BAREKENG: Journal of Mathematics and Its Applications
June 2023 Volume 17 Issue 2 Page 0955–0968
P-ISSN: 1978-7227 E-ISSN: 2615-3017
https://doi.org/10.30598/barekengvol17iss2pp0955-0968

IMPLEMENTATION AND COMPARISON IN USING STATE PATTERN ON MAIN CHARACTER MOVEMENT (CASE STUDY: POCONG JUMP VIDEO GAME VERSION 1.0)

Sanriomi Sintaro[1], Deiby Tineke Salaki[2], Luther Alexander Latumakulita[3*], Mahardika Inra Takaendengan[4], Bernard[5], Ade Surahman[6], Noorul Islam[7]

[1,3,4] Information System Study Program, Faculty of Mathematic and Nature Science, Sam Ratulangi University*
Bahu, Malalayang, Manado City, North Sulawesi, Indonesia

[2] Mathematic Study Program, Faculty of Mathematic and Nature Science, Sam Ratulangi University
Bahu, Malalayang, Manado City, North Sulawesi, Indonesia

[5] Informatic Department, Faculty of Engineering and Computer Science, Teknokrat Indonesia University
Jl. ZA. Pagar Alam No.9 -11, Labuhan Ratu, Kedaton, Bandar Lampung, Lampung, Indonesia

[6] Computer Engineering Department, Faculty of Engineering and Computer Science, Teknokrat Indonesia Univerisy
Jl. ZA. Pagar Alam No.9 -11, Labuhan Ratu, Kedaton, Bandar Lampung, Lampung, Indonesia

[7] Kanpur Institute of Technology, Kanpur, India

A1, UPSIDC Industrial Area, Chakeri Ward, Rooma, Uttar Pradesh 208001, India

*Corresponding author e-mail: * latumakulitala@unsrat.ac.id*

Article History:

Received: 14th January 2023
Revised: 24th April 2023
Accepted: 28th April 2023

Keywords:

Game;
Design pattern;
Finite state machine

ABSTRACT

Game development success is often hard to achieve due to various problems such as performance issues, malfunctioning features, and poorly organized program structure. The problems that arise can be prevented by using the design pattern as a game programming architecture from the beginning of development. By implementing a design pattern, the process of developing video games can be made easier and simplified. The development team can focus its efforts on producing better quality video games. In this study, design patterns that would be used are state patterns and finite state machines. The state pattern is implemented by encapsulating the character's behavior in a class called state. The finite state machine will then facilitate the transition of states caused by user/player input or variable value changes. State pattern and the finite state machine are tested with a test case and game performance is tested with software metrics. The result obtained from this study are state patterns and finite state machines have a valid component structure and could improve performance efficiency in video games.

How to cite this article:

S. Sintaro, Bernard, A. Surahman, L. A. Latumakulita, M. I. Takaendengan and N. Islam., "IMPLEMENTATION AND COMPARISON IN USING STATE PATTERN ON MAIN CHARACTER MOVEMENT (CASE STUDY: POCONG JUMP VIDEO GAME VERSION 1.0)," *BAREKENG: J. Math. & App.*, vol. 17, iss. 2, pp. 0955-0968, June, 2023.

Journal homepage: *https://ojs3.unpatti.ac.id/index.php/barekeng/*
Journal e-mail: *barekeng.math@yahoo.com; barekeng.journal@mail.unpatti.ac.id*

Research Article • Open Access

1. INTRODUCTION

Eternal Dream is a video game development startup company located in Bandar Lampung City, Lampung and has been established since 2017. This development team is one of the video game developers that has experienced development in Indonesia. The revenue earned by video game developers in Indonesia in 2019 reached $1.08 Billion [1].

Based on the information provided by Lucky Putra Dharmawan, the CEO of Eternal Dream and who at that time was the programmer of the Video Game Pocong Jump, Pocong Jump had problems in program structure, functionality, and performance. The structure of the programs contained in video games is not neatly arranged and systematically. In addition, Pocong Jump also has problems with inefficient memory allocation, ineffective coroutine use, and improper use of animations, resulting in video games experiencing poor performance when running on platforms that have 1GB of RAM. In Game Development, we have to use Research and Development to make games. One of the ways to make R&D is to do a comparison between the previous program code and the better program code to produce increased performance [2]. In this research, we are using design pattern for the code.

The use of design patterns as a game programming architecture has been widely applied in the video game development process, with the aim of simplifying and simplifying the video game development process so that it can be easily understood by the development team [2]. We also must to awareness about Context awareness and adaptation, which are crucial aspect in mobile game based learning [3]. The use of design pattern has multiple benefits for game software, because hardware will communicate with input, data transform, and output [4]. The development team can allocate its resources to build video games that have better performance, functionality, and structure, because of which the quality of the video games developed improves. One of the design patterns that can be used to improve video game quality is to apply a state pattern [5], [6] with a finite state machine [5]. FSM not only can be used for games but also another platforms, such as medical area, because FSM is an abstract machine that can allow the storage and also processing of all information with order-sensitive patterns [7].

In this research, We decided to use state pattern which can be implemented in a synchronous approach with FSM [8]. The authors will apply state patterns and finite state machines to Pocong characters in eternal dream's Pocong Jump video game. The state pattern encapsulates the character's behavior on the Pocong character then creates three states, namely silence, jump, and death. The finite state machine oversees handling the transfer between states and then the system will execute the currently active state.

