 usage
public void drawLanes(Graphics2D g2d) {
    g2d.drawLine( x1: 100,  y1: 0,  x2: 100,  y2: 800);
    g2d.drawLine( x1: 200,  y1: 0,  x2: 200,  y2: 800);
    g2d.drawLine( x1: 300,  y1: 0,  x2: 300,  y2: 800);
    g2d.drawLine( x1: 400,  y1: 0,  x2: 400,  y2: 800);
    g2d.drawLine( x1: 500,  y1: 0,  x2: 500,  y2: 800);
}
```

g2d.drawLine(x1, y1, x2, y2): This method draws a line between two points (x1, y1) and (x2, y2). For example g2d.drawLine(100,0,100,800), draws a vertical line from the top of the screen (y = 0) to the bottom (y = 800) at x = 100. These lines are drawn on the screen to visually separate the four lanes where the notes and sliders can spawn in.

```
// Draw the rectangles representing the keys
1 usage
public void drawRectangles(Graphics2D g2d, Color dRectangleCurrentColor, Color fRectangleCurrentColor, Color jRectangleCurrentColor, Color kRectangleCurrentColor) {
    g2d.setPaint(dRectangleCurrentColor);
    g2d.fillRect( x: 101,  y: 700,  width: 98,  height: 100);
    g2d.setPaint(fRectangleCurrentColor);
    g2d.fillRect( x: 201,  y: 700,  width: 98,  height: 100);
    g2d.setPaint(jRectangleCurrentColor);
    g2d.fillRect( x: 301,  y: 700,  width: 98,  height: 100);
    g2d.setPaint(kRectangleCurrentColor);
    g2d.fillRect( x: 401,  y: 700,  width: 98,  height: 100);
}
```

The ‘drawRectangles’ method is responsible to visually show when the players need to hit the notes and sliders. These rectangles are hit boxes for the keys 'D', 'F', 'J', and 'K'. Each rectangle is filled with a specified color, which allows the players to visually differentiate the keys and give feedback when the key is pressed.

```
// Draw the key binds (D, F, J, K) above their respective rectangles
1 usage
public void drawKeyBinds(Graphics2D g2d) {
    g2d.setPaint(Color.black);
    g2d.setFont(new Font( name: "Arial", Font.BOLD, size: 50));
    g2d.drawString( str: "D", x: 135, y: 770);
    g2d.drawString( str: "F", x: 235, y: 770);
    g2d.drawString( str: "J", x: 335, y: 770);
    g2d.drawString( str: "K", x: 435, y: 770);
}
```

The ‘drawKeyBinds’ method is responsible for drawing the strings to show which key corresponds to which lane. It sets the text color to black, font to ‘Arial’, font to bold, and font size to 50. Then it draws the string “D”,”F”,”J”, and “K” on the middle of each track or lane.

## 3. Game Timer

First we start with the start game timer method.

```
// Method to start the game timer
2 usages
private synchronized void startGameTimer() {
```

The synchronized Keyword ensures that the method is thread-safe, preventing multiple threads from executing it simultaneously.

```
// If the timer is already running, stop it first
if (gameRunning) {
    stopGameTimer();
}

// Reset elapsed time
elapsedTime = 0;

// Start the game loop in a separate thread to avoid blocking the main thread
gameRunning = true;
```

The method will first check if ‘gameRunning’ is true, if it is true then it will execute the ‘stopGameTimer’ method. It will then reset ‘elapsedTime’ to zero and set ‘gameRunning’ equals to true.

```
gameThread = new Thread(() -> {
```

Starts a new thread to handle the game loop, ensuring it doesn't block the main thread.

```
while (gameRunning && !Thread.currentThread().isInterrupted()) {
    long currentTime = System.nanoTime();
    long deltaTime = currentTime - lastTime;
```

Initializes ‘lastTime’ with the current time in nanoseconds. Continuously runs the loop while gameRunning is true and the thread is not interrupted.

```
// If the elapsed time is greater than or equal to the interval
if (deltaTime >= interval) {
    lastTime += interval;

    // Play background music when elapsed time reaches 300 milliseconds
    if (elapsedTime == 350) {
        swayToMyBeat.play();
    }
    // Stop background music and end the game when elapsed time exceeds 43400 milliseconds
    else if (elapsedTime >= 43400) {
        swayToMyBeat.stop();
        createEndMenu();
        toggleEndMenu();
        gameRunning = false; // Stop the loop
        break;
    }

    // If the game is not paused, update the game state
    if (!isPaused) {
        // Generate a new note
        generateNote();

        // Update game state
        updateGame();

        // Increment the elapsed time by 10 milliseconds (10,000,000 nanoseconds)
        elapsedTime += 10;

        // Repaint the game panel on the EDT
        SwingUtilities.invokeLater(() -> repaint());
    }
} else {
    // Sleep for a short period to prevent CPU hogging
    try {
        Thread.sleep( millis: 1);
    } catch (InterruptedException e) {
        // If the thread is interrupted during sleep, exit the loop
        Thread.currentThread().interrupt();
    }
}
```

It first checks if the ‘deltaTime’ is greater than ‘interval’ which was set to 15.85 ms. If it is equal to or over the interval time then it will execute the following code. It will play the game’s music if ‘elapsedTime’ is equal to 350 and stop the game’s music when ‘elapsedTime’ equals 43400. When the ‘elapsedTime’ reaches 43400, it means the game has ended, so it will run the method to create the end menu panel, toggle the end menu panel to be visible on the screen and set ‘gameRunning’ to false. It will then check for the ‘isPaused’ variable which checks if the game is currently paused or not. If the game is not paused, then it will run the ‘generateNote’ method and ‘updateGame’ method. It will also increment the elapsed time by 10 milliseconds and repaint the game panel. It will sleep for 1 millisecond to prevent CPU hogging when not enough time has passed to meet the interval requirement.

