Implementation of Finite State Machine in Flowchart-Based Visual Programming Game

Iqbal Al Mahdi
Faculty of Computer Science
University of Jember
Jember, Indonesia
182410103030@mail.unej.ac.id

Saiful Bukhori
Faculty of Computer Science
University of Jember
Jember, Indonesia
saiful.ilkom@unej.ac.id

Muhammad Ariful Furqon
Faculty of Computer Science
University of Jember
Jember, Indonesia
ariful.furqon@unej.ac.id

Abstract*—This study explores the implementation of Finite State Machines (FSMs) in a visual programming game based on flowcharts, aimed at enhancing the learning experience of programming concepts. Traditional programming education methods often struggle to engage beginners, leading to the development of interactive and intuitive approaches such as visual programming games. In this context, FSMs are integrated to manage the behavior of game units, allowing for dynamic state transitions based on user-defined flowcharts. The research adopts the Game Development Life Cycle (GDLC) approach, focusing on initialization, pre-production, production, and alpha testing stages. The primary objective is to implement and validate the FSM's effectiveness in controlling unit behavior within the game. Users can design strategies through a drag-and-drop interface, creating flowcharts that translate into FSM models, which dynamically control unit actions during gameplay. Results from the alpha testing indicate that the FSM implementation successfully manages the transitions and behaviors of game units according to the conditions specified in the flowcharts. This demonstrates the technical feasibility and effectiveness of the approach. Although the study does not extend to beta testing and release stages, the alpha testing provides a solid foundation for future research and development focused on user experience and broader feedback.

***Keywords*—Finite State Machine, Visual Programming, Game Development, Flowchart**

I. Introduction

The rapid rise of Industry 4.0 has turned programming into a core competency for many professions [1]. Many programmers face difficulties when they first start learning about the concepts in programming logic [2] and often lose motivation in the [3]. Games are a solution that can help maintain motivation in learning programming logic [4]. According to Schrader [5], in recent decades, game-based learning methods have become increasingly popular and sought after as a digital learning medium. Game-based learning has proven effective in increasing learning motivation and can help teach material that is difficult to explain with conventional methods.

Robocode is a game designed to introduce programming logic, where users must create artificial intelligence to control characters within the game [6]. However, the conventional programming languages used to design artificial intelligence pose a barrier for novice users who lack experience with complex code syntax. Visual programming can be a solution to this problem [7]. The advantage of visual programming languages lies in their ability to overcome the complexity of programming tasks through graphical representation [8]. Visual programming based on flowcharts is used by some programmers to design algorithms, which are then implemented into program code based on the algorithm design. Besides being used to design program algorithms, flowcharts are also beneficial for novice programmers to understand how to structure programs better [9].

Well-developed artificial intelligence agents can enhance users' decision-making skills, as the outcomes in the game depend on these skills [10]. Artificial intelligence in games refers to the cognitive abilities and decision-making processes exhibited by game characters, playing a crucial role in creating dynamic and engaging interactive experiences for users. Typically, game characters use rule-based artificial intelligence systems with limited behavior patterns, performing the same actions repeatedly until an event triggers a change in behavior [11]. As the game industry evolves, game developers have begun to develop various rule-based programming architectures to manage more complex rules, such as Finite State Machines (FSM) [12]. In game development, FSM is used to design and determine appropriate responses to changing conditions. Game developers can create flexible and easily manageable behavior models for game characters using FSM, resulting in more dynamic and realistic interactions with game characters [13].

The research objectives are to implement and validate the FSM's effectiveness in controlling unit behavior within the game. To address this gap, this paper pursues three concrete goals: (1) Design and integrate an FSM engine into the Flowchart League visual programming game, enabling users to specify unit behavior through drag-and-drop flowcharts; (2) Evaluate the FSM-driven gameplay in an alpha-testing phase, measuring both functional correctness (state-transition fidelity) and learner engagement (motivation, perceived ease of use); and (3) Compare the FSM-based approach with traditional script-based control schemes to demonstrate its pedagogical advantage for novice programmers. Consequently, the primary research objective of this study is to implement and empirically validate the effectiveness of FSMs for controlling unit behavior in a flowchart-based visual programming game.

II. Literature Review

A. *Visual Programming Language*

Visual programming languages are a type of programming language that allows users to create programs by combining graphical elements rather than writing text. Visual programming is used to make it easier for beginners to learn the basics of programming by visualizing program elements. This approach enables beginners to create, develop, and customize software applications using two-dimensional graphical elements. Today, visual programming is increasingly used in various fields to create and customize useful applications beyond educational contexts, such as in the Internet of Things (IoT), mobile application development, robotics, and Virtual Reality or Augmented Reality. Kuhail, Mohammad Amin et al. [14] classify visual programming languages into four categories based on their visual representation:

- Block-Based: In block-based visual programming languages, users can create programs by dragging and dropping blocks of program elements from a predefined list of commands into a development area. This approach helps avoid syntactical errors and allows users to focus more on concepts rather than implementation details. Examples of applications that have implemented this type of visual programming language include Scratch and App Inventor.

- Icon-Based: Icon-based visual programming languages use graphic symbols to represent objects or actions. There are two types of icons: simple and complex. Simple icons represent basic objects or actions like files, delete, and edit, while complex icons are a combination of simple icons to form visual sentences. This type of visual programming language has been used to help users without programming experience create applications based on triggers and actions, such as sending alerts if the room temperature is below 40°F.

- Form-Based: Form-based visual programming languages allow users to set triggers and actions using forms. Most form-based approaches use drag-and-drop configuration of visual components, while some also use dropdown text. Users of form-based visual programming languages can create or configure cells and specify parameters for these cells. These parameters refer to values contained in other cells and use them in calculations. Whenever a cell's formula is defined, the underlying evaluation engine calculates the cell's value, recalculates the values of cells that reference the recalculated cell, and displays the new results on the screen. However, Huang, Gaoping et al. [15] argue that form-based programming languages are not effective because they are not flexible enough to handle dynamic and complex workflows compared to other visual programming languages such as block, data-flow, flowchart, event-based, or state-flow.

- Diagram-Based: Diagrams have been used as communication tools in many fields. Diagram-based visual programming languages, also known as diagram or data-flow languages, are characterized by connecting graphical objects such as boxes with arrows, lines, or arcs representing relationships. To understand a diagram-based program, users can follow the flow of the diagram. Diagrams use various perceptual encoding methods to represent the flow of the program. For example, flowcharts use connections and directions to show how information relates to each other and how it flows from one point to another.

B. Flowchart

A flowchart is a part of diagram-based visual programming languages [14]. It is a visual representation of an algorithm, depicted using symbols or images to solve a particular problem [16]. According to Chaudhuri [17], a program flowchart is a useful tool in program development as it helps to detect errors or omissions in the program logic more easily compared to text-based program representations. A flowchart can also serve as useful documentation if the program needs to be modified in the future. Finite State Machine.

According to Yannakakis & Togelius [18], Finite State Machine (FSM) is a primary artificial intelligence method for controlling and making decisions for game characters. The FSM system falls under the category of expert systems, represented as a simple graphical network of objects, symbols, events, actions, or properties. This graph consists of states and transitions that represent conditional relationships between states. An FSM can only be in one state at a time, and state changes occur only when the conditions of the corresponding transition are met. There are three main components that define an FSM:

- States that store information about a task, such as the "explorer" state being worked on.

- Transitions between states that indicate state changes and are described by conditions that must be met, for example, if you hear gunfire, transition to the "alerted" state.

- Actions to be performed in each state, such as moving randomly and searching for opponents while in the "explorer" state.

