The Implementation of A* Algorithm for Developing Non-Player Characteristics of Enemy in A Video Game Adopted from Javanese Folklore "Golden Orange"

Subari[1], Nira Radita[2*], Bimo Prakoso[3]

[1,2,3]Informatics, Sekolah Tinggi Informatika & Komputer Indonesia, Malang, East Java, Indonesia
Email: [1]subari@stiki.ac.id, [2*]niraradita@stiki.ac.id, [3]bimo.prakoso@gmail.com

(Received: 23 Apr 2024, revised: 24 May 2024, accepted: 27 May 2024)

Abstract

Video games are a means of entertainment for everyone, from children to adults. The genre of games now is also very diverse, ranging from adventure, puzzles to storytelling, and even many folk stories have been made into video games by several developers in Indonesia. Starting from folk tales with horror themes such as kuntilanak, legends such as cucumber mas, to folk tales that rarely sound like golden oranges. The folklore video game of buah jeruk emas is a video game that tells of a king who gets a whisper from the gods to get golden oranges. The king then held a competition to get the golden orange fruit. The player must be able to take the golden orange fruit from the enemy in the form of a Non Playable Character (NPC) who will chase the player. In making NPCs, algorithms are used to help play video games. Therefore, the author wants to apply the A * algorithm in the game of golden oranges so that npc can catch up to players according to the planned system. The main method used is A * and then the addition of the FSM method for other methods. The golden orange fruit is a video game using the A * algorithm and the FSM method after testing it can be concluded that it is enough to make the game run. With the results according to the planned system.

Keywords: A*, FSM, Video Game, golden orange fruit.

I. INTRODUCTION

The golden orange fruit folklore game is a video game that tells a king who gets a whisper from the God to get golden oranges. The king then held a contest to get the golden oranges [1]. The player must be able to take the golden orange from the enemy in the form of a Non-Playable Character (NPC) who will chase the player.

In making NPCs, algorithms are used to help in running video games. These algorithms are very diverse, starting from the simplest algorithms to algorithms that use Artificial Intelligence (AI) which means that artificial intelligence is created and then inserted into a system so that it can behave like humans [2] [3]. The examples of algorithms used are A* (A-star) and Finite State Machine (FSM) [4].

Finite state machine (FSM) is a control system design methodology that describes the behavior or working principle of the system using the following three things: state, event and action. This state transition is generally also accompanied by actions performed by the system when responding to inputs that occur. The taken actions can be simple actions or involve a series of relatively complex processes [5]. FSM in this golden orange game is used for decision making on enemy NPC characters driven by artificial intelligence. This method aims at supporting NPC characters so that the movements and actions of the enemy can run without user intervention from the game [6].

In addition to the FSM method, this study also uses the A* method for the pathfinding algorithm. A* itself is one of the most popular methods and pathfinding algorithms that have the best performance. The A* algorithm checks the feasibility of the costs required to reach a node from another node. This algorithm is a Best First Search algorithm that combines Uniform Cost Search and Greedy Best-First Search [7].

The FSM and A* algorithms have previously been applied in a study entitled "An intelligent agent of finite state machine in the educational game "Flora the Explorer" but their usage is only for chasing players [8], so this research is expected to improve the use of the Finite State Machine algorithm. (FSM) and the A* algorithm that can make the action and rate of enemy NPCs more diverse and efficient [9] [10].

II. RESEARCH METHODS

The 3D games that contain folklore in Indonesia are less than platformer games with 2D dimensions. Many developers prefer to develop 2D games rather than 3D. This is due to easier development and application of algorithms than 3D games. This causes the actions carried out by enemy NPCs cannot be varied and becomes more monotonous. In addition, in the games like Pacman that apply a pathfinding algorithm, sometimes its' enemy NPCs prefer a longer path to pursue the player's character. This can reduce the efficiency of the NPC to catch up with the enemy. The limitations in this study include the use of the A* algorithm to find routes to players, the use of the FSM method for state patrol, chase, and attack, and the game has only 1 map.

NPCs are created by applying the Finite State Machine (FSM) algorithm so that they can react differently based on player treatment. The A* algorithm is also used to make NPCs more efficient in finding player locations. The use of these two algorithms aims to prove the efficiency of the FSM and A* algorithms [11].

The A* algorithm is used to find the shortest route to the player and will always chase the player. In addition to A*, the FSM algorithm will also be used for actions such as attacking and destroying obstacles against the players. The steps of finding the shortest route using the A* algorithm is started from entering the initial node into the open list then checking the nodes adjacent to the initial node and finally adding all these nodes to the open list [12].

The features users have access to start the game to play that the users can win or lose, Figure 1 shows the treatment process for NPCs.

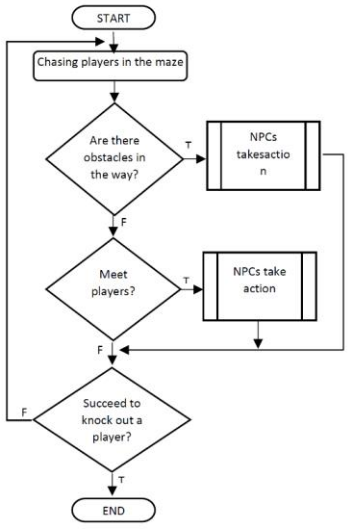

START
Chasing players in the maze
Are there obstacles in the way?
T
NPCs takesaction
F
Meet players?
T
NPCs take action
F
Succeed to knock out a player?
F
T
END

Figure 1. Flow of NPCs in the Game

The flowchart in Figure 2 describes if an NPC encounters an obstacle while chasing a player. NPCs will choose the action between destroying or looking for another route.

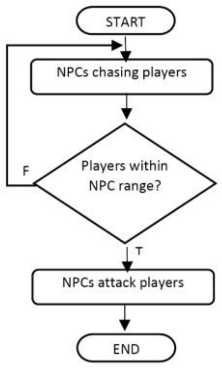


Figure 2. The process when enemy NPCs attack players

The flowchart in Figure 3 describes if an NPC encounters an obstacle while chasing a player. NPCs will choose the action between destroying or looking for another route.