2. RESEARCH METHODS

This research was used to find whether using the state pattern makes the video game develop better. This was done because A well-developed video game will give players a special gaming experience [9], [10]. Video games have several genres referred to as "The Classics Game Genre". These genres are shooter, action and arcade, platformer, fighting, strategy, role-playing, sport, vehicle, construction and simulations, adventure, and puzzle. Meanwhile, in this study, the Pocong jump video game has a platformer genre where Platformer is one of the genres of video games where characters move by jumping from one platform to another. Platformers give players a challenge that can be fighting or avoiding a trap [11]. Platformers can be divided into two types, namely single-screen platformers and scrolling platformers. In a single-screen platformer, characters can only move within a scope as large as the monitor screen, while in scrolling platformers, the screen moves following the character's movements [12].

2.1 Data Collection Technique

2.1.1 Interview

The interview method is carried out by conducting interview directly with the source. The author conducted an interview with Lucky Putra Dharmawan, the CEO of Eternal Dream and who at that time was the programmer of the video game Pocong Jump.

2.1.2 Observation

The observation method is carried out by studying the behavioral function of the Pocong character by playing the Pocong Jump video game and reading the program code contained in the Pocong character. Here is the flowchart of the program code on the Pocong character that we can see in **Figure 1** below.

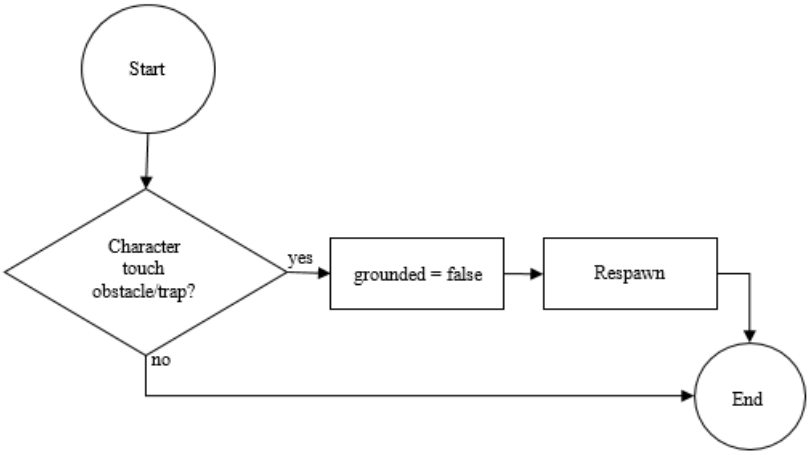

Figure 1. Functions inside OnTriggerEnter2D

Figure 1 shows the simple logic for playing this game is when the main character touches an obstacle or trap that we put somewhere inside the game. The main character will respawn if the main character touch the ground. For this logic, we can put ground below the screen, so the player cannot see it, but that logic will make the main character respawn when fall into the trap that not visible on the screen. **Figure 2**, **Figure 3** and **Figure 4** are made to Shown an update function, it will be use to set the main character movement and other feature that we put inside the game.

