United International University
Department of Computer Science & Engineering

Street Fighter

2D Fighting Game with Network Multiplayer

Course: Advanced Object Oriented Programming Laboratory

Section: [N]

Semester: [Summer 25]

Presented by:

AL-Farhan	112310262
Shabiha Tasnim	112330915
Miftahul Kabir	112310251

Instructor:
Mr. Rizvan Jawad Ruhan

Submission Date: October 22, 2025

Contents

1. Introduction

The **Street Fighter Game** is a 2D fighting game inspired by the classic arcade game Street Fighter. This project implements a complete combat system with both local and online multiplayer capabilities. The game features two playable characters (Ryu and Ken) with diverse move sets including punches, kicks, jumps, flips, and blocks.

1.1 Project Objectives

- Develop a responsive 2D fighting game with sprite-based animations
- Implement a robust combat system with hitbox detection
- Create both local (same device) and network multiplayer modes
- Build user authentication and leaderboard tracking systems
- Achieve smooth 60 FPS gameplay with synchronized network play

1.2 Project Scope

The game includes user authentication, character and map selection, local and network multiplayer modes, a comprehensive combat system with multiple attack types, and a leaderboard system for competitive play.

2. System Architecture

2.1 Technology Stack

The Street Fighter Game was developed using a combination of modern Java technologies:

- **JavaFX**: For UI rendering, canvas-based graphics, and game scene management
- **MySQL Database**: For user authentication, player statistics, and leaderboard data
- **UDP Networking**: For real-time multiplayer synchronization
- **Java OOP**: Core game logic implemented with object-oriented principles

2.2 Design Patterns

The project follows the MVC (Model-View-Controller) pattern with clear separation between:

- **Frontend**: JavaFX Controllers (Login, Home, ChampSelect, MapSelect, GameScene)
- **Backend**: Game logic (Fighter, CombatSystem, AnimationStateMachine)
- **Database**: MySQL with DatabaseManager for data persistence
- **Network**: UDP client-server architecture for multiplayer

Additional design patterns implemented include:

- **State Pattern**: Animation state machine management
- **Singleton Pattern**: Asset and resource management
- **Observer Pattern**: Network event callbacks

3. Core Game Systems

3.1 User Authentication

Login/Registration: Users create accounts stored in MySQL database with password-based authentication and session management.