```
2 usages
private synchronized void stopGameTimer() {
    // Set gameRunning to false to stop the loop
    gameRunning = false;
    if (gameThread != null) {
        gameThread.interrupt();
        try {
            gameThread.join(); // Wait for the thread to finish
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        gameThread = null;
    }
}
```

The ‘stopGameTimer’ method sets the ‘gameRunning’ variable to false. It will check if ‘gameThread’ is not equal to null, if the ‘gameThread’ exists, it interrupts the thread. This line waits for the game thread to finish executing before continuing. The join() method blocks the current thread until the thread it's called on (in this case, gameThread) terminates. If an InterruptedException occurs while waiting, it's caught, and a stack trace is printed. This is a standard error handling practice when dealing with thread synchronization. Lastly, it will set ‘gameThread’ equals to null.

## 4. Making the beatmap

```
// Method to initialize the maps by adding the beats in order into an arraylist.
2 usages
private void initializeMap() {
    beatMap = new ArrayList<>();
    int speed = 10;
    beatMap.add(new BeatInfo( lane: 1, new Sliders( x: 101,  y: -400,  width: 98,  height: 400, speed, Color.PINK),  timing: 0));
    beatMap.add(new BeatInfo( lane: 4, new Sliders( x: 401,  y: -400,  width: 98,  height: 400, speed, Color.PINK),  timing: 800));
    beatMap.add(new BeatInfo( lane: 3, new Sliders( x: 301,  y: -400,  width: 98,  height: 400, speed, Color.PINK),  timing: 1600));
    beatMap.add(new BeatInfo( lane: 2, new Sliders( x: 201,  y: -400,  width: 98,  height: 400, speed, Color.PINK),  timing: 1600));
    beatMap.add(new BeatInfo( lane: 1, new Sliders( x: 101,  y: -400,  width: 98,  height: 400, speed, Color.PINK),  timing: 2400));
```

The ‘initializeMap’ method is responsible for setting up a list of beats for the game. Firstly, it will initialize an Arraylist called ‘beatMap’ to store instances of ‘BeatInfo’. It also defines an integer variable named ‘speed’ which has a value of 10. It adds various beats to the beatMap ArrayList. Each beat is represented by a BeatInfo object, which contains information about the beat's type, position, and timing. The beats are added in a specific order, with each beat having its own timing (specified in milliseconds) and characteristics. Beats can be of two types: sliders and notes.

## 5. Generating Notes

```
// Method to generate notes based on the current elapsed time
1 usage
private void generateNote() {
    // Iterate through the beat map to find notes scheduled for the current elapsed time
    for (BeatInfo beatInfo : beatMap) {
        // Check if the timing of the beatInfo matches the current elapsed time
        if (beatInfo.getTiming() == elapsedTime) {
            // Get the lane of the beatInfo
            int lane = beatInfo.getLane();
```

The ‘generateNote’ method is responsible for taking the ArrayList ‘beatMap’, which stores instances of ‘BeatInfo’, and organizes the beats to go to their specific lane when it reaches the right elapsed time. It first iterates over every item in the list and gets the timing of each beat, if the timing is equal to ‘elapsedTime’ then it will get what lane it's supposed to spawn at (1,2,3, or 4).

```
// Determine the type of note or slider and add it to the corresponding lane list
switch (lane) {
    case 1:
        if (beatInfo.getNoteType() instanceof Sliders) {
            sliders1stLane.add((Sliders) beatInfo.getNoteType());
        } else if (beatInfo.getNoteType() instanceof Notes) {
            notes1stLane.add((Notes) beatInfo.getNoteType());
        }
        break;
    case 2:
        if (beatInfo.getNoteType() instanceof Sliders) {
            sliders2ndLane.add((Sliders) beatInfo.getNoteType());
        } else if (beatInfo.getNoteType() instanceof Notes) {
            notes2ndLane.add((Notes) beatInfo.getNoteType());
        }
        break;
```

```
                case 3:
                    if (beatInfo.getNoteType() instanceof Sliders) {
                        sliders3rdLane.add((Sliders) beatInfo.getNoteType());
                    } else if (beatInfo.getNoteType() instanceof Notes) {
                        notes3rdLane.add((Notes) beatInfo.getNoteType());
                    }
                    break;
                case 4:
                    if (beatInfo.getNoteType() instanceof Sliders) {
                        sliders4thLane.add((Sliders) beatInfo.getNoteType());
                    } else if (beatInfo.getNoteType() instanceof Notes) {
                        notes4thLane.add((Notes) beatInfo.getNoteType());
                    }
                    break;
            }
        }
    }
}
```

Based on the lane, it determines whether the beat is a slider or a note, and adds it to the corresponding list for that lane (sliders1stLane, notes1stLane, sliders2ndLane, notes2ndLane, and so on). It repeats this process for each beat in the beatMap.

