High Level Architecture of The Game

As shown in Figure 3.10, the architecture of the game involved the client-side system and server-side system. However, since the game is made to be accessed via web browser, the client-side program has been kept within the server-side system so that it can be fetched by a student to be executed via his/her web browser. In other words, the student will first interact with the client-side system before transferring data to the server-side system to allow communication among others. Additionally, the components of the architecture have been coloured differently to denote different types of MDA elements. Specifically, red represents mechanic components, orange represents dynamic components and blue represents aesthetic components.

In detail, the server-side system consists of multiple components that have been developed within the Node JS environment. The components include Client Request Handler, Game Session Manager, Server-Side Engine, Server-Client Data Broadcaster and Client-Server Trigger Mechanism.

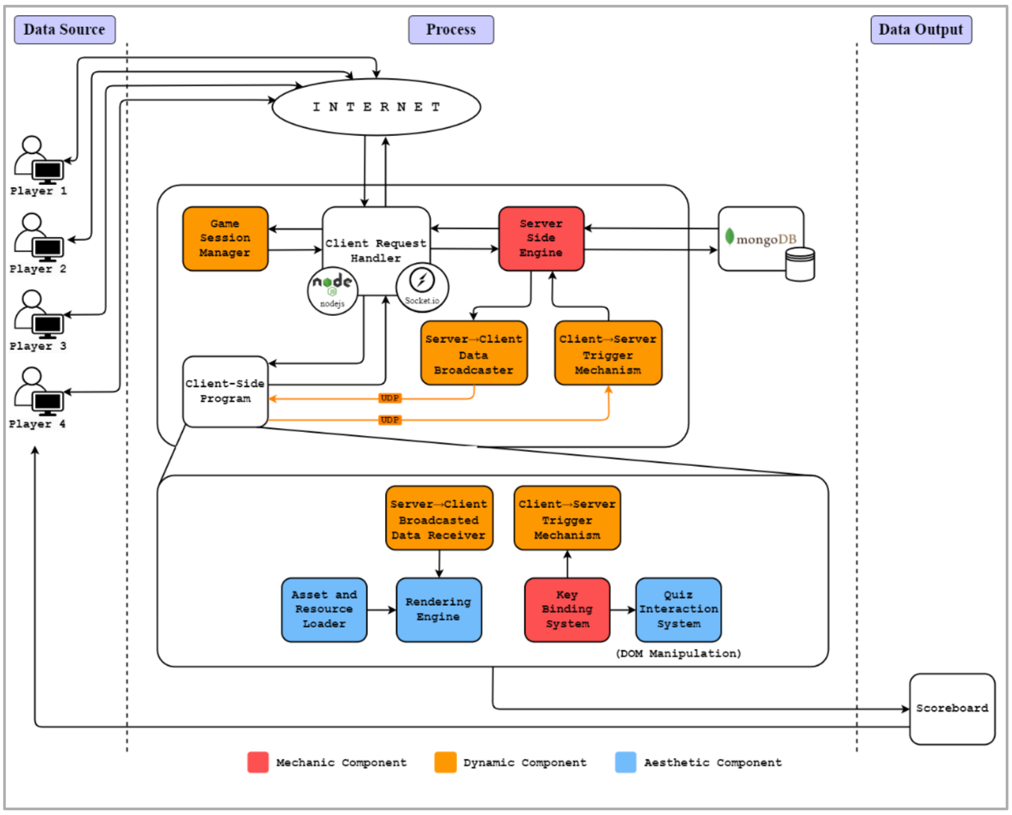


Figure 3.10: High level architecture of the Game-Based Learning in Java Programming Language

In this system, Request Handler has the functionality to process any incoming requests from students that may be in the form of user interfaces, stored data or even the Client-side Program itself. In other words, it manages the flow of requests through the server and coordinate the necessary actions to fulfil those requests. One of the implemented features within the Request Handler is Socket.IO, which is a tool that enables real-time and bidirectional communication between clients and servers. In this system, whenever the Request Handler receives an input from one of the students, Socket.IO will process the input and broadcast it to the other students, to ensure that every student experience a consistent and up-to-date representation of the game world. In other words, it enables the server to send real-time updates to the connected clients.

Aside from that, the system also consists of Game Session Manager that handles all sorts of functionality to initiate the multiplayer features. One of its core functionalities is handling the player assignments to a specific role and team within the game as well as initiating the creation of game sessions once the player assignments process has been executed.