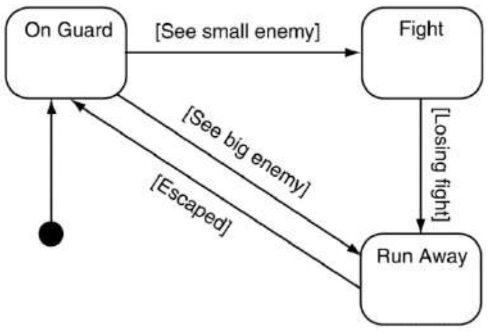


Fig. 1. Example of a simple FSM [11]

Fig. 1. shows an example of a simple state machine with several states, including On Guard, Fight, and Run Away, each with its own transitions.

The state machine monitors various potential conditions and notes the current condition. In addition to conditions, the state machine also observes a series of transitions. At each iteration (usually every frame), the state machine's update function processes to check if there are any changes from the current state that are activated. The first change that is activated will be scheduled to run. Then, a list of actions to be performed from the currently active state is compiled. The separation between the triggering and execution of changes allows transitions to have their own actions as well. In its performance, the state machine only uses memory to hold the triggered transitions and the current state [11].

C. Game Development Life Cycle

The Game Development Life Cycle (GDLC) is a game development method that follows an iterative approach with six stages, starting from initialization, pre-production, production, testing, beta, and release [19]. The stages of GDLC can be seen in Fig. 2.

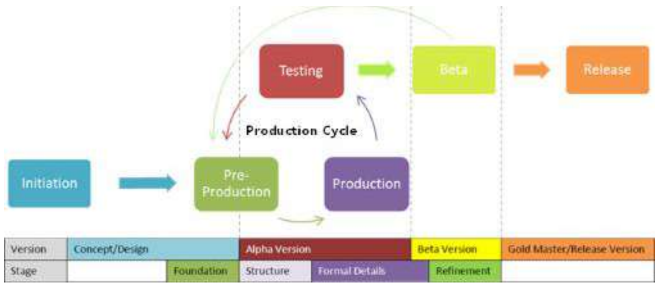


Fig. 2. GDLC [19]

1. Initialization

The initial step in creating a game is to outline the rough components of the game. The outcome of this stage is the game concept and a simple description of the game to be developed.

2. Pre-Production

The pre-production stage involves refining the game design and creating its prototype. Game design focuses on determining the genre, gameplay, mechanics, storyline, characters, challenges, fun aspects, and technical aspects.

3. Production

Production involves asset creation, programming, and integrating these two elements. The production stage also deals with creating and refining formal details such as game balancing, adding new features, improving performance, and fixing bugs.

4. Alpha Testing

Testing phase occurs after the production stage is completed. This internal testing aims to evaluate game functions and performance. Results from this testing phase include bug reports, change requests, and decisions regarding further development. Conclusions from this stage influence decisions to proceed to the next stage or iterate the production cycle.

5. Beta Testing

Unlike the previous stage, beta testing involves external participants. There are two types of beta testing based on participants: closed beta and open beta. Closed beta allows invited participants only, while open beta opens up testing to anyone who registers. Results from beta testing include bug reports and user feedback. If no significant issues arise from the testing results, the process can proceed to the next stage.

6. Release

At this stage, game development has reached its final phase and is ready for public release. The release process involves product launch, project documentation, and planning for game maintenance and expansion.

III. RESEARCH METHODOLOGY

The stages of this research will adopt the Game Development Life Cycle (GDLC) approach, but will be limited to alpha testing. This study focuses on implementing FSM in games to control character behavior. Therefore, the emphasis is on technical aspects involving development, implementation, and internal game mechanism testing, such as FSM application. Beta testing and release stages, which typically focus on user experience and satisfaction with the released game, are beyond the primary scope of this research. Hence, beta testing and release are not prioritized, as alpha testing is sufficient to verify that FSM functions according to predefined objectives. The steps involved in this research include initialization, pre-production, production, and alpha testing.

$$\begin{tikzcd}
\boxed{\text{Initialization}} \arrow[r] & \boxed{\text{Pre-Production}} \arrow[r] & \boxed{\text{Production}} \arrow[r] & \boxed{\text{Alpha Testing}}
\end{tikzcd}$$

Fig. 3. Research stages

1. Initialization

The initial stage of this research is the initialization phase. It involves defining the game's function, functional and non-functional requirements, game concept, and wireframing as an initial visual foundation aiding in the detailed interface design development in subsequent stages.

2. Pre-Production

The next stage is pre-production, where the rough concepts generated in the initialization phase are detailed and explained in the Game Design Document (GDD).

3. Production

In the production stage, the game development begins based on the GDD and previously established wireframes. This phase explains the FSM implementation and the creation of features and interfaces. FSM is developed to control characters in the game, with predefined states and actions. However, state transitions are dynamic based on user-created flowcharts. Table 1 outlines several states that control character behavior along with their descriptions.

TABLE 1. TYPES OF STATES AND THEIR DESCRIPTION

State	Description
Idle	State where the character is not performing any actions.
MoveTo	State where the character moves towards a specified location or target.
FleeFrom	State where the character moves away from a specific target.
Attack	State where the character attacks a specified target.
Dead	State where the character cannot perform any actions due to being deceased.

4. Alpha Testing

Alpha testing is the final step in game development for this research. This testing phase focuses on the developed game features using Blackbox Testing with Functional Testing type. Additionally, evaluation is conducted on the FSM implemented in the game. This testing employs three FSM examples, which are then translated back into flowchart forms within the game. Below are explanations of each FSM example:

- FSM Example 1

The first FSM example is the simplest form. This FSM only implements two states: MoveTo and Attack. The character moves towards the enemy if the nearest enemy is more than 10 units away, and attacks the enemy if the distance is less than 10 units.

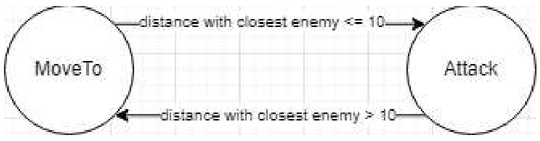

MoveTo
Attack
distance with closest enemy <= 10
distance with closest enemy > 10

Fig. 4. FSM example 1

- FSM Example 2

The second example FSM implements three states: MoveTo, Attack, and FleeFrom. This FSM controls the character to approach the enemy if the character's health is better than the enemy's, and vice versa to move away. If the character is in the MoveTo state and the nearest enemy is within 10 units, it switches to the Attack state.

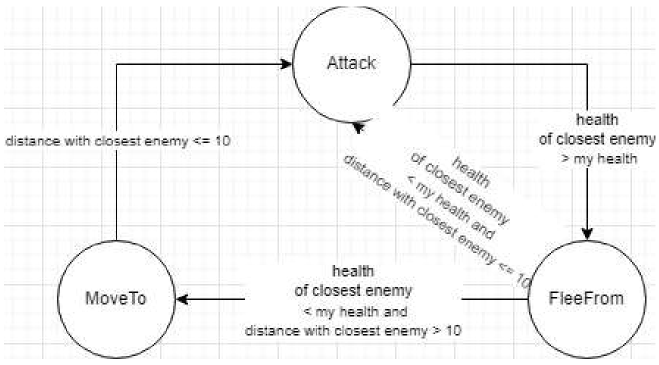

Attack
distance with closest enemy <= 10
health of closest enemy > my health
health of closest enemy < my health and distance with closest enemy <= 10
MoveTo
health of closest enemy < my health and distance with closest enemy > 10
FleeFrom

Fig. 5. FSM example 2

- FSM Example 3

The third FSM example is more complex than the previous two, with four different states and diverse transitions. The character is directed to approach the enemy until within 10 units, then attack. However, if the character's health drops below 50%, it will move away from the enemy and return only when its health is back above 50%.