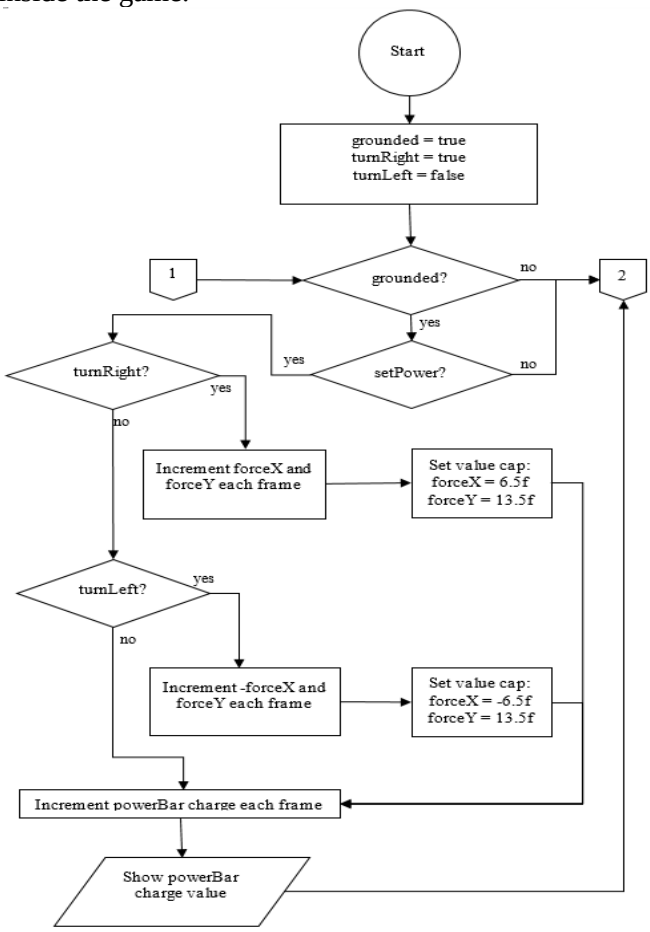

Figure 2. Update Function

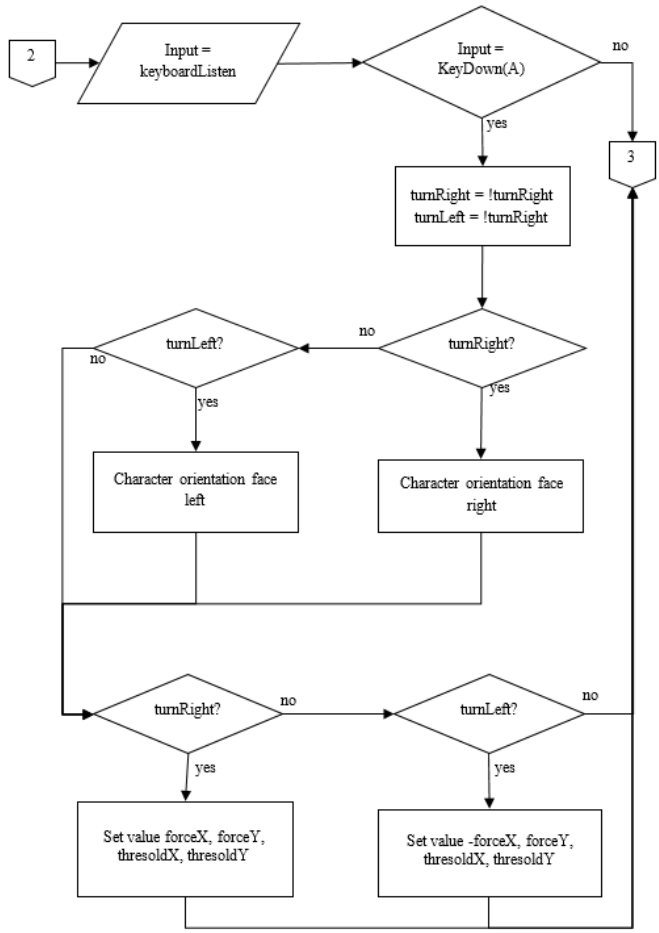

Figure 3. Update Function (2)

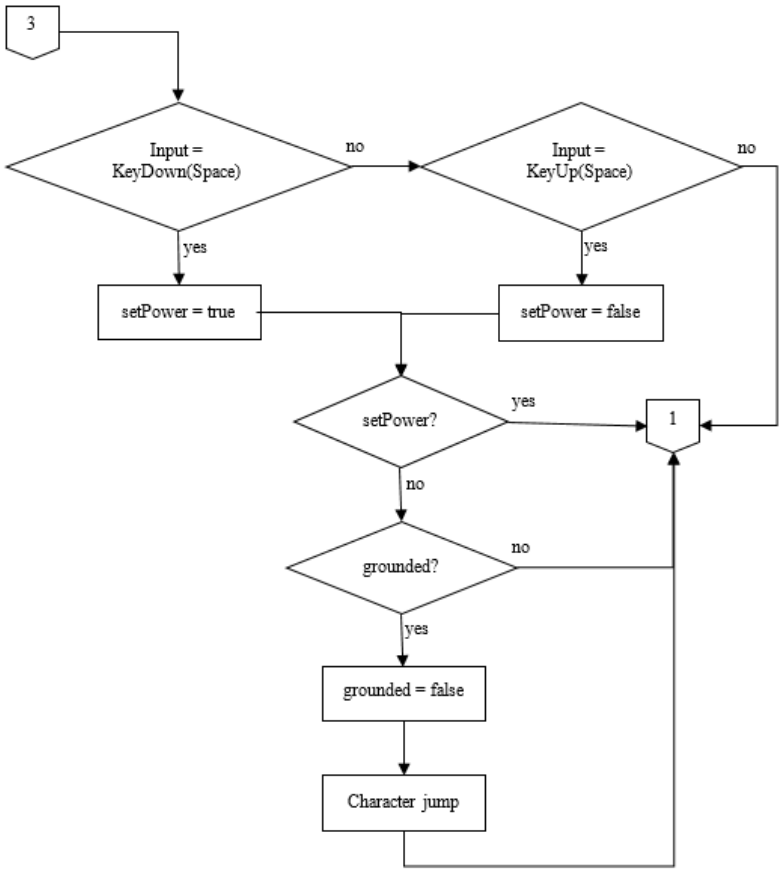

Figure 4. Update Function (3)

Figure 2, **Figure 3**, and **Figure 4** shown above show the update process for the character for moving and control. As we can see in **Figure 2**, at the start program will ask abour where player is, whether it is on ground and is it mirrored or not. When the program knows that the character is at the ground, it will check about setPower, and setPower are true then the last check checks the mirrored position of a character. If it turnRight (not mirrored), hen the program will calculate increment forceX and forceY for each frame and set the value cap for forceX to 6.5f and forceY to 13.5f while it is mirrored (turn left). The program will calculate the different values. While the forceY is still the same value, forceX will be set to -6.5f, which means that the X axis will send back to the left of the character. In this progress and calculation, the program will show the increment of the PowerBar charge value and show the user the charge meter. This progress will show no change to the character because we do not input anything to the character when the program starts, but the process will continue to **Figure 3**. To change the direction of the character, shown in **Figure 2** that the program will take the input and check the right input, we can see that if we press KeyDown(A) while the character is faceRight, it will change the direction to faceLeft and if we press KeyDown(A) while the character is faceLeft, it will change the direction to faceRight. While the program knows the direction of the character, this process will continue to keep checking the forceY, forceY, thresoldX, and thresoldY. The last step in **Figure 4**, shows that the user press the input KeyDown(Space), program will checking the value of power and if the power already released the program will check the character are on the ground or not, the character will jump with the certain value that the program already collect before.

2.2 Analysis Method

2.2.1 Research Framework

The research stage is needed so that the research carried out is carried out regularly and systematically to achieve the goal. The stages of research carried out in this study can be seen in the form of a diagram in **Figure 5**.

