DOI: http://dx.doi.org/10.21111/fij.v6i2.4678
Diterima: 27 Juli 2020 Revisi: 22 Mei 2021 Terbit: 27 Juni 2021

Development of Non-Player Character for 3D Kart Racing Game Using Decision Tree

Nashrul Azhar Mas'udi [1)*], Muhammad Aminul Akbar [2)], Eriq Muhammad Adams Jonemaro [3)], Tri Afirianto [4)]

Faculty of Computer Science, Brawijaya University, Indonesia [1,2,3,4)]
nashazhar25@gmail.com [1)*], eriq.adams@ub.ac.id [2)], muhammad.aminul@ub.ac.id [3)], tri.afirianto@ub.ac.id [4)]

Abstract

Racing game is one of the genre that's still popular today. Unity is one of many game engines one can use to develop a racing game. At Unity Asset Store, there is a free template called Micro-Game Karting which can only be played alone. In order to play player versus enemy mode, an artificial intelligence (AI) is needed for directing non-player character (NPC) who acts as the opponent. In racing game, the AI requires the use of movement algorithm and decision making system. For this study, the movement algorithm will use pathfinding. The algorithm is used as a guiding path when NPC is moving and avoiding obstacles in the way. Pathfinding will use waypoint system and raycasting to accomplish it. The decision making technique that will be used is decision tree. It functions as decision maker for NPC so it can determine the correct action to be done at certain time. As for FPS (frame per second) test, performance suffers 0.2-0.3 FPS decrease for every addition of 2 NPCs. According to lap time test, the developed NPCs are faster than the machine learning NPC examples provided by the template and driving test showed favorable outcome.

Keywords: *racing game, Unity, non-player character (NPC), waypoint system, raycasting, decision tree*

Abstrak

Game balap masih menjadi salah satu genre yang diminati saat ini. Unity adalah salah satu game engine yang dapat digunakan untuk mengembangkan game balap. Pada Unity Asset Store, terdapat suatu template gratis dengan nama Micro-Game Karting. Template ini hanya dapat dimainkan seorang diri dan belum ada lawan yang tersedia. Supaya dapat dimainkan untuk melawan komputer, dibutuhkan kecerdasan buatan atau artificial intelligence (AI) untuk menjalankan non-player character (NPC) yang bertindak sebagai musuh. AI pada NPC dalam game balap harus memiliki kecerdasan seperti movement dan decision making. Pada movement, metode yang digunakan pada penelitian ini adalah pathfinding. Metode ini digunakan oleh NPC sebagai panutan dalam bergerak dan menghindari rintangan dari suatu lintasan. Pathfinding akan dilakukan dengan menggunakan sistem waypoint dan raycasting. Pada decision making, metode yang digunakan adalah decision tree. Metode ini berfungsi sebagai pembuat keputusan untuk NPC agar dapat menentukan aksi yang tepat untuk dilakukan pada waktu tertentu. Hasil pengujian FPS menunjukkan kinerja game hanya mengalami penurunan 0,2-0,3 FPS untuk setiap penambahan 2 NPC, sedangkan hasil pengujian waktu putaran menunjukkan NPC yang telah dikembangkan lebih cepat daripada contoh NPC machine learning yang disediakan templat. Hasil pengujian berkendara menunjukkan hasil yang cukup baik.

Kata kunci: *racing game, Unity, non-player character (NPC), sistem waypoint, raycasting, decision tree*

1. INTRODUCTION

Nowadays, there are many games with different genres, themes, and objectives. One of the genre that's still popular today is racing game. According to an article [1], *Mario Kart Wii*, a racing game, is ranked 7th in *Top 10 Best-Selling Video Games of All Time*. In 2019 alone, there are at least 16 racing game being released, such as *Dirt Rally 2.0*, *Team Sonic Racing*, *F1 2019*, *GRID*, *Need for Speed Heat*, etc [2].

To develop a game, one surely need to use a game engine. Game engine is a software created specifically for developing a game. One of many game engines one can use is *Unity*. The game development process can be made faster by using templates provided from *Unity Asset Store*.

At *Unity Asset Store*, there is a free template called *Micro-Game Karting*. This template can only be played in one player mode and there are no available

opponents. In order to play in another mode, namely player versus enemy (PvE), an artificial intelligence (AI) is needed for directing non-player character (NPC) who acts as the opponent.