Besides that, there is also a Server-Side engine that handles and manipulates the in-game data, including managing sprites and animations, controlling the game timer, and loading data from the MongoDB database server. Additionally, this component is integrated with the Server-Client Data Broadcaster and the Client-Server Trigger Mechanism, which serve as connecting points for transferring real-time data between the server and client so that seamless communication and synchronization can be maintained during gameplay.

As for the client-side system, it is composed of several integral components such as Asset and Resource Loader, Rendering Engine, Key Binding System and Quiz Interaction System. The Asset and Resource Loader is responsible for loading all necessary game assets and resources to ensure they are available and ready for use by the Rendering Engine before the game begins. Other than that, The Rendering Engine handles the visual presentation of the game by processing graphical data and rendering it on the screen to ensure all visual elements, such as sprites and animations are displayed smoothly and correctly. Besides that, the Key Binding System handles user inputs through keyboards, thus allowing the game system to translate these inputs into in-game actions, thus enhancing the game's responsiveness and playability. Finally, the Quiz Interaction System manipulates the Document Object Model (DOM) to dynamically update and display quiz questions, collect player responses and provide feedback, thus seamlessly integrating quizzes into the gameplay to offer an interactive and educational experience

Low Level Architecture of The Game

Based on the high-level game architecture shown in Figure 3.10, some of the components can be explored in more detail to understand their specific functions and interactions within the system. Therefore, in this section, a few core components have been analysed deeply to get an insight on how the components of mechanics, dynamics and aesthetics of the game have been developed.

i. Development on Mechanic Components

From the high-level game architecture shown in Figure 3.10 there are two components involved in developing the game mechanics which are Key Binding System from the client-side program and Server-Side Engine from the server-side program. Shown in Figure 4.5 is the low level architecture of the Key Binding System that reveals other four modules within the system. It includes 'Movement Key Bind' which takes input of 'keydown' event for moving the game avatars, 'Quiz Initiation Key Bind' which handles the key input for initiating quizzes within the game, 'Quiz Submission Key Bind' which processes the key input for submitting quiz answers and 'Quiz Termination Key Bind' which manages the key input for terminating or exiting a quiz.

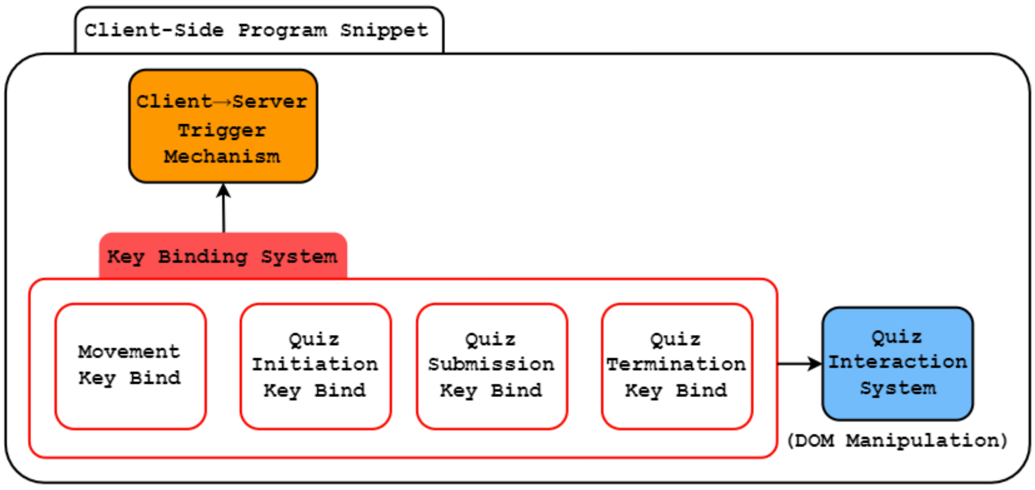


Figure 4.5: Low level architecture of the Key Binding System

However, shown in Figure 3.16 is the low level architecture of the Server-Side Engine which consists of four sub-modules that include Sprite System to handle the changes of coordinates and animations of the game's avatars, Quiz Event System to manage the events related to the quizzes, Timer System to keep track of the total time left for a gameplay and Data Loader to load the data of quiz from MongoDB database server.