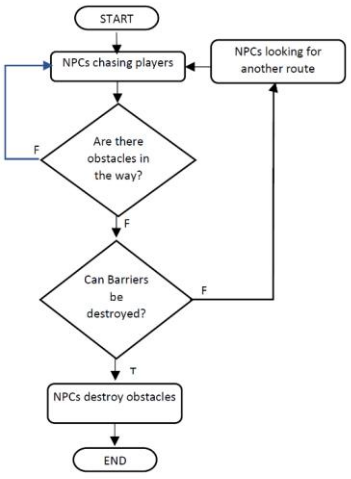

START
NPCs chasing players
NPCs looking for another route
Are there obstacles in the way?
F
F
Can Barriers be destroyed?
F
T
NPCs destroy obstacles
END

Figure 3. NPC Action Meets Barrier

Figure 4 shows a picture while NPCs is chasing players and while they are chasing players, they drop a block of wood as a barrier on the right and left is a forest background.

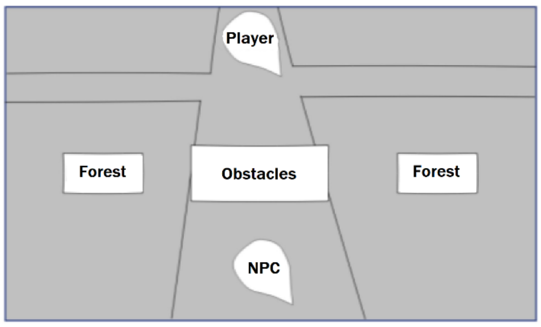


Figure 4. NPC Environment Chasing Players

Figure 5 shows the controls in the golden orange fruit game to manage the player in the form of a keyboard and mouse. The keyboard is useful for moving players and pausing the game. The mouse is useful for moving the camera by using a left click to interact with objects [13].

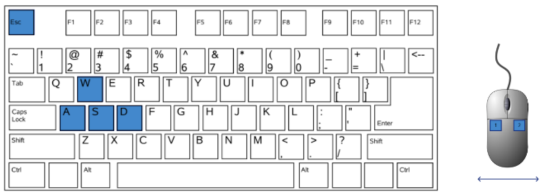


Figure 5. Control Scheme in the Golden Orange Fruit Game

The A* algorithm on the enemy will run when the game starts. The enemy will follow the path determined by the A* algorithm to reach the player. If the enemy reaches the player's position, the enemy will launch an attack.

III. RESULTS AND DISCUSSION

The objects created in this algorithm testing experiment are forest (obstacles), players, enemies, and areas. The first object to be created in Unity is the area object, shown in Figure 6 (scene, 3D Object and Plane).

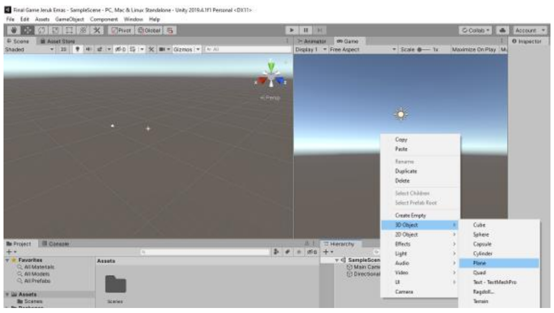


Figure 6. Area Creation

Figure 7 shows the plane position settings at (0, 0, 0) with the desired size.

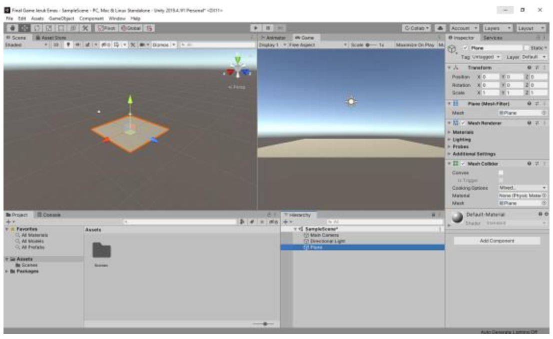


Figure 7. Setting the Position of the Area

Figure 8 shows the creation of obstacles using a cube object and the creation of obstacles from multiple cubes on a new layer.

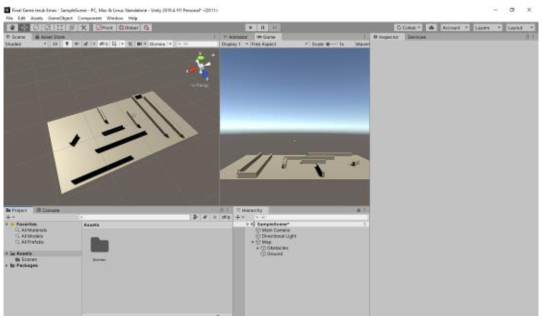


Figure 8. Obstacle Object Layer

For the creation of player and enemy objects, as a dummy, the authors use a capsule object. The authors use 2 capsule objects namely Enemy and Player, in which the 2 objects are placed opposite each other. The placement of enemy and player objects can be seen in Figure 9.

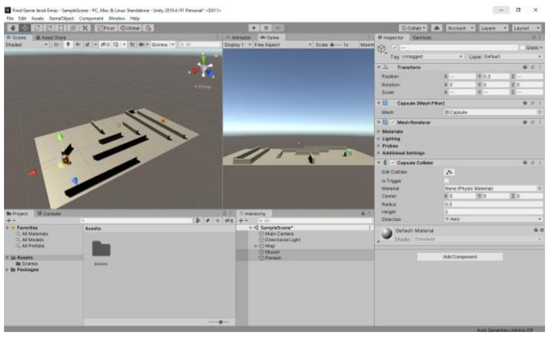


Figure 9. Creation of Enemy and Player Objects

In Figure 10, for the object to be installed, the material that has been made is given a color.

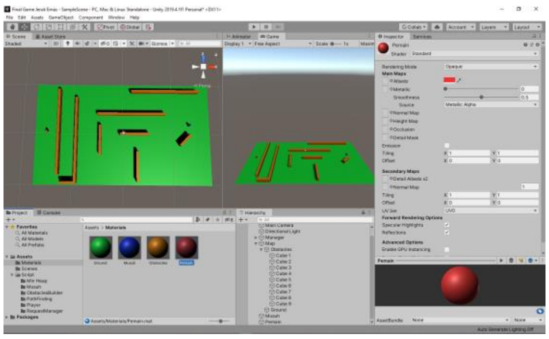


Figure 10. Adding Color to the Object

After the object is created, then the code is implemented into the object so that the A* algorithm can run [14]. After that it was created an empty object namely "Pathfinding Manager". Then it is added the Astar_manager, Pathrequest, and Pointgrid code files into the Pathfinding Manager object.