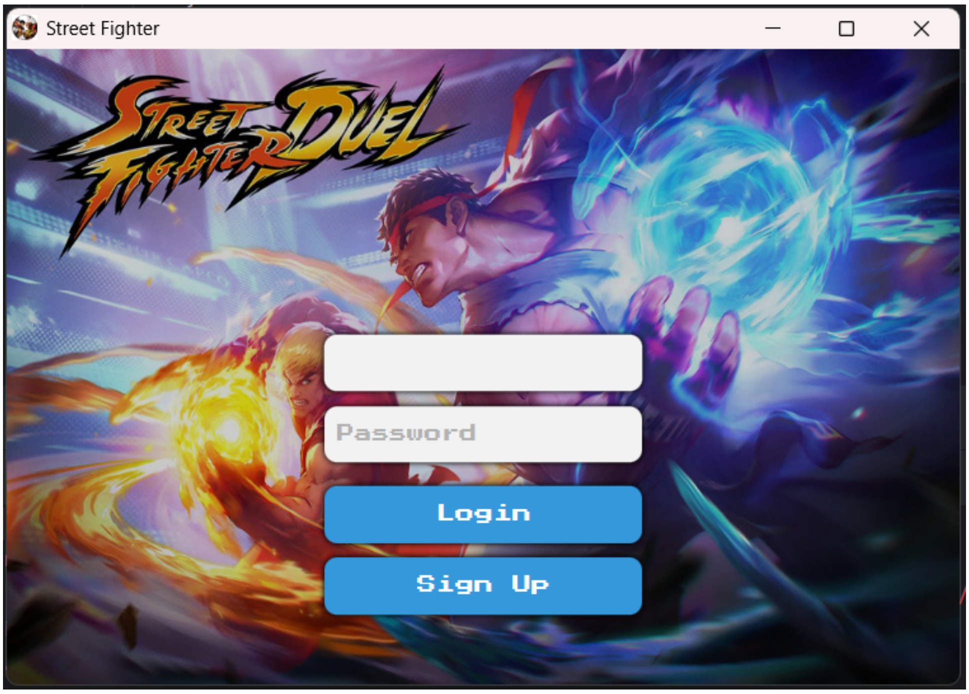


Figure 1: Log In and Sign Up

Implementation: LoginUIController validates credentials through DatabaseManager.loginPlayer() which executes SQL queries against the players table.

3.2 Navigation and Menus

3.2.1 Home Screen

The main menu provides access to all game modes and features:

- **Local Multiplayer**: Two players on the same machine
- **Network Multiplayer**: Players on different machines via WiFi
- **Leaderboard**: View top player rankings
- **Logout**: Return to login screen

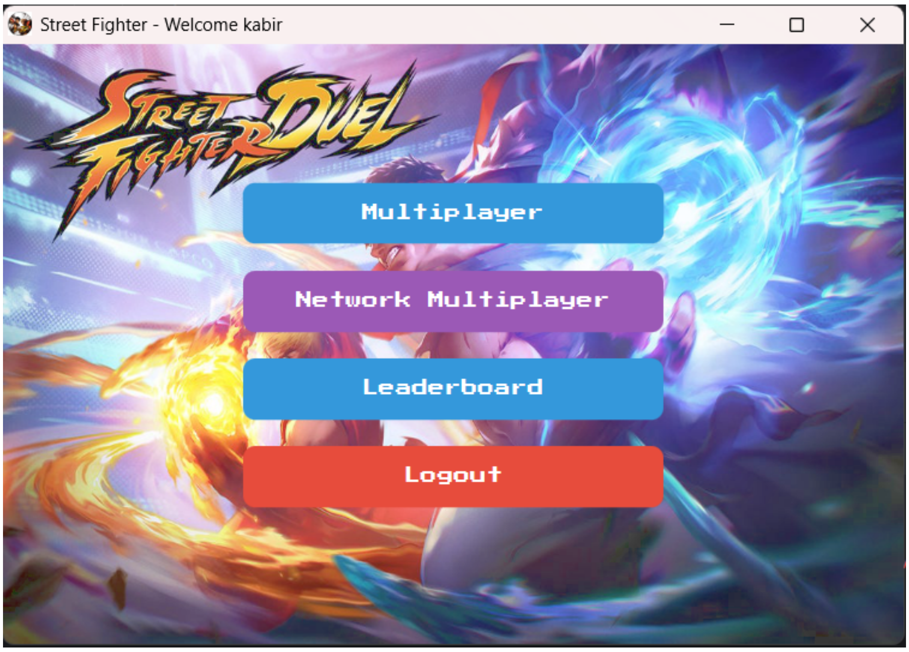


Figure 2: Main menu with game mode options

3.2.2 Character Selection

Players choose between Ryu and Ken, each with 20+ unique animations including idle, walking, jumping, attacking, and hit reactions. Navigation uses keyboard controls (A/D or Arrow keys, Q/Enter to confirm).