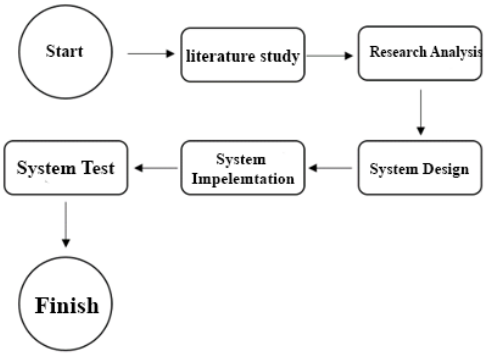

Figure 5. Research Stage

Figure 5 shows the steps for doing this research. The first step is a literature study, like collecting the data and interviews. After that, we do some research analysis to determine the core problem to search for a solution. After that, we do the system design that will manage the next steps, which are system implementation, the code is made, and after that, the last step is collecting the data with a system test. All of the data that are collected will then be used for the result of this research.

2.2.2 Designing the State Pattern

The state pattern is a class that takes the form of *an interface*. The class has several components that will be passed on to the child called the *state*. The components contained in *the state pattern* will be executed inside the *character* class *loop.* In the Unity *game engine*, the class is the class that implements the *MonoBehaviour* class. The *controller class* plays a role in helping *the finite state machine* function to regulate the course of transitions between *states*. The *state pattern* class diagram can be seen in **Figure 6**.

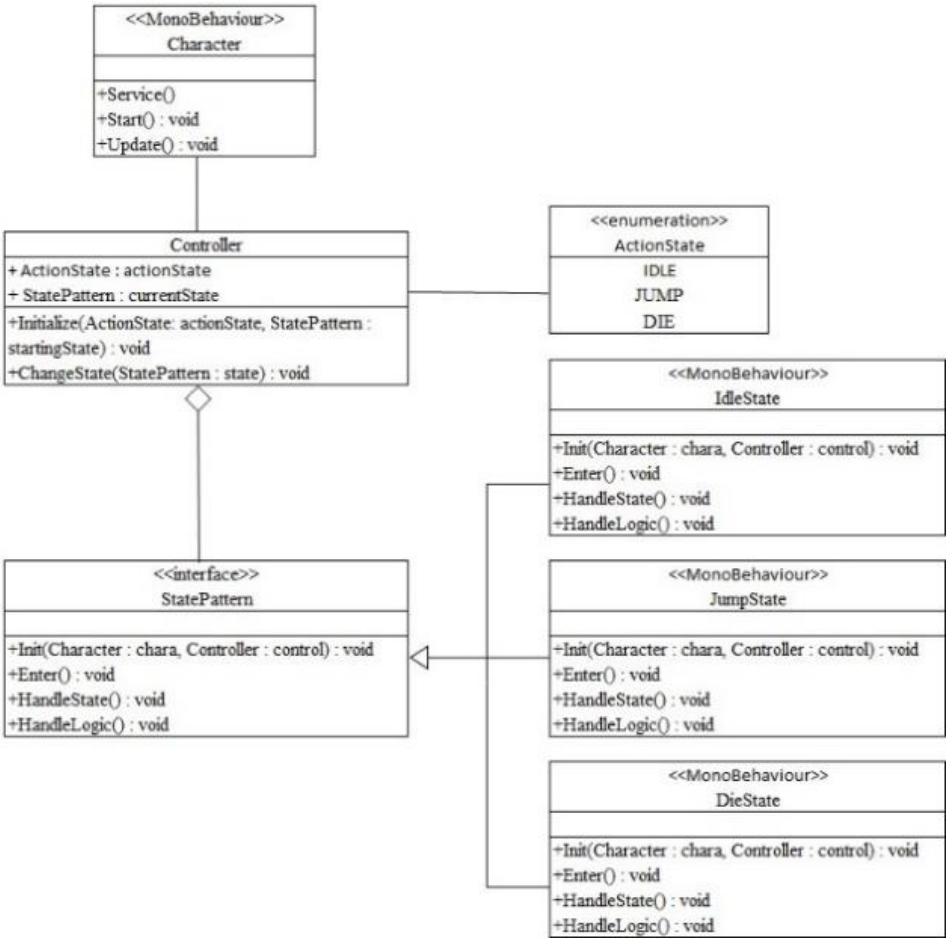

Figure 6. State Pattern Class Diagram

Figure 6. There are four components contained in the state pattern. These components are Init (Initialization), Enter, HandleState, and HandleLogic. The Init (Initialization) component contains the initial variables of character behavior within each of the states where the video game has just started. The Enter component is a component that is executed at the beginning of one time each state switches transitions. The HandleState component contains a finite state machine function where the system checks the conditions that regulate the displacement between states. The HandleLogic component executes the main line of program code character behavior.

2.2.4 Designing the Finite Stage Machine

Finite state machines are used to regulate transitions between states that occur within the state pattern [13]. In Figure 7. the design of the finite state machine is depicted using a state machine diagram.

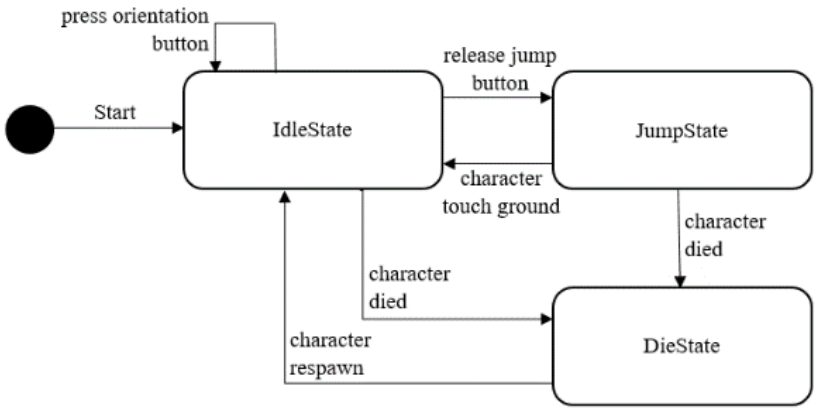

Figure 7. State Machine Diagram.