In the Point Grid section, the World Size in Figure 11 is set with the size of the ground object x and z times 10. So, if x and z are from ground objects 3.5 and 2, then fill the World Size with 35 and 20. Then arrange the Un Walkable Layer with obstacle layer that has been set previously.

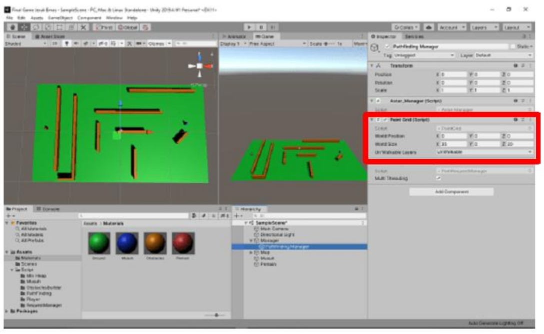


Figure 11. Setting the Grid

The next step is selecting the enemy object by entering the Enemy code file in the settings section, and selecting the player object as the target as shown in Figure 12.

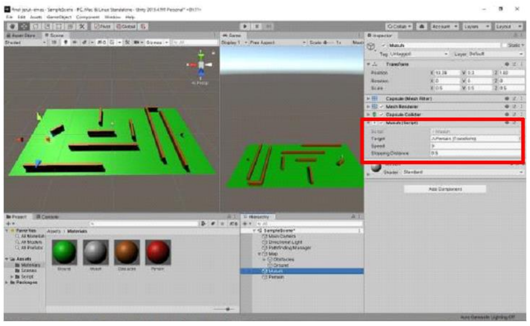


Figure 12. Enemy Code Settings on Enemy Objects

Figure 13 is the step of entering the Move Player code file on the player object to move the player object, and setting the speed on the move speed variable.

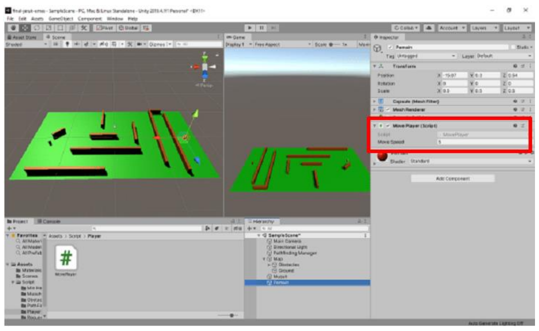


Figure 13. Code File for Moving Player Object

This function is to calculate the border map and create a grid based on the size of the map or ground created. This function will calculate the map boundaries and calculate the grid which is used to calculate neighbors between grids. It provides the Calculate World Map Borders function to measure the map boundaries, and the Generate grid function to create a grid used to place and count neighbors as shown in Figure 14.

Function used to update enemy and player locations when the game has started. Since the condition of the player is always moving, the nodes will always change and the enemy will move related to the node.

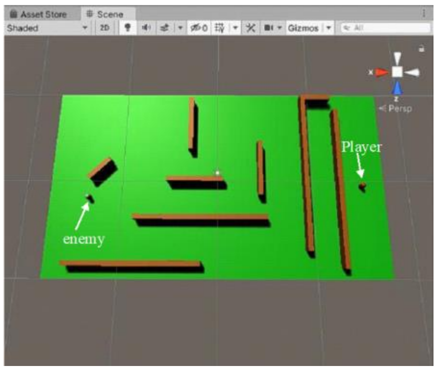


Figure 14. Location of Players and Enemies

In Figure 15, the position of the player and the enemy will always be updated when the player moves and the enemy chases. These positions will always change related to the nodes that have been calculated. When the player moves, the enemy will ask for a new path according to the calculated node and will update the taken path to get to the player's position.

The function is used to determine the starting node of the enemy's position and the destination node of the player's position. In addition, this code also contains the calculation of the A* algorithm. In Figure 15, the starting and destination nodes are determined on a map that already has grids. The starting node will look for a route to reach the destination node. In this game, agent Node is used as a variable for enemy objects, and target Node is a variable for player objects. These two variables will be calculated to determine the path that connects the 2 nodes.

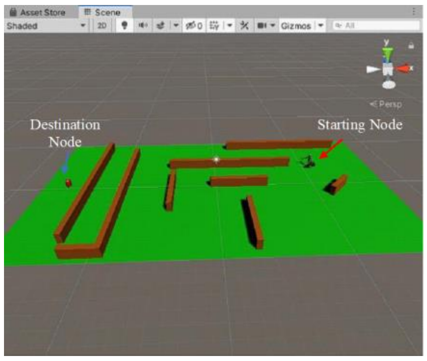


Figure 15. Starting Node and Destination Node

The function is used to store the checked nodes into the open list. After getting a new node, the old node will be removed from the open list and added to the closed list. This function applies heap optimization for route search optimization.

The function is used to add a path leading to the player's location. Enemies will follow these paths to chase the player. This path is a line and a point. In Figure 16, there is a black line and some dots. These lines are the paths that the enemy will follow to reach the player's position. This function is useful for drawing paths on nodes that have been found. This path consists of lines and points, where the line functions as a path and the point functions as a path displacement [15] [16].

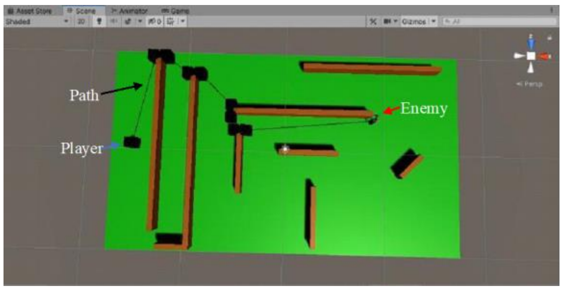


Figure 16. Path between Player and Enemy

This function is useful for the enemy to follow the existing path to reach the player's position. In Figure 17, the two enemies are seen following the existing path to reach the enemy position. The path for each enemy is different since the position of the enemy is different each other. The value of the distance between the enemy and the player will be entered into the Vector3 array, this array will later function to determine where the path will be placed based on the nodes that have been calculated by the A* algorithm.