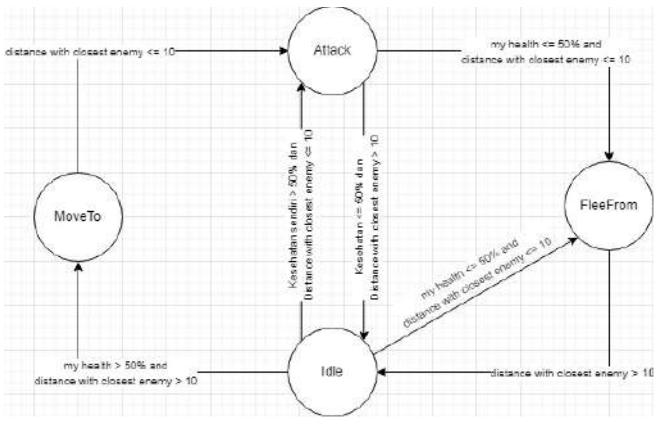

distance with closest enemy <= 10
Attack
my health <= 50% and distance with closest enemy <= 10
Kesehatan sendiri > 50% dan Distance with closest enemy <= 10
Kesehatan <= 50% dan Distance with closest enemy > 10
MoveTo
FleeFrom
my health <= 50% and distance with closest enemy <= 10
my health > 50% and distance with closest enemy > 10
Idle
distance with closest enemy > 10

Fig. 6. FSM example 3

IV. RESULT

A. Initialization

The main objective of this stage is to ensure that every aspect of the game requirements has been thoroughly analyzed and considered. This is crucial to facilitate a smooth development process in the subsequent stages. By ensuring all game requirements are well-considered and prepared, development becomes more efficient and organized, thereby reducing the likelihood of unforeseen obstacles later on.

1) Game Function

The function of this game is to enhance understanding of basic programming concepts and improve programming skills interactively and enjoyably. Additionally, users can sharpen their creativity and logic through this game.

2) Functional Requirements

Functional requirements analysis involves identifying and documenting all features and functions expected by users from the game being developed. The focus of functional requirements analysis is on aspects related to the operational or functional aspects of the game. Functional requirements for the game to be developed include:

- Users can design algorithms for their characters to take various actions in each situation that occurs in battle simulations using flowcharts.
- Users can select different algorithms for each of their characters.
- Users can save their character algorithm data online on their account.
- Users can test and evaluate the algorithms they have created for their characters.

3) Non-Functional Requirements

Non-functional requirements analysis involves identifying and documenting all requirements not related to the features or operation of the game being developed. This analysis focuses on technical, performance, and display aspects of the game. Non-functional requirements for the game to be developed include:

- Data security and user account information must be ensured.
- User interaction with the game must be easy to understand and user-friendly.
- Game response time must be fast and efficient.
- The game must have stable performance and not easily disrupted by external factors such as poor internet connection.

4) Game Concept

The concept of the game developed in this research is a strategy game where users can design and program behaviors of their characters, referred to as units in this game. Users can use flowchart symbols to design complex action sequences, such as attack strategies, avoiding enemy attacks, or maintaining specific positions in battle. Each flowchart symbol represents an action or decision to be taken by the unit, allowing users to create diverse strategies.

Before starting the battle simulation, users must assemble a team consisting of four unit members. Users can compose their team with various types of available units. Each unit type has unique abilities, skills, and roles in battle. Users can choose different unit types and flowcharts for each unit to complement or support each other, considering specific needs in particular battles. This process allows users to design a strong and effective team according to their gameplay style and objectives.

After assembling the team and programming the strategy, users can test the effectiveness of their strategy through battle simulations. In this simulation, the user's team will fight against the opponent's team, with behaviors programmed by the user beforehand. This simulation provides an opportunity for users to see the results of their created strategies, evaluate strengths and weaknesses, and make adjustments if necessary.

5) *Wireframe*

Based on the developed concept, a wireframe will be designed as the basis for creating the game interface.

- Flowchart Editor

The wireframe of the flowchart editor serves as an interface for designing flowcharts. In this view, users are presented with a list of their flowcharts, a workspace view for designing flowcharts, a list of node options that can be inserted, and tools for editing nodes.

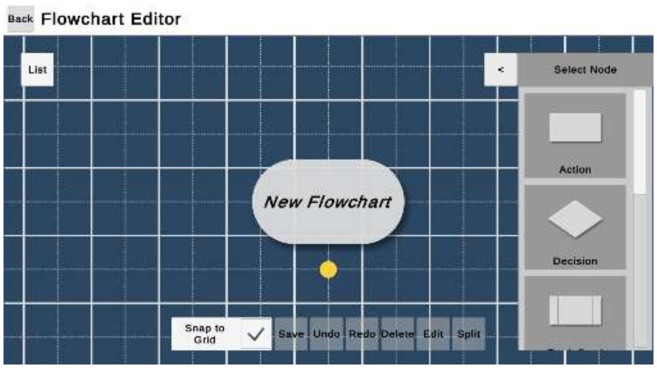


Fig. 7. Flowchart editor wireframe

- Team Setup

This wireframe includes a list of units and unit settings to manage names, unit types, and flowcharts of each unit. Users can organize the appropriate strategy to handle each challenge by arranging each unit individually and combining various unit types and flowcharts.

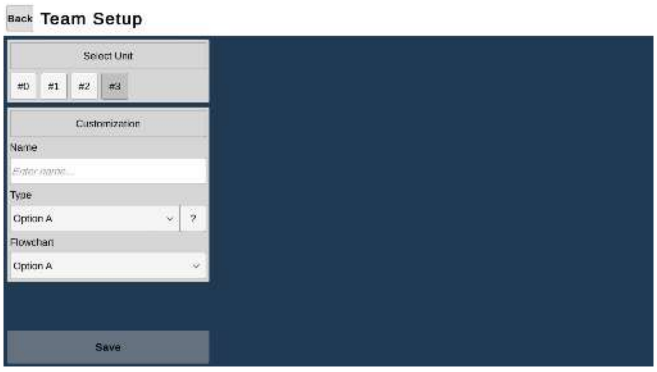


Fig. 8. Team setup wireframe

- Game

This wireframe functions to display battle simulations. In this view, users can observe the results of the flowcharts they have applied to their units. This view includes game controls such as play, pause, speed up, speed down, and current speed indicator. Additionally, there is a panel to monitor the status of the currently selected unit.

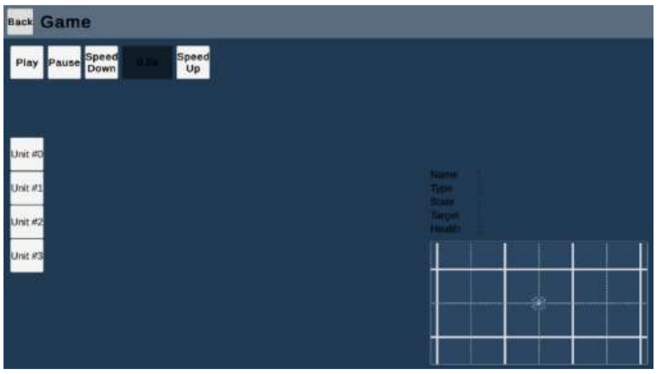


Fig. 9. Game wireframe

These wireframes provide a visual guide for developing the game interface and functionality according to the specified requirements and concepts.

B. *Pre-Production*

At this stage, the game concept that has been created will be further elaborated in the Game Design Document (GDD). The following Table 2 is the GDD that has been prepared for the game to be developed.

TABLE 2. GAME OVERVIEW

Game Title	Flowchart League
Platform	Android
Genre	Strategy
Description	Flowchart League is a strategy game where users have full control to design and program the behavior of their units using flowchart symbols. Through this mechanism, users can create complex and varied strategies to handle different battle situations. Users can assemble a team consisting of various unit types, each with unique abilities and roles, to create optimal formations and effective tactics.
Target User	All ages, especially those who are studying programming algorithms.