Racing game with no opponents is not really fun. This is because, in general, playing a racing game requires at least 2 players. Game with 2 players or more is often in the form of competition or rivalry between players with a clear outcome, winning or losing. Even if the feeling of having fun is subjective, the feeling when one wins can contribute to giving the feeling of having fun to some players [3].

The proposed methods for developing AI in racing game is pathfinding and decision tree. Pathfinding is a movement algorithm which is part of the game AI model [4]. It is accomplished by using waypoint system and raycasting. Decision tree is a decision making technique which is part of game AI model [4]. It functions as the NPC's brain so it can determine the appropriate action to be done at the correct time.

The reason waypoint system is chosen is because in [5], the use of waypoint system with vector calculations, conditional monitoring system, and artificial environment perception can provide the player with a challenging and enjoyable experience. They also added that *Unity* supported efficient development of racing game because component of waypoint system, physics engine, and vector calculation vector is provided by *Unity*.

Waypoint system is complemented with the addition of raycasting as visual sensor for the NPC. It is used for collision detection inside the game environment [6]. Raycasting helps the NPC to detect obstacles, such as another NPC in the race, so the NPC can avoid it while the NPC is moving in the course.

Decision tree is chosen because [7] said in their paper that by using decision tree and finite state machine, the NPC can be made better into a creative and diversified game-agents which leads to increased fun factor and life cycle of games. Another paper [8] showed that decision tree have better performance than neural network and table lookup.

The performance of a game can be known by doing FPS (frame per second) test to measure the frame rate of said game. The value of this frame rate directly impacts the player's ability to enjoy some, if not all, games. According to [9], frame rate has a higher impact than frame resolution in player performance. The test result in first-person shooter game shows a good frame rate is very important to the player performance. When the frame rate is very low (3-7 FPS), the player cannot adequately target opponents. This is very different compared to when the frame rate is high (60 FPS), the player performance increased by 7-fold over a frame rate of 3 FPS. On another note, frame resolution has little effect on player performance. Players are still able to effectively target opponents even at low frame resolution. Based on these results, it can be concluded that a high frame rate means the player can provide their desired action in response which leads to giving enjoyable gaming experience.

2. RESEARCH METHOD

Steps for developing AI of NPC starts with identifying attributes and behaviors the NPC has then designing waypoint system, raycasting, and decision tree. These designs will be used as references for implementing them.

2.1. IDENTIFYING ATTRIBUTES AND BEHAVIORS

NPC's attributes and behaviors can be obtained by looking at inputs that can be done by the player for directing their kart in *Micro-Game Karting*. The obtained attributes and behaviors is added with sensor components for NPC. Table 1 shows NPC's attributes and behaviors.

Tabel 1. NPC's attributes and behaviors

No	Attributes	*Behavior*
1.	Accelerate	NPC can accelerate
2.	Brake	NPC can brake
3.	Steer left	NPC can turn left
4.	Steer right	NPC can turn right
5.	Check front	NPC can look at front to check for kart
6.	Check front left	NPC can look at front left to check for kart
7.	Check front right	NPC can look at front right to check for kart
8.	Check left	NPC can look at left to check for wall
9.	Check right	NPC can look at right to check for wall

2.2. WAYPOINT SYSTEM DESIGN AND IMPLEMENTATION

Waypoint system is one of many pathfinding techniques. Pathfinding is a method for directing agent from point A to point B [10]. This direction is used by agent as the main way to move and to avoid obstacles from the course.

All places which can be reached by waypoints should be reachable from any waypoints by traveling along one or more waypoints, resulting in a path the agent can move on [11]. For better navigation, the agent needs to know what the current waypoint is and whether the path can only be passed once (open path) or the path is a circuit, which means it will return to the first waypoint in the list and start all over again (closed path) [12].

Figure 1 shows the flowchart of waypoint sytem design. After `waypoints` have been initialized, the loop is done to check all waypoints one by one, if the current waypoint is close to NPC, then proceeds to check the next waypoint in the list. This is done until all waypoints has been checked. Current waypoint is checked whether it is on the left side or the right side of NPC. If it is on the left side, then NPC turns left. If it is on the right side, then NPC turns right. If it is in neither side or does not exist, then NPC does not turn.