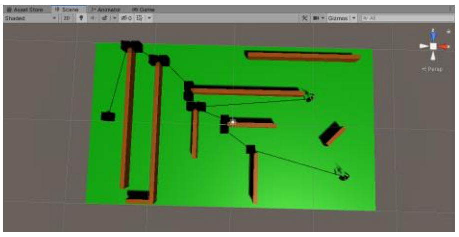


Figure 17. Two Enemies Approaching the Player

The A* test is carried out to find out the prices of F, G, and H according to Table 1, as well as the algorithm's way of finding routes. This process is carried out by taking the coordinates of the player and the enemy.
Player Position Coordinates:
X: 12.3
Y: 0.7
Z: -9.9
Enemy Position Coordinates:
X: 1.5
Y: 0.7
Z: 8

Table 1. Trial Results of the A* Algorithm

Steps	Score	Coordinate Start Node	Coordinate End Node
1	G : 7.6 H : 6.6 F : 14.2	(1.5, 0.4, 7.5)	(12.3, 0.7, -9.9)
2	G : 7.2 H : 5.9 F : 13.1	(1.2, 0.4, 7.1)	(12.3, 0.7, -9.9)
3	G : 6.5 H : 4.8 F : 11.3	(0.7, 0.4, 6.5)	(12.3, 0.7, -9.9)
4	G : 5.8 H : 3.7 F : 9.5	(0.3, 0.4, 5.8)	(12.3, 0.7, -9.9)
5	G : 4.8 H : 1.9 F : 6.7	(-0.5, 0.2, 4.8)	(12.3, 0.7, -9.9)
6	G : 3.6 H : 0.7 F: 4.3	(-0.5, 0.2, 3.6)	(12.3, 0.7, -9.9)
7	G : 2.6 H : 0.9 F : 3.5	(-1.5, 0.2, 3)	(12.3, 0.7, -9.9)
8	G : 0.7 H : 2.3 F : 3	(-2.5, 0.2, 2.6)	(12.3, 0.7, -9.9)
9	G : -2.3 H : 3.3 F : 1	(-3.5, 0.2, 2.6)	(12.3, 0.7, -9.9)
10	G : -3.5 H : 2.5 F : -1	(-3.5, 0.2, -0.4)	(12.3, 0.7, -9.9)
11	G : -2.9 H : 6.3 F : 3.4	(-2.5, 0.2, -1.4)	(12.3, 0.7, -9.9)
12	G : -2.7 H : 6.2 F : 3.5	(-1.5, 0.1, -2.3)	(12.3, 0.7, -9.9)
13	G : -3.2 H : 6.1 F : 2.9	(-0.5, 0.1, -3.2)	(12.3, 0.7, -9.9)
14	G : -4.1 H : 6.1 F : 2	(0.5, 0.1, -4.2)	(12.3, 0.7, -9.9)
15	G : -5 H : 6 F : 1	(1.7, 0.1, -5.3)	(12.3, 0.7, -9.9)
16	G : -1.8 H : 5.9 F : 4.1	(2.7, 0.1, -6.2)	(12.3, 0.7, -9.9)
17	G : -5.7 H : 5.2 F : -0.5	(4.5, 0.1, -7.3)	(12.3, 0.7, -9.9)
18	G : -4.7 H : 4.1 F : -0.6	(5.6, 0.1, -7.3)	(12.3, 0.7, -9.9)
19	G : -3.3 H : 3.2 F : -0.1	(6.5, 0.1, -7.3)	(12.3, 0.7, -9.9)
20	G : 1.7 H : 2.2 F : 3.9	(7.5, 0.1, -7.3)	(12.3, 0.7, -9.9)
21	G : 0 H : 2.4 F : -2.4	(8.4, 0.2, -8.4)	(12.3, 0.7, -9.9)
22	G : 2.7 H : 2 F : 4.7	(9.6, 0.2, -9.2)	(12.3, 0.7, -9.9)
23	G : 5 H : 1.1 F :6.1	(10.5, 0.2, -9.2)	(12.3, 0.7, -9.9)
24	G : 6.6 H : 0.5 F : 7.1	(11.3, 0.2, -9.4)	(12.3, 0.7, -9.9)
25	G : 7.1 H : 0.1 F : 7.2	(12.2, 0.2, -9.9)	(12.3, 0.7, -9.9)

This object is basically the same as the object that was created in the application of the A* algorithm, the difference lies in the addition of a waypoint object. This object will be used as a point for enemy characters in state patrol. The waypoints are made by creating 4 sphere objects placed in several places, these objects will later be used as enemy patrol points.

After giving the waypoint object, the next section is adding the creation of the animator for the state patrol enemy character which can be seen in Figure 18. This animator functions to create a different state for the enemy.

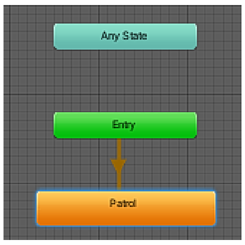
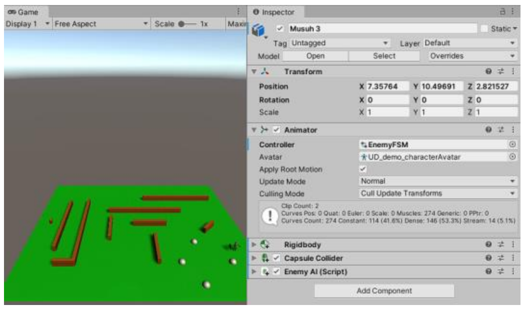


Figure 18. Creating a State Patrol

At each state patrol creation, a behavior will be added to adjust the enemy's behavior while in the patrol state. The *Awake()* method in the state patrol coding serves to define a waypoint object with a waypoint tag that has been created on the sphere object. The *OnStateEnter* method will be executed when the state patrol is called then the enemy will go to the waypoint 0 position. This will also happen when the enemy loses distance while chasing the player, then the enemy will return to the patrol state and go to waypoint 0. The *OnStateUpdate* method will run continuously while the enemy is in patrol state and run according to the waypoint which has been set. In this method there are some codes to add the order of the enemy waypoints. If the enemy has reached waypoint 0, the waypoint value automatically becomes 1 and the enemy will walk to waypoint 1, and so on until it returns to the value 0. The treatment that has been described in *OnStateUpdate* is part of the role in the implementation of this animation controller. It can be seen in Figure 19.