```java
// Method to update active notes and remove inactive ones
4 usages
private void updateNotes(List<Notes> notes) {
    // Iterate through the list of notes
    Iterator<Notes> iterator = notes.iterator();
    while (iterator.hasNext()) {
        Notes note = iterator.next();
        // Update active notes
        if (note.isActive()) {
            combo = note.update(combo, comboBreak);
        } else {
            // Remove inactive notes and update accuracy
            iterator.remove();
            numOfNotes += 1;
            updateAcc();
        }
    }
}
```

```java
// Method to update active sliders and remove inactive ones
4 usages
private void updateSliders(List<Sliders> sliders) {
    // Iterate through the list of sliders
    Iterator<Sliders> iterator = sliders.iterator();
    while (iterator.hasNext()) {
        Sliders slider = iterator.next();
        // Update active sliders
        if (slider.isActive()) {
            combo = slider.update(combo, comboBreak);
        } else {
            // Remove inactive sliders and update accuracy
            iterator.remove();
            numOfNotes += 1;
            updateAcc();
        }
    }
}
```

The ‘updateSliders’ and ‘updateNotes’ methods are responsible for making the beats descend from the top of the screen to the bottom and removing the beats if it got hit or went outside of the screen. Both of the methods initialize an iterator to transverse the list of notes or sliders. It goes through each note or slider and checks whether it has an active status or not. If it does have an active status, it will update the current slider or note by running the corresponding update methods. Otherwise, it will remove the current note or slider from the list of active notes or sliders. It will also increment the number of notes so far that have spawned by 1 and also run the ‘updateAcc’ method.

```java
/**
 * Update the position of the note to go down the screen based on the speed it was set.
 * If the note reaches the bottom of the screen, it becomes inactive and triggers a miss event.
 * If the combo is not zero, it resets the combo and plays the combo break sound.
 *
 * @param combo         The current amount of combos the player has.
 * @param comboBreak    comboBreak is the sound played when combo breaks.
 * @return combo        The updated combo count.
 */
2 usages  1 override
@Override
public int update(int combo, Music comboBreak) {
    y += speed;
    if (y > 730) {
        active = false;
        GamePanel.scoreTypeDisplay = "miss";
        GamePanel.missCounter += 1;
        if(combo != 0) {
            combo = 0;
            comboBreak.play();
        }
    }
    return combo;
}
```

```java
/**
 * Updates the position of the slider to move down the screen based on the speed.
 * If the top node reaches the bottom of the screen, the slider becomes inactive and triggers a miss event.
 * The positions of the top and bottom nodes are updated to match the new position of the slider.
 *
 * @param combo      The current combo count of the player.
 * @param comboBreak The sound played when the combo breaks.
 * @return The updated combo count.
 */
2 usages
@Override
public int update(int combo, Music comboBreak) {
    y += speed;
    if (topNode.y > 730) {
        active = false;
        GamePanel.scoreTypeDisplay = "miss";
        GamePanel.missCounter += 1;
        if(combo != 0) {
            combo = 0;
            comboBreak.play();
        }
    }
    topNode.setLocation(x, y);
    bottomNode.setLocation(x,  y: y + height - 10);
    return combo;
}
```

The two images above show the update methods from both sliders and note class. The update method for the note class, updates the y position of the note by the defined speed, making the note go down from the top of the screen to the bottom. It then checks if the y position of the note is above the value of 730. If it is above that value, then it will set the active status of the notes to false. It will also display a miss image and increment the miss counter by 1. Next, it will check if the combo is not equal to zero, if it is not equal to zero then it will set the combo equal to zero and play the combo break sound effect. Lastly it will return the combo amount. The slider update method is similar to the note method, but

it checks the top node of the slider if it's above 730 instead. Also it updates the position of the top node and bottom node to follow the body of the slider.

## 6. User Input

```
// Method to set up key bindings for game control
1 usage
private void setupKeyBindings() {
    // Key binding for 'D' key press
    getInputMap().put(KeyStroke.getKeyStroke( s: "D"), actionMapKey: "D_Pressed");
    getInputMap().put(KeyStroke.getKeyStroke( s: "released D"), actionMapKey: "D_Released");
    getActionMap().put( key: "D_Pressed", (AbstractAction) (e) -> {
            // Change color of 'D' rectangle
            dRectangleCurrentColor = Color.decode( nm: "#ffa8a3");
            repaint();
            if (!dPressed) {
                // Play hit sound
                hitSound.play();
                if (!sliders1stLane.isEmpty()) {
                    // Update combo and score for notes in the 1st lane
                    slidersFirstBottomNodeY = sliders1stLane.getFirst().getBottomNode().y;
                }
                updateScoreAndCombo(notes1stLane);
                dPressed = true;
            }
    });
    getActionMap().put( key: "D_Released", (AbstractAction) (e) -> {
            // Restore original color of 'D' rectangle
            dRectangleCurrentColor = dRectangleInitialColor;
            if(dPressed){
                updateScoreAndComboSlider(sliders1stLane, slidersFirstBottomNodeY);
            }
            dPressed = false;
            repaint();
    });
```

The ‘setupKeyBindings’ method sets up key bindings for controlling the game. It binds specific actions to key presses and releases for the keys 'D', 'F', 'J', and 'K', which are the main keys for the game input. When a key is pressed, it will slightly alter the colour of the hit boxes to indicate that a key is pressed. When a key is pressed it will first check if it was pressed before. If it wasn't pressed before, it will play the hit sound and check if the current lane contains a slider or not. If it does contain a slider, then it will save the current Y position of the bottom node. Next it will update the score and combo for the notes and set the boolean flag for the corresponding key that is pressed to be true. When a key is released, the hit boxes will return to its original colour, set the boolean flag for the corresponding key that is pressed to be false, and repaint the screen. It also checks if the key was pressed beforehand, and updates the score and combo of the sliders.