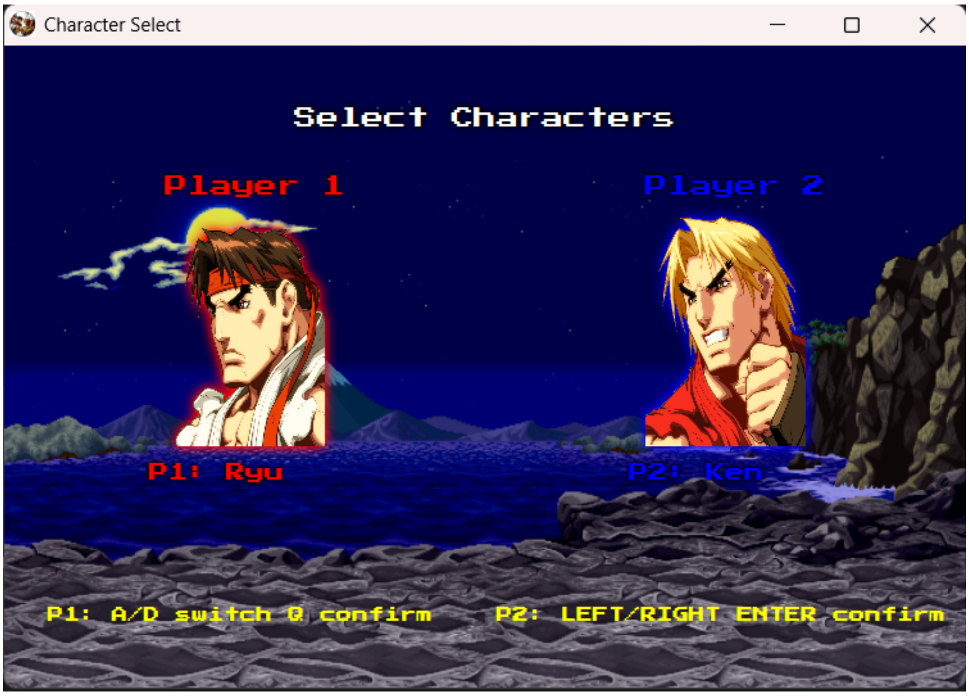


Figure 3: Character selection screen

Implementation: ChampSelectController manages character selection and transitions to map selection.

3.2.3 Map Selection

Six different maps provide visual variety with interactive keyboard-based navigation and preview images.

Figure 4: Map selection interface

Implementation: MapSelectController handles map selection and passes choices to GameScene.

3.3 Combat System

3.3.1 Move Set

Complete fighting mechanics include:

- **Light Punch (F)**: Fast attack, 12 damage
- **Heavy Punch (G)**: Slower attack, 25 damage
- **Light Kick (H)**: Fast kick, 18 damage
- **Heavy Kick (R)**: High damage kick, 35 damage
- **Air Attacks**: Air punch, air kick, punch down
- **Blocking (T/I)**: Reduces damage by 75%
- **Movement**: Jump, front flip, back flip

3.3.2 Advanced Mechanics

- **Combo System**: Attack canceling from light to heavy moves
- **Physics Engine**: Gravity simulation, knockback, friction

- **Frame Data**: Startup, active, and recovery frames for each attack

Figure 5: Main fight screen showing combat in action

Implementation: CombatSystem.java defines frame data for each attack. Fighter.java processes input and executes moves based on animation state.

3.3.3 Damage and Health System

- Visual health bars with color-coded status (green/yellow/red)
- Damage scaling for combo hits
- Win conditions: Best of 3 rounds with 99-second timer

3.4 Animation System

The game features 20+ sprite-based animations per character managed by Animation-StateMachine:

- **State Machine**: Manages transitions between animations
- **Frame-Perfect Timing**: Synchronized at 60 FPS
- **Non-Interruptible States**: Attack animations complete before new actions
- **Sprite Management**: Efficient loading and caching

Figure 6: Ryu punch sprite animation frames

Implementation: AnimationStateMachine.java tracks animation states and frame counts. Assets.java loads sprite sheets and crops individual frames.

3.5 Asset Loading Pipeline

Example workflow for loading punch animation:

1. ImageLoader.loadImage("/images/ryu/punch.png")
2. SpriteSheet.crop(606, 102, x, y) extracts each frame
3. Assets.punch[6 frames] stores the array
4. AssetManager.mapRyuAnimations() registers animation
5. AnimationStateMachine retrieves and displays frames
6. Fighter.render() draws frame to canvas

3.6 Audio System

- **Background Music**: Themed music for menus and fight scenes
- **Sound Effects**: Punch, kick, jump, hit, death, and unique announcement sounds.

Implementation: AudioManager.java handles all audio playback using JavaFX MediaPlayer.