Musuh 3 Static
Tag Untagged Layer Default
Model Open Select Overrides
Transform
Position X 7.35764 Y 10.49691 Z 2.821527
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Animator
Controller EnemyFSM
Avatar UD_demo_characterAvatar
Apply Root Motion ✓
Update Mode Normal
Culling Mode Cull Update Transforms
Clip Count: 2
Curves Pos: 0 Quat: 0 Euler: 0 Scale: 0 Muscles: 274 Generic: 0 PPtr: 0
Curves Count: 274 Constant: 114 (41.6%) Dense: 146 (53.3%) Stream: 14 (5.1%)
Rigidbody
Capsule Collider
Enemy AI (Script)

Figure 19. Animation Controller Implementation

After implementing state patrol on the enemy, a state chase is created so that the enemy the will chase the player is in a certain range. It is necessary to create a new script to make it easier to adjust the enemy's speed in chasing, turning and accuracy. In this script, the code will be added so that it can control the enemy objects and players in terms of speed, rotation speed and accuracy of the enemy itself.

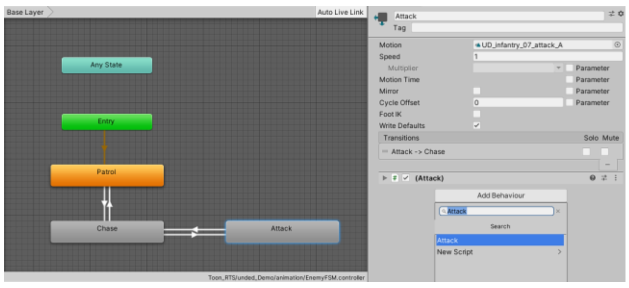
In the script there are several things mentioned, such as the creation of Game Objects that will later function to define enemy and player objects. As for speed, rot speed and accuracy, the function is to regulate this script without having to change from another script.

In the OnStateEnter method, there are NPCs and opponents. These two Game Objects will always be called when they are in state. The opponent will take a component from the EnemyAI script functioning to define enemy objects. Next, a new state called chase is created in the animator. This state will later contain the code so that the enemy chases the player. After creating a state chase, it is necessary to have a transition between state patrol and chase as shown in Figure 20. This transition is useful when the player is within a certain distance from the enemy, the enemy in the patrol state will move to the chase state.

Next, a parameter is created as a determinant when the enemy will chase the player. In this project, the enemy is made to chase the player if the parameter value is less than 20. If it is more than 20, the enemy will return to the patrol state. It is also required is a parameter namely distance of type float and filled with a value of 100 as the default value which will decrease if the player gets closer.

To set the condition when the enemy will chase the player, it needs to be set with a distance condition less than 20. This works if the enemy to player distance is less than 20, then the enemy will enter the chase state. If the enemy's distance condition on the player is greater than 20, then the enemy will return to the patrol state. To make the transition from chase to patrol, the conditions must be set to more than 20.

The attack state in Figure 20 is used for enemies that will attack players within a predetermined range, also transitions from chase to attack and vice versa. This transition is useful for changing from the chase state to the attack state if a certain distance has been met, and vice versa. The transition sets the value of the distance parameter for each transition, from chase to attack if it is less than 2 and vice versa. This means that if the distance between the enemy and the player is less than 2, the enemy will move to the attack state and attack the player. However, if the distance is more than 3, then the enemy will return to chasing the player.



Toon_RTS/undead_Demo/animation/EnemyFSM.controller
Attack
Tag
Motion UD_infantry_07_attack_A
Speed 1
Multiplier Parameter
Motion Time Parameter
Mirror Parameter
Cycle Offset 0 Parameter
Foot IK
Write Defaults ✓
Transitions Solo Mute
Attack -> Chase
(Attack)
Attack
Attack
New Script

Figure 20. Adding Script Behavior to the Attack State

This test aims to test whether the game has been running in accordance with the system design that has been arranged. In gameplay testing, there are several stages of testing are carried out including the early part and while the game is running.

At the beginning of the game as shown in Figure 21 which presents several main menu options.

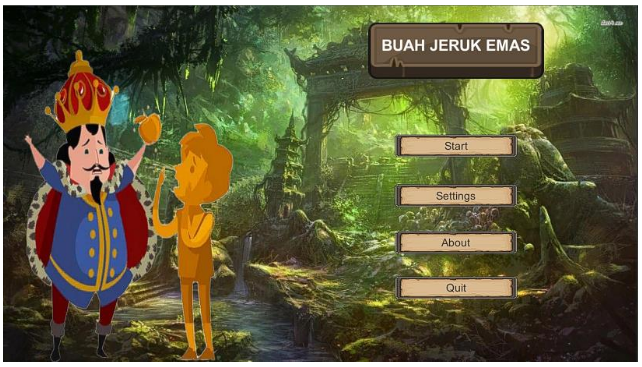


Figure 21. Main Menu Page Display

The game is as shown in Figure 22, the top view shows 3 enemies, 2 enemies are using the A* algorithm in path searching, and 1 enemy is using FSM to adjust the enemy's behavior.

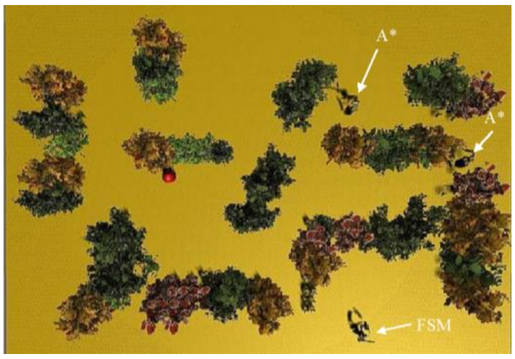


Figure 22. Top View

The camera setting in Figure 23 is for a rear view of the enemy showing the enemy's rear while chasing players or while on patrol.

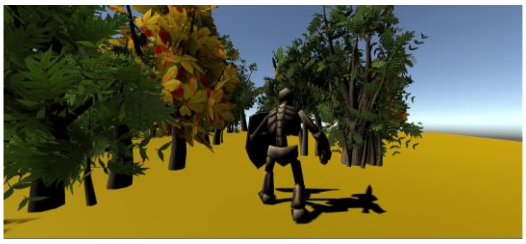


Figure 23. Enemy's Rear View

Enemy route display consists of 3 enemy routes in which 2 enemies use A* for route search, while 1 enemy uses FSM for behavior, as shown in Figure 24.