```
// Bind Escape key to togglePause action
getInputMap(JComponent.WHEN_IN_FOCUSED_WINDOW).put(KeyStroke.getKeyStroke( s: "ESCAPE"), actionMapKey: "pause");
getActionMap().put( key: "pause", (AbstractAction) (e) -> { togglePause(); });
```

It also binds the escape button for the pause action. When the escape key is pressed, it will run the ‘togglePause’ method which will activate the pause screen and game state.

7. Scoring, Combo, and Accuracy

```
// Method to update score and combo for notes
4 usages
private void updateScoreAndCombo(List<Notes> notes) {
    if (!notes.isEmpty() && notes.getFirst().getY() > 650) {
        notes.getFirst().setActive(false);
        score += (combo != 0) ? 30*combo : 30;
        noComboScore += 30;
        combo += 1;
        scoreTypeDisplay = "perfect";
        perfectHitCounter += 1;
    } else if(!notes.isEmpty() && notes.getFirst().getY() > 625) {
        notes.getFirst().setActive(false);
        score += (combo != 0) ? 10*combo : 10;
        noComboScore += 10;
        combo += 1;
        scoreTypeDisplay = "okay";
        okayHitCounter += 1;
    } else if(!notes.isEmpty() && notes.getFirst().getY() > 600){
        notes.getFirst().setActive(false);
        score += (combo != 0) ? 5*combo : 5;
        noComboScore += 5;
        combo += 1;
        scoreTypeDisplay = "bad";
        badHitCounter += 1;
    }
    if (combo > playerMaxCombo){
        playerMaxCombo = combo ;
    }
}
```