Figure 7. There are three states in the finite state machine system. Those states are IdleState, JumpState, and DieState. At the beginning of the first time the game is run, the character will enter the IdleState. In this state, the character stays in place and the player can change the orientation of the Pocong character to face left or right.

When the player presses and holds the jump button, the Pocong character will fill the jump bar to determine the strength of the jump. When the player releases this button, the state will transition to jumpstate. In this state, Pocong characters will perform jumping actions to move or avoid obstacles. When the character hits the ground, the state will move back to IdleState.

When the character is hit by a trap that causes lives to decrease, the system will force the character to make a transition to DieState. In this state, the character will play a dying animation according to how the Pocong character died. The system then runs a three-second countdown to return the character to the checkpoint position (respawn).

3. RESULTS AND DISCUSSION

3.1. State Pattern Implementation

The implementation of the state pattern begins with creating an interface to facilitate the development and maintenance of video games. Interfaces are also used to homogenize structures within the state class.

```
INTERFACE IstatePattern
void Init (parameters: chara, controller)
void Enter()
void HandleLogic()
void HandleState()
```

In *the* IStatePattern interface, there are four framework *methods* in the IStatePattern interface, namely Init, Enter, HandleLogic, and HandleState. One example of an implementation of *the* IStatePattern interface is the creation of a *state* that will be shown in the following IdleState class.

```
Class IdleState IMPLEMENT MonoBehaviour EXTEND IStatePattern
Initialization _character As Character
Initialization _stateController As Controller

void Init (parameters: chara, controller)
begin
  _character ← chara
  _stateController ← control
end

void Enter()
begin
  CALL _character.AnimController.SetBool WITH "isGrounded", true
End Enter

void HandleLogic()
begin
  IF keyboard input key down IS 'A'
    _character.SpriteRender.flipX ← NOT _character.SpriteRender.flipX
  ENDIF

  IF keyboard input key IS 'Space'
    Initialization _jumpForce As Vector2
    _jumpForce ← _character.JumpCharge
    _jumpForce ← _jumpForce + (_character.MultiplierJumpForce * Time.deltaTime)
    _jumpForce ← MINIMUM VALUE OF_jumpForce, _character.MaxJumpForce
    _character.JumpCharge ← _jumpForce;
  ENDIF

  IF keyboard input key release IS 'Space'
    _stateController.ActionState ← Controller.State.JUMP
  ENDIF
END HandleLogic

void HandleState()
begin
  CASE _stateController.ActionState OF
    Controller.State.JUMP : CALL _stateController.ChangeState WITH _character.jumpState
    Controller.State.DIE : CALL_stateController.ChangeState WITH _character.dieState
  ENDCASE
END HandleState
```

```
void OnTriggerEnter2D (parameter: target)
begin
  IF target.gameObject.tag IS MEMBER OF _character.ObstacleTags
    _character.CauseOfDeath ← target.gameObject.tag;
    _stateController.ActionState ← Controller.State.DIE;
    RETURN null;
  ENDIF
end OnTriggerEnter2D (parameter: target)
```

The IdleState class is built by *implementing* MonoBehaviour and *extending* the IStatePattern interface. The MonoBehaviour implementation is used to access Unity components, in this case, accessing collider2D components to use the OnTriggerEnter2D function.

State initialization *is* executed when the *video game* starts inside the Awake *method* inside the Character class. *The* HandleLogic and HandleState methods are executed once each *frame* is inside the Update *method* inside the Character class. The program below shows the Character class.

```
Class Character IMPLEMENT MonoBehaviour
Initialization StateController As Controller
Initialization idleState As IdleState
Initialization jumpState As JumpState
Initialization dieState As DieState

void Awake()
Begin
  CALL idleState.Init WITH this, StateController
  CALL jumpState.Init WITH this, StateController
  CALL dieState.Init WITH this, StateController
End Awake

void Update()
Begin
  CALL StateController.CurrentState.HandleLogic
  CALL StateController.CurrentState.HandleState
End Update
```

In the Controller class, the program calls *the Enter method* to execute the program in The CurrentState once each state switch occurs. The program below shows the Controller class.

```
Class Controller
Enumeration State
Begin
  IDLE
  JUMP
  DIE
End State

Initialization ActionState As State
Initialization CurrentState As IStatePattern

void Init (parameters: actionState, startingState)
Begin
  ActionState ← actionState
  CurrentState ← startingState
  CALL CurrentState.Enter
End Init

void ChangeState (parameter: newState)
Begin
  CurrentState ← newState
  CALL CurrentState.Enter
End ChangeState
```