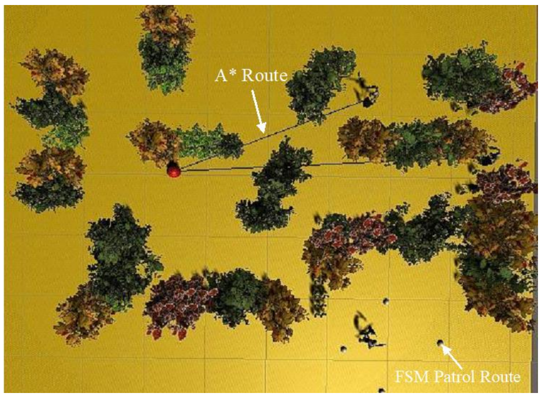


Figure 24. Display of Route A* and FSM

The route taken by this enemy will be different, the enemy A* will automatically chase the player because the route has been determined when the game is running with the A* algorithm calculation, while for FSM enemies will follow the patrol route. If the player is within range, the enemy will chase the player, as shown in Figure 25.

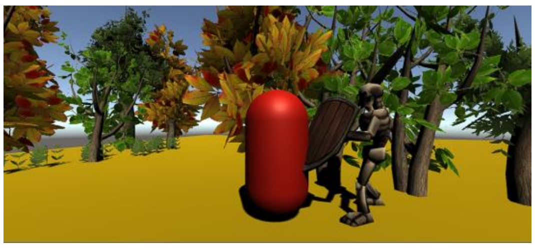


Figure 25. Display of the Enemy Attacking the Player

Figure 26 shows when the player is within a certain range, the enemy will attack the player. In the A* and FSM methods, this method is different. In A*, enemies see if the player is within range by detecting the player's collider. If the enemy hits the collider, the enemy will stop chasing and switching to attack mode. For FSM, if the range of the enemy and player is less than 5 then the enemy changes from state chase to chase attack. This functional testing stage aims to display the results that have been carried out, where the tests include A* testing and FSM testing.

This section is the results of testing against 2 enemies using the A* algorithm to find the player's route when approaching the player.

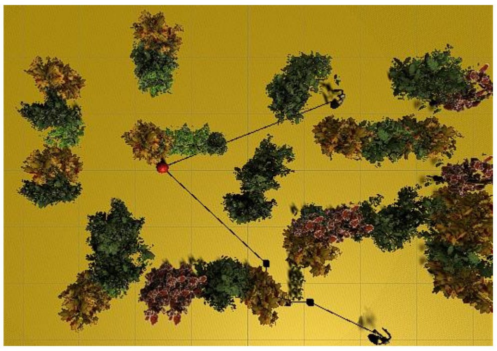


Figure 26. Route Enemy to Player using A*

In Figure 27, when the game starts, the enemy will immediately have a route that has previously been calculated by the A* algorithm to determine the closest route to reach the player. This route is a black line. The route will always change according to the player's position while moving. Enemies will always follow this route to approach the player's position.

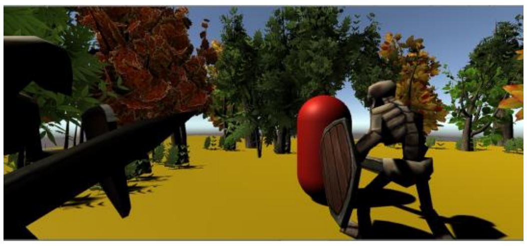
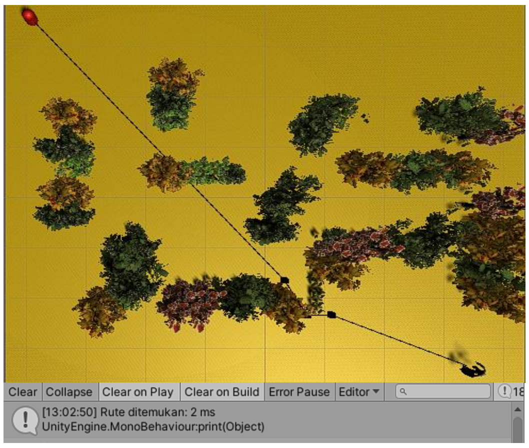


Figure 27. Enemy Attacks Players

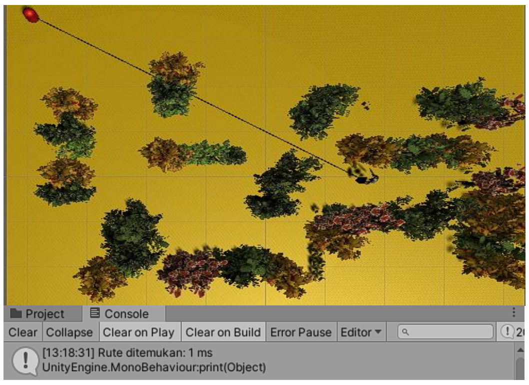
In Figure 28, if the enemy is approaching the player's position, the enemy will detect the player's collider object, if it collides with the collider, the enemy will stop and attack the player.



```
[13:02:50] Rute ditemukan: 2 ms
UnityEngine.MonoBehaviour:print(Object)
```

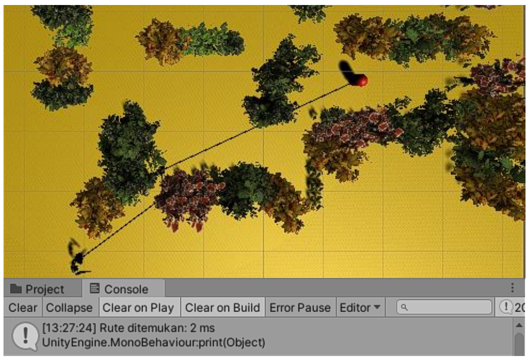
Figure 28. Path 1 for Enemy and Time to Find Route Path 1

In Figure 29, the authors make an example of 1 path for enemy A*, the time allocation to find a path to the player is 2 ms (Miliseconds). The authors make the different paths to see how long it takes to find a route as in line 1. The time allocation to find a route on line 2 is 1 ms.



```
[13:18:31] Rute ditemukan: 1 ms
UnityEngine.MonoBehaviour:print(Object)
```

Figure 29. Path 2 for Enemy and Time to Find Route Path 2



```
[13:27:24] Rute ditemukan: 2 ms
UnityEngine.MonoBehaviour:print(Object)
```

Figure 30. Path 3 for the Enemy and The Time of Finding the Route of Path 3 When the Game Starts

In Figure 30, the authors make another path to calculate how long it will take to find a route to the player. In line 3 it takes the same time allocation as line 1, which is 2 ms.