The ‘updateScoreAndCombo’ method is responsible for updating the score and combo of the notes based on when the player presses the corresponding buttons to the lanes. The method first checks if there are any notes and if the Y-coordinate of the first note exceeds certain thresholds. Depending on the Y-coordinate, it classifies the hit as "perfect", "okay", or "bad". Each classification awards different points and updates counters. It also sets the active state of the first note in the list to be false. It increments the counters such as ‘noComboScore’ , ‘combo’, and hits counters by a specific value. The score the player gets also depends on the combo streak they have since the combo acts as a score

multiplier. Lastly, it updates the player's highest combo if the current combo exceeds the player's maximum combo in that one round.

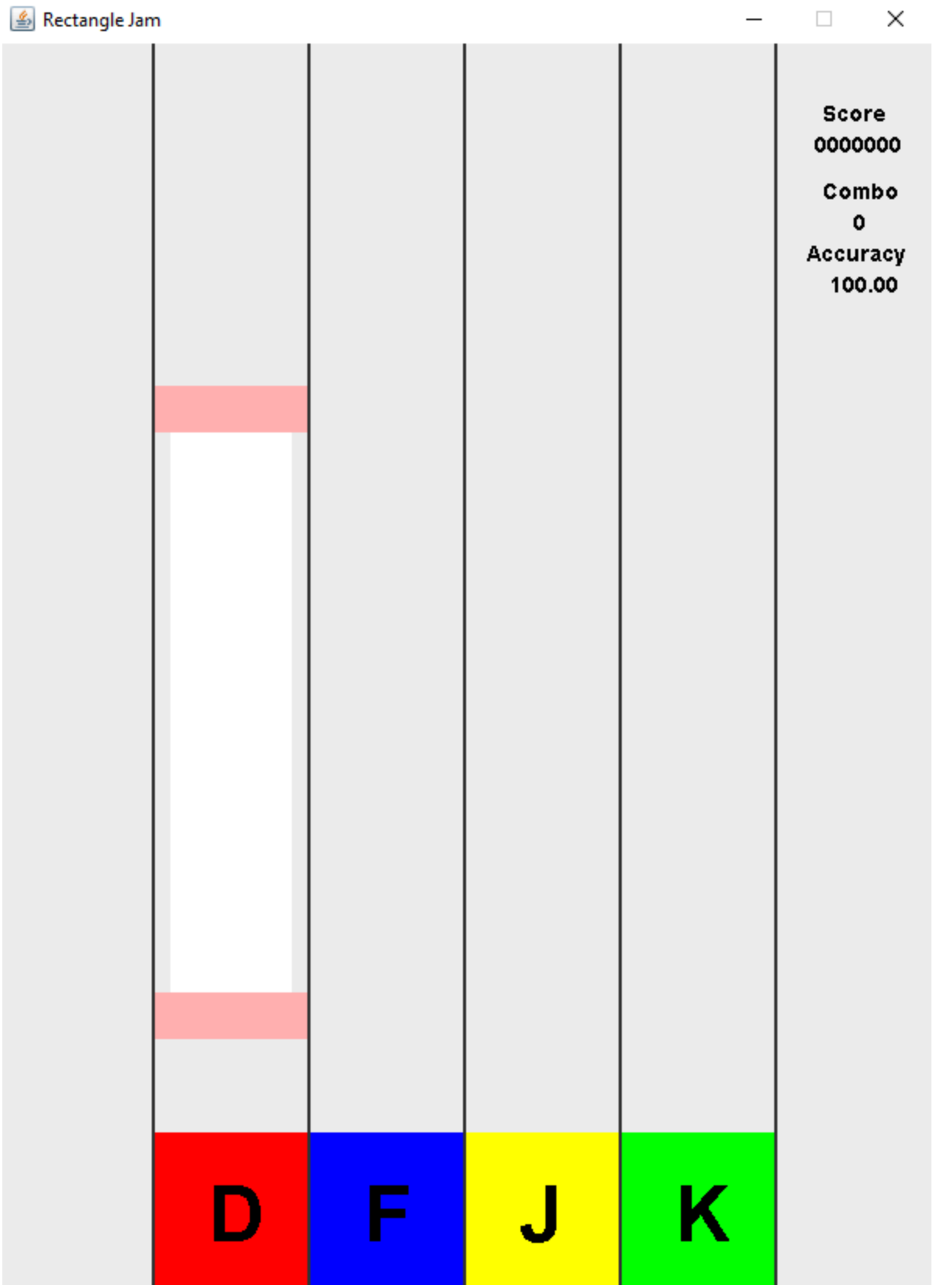
```
// Method to update score and combo for sliders
4 usages
private void updateScoreAndComboSlider(List<Sliders> sliders, int bottomNodeY) {
    if(!sliders.isEmpty()) {
        int sliderLane = sliders.getFirst().getX() / 100;
        int topNodeY = sliders.getFirst().getTopNode().y;
        if (topNodeY > 650 && bottomNodeY < 730 && topNodeY < 730) {
            sliders.getFirst().setActive(false);
            switch (sliderLane){
                case 1:
                    hitSound.play();
                    break;
                case 2:
                    hitSound2.play();
                    break;