3.2 Finite State Machine

The *finite state machine* implementation is done inside the HandleState method contained in each state as in the IdleState class. In the Pocong character, there are three actions that can be divided into three *states*, namely IdleState, JumpState, and DieState. Below is a program implementing *the finite state machine* by using the Controller class to replace the active state.

```
Class IdleState IMPLEMENT MonoBehaviour EXTEND IStatePattern
void HandleState()
begin
  CASE _stateController.ActionState OF
    Controller.State.JUMP : CALL _stateController.ChangeState WITH _character.jumpState
    Controller.State.DIE : CALL_stateController.ChangeState WITH _character.dieState
  ENDCASE
END HandleState
```

```
Class JumpState IMPLEMENT MonoBehaviour EXTEND IStatePattern
void HandleState()
begin
  CASE _stateController.ActionState OF
    Controller.State.IDLE : CALL _stateController.ChangeState WITH _character.idleState
    Controller.State.DIE : CALL_stateController.ChangeState WITH _character.dieState
  ENDCASE
END HandleState
```

```
Class DieState IMPLEMENT MonoBehaviour EXTEND IStatePattern
void HandleState()
begin
  CASE _stateController.ActionState OF
    Controller.State.IDLE : CALL _stateController.ChangeState WITH _character.idleState
  ENDCASE
END HandleState
```

Tabel 1. Pocong character action

Picture	Active State	Information
	IdleState	No input is given. The active state is IdleState. This state is active when a Pocong character hits the ground or respawns.
	IdleState Orientation	The player presses the 'A' key or the like to change the orientation of the character. The active state is IdleState.
	IdleState Hold Power	Players press and hold the 'Space' button or the like to fill the jump power bar. The active state is IdleState.
	JumpState	Players release the 'Space' button or the like to perform jumping stunts. The active state transitions to jumpstate.
	DieState	When a character touches a trap, the active state automatically transitions to DieState and runs a countdown to respawn.
	IdleState Revive	After the countdown is over, the Pocong character respawns at the checkpoint point. The active state transitions to IdleState.

In **Table 1**, you can see the actions performed by the Pocong character. Pocong character has an active state that is shown above. We use IdleState for 4 states (idle, orientation, hold, and revive), jumpState when the character performs jumping, DieState when the character die. While the player can do something for the character, like touching space to fill power jump or A to change orientation, IdleState will be shown.

3.3 Gameplay Comparison

Tabel 2. Gameplay Comparison

Without Design Pattern	With Design Pattern
	

In Table 2, shows the gameplay comparison of Pocong characters between those who are not and those who use the design pattern. To make it different in design when comparing this research, we made a few changes in the bottom right of the design pattern. While there is no difference in design, there are so many results that will be shown in this research later below.

3.4 State Pattern Implementation

State pattern testing is carried out with a test case to find out the correctness of the system to be executed. The results of the tests can be seen in Table 3.

Table 3. State pattern testing

Test Case	Component	Expected Results	Test Results
Program Executed	Init	The Enter, HadleLogic, and HandleState methods work	Appropriate
	Enter	The Enter method works	Appropriate
	HandleLogic	HandleLogic method works	Appropriate
	HandleState	State can transition	Appropriate
Program Not Executed	Init	NullReference error, state cannot transition	Appropriate
	Enter	The Enter method doesn't work	Appropriate
	HandleLogic	HandleLogic method doesn't work	Appropriate
	HandleState	State cannot transition	Appropriate

Table 3 shows state pattern testing that we test method for handling the game. From Table 3 we can see there are two main components that we are testing when the test case is the program being executed, component Init, Enter, HandleLogic, and Handle State, which is already explained in 3.1. State Pattern Implementation, all of the test case gives appropriate test results. Program Not Executed also gives Appropriate test results when component init, enter, handlelogic, and handlestate are being tested and show the right expected results.

3.5 Finite State Machine Implementation

Finite state machine testing is carried out with a test case to determine the validity of the state displacement. The results of the tests can be seen in Table 4.

Table 4. Finite state machine testing

Test Case	Precondition	Steps/ Conditions	Expected Results	Test Results
IdleState Transition Test	Idle State running	No input	State does not transition	Appropriate
		The spacebar is released after pressing/holding	State transitions to JumpState	Appropriate
		Characters hit traps	State transitions to DieState	Appropriate
JumpState Transition Test	Jump State running	No input	State does not transition	Appropriate
		Character touches the ground	State transitions to IdleState	Appropriate
		Characters hit traps	State transitions to DieState	Appropriate
DieState Transition Test	DieState running	No input	The state transitions to idlestate and the character respawns at the checkpoint after a delay of 3.5 seconds	Appropriate

Table 4 shows Finite State matching Testing, and test cases are the transition between states. In the test case IdleState, we can see steps or conditions we made for testing this state when no input state does not transition, but when the spacebar is released after pressing or holding it, IdleState changes into JumpState. It also changes into DieState when the character hits traps. For the JumpState, the transition we want to see is from JumpState to IdleState, and DieState is already shown in Tabel 4 above. DieState also gives appropriate test results when we give no input, and automatically after 3.5 seconds, DieState will change into IdleState to Respawn the character.

3.6 Performance Testing

For testing, we use a test case, which is a software test based on several predefined input scenarios. The test case is carried out by comparing the expected results with the actual results that occurred at the time of testing. If there is a discrepancy between the two, the program code must be corrected [14]. Performance testing was carried out by comparing Eternal Dream's Pocong character behavior program with the program that has been designed in this study. Testing is carried out using *a profiler* by turning on *the deep profile* to determine the program execution time of each *frame*. To facilitate the testing process, this study used 100 pieces of *prefab* characters. The results of algorithm performance testing can be seen in Table 5 and Table 6.