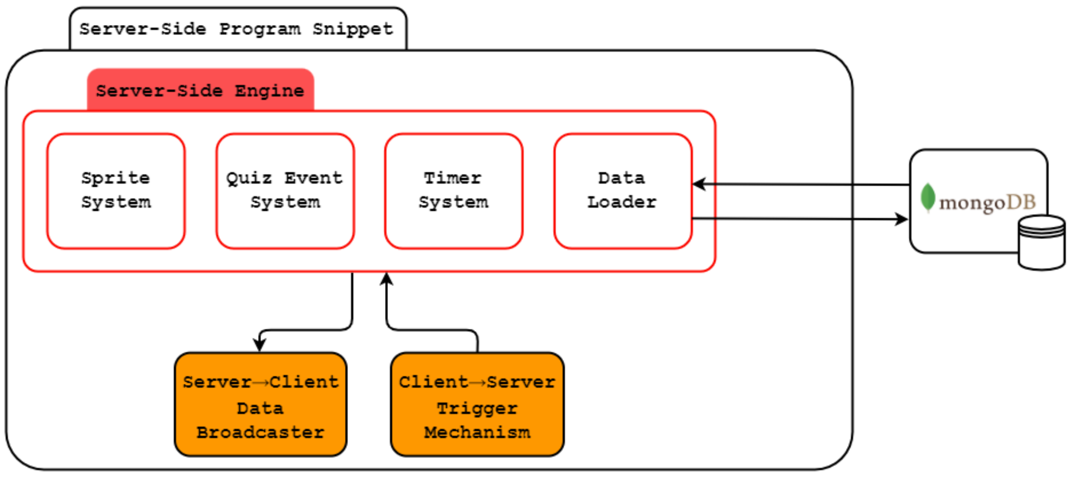

Server-Side Program Snippet
Server-Side Engine
Sprite System
Quiz Event System
Timer System
Data Loader
mongoDB
Server→Client Data Broadcaster
Client→Server Trigger Mechanism

Figure 4.6: Low level architecture of the Server-Side Engine

Out of all the modules illustrated within the low-level architecture, there are few modules that play major roles in managing the movement and animation of the Game Avatar as well as the quiz event within the gameplay. These include all the modules within the Key Binding System as well as the Sprite System and Quiz Event System from the Server-Side Engine.

Movement and Animation Mechanic

```
function processEventQueue(id) {
    if (keyEventQueue[id].length > 0) {
        const event = eventQueue[id][0];
        const { details, keyEvent } = event;

        updatePosition(details, key);
        mpr[id] -= 1;

        if (mpr[id] > 0) {
            interval_[id] = setTimeout(() => {
                processEventQueue(id);
            }, 15);
        } else {
            keyEventQueue[id].shift();


            if (keyEventQueue[id].length > 0 && mpr[id]
                        === 0) {
                mpr[id] = 10;
                processEventQueue(id);
            } else {
                updateSprite(key, details);
                clearInterval(interval_[id]);
                delete interval_[id];
            }
        }
    }
}
```

Figure 4.7: Code snippet on Sprite System

Shown in Figure 4.7 is the code snippet on the process of managing the movement and animation of the game avatar in a flowchart form. Initially, this process will be executed whenever there is 'keyEvent' received from the 'Movement Key Bind' of the Key Binding System. The 'keyEventQueue' from the process ensures that only one event is handled at a time to avoid conflicts and maintain sequential execution of movements. Apart from that, the 'mpr' variable which stands for 'Moving Progress Remaining' handles the grid-based movement of the avatars to keep them at the centre of a tile all the time.

Aside from that, the implemented process of managing quiz events can be illustrated in the code snippet of Figure 4.8. Initially, the process will be triggered whenever it receives an input of quiz initiation event from students via Client-Server Trigger Mechanism. Within the gameplay, each quiz terminal will allow students to have a time limit of 20 seconds. Therefore, 'countDown[UID]' will be used as a timer to track the remaining time for each player's quiz attempt. This ensures that students have a fair and consistent timeframe to respond to quiz questions during their gameplay session. Once 'countDown[UID]' has been initialized, the program will send the initial countdown value to the player and designate the specific quiz terminal they are interacting with as 'inUse'. This prevents multiple students from accessing the same quiz simultaneously, maintaining fairness and avoiding conflicts in gameplay interactions.