Waypoint system implementation starts with placing waypoints in the game environment or the track

course. Waypoint is represented with green color and connected by green line, for better visualization when doing implementation process. Figure 2 shows waypoints in two circuit track. Figure 3 shows pseudocode for waypoint system.

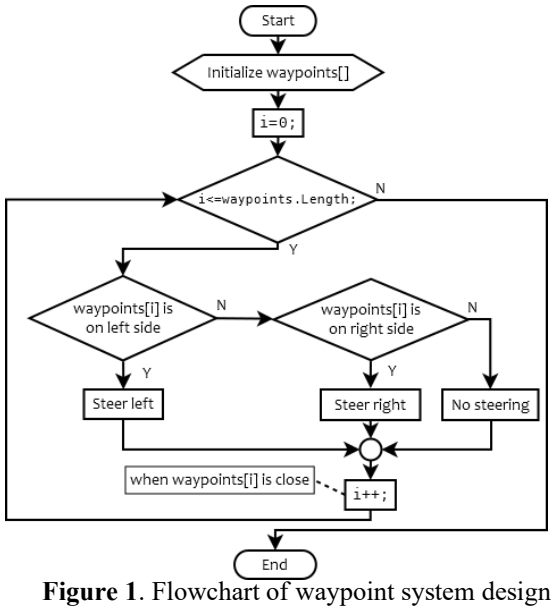


Figure 1. Flowchart of waypoint system design

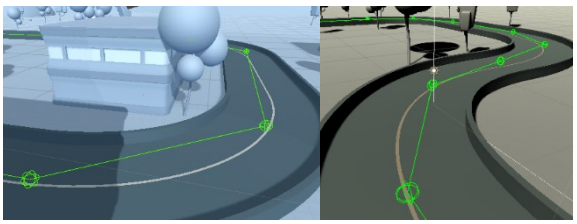


Figure 2. Waypoints in two circuit tracks

```
Pseudocode 1: Waypoint System
 1 BEGIN
 2  Transform[] waypoints;
 3  int currentWpIdx = 0;
 4  Transform activeWaypoint =
   waypoints[currentWpIdx];
 5  REPEAT
 6     IF activeWaypoint exists THEN
 7       IF activeWaypoint is on left side of
   NPC THEN
 8         steer left;
 9       ELSEIF activeWaypoint is on right
   side of NPC THEN
10         steer right;
11       ELSE
12         don't steer;
13       ENDIF
14     ENDIF
15     IF activeWaypoint is close THEN
16       IF currentWpIdx >= waypoints.Length
   THEN
17         currentWpIdx = 0;
18       ELSE
19         currentWpIdx++;
20       ENDIF
21       activeWaypoint =
   waypoints[currentWpIdx];
22     ENDIF
23  UNTIL Player/NPC reaches finish line on
   last lap
24 END
```

Gambar 3. *Pseudocode* untuk sistem *waypoint*

2.3. RAYCASTING DESIGN AND IMPLEMENTATION

Raycasting is the process of shooting an invisible ray from a point, in a specified direction to detect whether any colliders (used by game objects) lay in the path of the ray [6]. Raycasting here is used as visual sensor by the agent to observe and detect obstacles.

The first step of the design is installing ray on the NPC. Figure 4 shows ray installation on NPC. Three rays on the front (F, FL, and FR) is for detecting karts. Their lengths are 7, 5, and 5 cm, respectively. Two rays on the side (L and R) is for detecting walls. Both are 1 cm in length. Figure 5 shows the flowchart of raycasting design.