Table 2. Track Search Time Table

No	Route	Time Allocation
1	Route 1	2 ms
2	Route 2	1 ms
3	Route 3	2 ms

In Table 2, the test results of checking the time needed by the enemy to find a path to the player are very good in which the average time required is less than 3 ms, this is in accordance with the purpose of this study.

Table 3. A* Testing Table

No	Condition	Action	Description
1	*Game starts*	The route appears and the enemy follows the route	Matched
2	Players change positions	The route changes and the enemy follows the new route	Matched
3	Enemy is close to players	Enemy checks collider and attacks players	Matched
4	Enemy meets obstacles	Enemy avoids obstacles	Matched
5	Enemy doesn't hit the player collider	Enemy is still chasing following the route	Matched
6	Appears the time it takes to find the route	Time appears on console	Matched

In Table 3, the results of testing the A* algorithm applied to the enemy run well and relate. In the next stage, the enemy testing is carried out using the FSM method.

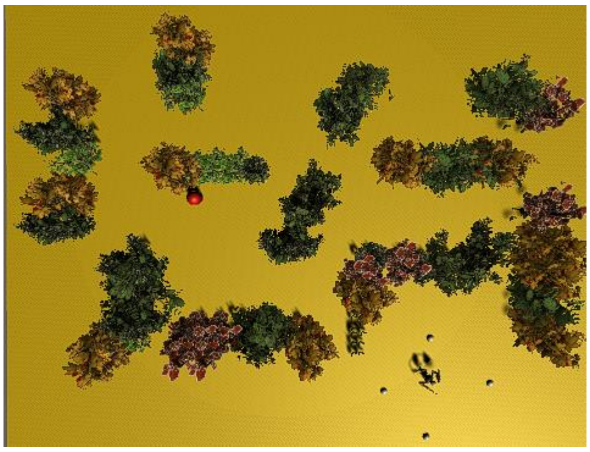


Figure 31. State Patrol on Enemy

In this FSM method, Figure 31 shows the enemy has 3 states, namely patrol, chase, and attack. Each state will change if facing the certain condition. Enemy will follow the waypoint (red circle) for the patrol route and always moves from waypoint to waypoint as long as the player is not close. The distance parameter is 12.3 in which the value is less than 20, then the enemy will transition to the chase state and chase the enemy. It can be seen in Figure 32, the enemy is chasing the player.

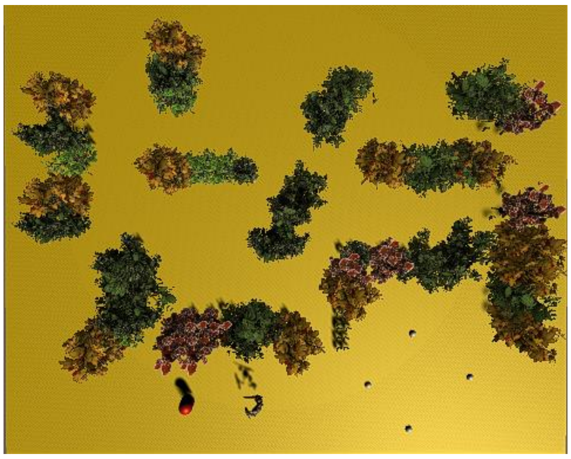
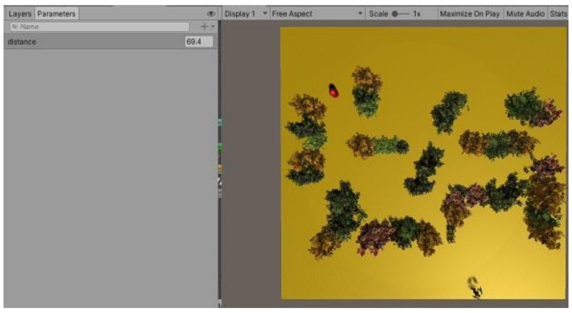


Figure 32. State Chase on Enemy



distance 69.4

Figure 33. The Value of the Distance Parameter is More Than 20

In Figure 33 the distance parameter is 69.4 in which if the value is more than 20, then the enemy will transit to the patrol state and return to the patrol route. Figure 34 shows the enemy back into the patrol state.

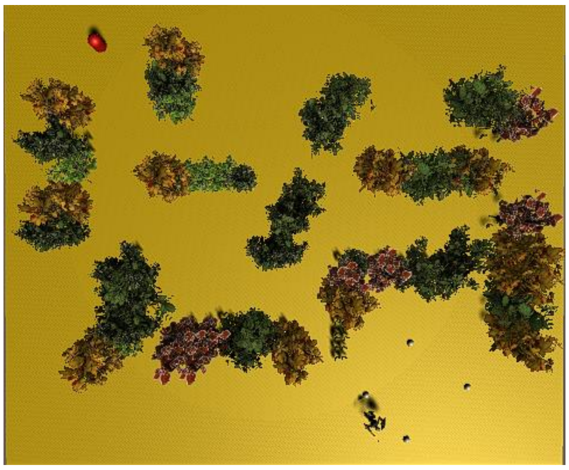


Figure 34. Enemy Returns to Patrol State

The distance parameter is 3.7 in which if the value is less than 5, then the enemy will transit to the attack state and attack the player. Figure 35 shows the enemy performing an attack animation.

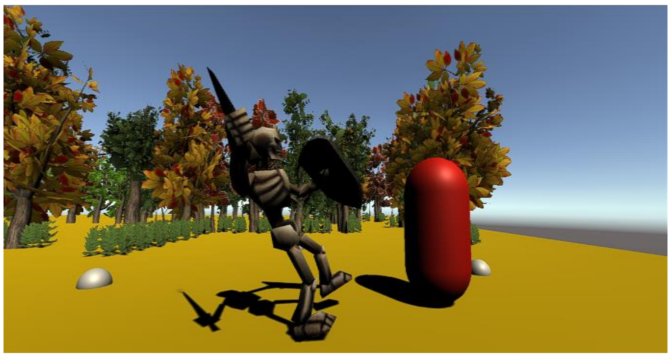


Figure 35. State Attack on Enemy

Table 4. FSM Method Testing Table

No	Condition	Action	Description
1	> 20	Patrolling	Matched
2	< 20	Chasing	Matched
3	< 5	Attacking	Matched
4	> 5	Chasing	Matched

Based on the results in Table 4, the parameter values determined by the authors are the enemy transitions from one state to another has run well. If the value of the distance parameter is more than 20, then the enemy is still in the patrol state. If the value is less than 20, then the enemy enters the chase state. If the value of the distance parameter is less than 5, then the enemy enters the attack state. The implementation phase showed that the model emphasized the importance of providing immediate information to the player [17].