                case 3:
                    hitSound3.play();
                    break;
                case 4:
                    hitSound4.play();
                    break;
            }
            score += (combo != 0) ? 30 * combo : 30;
            noComboScore += 30;
            scoreTypeDisplay = "perfect";
            combo += 1;
            perfectHitCounter += 1;
```

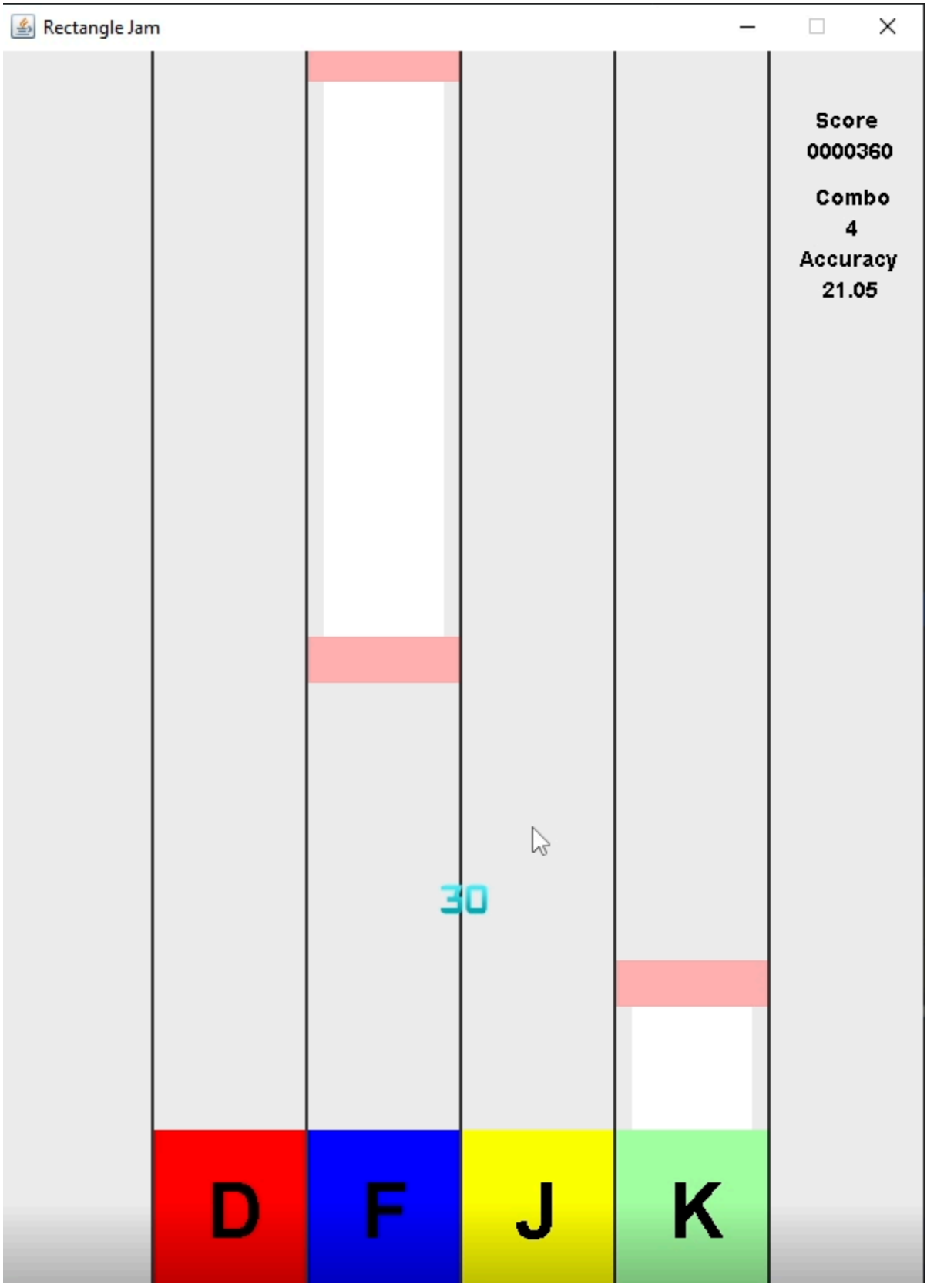
The 'updateScoreAndComboSlider' works similarly with the 'updateScoreAndCombo' method. It first checks if the current lane contains a slider in it. If it does contain a slider, check the Y-coordinates of the slider's top and bottom nodes. Depending on the Y-coordinate range, it classifies the hit as "perfect", "okay", or "bad". If it fulfills any of the conditions, it will mark the first slider as inactive. It will then play a hitsound corresponding to which lane the slider is in. Then it will increase the score by a value that scales with the current combo. It will increment 'noComboScore' with the corresponding score and also increment the combo. Then it will set the 'scoreTypeDisplay'and increments the corresponding hit counter (perfectHitCounter, okayHitCounter, badHitCounter).

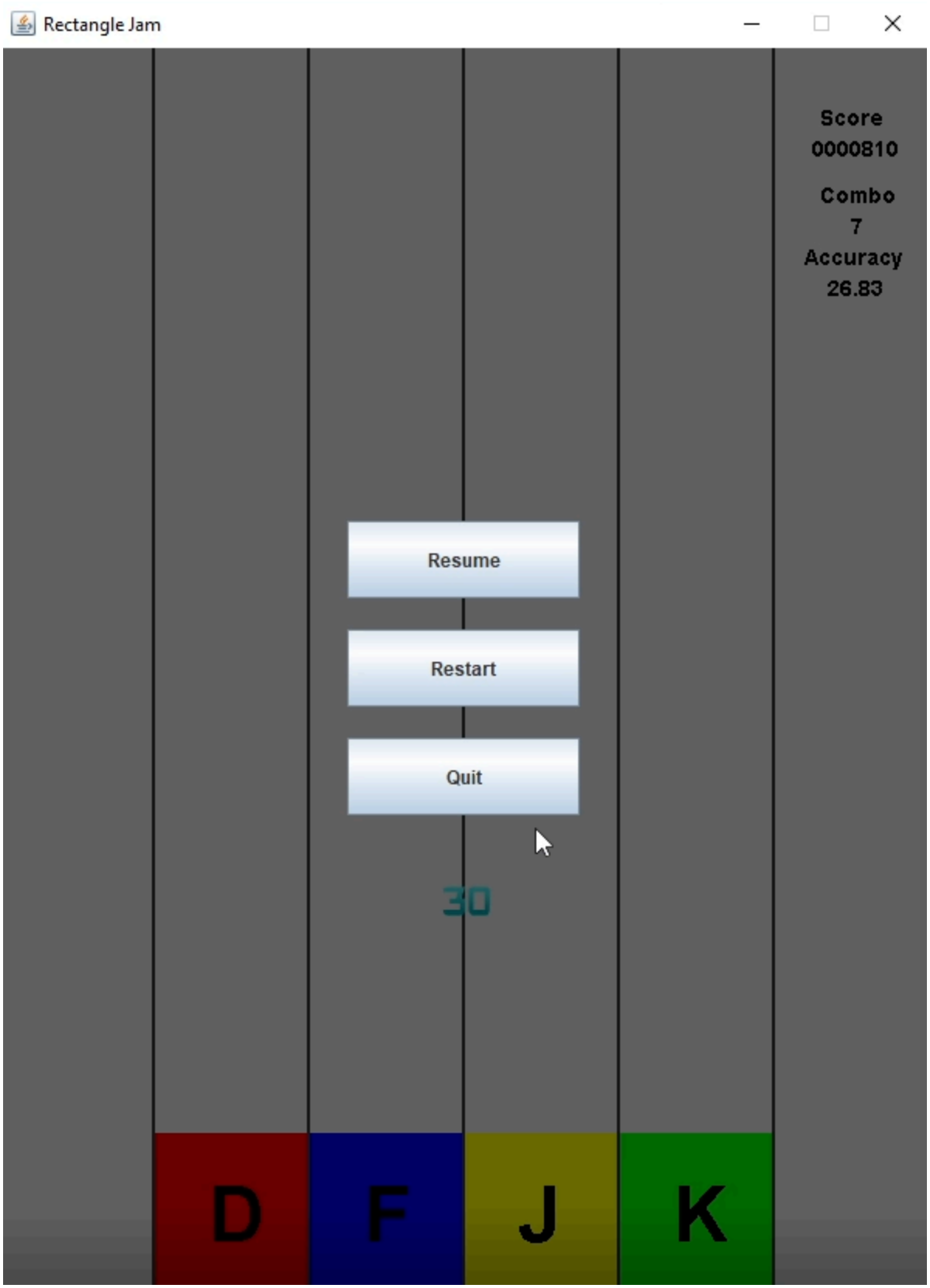
```
// Method to update the accuracy based on the number of correctly hit notes
2 usages
private void updateAcc() {
    // Calculate accuracy if there are notes in the game
    if (numOfNotes != 0) {