Table 5. Performance testing without design pattern

Action	Data retrieval					Average
	1	2	3	4	5	(ms)
Idle	1.27	1.39	1.26	1.29	1.26	1.3
Orienta -tion Change	1.28	1.26	1.34	1.34	1.28	1.3
Jump Charge	1.61	1.51	1.40	1.47	1.43	1.5
Jump Button Release	6.,8	55.3	55.5	55.4	61.5	57.7
Mid Air	0.91	0.92	0.92	0.93	0.92	09
Die	1.18	1.17	1.25	1.16	1.17	1.2

Table 5 shows performance testing without a design pattern. We can also see the result in Figure 8. With this testing, we take five times data retrieval for each State. For IdleState the average time for executed the code is 1.3ms, the average Orientation is 1.3ms, the average of JumpCharge is 1.5ms, and the average shown below.

Table 6. FPS testing without *design pattern*

Action	Data retrieval (*frame per second*)					Average
	1	2	3	4	5	
Idle	371	368	311	315	309	335
Orienta-tion Change	344	343	343	360	329	344
Jump Charge	284	277	291	295	291	288
Jump Button Release	205	211	213	217	220	213
Mid Air	397	407	426	422	404	411
Die	295	317	281	309	266	294

Table 6 shows FPS testing without design pattern. With this testing we also take five times data retrieval for each State. For IdleState the average FPS is 335 Frame per Second, the average of Orientation is 344 FPS, the average of JumpCharge is 288 FPS, the average of JumpButtonRelease is 213 FPS, the average of MidAir is 411 and, the average when Die is 294.

In Table 7 and Table 8, you can see the test results of algorithms that have used design patterns,

Table 7. Performance testing with design patterns

Action	Data Retrieval (*milisecond*)					Average
	1	2	3	4	5	
Idle	0.39	0.39	0.40	0.39	0.39	0.4
Orienta-tion Change	0.51	0.51	0.60	0.52	0.52	0.5
Jump Charge	0.62	0.62	0.62	0.69	0.62	0.6
Jump Button Release	0.94	0.92	0.91	0.92	0.98	0.9
Mid Air	0.32	0.31	0.30	0.31	0.30	0.3
Die	0.19	0.21	0.19	0.19	0.20	0.2

Table 7 shows performance testing with design pattern that already implemented. With this testing we take five times data retrieval for each State. For IdleState the average time for executed the code is 0.4ms, the average of Orientation is 0.5ms, the average of JumpCharge is 0.6ms, the average of JumpButtonRelease is 0.9ms, the average of MidAir is 0.3ms, and the average when Die is 0.2ms.

Table 8. FPS testing with design pattern

Action	Data Retrieval (*frame per second*)					Averages
	1	2	3	4	5	
Idle	587	537	555	504	514	539
Orienta-tion Change	530	560	528	542	553	543
Jump Charge	488	486	486	525,	477	492
Jump Button Release	507	491	520	553	528	520
Mid Air	515	554	539	587	588	557
Die	507	558	534	568	504	534

Table 8 shows FPS testing with a design pattern that has already been implemented. With this testing, we also take five times data retrieval for each State. For IdleState, the average FPS is 539 Frames per Second, The average for Orientation is 543 FPS, The average for JumpCharge is 492 FPS, The average of JumpButtonRelease is 520 FPS, The average of MidAir is 557, and The average when Die is 534.

We use Software metrics as indicators for software development to measure the quality of software. Software metric contains information that is quantitative in nature to evaluate the efficiency of certain programs or features contained in the software. One of the indicators that are often used in metric software is execution time. Execution time is the time it takes the CPU (Central Processing Unit) to execute a program from start to finish **[15]**.

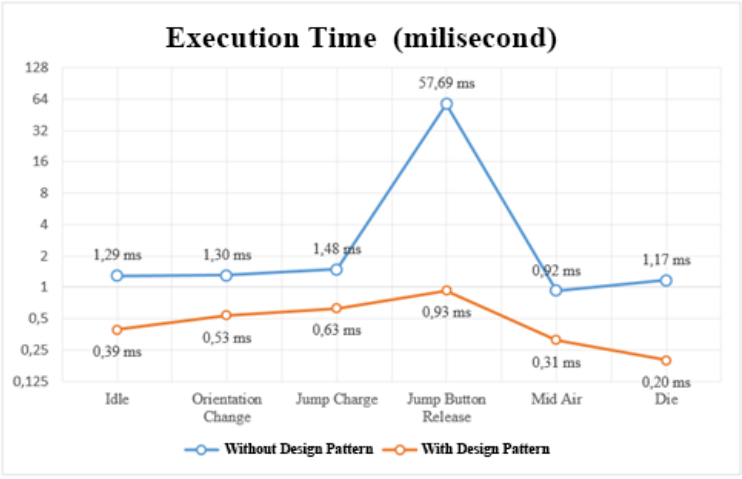

Figure 11. Execution time comparison graph

In **Figure 11**, information is obtained that both algorithms have a relatively stable execution time, with programs that use *the design pattern* having a faster execution time. When the *jump* button is released, there is a very significant spike in execution time occurring in Eternal Dream's algorithm.