Quiz Event Mechanic

```
socket.on('init-quiz-event', ({ details, key, quizCoor })
=> {

    if(!countDown[UID]){
        countDown[UID] = timeLimit
    } else {
        console.log('count already exist');
    }

    socket.emit('quizCountDown', countDown[UID]);
    serverQuizStation.inUsed = socket.id;


    tickDownTimer[socket.id] = setInterval(() => {
        countDown[UID]--;
        socket.emit('quizCountDown', countDown[UID]);
        if (countDown[UID] <= 0){
            clearInterval(tickDownTimer[socket.id]);
            delete countDown[UID];

        io.to(details.roomCode).emit('outUsedTerminal',
        {quizKey: `${quizCoor.x},${quizCoor.y}`});

        serverQuizStation[details.roomCode]
        [`${quizCoor.x},${quizCoor.y}`].inUsed = '';
        }
    }, 1000)

    // Broadcast on the in-used terminal
    io.to(details.roomCode).emit('inUsedTerminal',
    {quizKey: `${quizCoor.x},${quizCoor.y}`, usedBy:
    socket.id});
})
```

Figure 4.8: Code snippet on Quiz Event System

After that, an interval function will decrement the value of 'countDown[UID]' by one every second. This process continues until 'countDown[UID]' reaches a value of 0. Once the countdown reaches zero, the interval function will stop executing. Subsequently, 'countDown[UID]' will be removed, and the quiz terminal that was marked as 'inUse' will be cleared, allowing other students to access it. This sequence ensures that each player has a defined timeframe to complete their quiz interaction after which the system resets for the next player, thus maintaining orderly gameplay and preventing simultaneous access issues.

ii. Development on Dynamic Components

Regarding the dynamic components, the high-level architecture illustrates three components within the server-side program which include Game Session Manager, Server-Client Data Broadcaster and Client-Server Trigger Mechanism. However, on the client-side program, there are two components which include Server-Client Broadcasted Data Receiver and Client-Server Trigger Mechanism.

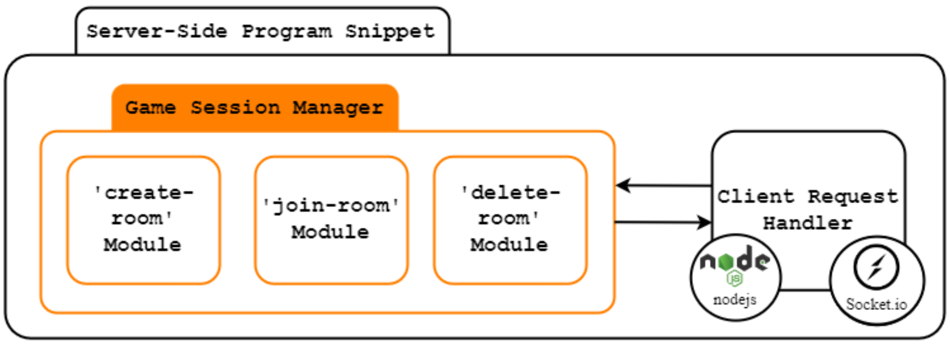
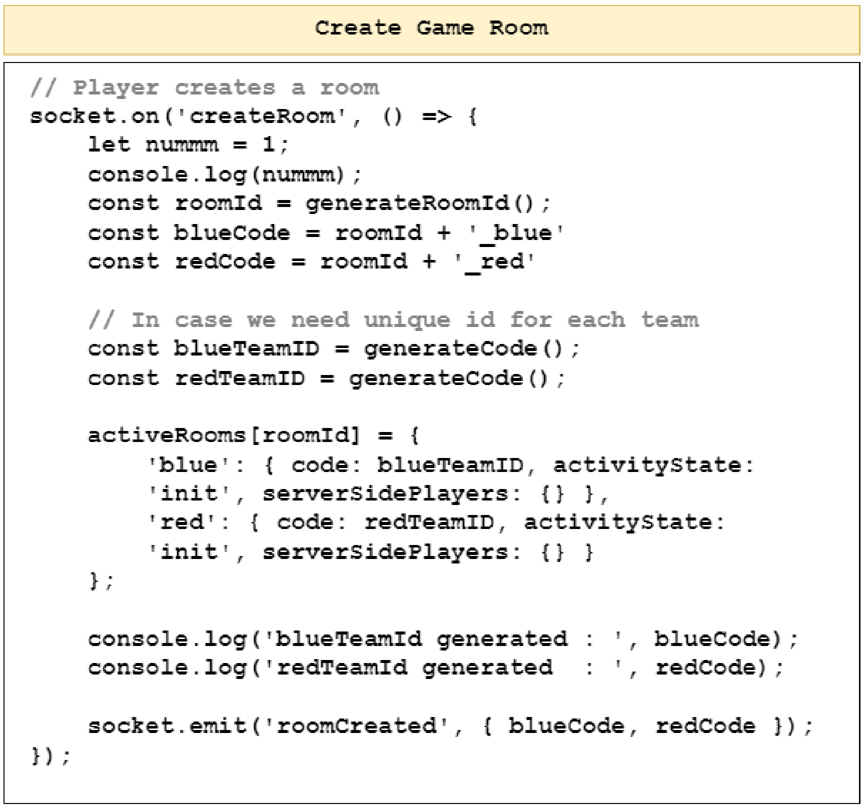