3.7 Leaderboard System

Tracks player statistics with automatic updates after each match:

- Win/loss tracking and win rate calculation

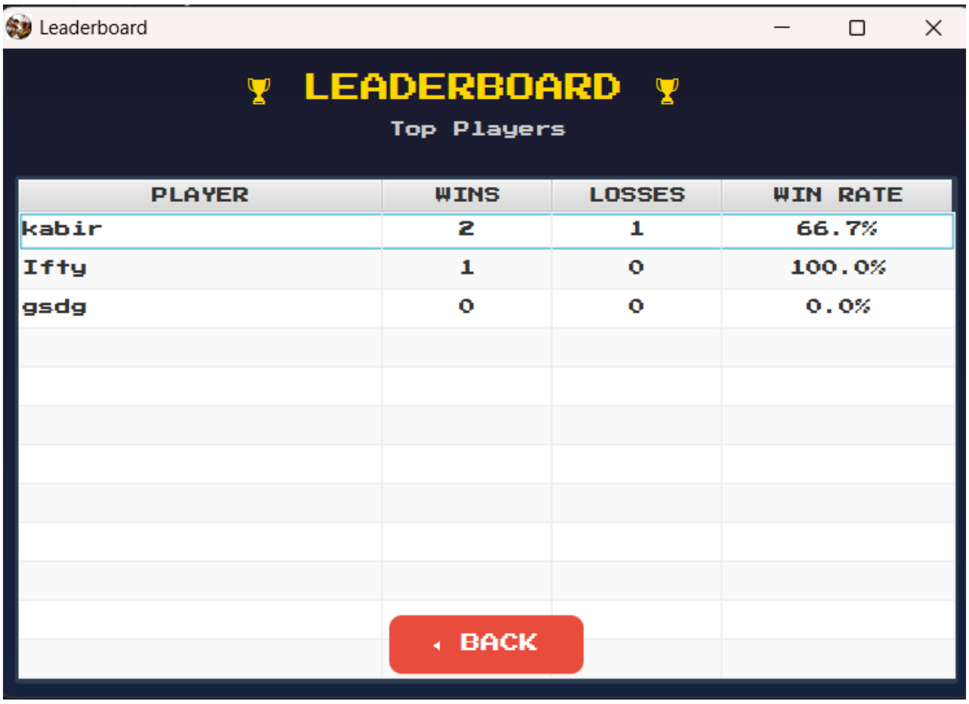


Figure 7: Player statistics leaderboard

Implementation: DatabaseManager.java executes SQL queries for statistics updates and leaderboard retrieval.

4. Network Multiplayer Architecture

4.1 Overview

The network system enables real-time multiplayer combat with:

- Host-client architecture with UDP protocol
- Low-latency communication (27-47ms total)
- 60 FPS input synchronization
- Lobby system with connection status
- Host-controlled character and map selection
- Synchronized pause/resume and rematch functionality

Figure 8: Network multiplayer lobby

4.2 Network Flow Diagram

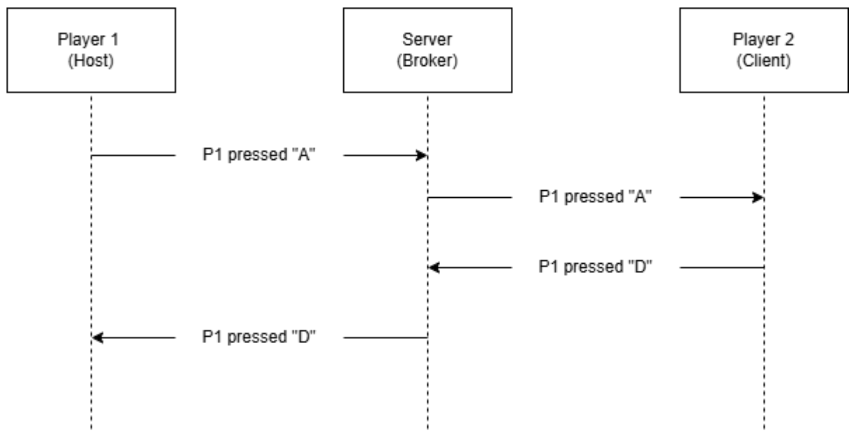


Figure 9: Complete network communication flow

4.3 Connection Establishment

1. Host starts NetworkGameServer on port 5555
2. Client connects to host's IP using NetworkClient
3. Server stores both clients' addresses and port numbers
4. Connection established via UDP DatagramSocket
5. Both clients receive CONNECT_ACCEPTED packets
6. When 2 players connected, server broadcasts GAME_START