        accuracy = ((float) noComboScore / (30 * (float) numOfNotes)) * 100;
    } else {
        // If there are no notes, set accuracy to 100%
        accuracy = 100.00;
    }
}
```

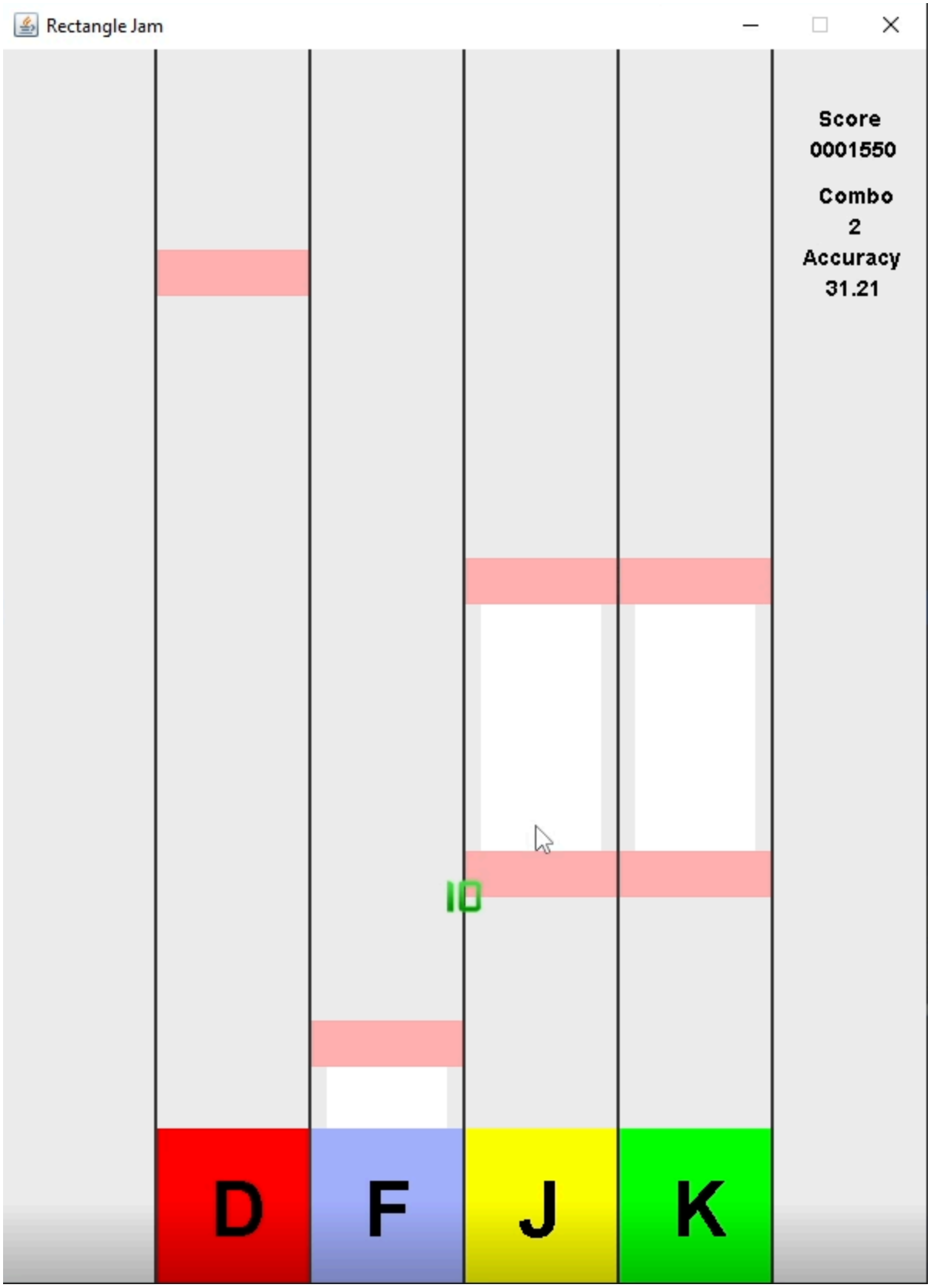
The ‘updateAcc’ is in charge of updating the player’s accuracy of hitting the notes. How it works is that it first checks if the ‘numOfNotes’, which is the amount of notes that has spawned so far on the screen, is not equal to zero. If it is equal to zero then the accuracy is 100. Otherwise, it will count the accuracy by dividing the ‘noComboScore’ which is the total amount of score the player got without the combo multiplier, by the ‘numOfNotes’ multiplied by 30, which is the maximum amount of score the player could get without combo multipliers. Then it will multiply by 100 percent and that would be the accuracy of the player.

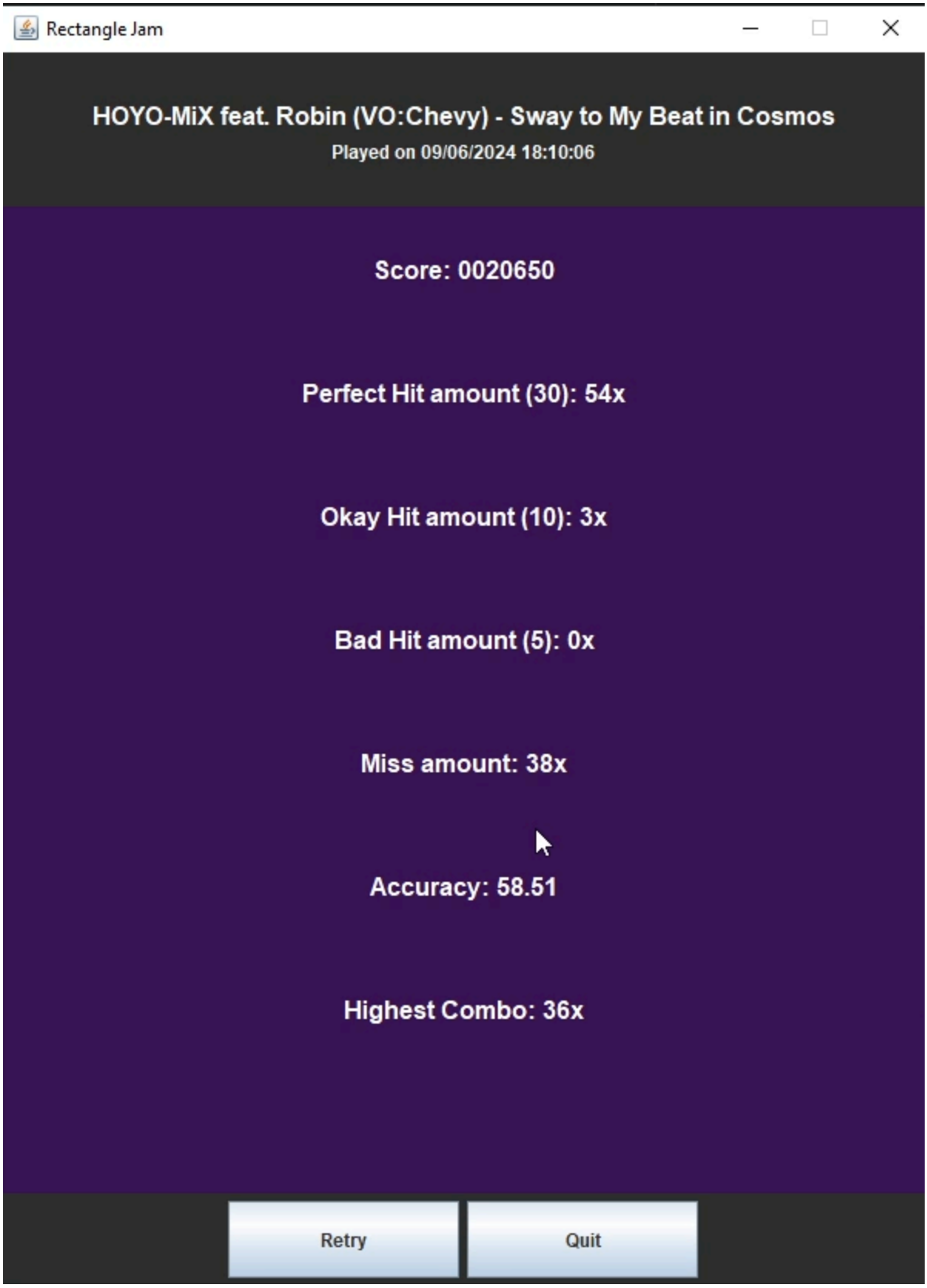
# E. Screenshots of the Game


Rectangle Jam
Score
0000000
Combo
0
Accuracy
100.00
D
F
J
K


Rectangle Jam
Score
0000360
Combo
4
Accuracy
21.05
30
D
F
J
K


Rectangle Jam
Score
0000810
Combo
7
Accuracy
26.83
Resume
Restart
Quit
30
D
F
J
K


Rectangle Jam
Score
0001550
Combo
2
Accuracy
31.21
10
D
F
J
K


Rectangle Jam
HOYO-MiX feat. Robin (VO:Chevy) - Sway to My Beat in Cosmos
Played on 09/06/2024 18:10:06
Score: 0020650
Perfect Hit amount (30): 54x
Okay Hit amount (10): 3x
Bad Hit amount (5): 0x
Miss amount: 38x
Accuracy: 58.51
Highest Combo: 36x
Retry
Quit

# F. Lessons Learned/ Reflection

This project has taught me that time management is a very important skill to have. Making this project when deadlines for university assignments are piling up, and the organization has meetings almost every weekend, was not an easy thing to do. This made me realize that my time management skills suck and if I had managed it better, maybe I would have suffered less. Despite the challenges, making this game was quite fun and challenging. I love playing rhythm games in my free time ever since the pandemic, so I always wondered how they might work. I wanted to make a rhythm game in semester 1 for algorithm and programming course, but I couldn't because of the time constraint, but now since I have more knowledge about programming, I decided to go ahead and try to make my own implementation of a rhythm game. I learned that the Java Timer class is pretty affected by the performance of the computer. I was planning on presenting on the third of June, however, when I tested my game on my laptop monitor alone, I found out that the timing was two times faster compared to while I am playing on my second monitor. That was the reason why making the beatmap was challenging for me too because it felt like the timing was always different every time I added more and more notes and sliders. Doing this project made me learn a bit of Swing and I learned how music in java works, which was way more complicated compared to python.