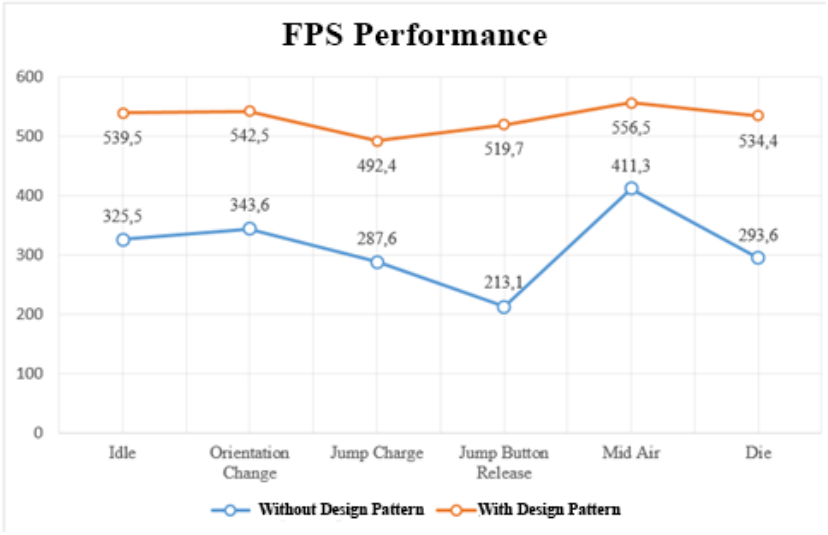

Figure 12. FPS comparison graphics

In **Figure 12**, information obtained from algorithms with a design pattern has a higher FPS and is more stable than algorithms that do not use design patterns. The weight of computing during JumpButtonRelease makes video game performance decrease for a moment, so players feel a lagging when characters move.

4. CONCLUSIONS

Based on the results of research conducted on the behavior of Pocong characters, it was concluded that Pocong characters in the Pocong Jump video game could be developed by applying design patterns in the form of state patterns and finite state machines. The program code that implements the design pattern has a more organized structure and a faster program execution time. The test results of the state pattern and finite state machine components on the Pocong character show valid values. The value describes that the component has been appropriate and successfully tested.

Some suggestions that researchers can give to conduct further research, namely: The test results of the state pattern and finite state machine components on the Pocong character show valid values. The value describes that the component has been appropriate and successfully tested.

Some suggestions that researchers can give to conduct further research, namely:

1. The number of states carried out in this study is relatively small, it is hoped that in the future a study will be carried out with a larger number of states with different transitions.

2. It is hoped that in the future research related to the use of state pattern and finite state machine methods will be carried out in different cases, such as the behavior of non-player character movements or the interaction of the non-player character with players.
3. Develop this research by testing the design pattern and architecture of other video games, such as observers, object pooling, spatial partitions, and others.

ACKNOWLEDGMENT

With the creation of this study, the author expresses his deepest gratitude and gratitude to all parties who contributed to the study. Thank you to Lucky Putra Dharmawan for being willing to provide interviews, materials, and research objects. Thank you to a team from Teknokrat Indonesia University who are willing to help with this collaboration research. As well as thanks to team from Sam Ratulangi University..

REFERENCES

[1] Kneoema, "Top 100 Countries by Game Revenues," 2019. https://knoema.com/infographics/tqldbq/top-100-countries-by-game-revenues# (accessed Aug. 22, 2022).
[2] P. S. Dewi and S. Sintaro, "Mathematics Edutainment Dalam Bentuk Aplikasi Android," *Triple S (Journals Math. Educ.*, vol. 2, no. 1, pp. 1–11, 2019.
[3] L. Bennis, K. Kandali, and H. Bennis, "An Authoring Tool for Generating Context Awareness Mobile Game Based Learning," *Int. J. Emerg. Technol. Learn.*, vol. 17, no. 2, pp. 273–281, 2022.
[4] J. D. Bayliss, "Developing games with data-oriented design," in *Proceedings of the 6th International ICSE Workshop on Games and Software Engineering: Engineering Fun, Inspiration, and Motivation*, 2022, pp. 30–36.
[5] F. S. Pramana, "Penerapan Konsep State Pattern Pada Game Engine (Studi Kasus Game Wipe It Off)." Universitas Brawijaya, 2018.
[6] R. Nystrom, "Game Programming Patterns: Robert Nystrom: 9780990582908: Amazon. com: Books 1 edition., Genever Benning." 2014.
[7] S. Cao, F. Wang, L. Wang, C. Fan, and J. Li, "DNA nanotechnology-empowered finite state machines," *Nanoscale horizons*.
[8] V. André, R. A. S. S. Victório, and G. C. A. Coutinho, "Persistent State Pattern."
[9] M. F. Rahadian, A. Suyatno, and S. Maharani, "Penerapan metode finite state machine pada game 'The Relationship,'" 2017.
[10] S. Sintaro, "RANCANG BANGUN GAME EDUKASI TEMPAT BERSEJARAH DI INDONESIA," *J. Inform. dan Rekayasa Perangkat Lunak*, vol. 1, no. 1, pp. 51–57, 2020.
[11] M. Mustofa, S. Sidiq, and E. Rahmawati, "Penerapan Finite State Machine Untuk Pengendalian Animasi Pada Video Game Rpg Nusantara Legacy," *Jusikom J. Sist. Komput. Musirawas*, vol. 3, no. 1, pp. 1–10, 2018.
[12] T. Minkkinen, "Basics of Platform Games," 2016.
[13] F. Marzian and M. Qamal, "Game RPG 'The Royal Sword' Berbasis Desktop Dengan Menggunakan Metode Finite State Machine (FSM)," *J. Sist. Inf.*, vol. 1, no. 2, 2017.
[14] A. N. Hasibuan and T. Dirgahayu, "Pengujian dengan Unit Testing dan Test case pada Proyek Pengembangan Modul Manajemen Pengguna," *AUTOMATA*, vol. 2, no. 1, 2021.
[15] P. Kaur, "A Review of Software Metric and Measurement," *Int. J. Comput. Appl. Inf. Technol.*, vol. 9, no. 2, p. 187, 2016.