4.4 Input Synchronization

4.4.1 Input Capture and Packing

Running at 60 FPS:

1. Player presses button (e.g., F for punch)
2. GameSceneController.handleKeyPressed() detects input
3. InputManager stores key in pressedKeys set
4. NetworkClient.InputPacker compresses 9 buttons into 16-bit value
   - Example: LEFT + PUNCH = binary 0x11
   - Bit flags: LEFT=1, RIGHT=2, UP=4, DOWN=8, etc.

4.4.2 Sending Input to Server

- NetworkClient.sendInput(short inputBits) creates UDP packet
- Packet contains: playerId + 16-bit input + frame number
- Sent every 16ms (rate-limited to 60 FPS)
- UDP used for speed (30ms lower latency than TCP)

4.4.3 Server Processing

- NetworkGameServer.receiveLoop() constantly listens for packets
- handleInput() extracts data when packet arrives
- Server immediately broadcasts input to ALL clients
- **No game logic on server** - pure message broker

4.4.4 Client Reception and Game State Sync

- Remote client's receiveLoop() receives packet
- Packet unpacked: playerId identifies which player moved
- onInputReceived() updates InputManager.networkInputState
- GameSceneController.update() runs on BOTH clients at 60 FPS
- Each client reads InputManager for both players
- Both clients simulate the SAME game independently
- Deterministic logic ensures synchronization

4.5 Example: Player 1 Punch Sequence

Time	**Action**
0ms	Client 1: User presses F Client 1: InputManager.pressedKeys.add(KeyCode.F) Client 1: NetworkClient packs input → 0x10 Client 1: Sends UDP packet to Server
10ms	Server: Receives packet from Client 1 Server: Broadcasts to Client 1 & Client 2
20ms	Client 2: Receives packet Client 2: Unpacks → Player 1 pressed F Client 2: InputManager.networkInputState[P1] = 0x10
33ms	Client 1: player1.processInput() reads local input → PUNCH Client 2: player1.processInput() reads network input → PUNCH **Result:** Both clients show Player 1 punching in sync!

4.6 UDP Protocol Advantages

- **Fast**: 30ms lower latency than TCP
- **Lossy but Acceptable**: 1-2% packet loss tolerable
- **No Handshake**: Immediate transmission
- **Real-time Optimized**: Speed prioritized over reliability

4.7 Packet Loss Handling

- If packet lost: Next packet (16ms later) overwrites it
- No retransmission needed - latest state matters
- Example: Lost "LEFT pressed" → next packet has "LEFT still pressed"