In the game Flowchart League, there are several key features that form part of the core mechanism of this game. These features include:

1. Flowchart Editor

This feature is used to design the action flow for units in the game. Through a drag-and-drop interface, users can compose, save, and organize the action flow of units using flowchart symbols. This feature simplifies the process of creating logical flows by providing easy-to-understand visualizations, allowing users to design unit logic flows more easily. The flowcharts created will then be converted into FSM models and used to manage transitions between various states in the game, ensuring that units can transition between states dynamically according to the specified conditions. The following are the flowchart symbols used in this feature along with their functions:

- Action: The Action symbol is used to set the state of the unit in the FSM. Users can specify the state to be performed, the target of the action in that state, and specific criteria for the target. For example, a unit can be programmed to attack the nearest enemy with a health filter above 50%.

- Decision: This symbol decides the direction of transitions between states in the FSM based on evaluated conditions. Additionally, this symbol has several additional parameters for further control, such as specifying the target to be evaluated under certain conditions, filtering targets based on specific criteria, and determining the number of targets to be evaluated.
- Predefined Process: This symbol can process other prearranged flowcharts. It functions to simplify the main flowchart by summarizing complex or frequently used sections into a single symbol.
- Comment: The comment symbol is used to provide additional annotations or explanations about a process in the flowchart. By giving context or additional descriptions, this symbol helps readers understand the intent and purpose of specific parts, reducing confusion and misunderstandings. In battle simulations, this symbol will not be executed.
- Input/Output: When the Input/Output symbol is executed, the related unit will display a chat balloon that can contain important information, commands, or inter-unit communication. These chat balloons can give instructions, display unit status, or communicate the results of actions in the simulation. Thus, this symbol not only runs workflow logic but also enriches user interaction and understanding of the battle situation.
- On Page Connector: This symbol connects visually separated workflow sections on the same page. It prevents overlapping lines or arrows that cause confusion and allows efficient use of space by connecting distant steps without requiring long lines.

2. Units and Teams

Units in this game refer to characters or entities that users can control. Each unit has unique roles and abilities and can be programmed to perform various actions in battle according to instructions given by users using flowchart symbols. There are various types of units that users can choose from, each with different skills and statistics. The FSM manages how each unit transitions between various states, such as attacking, moving, or defending, based on the conditions set by the user in the flowchart. For example, the FSM can transition from a MoveTo state to a FleeFrom state when detecting an enemy unit of a specific type. This means the unit type can be one of the conditions triggering the FSM transition, allowing for more dynamic and responsive strategies in the game. Some unit types in Flowchart League include:

- Assault: A balanced unit type in both offense and defense.
- Scout: The fastest unit type, ideal for hit-and-run tactics.
- Artillery: This unit can provide long-range support, capable of area damage and ignoring obstacles in its line of sight when attacking.
- Medic: Medics can support the team by healing or restoring the health of other units, helping them fight longer.
- Sniper: Specializes in long-range precision attacks, capable of dealing high damage and its attacks can penetrate any unit in sight.

The attribute comparison of each unit can be seen in the following Table 3.

TABLE 3. UNIT ATTRIBUTE

Type	Health Point	Attack Cooldown	Attack Range	Attack Radius	Damage	Move Speed
Assault	200	1.2s	15m	-	50	1m/s
Scout	150	0.5s	10m	-	25	1.5m/s
Artillery	300	1.5s	25m	5m	70	0.8m/s
Medic	175	1.2s	5m	-	30	1.2m/s
Sniper	175	1.5s	30m	-	90	1.2m/s

These units are then assembled into teams of four members, where users can organize strategies and tactics to achieve victory in battles. Additionally, users can program the behavior of each unit using flowcharts, allowing them to create complex and diverse strategies in battle.

3. Battle Simulation

After composing the flowcharts and assembling the team, users can start the battle simulation to test the effectiveness of their strategies. In the battle simulation, the user's team will face off against an opposing team with varied strategies in each challenge. In the battle simulation, the flowcharts applied to each unit will be transformed into FSM models. The FSM has full control over each unit's behavior in battle, managing state transitions based on the conditions specified in the flowchart. Users can observe the results of the battle simulation to evaluate their strategies, identify strengths and weaknesses, and make adjustments if necessary.

Each challenge can also have unique battle rules. These rules govern various aspects of the battle, such as the number of team members, maximum endurance limits, attack power multipliers, and other parameters. With these rules, each challenge becomes more diverse and challenging, as users must adapt their strategies to the set parameters to achieve victory. This encourages users to develop different strategic approaches depending on the battle conditions in each challenge.

C. Production

During the production phase, the game is developed based on the GDD and wireframe previously created. A primary focus at this stage is the implementation of FSM to control the units in the game. The FSM will be integrated with the game system to manage unit behavior according to the flowcharts assigned to them.

1) FSM Implementation

In this game, the FSM is responsible for managing unit behavior, controlling transitions between states according to the instructions in the flowchart created by the user. Table 4 provides an overview of several states that control character actions, along with a brief explanation of the actions taken in each state.

TABLE 4. FSM IMPLEMENTATION RESULTS

State	Action
Idle	• Play the idle animation
Move To	• Walk toward the target • Rotate toward the target • Play the move animation
Flee From	• Walk away from the target • Rotate away from the target • Play the move animation
Attack	• Rotate toward the target • Attack the target • Play the attack animation
Dead	• Play the dead animation

The states MoveTo, FleeFrom, and Attack require a target to execute their actions. The action of each state depends on the specified target, such as the MoveTo state needing a target to determine the unit's movement direction. When setting a target, users can also use filters to specifically target, allowing them to give more focused instructions to the unit. The following Table 5 is a list of available filters for each target.

TABLE 5. TARGET FILTERS

		Target				
		Current Target	Myself	Team	Enemy Unit	Ally Unit Current Target
Filter	Distance From Me	✓		✓	✓	✓
	Distance From Me In Range	✓		✓	✓	✓
	Distance Within Attack Range	✓		✓	✓	✓
	Health	✓	✓	✓	✓	✓
	Health Percentage	✓	✓	✓	✓	✓
	Health Compared to Mine	✓		✓	✓	✓
	Current State	✓		✓	✓	✓
	Attack Cooldown	✓	✓	✓	✓	✓
	Unit Type	✓	✓	✓	✓	✓

The implementation of FSM in this game's development will use the C# programming language and be integrated into Unity to control the units. Fig. 10. will provide a detailed overview of the class diagram structure, encompassing each FSM component, the relationships between these components, and the FSM's connections with other game elements. This diagram provides a clear visual representation of the FSM implementation in the game.