Figure 4.9: Low level architecture of the Game Session Manager

Shown in Figure 4.9 are the modules built within the Game Session Manager Component which include 'create-room' Module and 'join-room' Module that are responsible for setting up and managing game sessions before the in-game activities commence. Additionally, the 'delete-room' Module handles the removal of game sessions after the in-game activities are completed.



Create Game Room

```
// Player creates a room
socket.on('createRoom', () => {
    let nummm = 1;
    console.log(nummm);
    const roomId = generateRoomId();
    const blueCode = roomId + '_blue'
    const redCode = roomId + '_red'

    // In case we need unique id for each team
    const blueTeamID = generateCode();
    const redTeamID = generateCode();

    activeRooms[roomId] = {
        'blue': { code: blueTeamID, activityState:
        'init', serverSidePlayers: {} },
        'red': { code: redTeamID, activityState:
        'init', serverSidePlayers: {} }
    };

    console.log('blueTeamId generated : ', blueCode);
    console.log('redTeamId generated  : ', redCode);

    socket.emit('roomCreated', { blueCode, redCode });
});
```

Figure 4.10: Code snippet on ‘create-room’ module

Figure 4.10 shows the detailed flow of the process involved within the 'create-room' module. Initially, when it receives an input indicating that a player wants to create a game room, the module generates a random string and concatenates it with '_blue' and '_red' to create two distinct codes for the blue team and the red team. These codes are then sent to the player to be distributed among other students.

```
Join Game Room
```

```
socket.on('joinRoom', ({ code, playername }) => {
    if(code.length < 13 || code.length > 14){
        socket.emit('joinFailed', 'invalid code')
        return;
    }

    if (Object.keys(selectedTeam.serverSidePlayers)
        .length === 0){
        selectedTeam.serverSidePlayers[socket.id] = {
            name: playername,
            team: teamCode,
            role: 'repair' // initial state
        }
    } else {
        selectedTeam.serverSidePlayers[socket.id] = {
            name: playername,
            team: teamCode,
            role: 'sabotage' // initial state
        }
    }

    // Join private team room
    socket.join(code);
    // Broadcast among team members
    io.to(code).emit('playerJoined', { relv_details:
    {selfId: socket.id, fullCode: code, roomCode:
    roomId, colorCode: teamCode}, totalPlayers });
});
```

Figure 4.11: Code snippet on ‘join-room’ module

However, Figure 4.11 shows the detailed flow of the process involved within the 'join-room' module. The process begins when it receives an input from a player who wants to join an existing game room. The player is prompted to enter the 'roomcode' on the client-side program, which is then sent to the module. The module checks the validity of the code. if the code is valid, it will proceed to assign the player's details as a game object on the server-side program. Finally, the module includes the player in the gameplay channel using the 'socket.join()' method.