IV. CONCLUSION

The results of the enemy test using the A* algorithm went well. By spending the time less than 3 ms in table 1, the use of the A* algorithm is in accordance with the research objectives. In table 2 the test of checking obstacles and colliders is also appropriate.

The results of the enemy test using the FSM method went well as expected. Table 3 shows that the enemy will enter according to the state based on the value of the distance parameter that has been set. With parameter values > 20, < 20, > 5, and < 5, the enemy has successfully transitioned into the preset state.

The authors hope that the future researchers can add another characters for enemies applying the other algorithm and add other states for enemies applying the FSM method.

REFERENCES

[1] Proyek Penelitian dan Pencatatan Kebudayaan Daerah, *Cerita Rakyat Daerah Jawa Timur*, Jakarta: Departemen Pendidikan dan Kebudayaan Proyek Penerbitan Buku Bacaan dan Sastra Indonesia dan Daerah, 1978.

[2] I. Ahmad and W. Widodo, "Penerapan Algoritma A Star (A*) pada Game Petualangan Labirin Berbasis Android," *Khazanah Informatika (Jurnal Ilmu Komputer dan Informatika),* vol. 3, no. 2, pp. 57-63, 2017.

[3] A. Candra, M. A. Budiman and R. I. Pohan, "*Application of A-Star Algorithm on Pathfinding Game,*" *Journal of Physics: Conference Series, 5 th International Conference on Computing and Applied Informatics (ICCAI 2020),* vol. 1898, no. 1, pp. 1-6, 2020.

[4] E. W. Hidayat, A. N. Rachman and M. F. Azim, "Penerapan *Finite State Machine* pada *Battle Game* Berbasis *Augmented Reality*," *JEPIN (Jurnal Edukasi dan Penelitian Informatika),* vol. 5, no. 1, pp. 54-61, 2019.

[5] D. S. Hormansyah, A. . R. T. H. Ririd and D. T. Pribadi, "Implementasi FSM (*Finite State Machine*) Pada *Game* Perjuangan Pangeran Diponegoro," *Jurnal Informatika Polinemae,* vol. 4, no. 4, pp. 290-297, 2018.

[6] F. Marzian and M. Qamal, "*Game RPG* "*The Royal Sword*" Berbasis Desktop Dengan Menggunakan Metode *Finite State Machine* (FSM)," *SISFO: Jurnal Sistem Informasi,* vol. 1, no. 2, pp. 62-96, 2017.

[7] C. J. Young, A. Suryadibrata and R. Luhulima, "*Review of Various A* Pathfinding Implementations in Game Autonomous Agent,*" *IJNMT (International Journal of New Media Technology),* vol. VI, no. 01, pp. 43-49, 2019.

[8] A. F. Pukeng, R. R. Fauzi, L. R. Andrea, E. Yulsilviana and S. Mallala, "*An intelligent agent of finite state machine in educational game "Flora the Explorer",*" *Journal of Physics: Conference Series, The 3rd International Conference On Science,* pp. 1-12, 2019.

[9] D. Foead, A. Ghifari, M. B. Kusuma, N. Hanafiah and E. Gunawan, "*A Systematic Literature Review of A* Pathfinding,*" *Procedia Computer Science,* vol. 179, no. 11, pp. 507-514, 2021.

[10] J. Smołka, K. Miszta, M. S. Paszkowska and E. Łukasik, "*A* pathfinding algorithm modification for a 3D engine,*" *MATEC Web of Conferences 252, 03007, CMES'18,* vol. 252, pp. 1-6, 2019.

[11] R. Stern, N. R. Sturtevant, A. Felner, S. Koenig, H. Ma, T. T. Walker, J. Li, D. Atzmon, L. Cohen, T. K. S. Kumar, E. Boyarski and R. Roman, "*Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks,*" *Proceedings of the Twelfth International Symposium on Combinatorial Search (SoCS 2019),* pp. 151-158, 2019.

[12] M. Espinoza-Andaluz, J. Pagalo, J. Ávila and J. Barzola-Monteses , "*An Alternative Methodology to Compute the Geometric Tortuosity in 2D Porous Media Using the A-Star Pathfinding Algorithm,*" *Journals Computation,* vol. 10, no. 4, pp. 1-17, 2022.

[13] H. F. Ramadhan, S. H. Sitorus and S. Rahmayuda, "*Game* Edukasi Pengenalan Budaya Dan Wisata Kalimantan Barat Menggunakan Metode *Finite State Machine* Berbasis Android," *Coding : Jurnal Komputer dan Aplikasi,* vol. 7, no. 1, pp. 108-119, 2019.

[14] S. H. Ligaputra, M. Anif, W. Gata and B. H. Prasetyo, "*Expert System for Identifying Damages of Panasonic NS1000 PABX Devices with A* Pathfinding,*" *JURNAL RESTI(Rekayasa Sistem dan Teknologi Informasi),* vol. 4, no. 3, pp. 558-568, 2020.

[15] D. Hermanto and S. Dermawan, "Penerapan Algoritma A-Star Sebagai Pencari Rute Terpendek pada Robot *Hexapod*," *Jurnal Nasional Teknik Elektro,* vol. 7, no. 2, pp. 122-129, 2018.

[16] S. Purnama, D. A. Megawaty and Y. Fernando, "Penerapan Algoritma A Star Untuk Penentuan Jarak Terdekat Wisata Kuliner Di Kota Bandar Lampung," *Jurnal TEKNOINFO,* vol. 12, no. 1, pp. 28-32, 2018.

[17] E. Handriyantini and S. Subari, "*Development of a Casual Game for Mobile Learning with the Kiili Experiential Gaming Model,*" in *11th European Conference on Games Based Learning (ECGBL),* Graz, Austria, 2017.