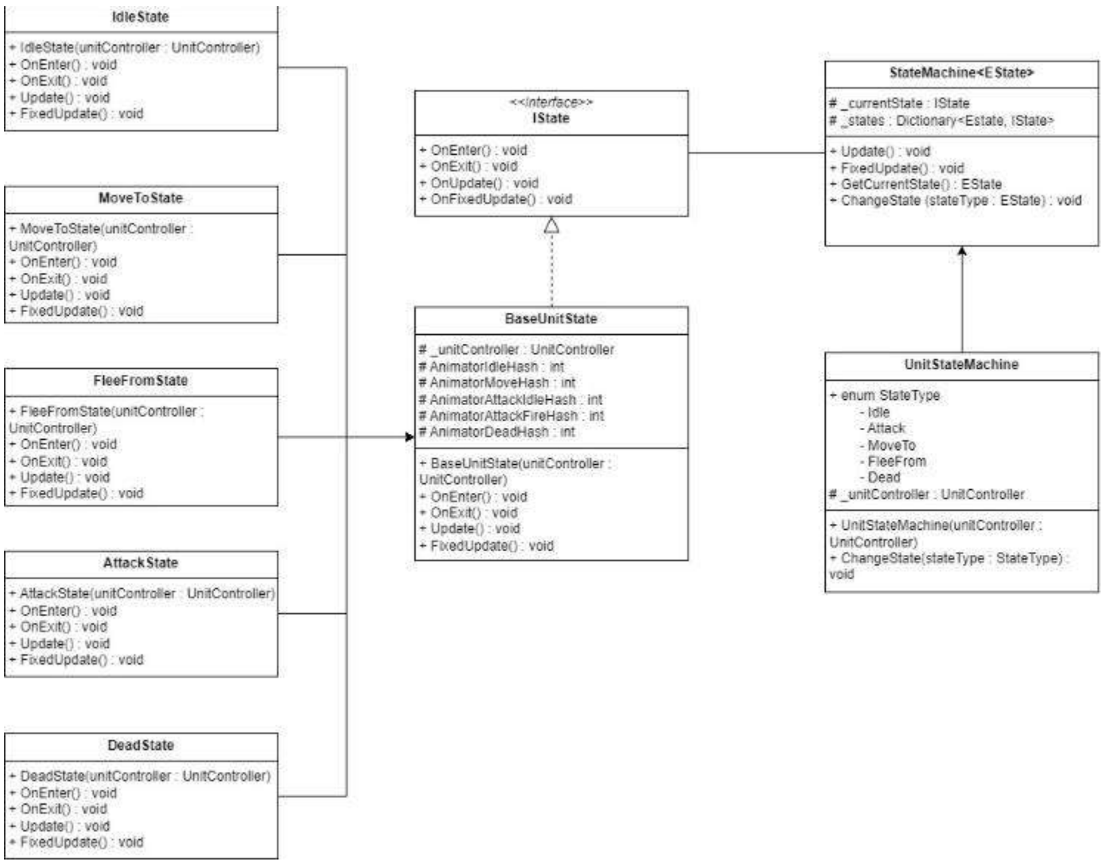


Fig. 10. FSM system class diagram

In this diagram, the application of FSM using the abstract class BaseUnitState, which implements the IState interface, is clearly visible. Additionally, there are several classes that inherit from BaseUnitState for each state. The StateMachine<EState> class is responsible for managing state transitions, while the UnitStateMachine class uses the FSM to control units in the game. The following is a script and explanation for each class mentioned in the class diagram.

1. IState Interface

The IState interface defines four basic methods that must be present in each state: OnEnter, OnExit, Update, and FixedUpdate. Through this interface, each state must have uniform basic behavior necessary for entering and exiting the state, as well as running per-frame and per-fixed frame logic. The interface specifies the four methods that each state must implement:

- OnEnter: Called when the object enters this state.
- OnExit: Called when the object exits this state.
- Update: Called every frame update.
- FixedUpdate: Called every fixed frame update.

2. BaseUnitState Class

BaseUnitState is an abstract class that implements IState and serves as the foundation for all unit states. This class is responsible for storing references to the UnitController and several animator hashes used to control animations.

3. IdleState, MoveToState, FleeFromState, AttackState, DeadState Classes

Each of these classes is a concrete implementation of each state defined in the StateType enum. Each state has different behaviors when entering, exiting, and being in that state. For example, AttackState plays the attack animation and attack logic when entering that state, while DeadState plays the death animation and logic when in the dead state.

4. StateMachine Class

The StateMachine class is a generic class responsible for managing state changes and storing the current state. The current state is stored in the _currentState variable, while all states are stored in a dictionary called _states. The Update() and FixedUpdate() methods ensure that the current state is updated regularly, while the ChangeState() method is used to transition between states.

5. UnitStateMachine Class

UnitStateMachine inherits from StateMachine and is tasked with managing various specific unit states such as Idle, Attack, MoveTo, FleeFrom, and Dead. The StateType enum defines all possible states, while the constructor initializes and sets the unit's initial state. The ChangeState() method is extended to notify state changes through the OnStateChanged event.

This FSM implementation provides a clear structure for managing the various states of units in the game. By abstracting each behavior into different states, the code becomes more modular, readable, and manageable. State changes are managed by the UnitStateMachine, ensuring smooth transitions between states and calling relevant logic at the appropriate times.

2) *Feature and Interface Implementation*

The following section outlines the design and implementation of features in the user interface of Flowchart League. These features and interfaces are designed based on the necessary aspects outlined in the GDD.

1. Sign In and Sign Up Panel

The Sign In panel, shown in Fig. 11, is the initial screen seen by users if they haven't authenticated within the game. Users must authenticate by filling out the email and password form. If users don't have an account, they can create one via the Sign Up panel.

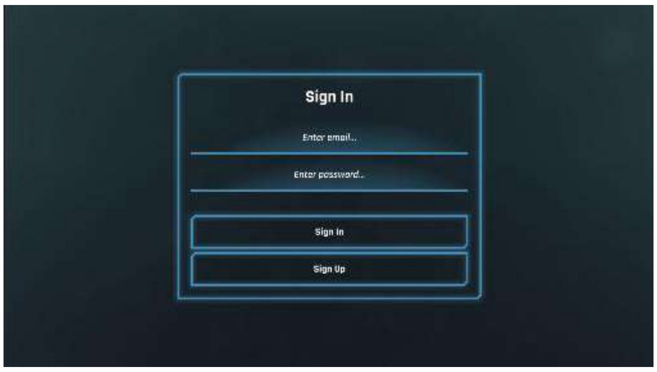


Fig. 11. Sign in and sign up panel

2. Main Menu Panel

The Main Menu panel functions as a navigation hub to other panels such as Challenge, Team Setup, Flowchart Editor, and Settings.

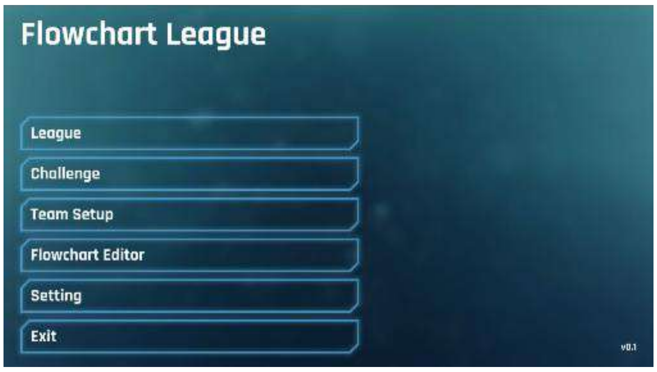


Fig. 12. Main menu panel

3. Settings Panel

The Settings panel allows users to manage game settings. Users can adjust the sound volume, change the language, copy their user ID, and log out.

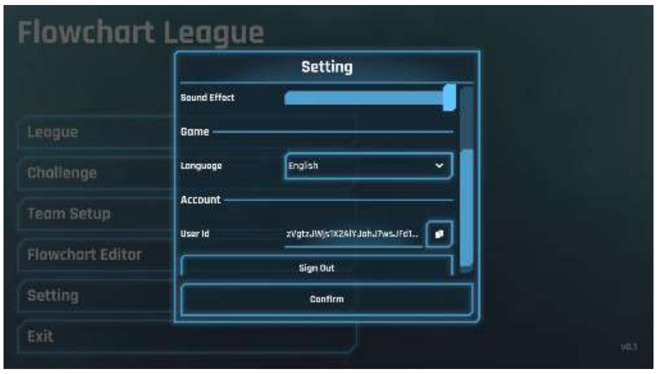


Fig. 13. Setting panel

4. Flowchart List in the Flowchart Editor Panel

When users select the Flowchart Editor menu from the Main Menu panel, they will see a list of flowcharts as shown in Fig. 14. Users can create new flowcharts or select existing ones to edit, rename, or delete.

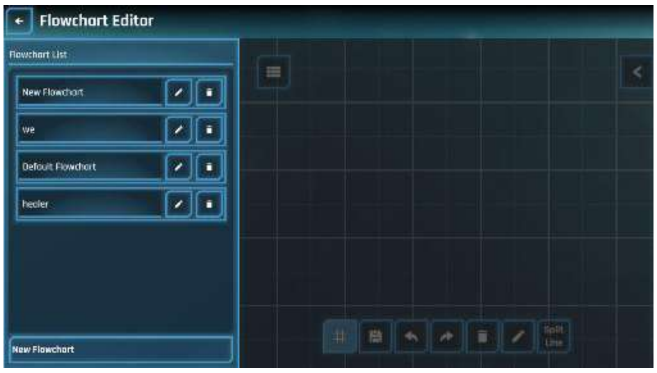


Fig. 14. Flowchart list in the flowchart editor panel

5. Workspace in the Flowchart Editor Panel

The Workspace in the Flowchart Editor panel serves as an interactive workspace where users can visualize the logic flow of their flowcharts. Various tools are available for adding, deleting, editing, and connecting flowchart symbols.