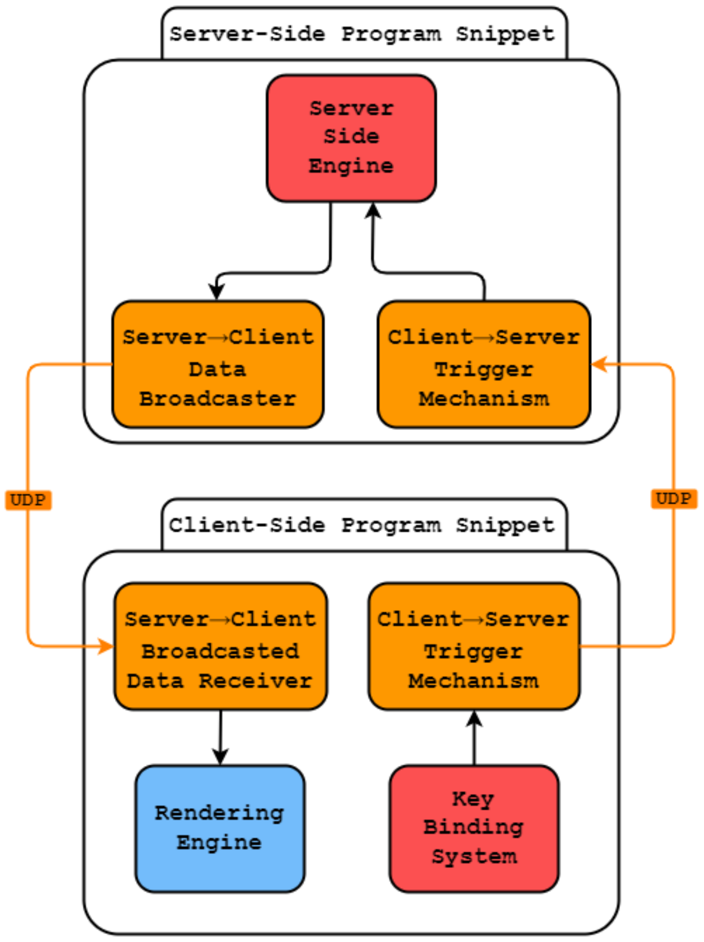

Server-Side Program Snippet
Server Side Engine
Server→Client Data Broadcaster
Client→Server Trigger Mechanism
UDP
UDP
Client-Side Program Snippet
Server→Client Broadcasted Data Receiver
Client→Server Trigger Mechanism
Rendering Engine
Key Binding System

Figure 4.12: Detailed architecture on the UDP client-server connection

The other four dynamic components of the high-level architecture do not have any subsystems or modules within them. However, it is essential to highlight the connectivity of these components via UDP connection, as shown in Figure 4.12. These components are the core components that developed the multiplayer features of the game.

Server-Client Data Broadcaster

```
setInterval(()=>{
    updateAnimationProgress();
    io.emit('updateGameObjects', activeRooms)
}, 15)
```

Server-Client Broadcasted Data Receiver

```
socket.on('updateGameObjects', (gameplayDetails)=>{
    if(this.playSession){
        playSession.updateGameObjectsEvent(
                gameplayDetails[this.details.roomCode]
        );
    }
});
```

Figure 4.13: Code snippet on Data Broadcaster and Data Receiver component

Figure 4.13 shows a snippet of the code from the Server-Client Data Broadcaster on the server-side program and the Server-Client Broadcasted Data Receiver on the client-side program. In simple terms, the Server-Client Data Broadcaster transmits all updates from the server to the students engaged in the gameplay. This implementation allows for real-time synchronization of game data and ensures that all students receive the latest information simultaneously.

Client-Server Trigger Mechanism (Client)

```
if (this.directionInput.direction) {
    // ... other code ...

    if (!debounceTimer) {
        debounceTimer = setTimeout(() => {
            this.socket.emit('key-down-event', {
                details: this.details,
                key: this.key
            });
            debounceTimer = null;
        }, 15);
    }
}
```

Client-Server Trigger Mechanism (Server)

```
socket.on('init-quiz-event',
   ({ details, key, quizCoor }) => {
        // ... other code ...
    }
);
```

Figure 4.14: Code snippet on Client-Server Trigger Mechanism for quiz event

```
Client-Server Trigger Mechanism (Client)
```

```
bindActionInput(){
    new keyPressListener("Enter", () => {
        // ... other code ...

        // If there is a quiz
        if(match && !isAnswering){
            // ... other code ...

            this.socket.emit('init-quiz-event', {
                details: this.details,
                key: this.key, quizCoor: nextCoords
            });
        }
    })
}
```

```
Client-Server Trigger Mechanism (Server)
```

```
socket.on('init-quiz-event',
   ({ details, key, quizCoor }) => {
        // ... other code ...
    }
);
```

Figure 4.15: Code snippet on Client-Server Trigger Mechanism for Sprite System

On the other hand, Figures 4.14 and 4.15 show snippets of the code from the Client-Server Trigger Mechanism on the client-side and server-side programs. Technically, the program in Figure 4.15 handles the game avatar's coordination and animation, while the program in Figure 4.14 manages quiz events. Both programs execute any necessary calculations or operations on the server-side when they receive a trigger event from students on the client-side.

iii. Development on Aesthetic Components

Apart from that, the high-level architecture presented in Figure 3.10 is equipped with three components that handle the aesthetic aspect of the game. As shown in Figure 4.16 the components include Asset and Resource Loader, Rendering Engine and Quiz Interaction System.