4.8 Latency Analysis

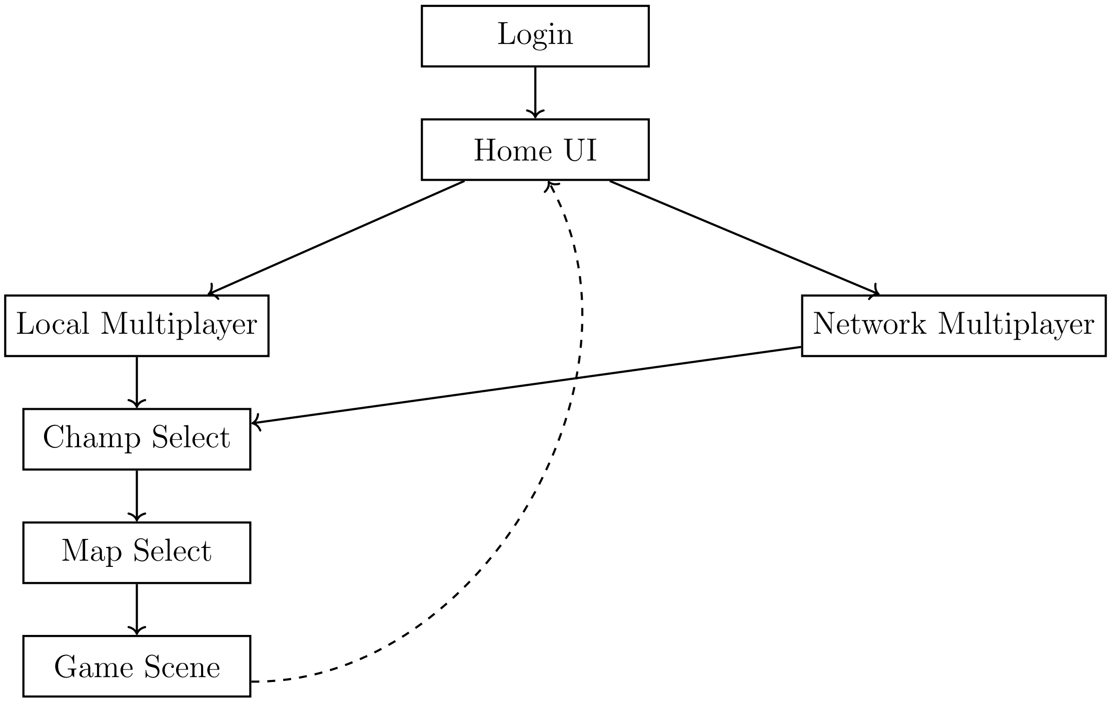
Total Latency: 27-47ms
Input Press → Pack (1ms) → Network Send (10-20ms) →
Server Broadcast (5ms) → Network Receive (10-20ms) →
Unpack & Apply (1ms)

4.9 Key Network Classes

- **NetworkClient.java**
  - sendInput() - Packs and sends input to server
  - receiveLoop() - Continuously listens for packets
  - onInputReceived() - Callback for remote input
- **NetworkGameServer.java**
  - receiveLoop() - Listens on port 5555
  - handleInput() - Processes incoming packets
  - broadcastInput() - Sends to all clients
- **InputManager.java**
  - Stores local input (pressedKeys)
  - Stores network input (networkInputState)
  - Abstracts input source
- **GameSceneController.java**
  - Implements NetworkCallback interface
  - Reads both local and network input
  - Runs identical game logic on both clients

5. Application Flow

5.1 Flow Diagram



5.2 User Journey

5.2.1 Authentication

- User enters credentials in LoginUIController
- DatabaseManager.loginPlayer() validates against MySQL
- SQL: `SELECT * FROM players WHERE username=? AND password=?`
- Success → Navigate to HomeUI

5.2.2 Mode Selection

From Home UI, players can choose:

- Local Multiplayer (same device)
- Network Multiplayer (WiFi connection)
- Leaderboard viewing
- Logout

5.2.3 Pre-Game Setup

- ChampSelectController: Players select Ryu or Ken
- MapSelectController: Choose from 6 maps
- Selections passed to GameSceneController

5.2.4 Game Initialization

- GameSceneController.setGameData() loads characters and map
- Fighter objects created with starting positions
- AnimationStateMachine initialized for each fighter
- Game loop starts at 60 FPS

5.3 Combat Loop

1. Player presses key (e.g., F for punch)
2. InputManager detects input
3. Fighter.processInput() checks if action allowed
4. AnimationStateMachine.transition(PUNCH) starts animation
5. CombatSystem calculates hitbox and damage
6. If hit detected: opponent.takeDamage()
7. Health bars update
8. Check win conditions (health = 0 or timer = 0)

5.4 Match Conclusion

- Winner performs victory animation
- Loser shows death animation
- DatabaseManager updates leaderboard
- Options: Next Round or Return to Home

6. Development Challenges and Solutions

6.1 Network Synchronization

Problem: Initial TCP implementation had 60-80ms latency causing noticeable input lag and client desynchronization.