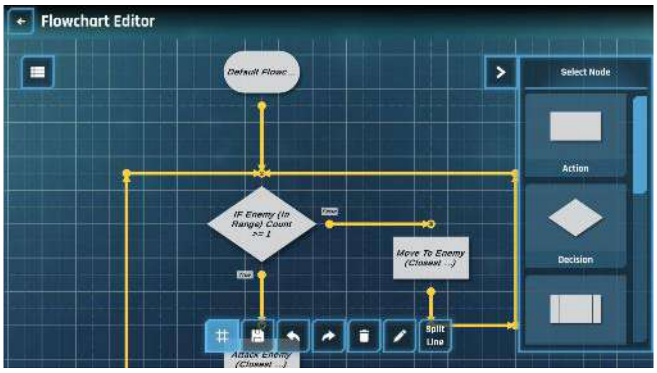

Flowchart Editor
Default Flowc...
IF Enemy (In Range) Count >= 1
False
True
Move To Enemy (Closest ...)
Attack Enemy (Closest ...)
Select Node
Action
Decision
Split Line

Fig. 15. Workspace in the flowchart editor panel

6. Edit Node Panel

When users use the tool to edit a symbol, the Edit Node panel will appear, as shown in Fig. 16. In this panel, users can modify various parameters related to the symbol.

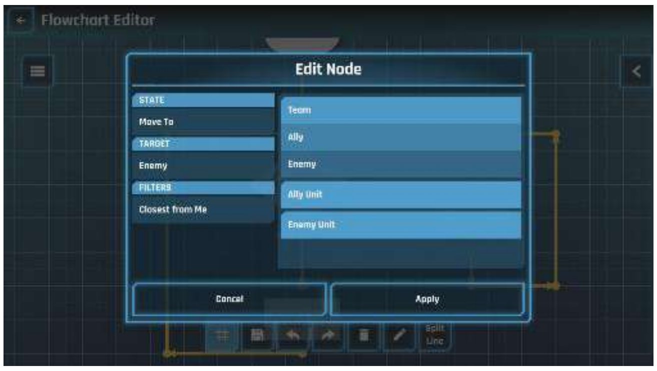


Fig. 16. Edit node panel

7. Team Setup Panel

This panel contains a list of units and unit settings that allow users to manage the name, unit type, and flowchart of each unit individually.

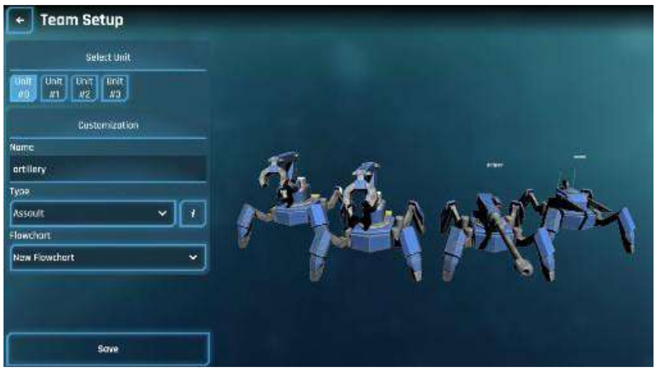
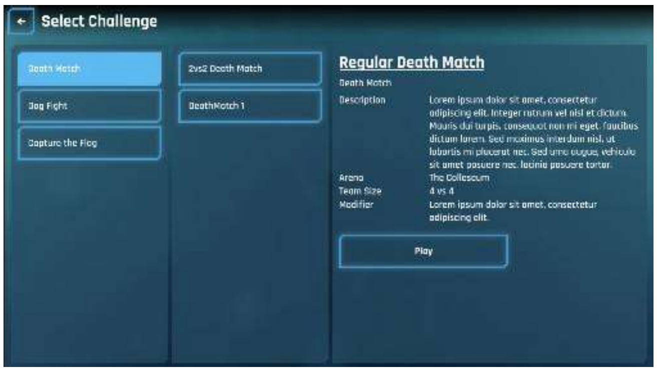


Fig. 17. Team setup panel

8. Select Challenge Panel

This panel includes a list of modes, available challenges, and detailed information about each challenge, such as name, mode, description, arena, and challenge rules. Users can select challenges from this panel.



Death Match

Description	Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer rutrum vel nisl et dictum. Mauris dui turpis, consequat non mi eget, faucibus dictum lorem. Sed maximus interdum nisl, ut lobortis mi placerat nec. Sed urna augue, vehicula sit amet posuere nec, lacinia posuere tortor.
Arena	The Colloseum
Team Size	4 vs 4
Modifier	Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Fig. 18. Select challenge panel

9. In-Game Panel

In this panel, users are presented with an isometric view of the battlefield. Users can see unit information such as name, unit type, current state, target, and health. There are also control options to speed up, slow down, or pause the battle.

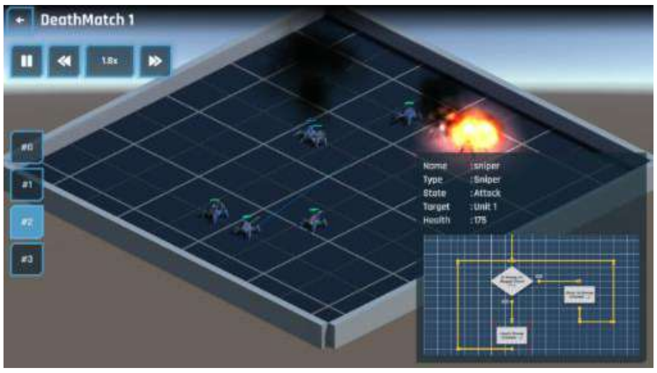


Fig. 19. In game panel

D. Alpha Testing

This testing phase focuses on validating the game's functions to ensure that all features work according to the specifications set out in the GDD. Below are some test scenarios implemented during the feature testing.

1) FSM Testing

The FSM testing aims to evaluate and ensure that unit behavior aligns with expectations. This testing references the FSM examples discussed in Chapter 3, which are converted into flowcharts and implemented on the units in the game. During testing, each unit is monitored to see how they react to various conditions and transitions defined in the flowchart. The testing includes various scenarios to ensure that units can smoothly transition between states such as Idle, MoveTo, Attack, and FleeFrom, and respond correctly to changes in conditions like distance from the enemy or health levels. The results of the FSM testing are as follows:

1. FSM Example 1

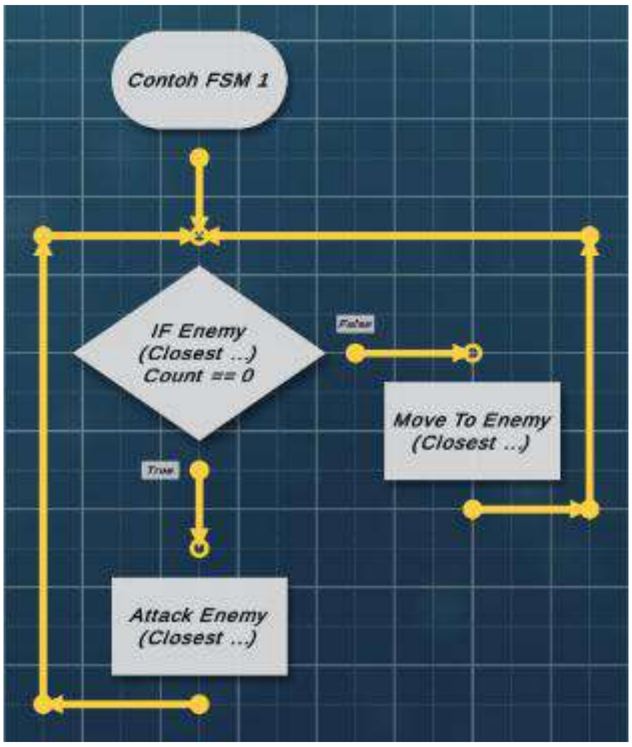

Contoh FSM 1
IF Enemy (Closest ...) Count == 0
False
True
Move To Enemy (Closest ...)
Attack Enemy (Closest ...)