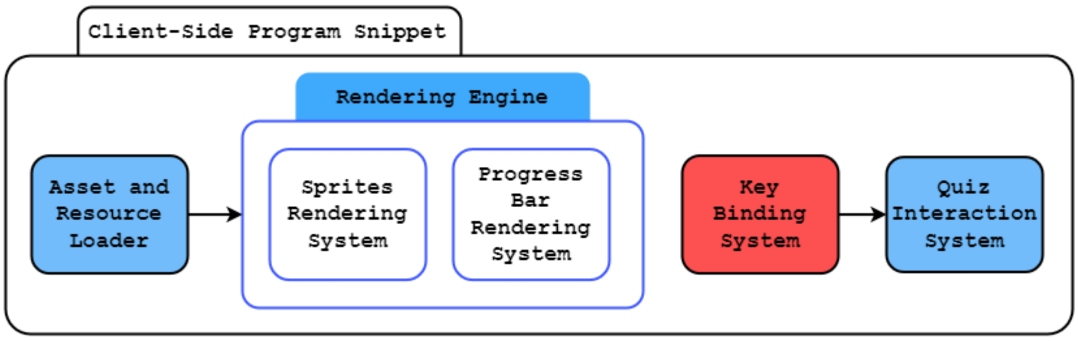

Client-Side Program Snippet
Rendering Engine
Asset and Resource Loader
Sprites Rendering System
Progress Bar Rendering System
Key Binding System
Quiz Interaction System

Figure 4.16: Low level architecture of the Rendering Engine

As for the Asset Resource Loader, it is equipped with a simple program that loads and ensures that all the game assets are readily available within the game before the game begins. However, for the Rendering Engine, it is equipped with two subsystems which are Sprites Rendering System and Progress Bar Rendering System.

Sprite Rendering System

```
this.ctx.drawImage (
    avatarImg,
    object.sprite.x * 16, object.sprite.y * 16,
    16, 16,
    object.position.x *16, object.position.y *16+4,
    16, 16
);
```

Progress Bar Rendering System

```
// Blue Team Progress Bar
blue_fills.forEach(element => {
    element.setAttribute("y", blueProgress + "%");
    let blueStateOfProgress =
              blueProgress > 50 ? "#034E9A" : "#0180FF";
    element.setAttribute("fill", blueStateOfProgress);
});

// Red Team Progress Bar
red_fills.forEach(element => {
    element.setAttribute("y", redProgress + "%");
    let redStateOfProgress =
              redProgress > 50 ? "#B90B0B" : "#FF0202";
    element.setAttribute("fill", redStateOfProgress);
});
```

Figure 4.17: Code snippet on Sprites and Progress Bar Rendering System

Shown in Figure 4.17 is a snippet of the code that builds the Sprites Rendering System and the Progress Bar Rendering System. In simple terms, as the data of the game is continuously received via the Server-Client Broadcasted Data Receiver, these two subsystems render the raw data to create visual representations of game elements. This includes updating the sprites for game characters and displaying the progress bars accurately.

Quiz Interaction System

```
// Create QnA Panel
this.element = document.createElement("div");
this.element.classList.add("qna-panel");
this.element.innerHTML = (`
    ${questPanel_element}
    <div class="answer-panel">
        ${ansPanel_element}
    </div>
`);
```

Figure 4.18: Code snippet on Quiz Interaction System

Besides that, the Quiz Interaction System handles the quiz interaction aspect of the game. As shown in Figure 4.18, this system is responsible for displaying and removing the quiz panel that shows a question and its multiple-choice answers, using DOM manipulation technique. This component ensures that the quiz interface is presented to the player when needed and removed once the interaction is complete, thus allowing for smooth and interactive quiz experiences within the game.