Solution: Switched to UDP protocol reducing latency to 27-47ms. Implemented 16-bit input compression and 60 FPS rate limiting. Used deterministic game logic for identical simulation on both clients. Accepted 1-2% packet loss as acceptable trade-off.

6.2 Animation State Management

Problem: Attack animations could be interrupted mid-execution, breaking combo system and allowing unrealistic attack spam.

Solution: Created AnimationStateMachine with canBeInterrupted flags. Marked attacks as non-interruptible until recovery frames complete. Added canCancelAttack logic for light-to-heavy attack chains. Implemented frame-perfect timing using System.currentTimeMillis().

6.3 Hitbox Detection Accuracy

Problem: Initial circular hitboxes caused incorrect hit registration, especially during jumps and flips.

Solution: Implemented rectangular hitboxes (Rectangle2D) with per-attack customization. Created CombatSystem.getAttackHitbox() adjusting position/size based on attack type and facing direction. Added active frame windows for hits during specific animation frames.

6.4 Sprite Sheet Management

Problem: Loading 40+ animations (2 characters $\times$ 20+ animations) caused 2-3 second startup lag.

Solution: Implemented AssetManager singleton with lazy initialization. Sprites loaded once and cached in HashMap. Used efficient SpriteSheet.crop() for frame extraction. Reused jump animation frames to reduce sprite count.

6.5 Database Connection Stability

Problem: MySQL connection timeouts during long sessions caused login failures and leaderboard errors.

Solution: Used connection pooling with try-with-resources for auto-closing. Each operation creates fresh connection via DatabaseManager.getConnection(). Added testConnection() to verify database availability.

7. Resources and References

7.1 Official Documentation

- **JavaFX Documentation**: https://openjfx.io/ and https://docs.oracle.com/javafx/2/ - Canvas rendering, Scene management, FXML controllers, AnimationTimer
- **MySQL Connector/J**: https://dev.mysql.com/doc/connector-j/ - JDBC driver for database integration
- **Java SE Documentation**: https://docs.oracle.com/javase/8/docs/api/ - Core Java API for networking and data structures

7.2 Game Development Resources

- **Game Programming Patterns** by Robert Nystrom - https://gameprogrammingpatterns.com/ - State and Singleton pattern implementation
- **Fighting Game Glossary**: https://glossary.infil.net/ - Frame data, hitboxes, and fighting game mechanics
- **2D Game Development Tutorial**: https://zetcode.com/javagames/ - JavaFX game loop and sprite rendering

7.3 Sprite and Asset Resources

- **Street Fighter Sprite Database**: https://www.spriters-resource.com/arcade/sf2/ - Ryu and Ken sprite sheets
- **OpenGameArt**: https://opengameart.org/ - Background images and assets
- **The Spriters Resource**: https://www.spriters-resource.com/ - Additional fighting game references

7.4 Project Repository

- **GitHub Repository**: https://github.com/Nihan2609/Street-Fighter-Game.git - Complete source code with JavaDoc, FXML layouts, sprite assets, and database schema

8. Conclusion

The Street Fighter Game successfully demonstrates integration of JavaFX, MySQL, and UDP networking for real-time multiplayer gameplay. The project achieves its core objectives with consistent 60 FPS performance, 27-47ms network latency, frame-perfect combat mechanics, and intuitive user experience.

8.1 Key Achievements

- 20+ sprite-based animations per character with state machine management
- Deterministic game logic ensuring client synchronization
- Robust input handling for local and network modes
- Scalable database architecture for user management
- Complete combat system with hitbox precision and combo mechanics

8.2 Technical Outcomes

Despite challenges with network synchronization, animation timing, and sprite loading, effective solutions were implemented using UDP protocols, frame-perfect state machines, and optimized asset management. The project balances technical complexity with playability, delivering a functional and enjoyable fighting game experience.

8.3 Future Enhancements

Potential improvements include:

- Additional characters with unique move sets
- Special moves (hadouken, shoryuken)
- Ranked matchmaking system
- Replay system for match review
- AI opponents for single-player mode
- Tournament bracket system
- Enhanced visual effects and particle systems

Thank you! :)