Fig. 20. FSM example 1 implementation result

Fig. 20. shows the first FSM example rewritten as a flowchart within the game. The flowchart for FSM Example 1 is quite simple, consisting of two main states, MoveTo and Attack, with two reciprocal transitions between them. These state transitions are described as follows:

- MoveTo to Attack: The unit transitions to the Attack state and starts attacking the enemy if the distance to the nearest enemy is less than 10.
- Attack to MoveTo: This transition occurs when the distance between the unit and the nearest enemy is less than 10, continuously measured.

The FSM test results can be seen in Table 7. Based on this table, the FSM testing shows that each unit can transition between states according to the conditions set in the flowchart.

TABLE 1. FSM EXAMPLE 1 TEST RESULT

Initial State	Transition	End State	Result
MoveTo	• Distance from closest enemy <= 10	Attack	Passed
Attack	• Distance from closest enemy > 10	MoveTo	Passed

2. FSM Example 2

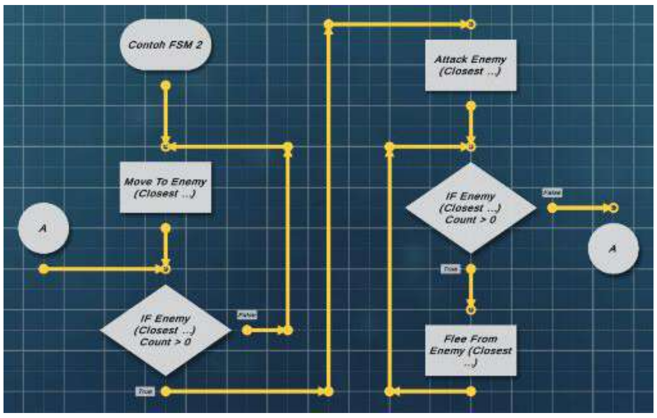

Contoh FSM 2
Move To Enemy (Closest ...)
A
IF Enemy (Closest ...) Count > 0
False
True
Attack Enemy (Closest ...)
IF Enemy (Closest ...) Count > 0
False
A
True
Flee From Enemy (Closest ...)

Fig. 21. FSM example 2 implementation result

Fig. 21. shows the second FSM example rewritten as a flowchart within the game. The flowchart for FSM Example 2 involves three states: MoveTo, Attack, and FleeFrom. These state transitions are described as follows:

- MoveTo to Attack: The unit transitions from MoveTo to Attack when the distance to the closest enemy is 10 or less.
- MoveTo to FleeFrom: The unit transitions from MoveTo to FleeFrom when the closest enemy's health is higher than its own.
- Attack to FleeFrom: The unit transitions from Attack to FleeFrom when the closest enemy's health is higher than its own.
- FleeFrom to MoveTo: The unit transitions from FleeFrom to MoveTo when the closest enemy's health is lower than its own.

The FSM test results can be seen in Table 8. Based on this table, the FSM testing shows that each unit can transition between states according to the conditions set in the flowchart.

TABLE 2. FSM EXAMPLE 2 TEST RESULT

Initial State	Transition	End State	Result
MoveTo	• Distance from closest enemy <= 10	Attack	Passed
MoveTo	• Health of closest enemy > my health	FleeFrom	Passed
Attack	• Health of closest enemy > my health	FleeFrom	Passed
FleeFrom	• Health of closest enemy < my health	MoveTo	Passed

3. FSM Example 3

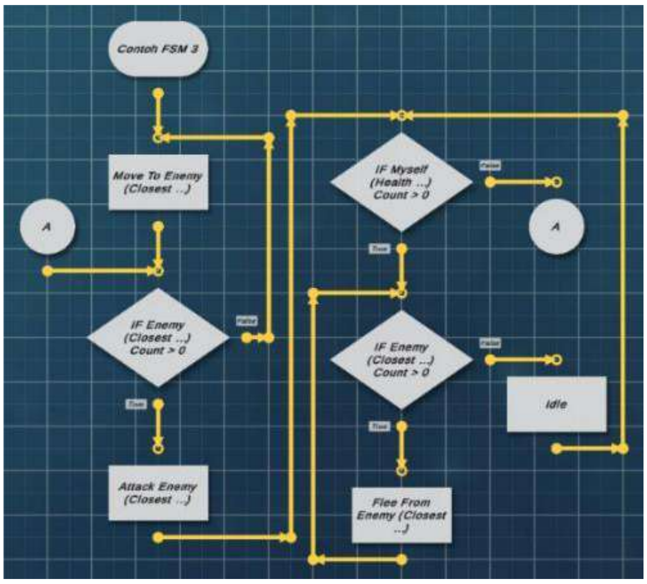

Contoh FSM 3
Move To Enemy (Closest ...)
A
IF Myself (Health ...) Count > 0
False
True
A
IF Enemy (Closest ...) Count > 0
False
True
IF Enemy (Closest ...) Count > 0
False
True
Idle
Attack Enemy (Closest ...)
Flee From Enemy (Closest ...)

Fig. 22. FSM example 1 implementation result

Fig. 22. shows the third FSM example rewritten as a flowchart within the game. This flowchart is the most complex, with four states: MoveTo, Attack, FleeFrom, and Idle. These state transitions are described as follows:

- MoveTo to Attack: The unit transitions from MoveTo to Attack when the distance to the nearest enemy is 10 or less.
- Attack to MoveTo: The unit transitions from Attack to MoveTo if its health drops to 50% or less, but the enemy is still close.
- FleeFrom to Idle: The unit transitions from FleeFrom to Idle when the distance to the nearest enemy is more than 10.
- Idle to FleeFrom: The unit transitions from Idle to FleeFrom if its health drops to 50% or less, and the enemy is close.
- Idle to Attack: The unit transitions from Idle to Attack if its health is more than 50%, and the enemy is close.
- Idle to MoveTo: The unit transitions from Idle to MoveTo if its health is more than 50%, but the enemy is still far.

The FSM test results can be seen in Table 9. Based on this table, the FSM testing shows that each unit can transition between states according to the conditions set in the flowchart.

TABLE 3. FSM EXAMPLE 3 TEST RESULT

Initial State	Transition	End State	Result
MoveTo	• Distance from closest enemy <= 10	Attack	Passed
Attack	• Health <= 50% • Distance from closest enemy <= 10	MoveTo	Passed
FleeFrom	• Distance from closest enemy > 10	Idle	Passed
Idle	• Health <= 50% • Distance from closest enemy <= 10	FleeFrom	Passed
Idle	• Health > 50% • Distance from closest enemy <= 10	Attack	Passed
Idle	• Health > 50% • Distance from closest enemy > 10	MoveTo	Passed

2) *Feature Testing*

Alpha testing using Blackbox Testing, specifically Functional Testing, ensures that all core features of the Flowchart League game function as expected. Each test case is designed to verify the functional aspects of various game features, including flowchart creation and management, unit and team setup, battle simulations, and account management. The expected outcomes of each test case must be achieved to ensure the quality and stability of the game.

TABLE 4. FEATURE TESTING RESULT

Scenario	Expected Result	Steps	Result
User successfully logs into the game	User successfully logs into the game and is directed to the Main Menu panel.	1. Open the Flowchart League application. 2. On the Sign In panel, enter a valid email and password. 3. Click the Sign In button.	Passed
User fails to log in with incorrect credentials	An error message appears informing that the email or password is incorrect.	1. Open the Flowchart League application. 2. On the Sign In panel, enter an invalid email and password. 3. Click the Sign In button.	Passed
User successfully registers a new account	User account successfully registered.	1. Open the Flowchart League application. 2. On the Sign Up panel, enter a valid email and password. 3. Click the Sign Up button.	Passed
Exiting the game	The application will close.	1. On the Main Menu panel, click the Exit button.	Passed
Navigating to the Settings panel	User is directed to the Settings panel.	1. On the Main Menu panel, click the Settings button.	Passed
Navigating to the Flowchart Editor panel	User is directed to the Flowchart Editor panel.	1. On the Main Menu panel, click the Flowchart Editor button.	Passed
Navigating to the Team Setup panel	User is directed to the Team Setup panel.	1. On the Main Menu panel, click the Team Setup button.	Passed
Navigating to the Challenge panel	User is directed to the Challenge panel.	1. On the Main Menu panel, click the Challenge button.	Passed

Adjusting the volume settings	Sound volume changes according to the specified settings.	1. In the Settings panel, adjust the sound settings.	Passed
Changing the language	Game language changes according to the specified settings.	1. In the Settings panel, change the language settings.	Passed
Logging out	User successfully logs out and is directed back to the Sign In panel.	1. In the Settings panel, click the Log Out button.	Passed
Creating a new flowchart	A new flowchart is successfully created and appears in the flowchart list.	1. In the Flowchart Editor panel, click the New Flowchart button.	Passed
Renaming a flowchart	Flowchart name change is successfully saved.	1. In the Flowchart Editor panel, click the Rename Flowchart button. 2. Modify the flowchart name.	Passed
Deleting a flowchart	Flowchart is successfully deleted from the list.	1. Select a flowchart from the list. 2. Click the Delete button.	Passed
Editing a flowchart	Flowchart is successfully edited and changes are saved correctly.	1. Select a flowchart from the list. 2. Add, remove, or edit the flowchart. 3. Save changes.	Passed
Modifying units within a team	Changes to units are successfully saved.	1. Select a unit from the team list. 2. Modify the unit's name, type, and assigned flowchart. 3. Save changes.	Passed
Selecting a challenge	Challenge is successfully selected and challenge information is displayed.	1. In the Select Challenge panel, choose one of the available challenges.	Passed
Initiating a battle simulation	Battle simulation starts correctly, units engage according to the configured flowchart.	1. In the Select Challenge panel, select a challenge and click Start. 2. Observe the battle simulation.	Passed
Using game controls during simulation	Game controls function correctly and affect the course of the simulation according to commands.	1. During the battle simulation, use the play, pause, speed up, and speed down controls.	Passed

V. CONCLUSIONS

The implemented Finite-State Machine (FSM) framework demonstrably provides flexible and dynamic control over character behaviour in Unity-based game prototypes. Empirical testing showed that agents reliably transition between states (Idle, MoveTo, Attack, Flee) in response to contextual cues such as distance to enemies and health levels. Moreover, novice users were able to construct functional flowcharts with minimal coding effort, confirming the claim that visual FSM design lowers the barrier to entry for gameplay scripting.

Despite these successes, the study reveals two noteworthy constraints. First, the steep learning curve for newcomers emerged when participants attempted to design complex state networks; over-engineered diagrams often led to "tunnel-vision" and unintuitive transitions. Second, the visual scalability of the drag-and-drop editor proved inadequate for diagrams exceeding five nodes, causing clutter and reduced comprehension. These limitations suggest that the current tool favours simplicity over depth, and that additional scaffolding (e.g., automated state-validation or hierarchical sub-FSMs) is needed to sustain larger-scale designs.

Building on the present work, we propose three concrete research directions in the future, namely: (1) AI-assisted design feedback – embed a constraint-checking engine that flags unreachable states, dead-ends, or contradictory transitions in real-time; (2) Hierarchical & multi-agent FSMs – extend the framework to coordinate squads or teams, exploring how distributed state machines can model collective tactics; and (3) Pedagogical scaffolding – develop guided tutorials and "design-patterns" libraries that teach learners systematic FSM construction while preserving creative freedom. Pursuing these paths will not only improve the usability of visual programming tools but also deepen our understanding of how systemic thinking can be cultivated through game-based learning environments.

REFERENCES

[1] S. Schez-Sobrino, D. Vallejo, C. Glez-Morcillo, M. A. Redondo and J. J. Castro-Schez, "RoboTIC: A serious game based on augmented reality," Springer Science+Business Media, LLC, part of Springer Nature 2020, 2020.

[2] S. Grey and N. A. Gordon, "Motivating Students to Learn How to Write Code Using a Gamified Programming Tutor," Educ. Sci. 2023, 13(3), 230, 2023.

[3] A. Vahldick, P. R. Farah, M. J. Marcelino and A. J. D. Mendes, "A blocks-based serious game to support introductory computer programming in undergraduate education," Computers in Human Behavior Reports, 2020.

[4] J. P. Silva, I. F. Silveira, L. Kamimura and A. T. Barboza, "Turing Project: An Open Educational Game to Teach and Learn Programming Logic," in 2020 15th Iberian Conference on Information Systems and Technologies (CISTI), Seville, 2020.

[5] C. Schrader, "Serious Games and Game-Based Learning," in Handbook of Open, Distance and Digital Education, Singapore, 2023.

[6] H. Kayakoku, M. S. Guzel, E. Bostanci, I. T. Medeni and D. Mishra, "A Novel Behavioral Strategy for RoboCode Platform Based on Deep Q-Learning," Complexity, 2021.

[7] B. A. Noval, M. Safrodin and R. Y. Hakkun, "Battlebot : Logic Learning Based on Visual Programming Implementation in Multiplayer Game Online," in 2019 International Electronics Symposium (IES), Surabaya, 2019.

[8] N. Bak, B.-M. Chang and K. Choi, "Smart Block: A visual block language and its programming environment for," Journal of Computer Languages, 2020.

[9] K. Charntaweekhun and S. Wangsiripitak, "Visual Programming using Flowchart," ISCIT 2006, 2006.

[10] Sugiyanto, G. Fernando and W.-K. Tai, "A Rule-Based AI Method for an Agent Playing Big Two," Applied Sciences, 2021.

[11] I. Millington, AI for Games, Boca Raton: CRC Press, 2019.

[12] Z. Hu, C. Fan, Q. Zheng, W. Wu and B. Liu, "Asyncflow: A visual programming tool for game artificial intelligence," Visual Informatics, 2021.

[13] E. W. Hidayat, A. N. Rachman and M. F. Azim, "Penerapan Finite State Machine pada Battle Game Berbasis Augmented Reality," Jurnal Edukasi dan Penelitian Informatika, 2019.

[14] M. A. Kuhail, S. Farooq, R. Hammad and M. Bahja, "Characterizing Visual Programming Approaches for End-User Developers: A Systematic Review," IEEE Access, 2021.

[15] G. Huang, P. S. Rao, M.-H. Wu, X. Qian, S. Y. Nof, K. Ramani and A. J. Quinn, "Vipo: Spatial-Visual Programming with Functions for Robot-IoT Workflows," in CHI '20: Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems, New York, 2020.

[16] W. E. Susanto and A. Syukron, Logika dan Algoritma untuk Pemula, Yogyakarta: Graha Ilmu, 2020.

[17] A. B. Chaudhuri, Flowchart and Algorithm Basics: The Art of Programming, Virginia: MERCURY LEARNING AND INFORMATION, 2020.

[18] G. N. Yannakakis and J. Togelius, Artificial Intelligence and Games, New York: Springer, 2018.

[19] R. Ramadhan and Y. Widyani, "Game development life cycle guidelines," in Conference: 2013 International Conference on Advanced Computer Science and Information Systems, Bali, 2013.