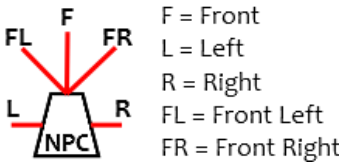


Figure 4. Ray installation

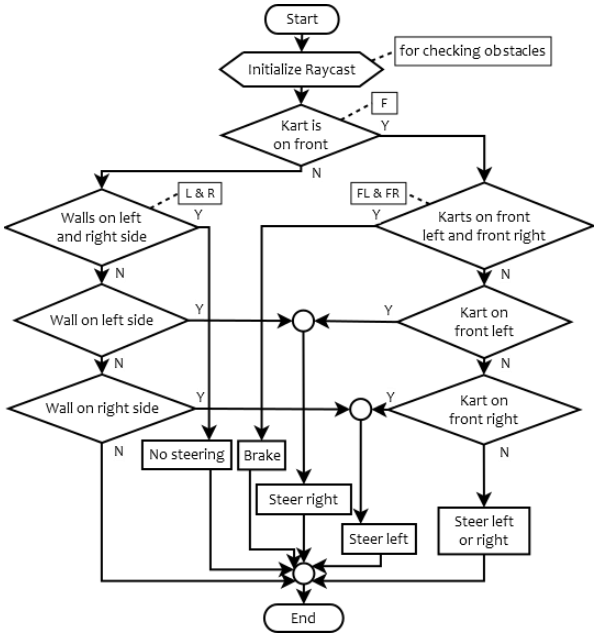


Figure 5. Flowchart of raycasting design

The next step is initializing raycasts with installed rays, then proceeds to check detection result from each raycasts. If kart is not detected in front, then check whether there are walls on both left and right side. If there are, then NPC stops by braking. If it is only on left side, then NPC turns right. If it is only on right side, then NPC turns left. If there are no walls on either side, then do nothing. However, if there is a kart in front, then check whether there are karts on both front left and front right. If on both, then NPC does not turn. If only in front left, then NPC turns right. If only in front right, then NPC turns left. If there is a kart in front and there

are no karts in front left and front right, then NPC can turn left or turn right. Figure 6 shows pseudocode for raycasting.

```
Pseudocode 2: Raycasting
 1 BEGIN
 2  Ray rayF = from in front of NPC to
   forward;
 3  Ray rayFL = from in front of NPC to
   front left;
 4  Ray rayFR = from in front of NPC to
   front right;
 5  Ray rayL = from left side of NPC to
   left;
 6  Ray rayR = from right side of NPC to
   right;
 7  REPEAT
 8    IF rayF hit kart with maxDistance of 7
   on layer kart THEN
 9      IF rayFL hit kart with maxDistance
   of 5 on layer kart THEN
10        IF rayFR hit kart with maxDistance
   of 5 on layer kart THEN
11          brake;
12        ELSE
13          steer right;
14        ENDIF
15      ELSE
16        IF rayFR hit kart with maxDistance
   of 5 on layer kart THEN
17          steer left;
18        ELSE
19          steer left OR right;
20        ENDIF
21      ENDIF
22    ELSE
23      IF rayL hit wall with maxDistance of
   1 on layer wall THEN
24        IF rayR hit wall with maxDistance
   of 1 on layer wall THEN
25          don't steer;
26        ELSE
27          steer right;
28        ENDIF
29      ELSE
30        IF rayR hit wall with maxDistance
   of 1 on layer wall THEN
31          steer left;
32        ENDIF
33      ENDIF
34    ENDIF
35  UNTIL Player/NPC reaches finish line on
   last lap
36 END
```

Figure 6. Pseudocode for raycasting

2.4. DECISION TREE DESIGN AND IMPLEMENTATION

Decision tree is one of the simplest decision making techniques because it is fast, easy to understand and implement [4] . Decision tree consisted of inter connected decision node. The beginning node of decision tree is called root. Starting from root, a collection of choices is chosen in successive from every decision. Each decision is chosen based on the knowledge of the character. This is done until the selection process could not find another decision to consider. On each leaf (the last node which has no branch), there is an action, when reached, will be immediately executed.

Both flowcharts in Figure 1 and 5 is converted into decision tree diagram. For each decision or branching point in the tree, classes will be created. It is likewise for each leaf on the diagram.

Figure 7 shows decision tree diagram of waypoint system design. It starts with checking whether there is an active waypoint or the waypoint which NPC is currently heading. If it does not exist, then NPC does not turn. If it exists, then proceed to check whether it is on the left side or right side of NPC. If on the left side, then NPC accelerates and turns left. If on the right side, then NPC accelerates and turns right. If it is in neither side, then NPC does not turn.

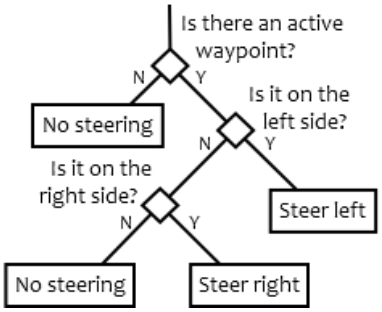


Figure 7. Decision tree diagram of waypoint system

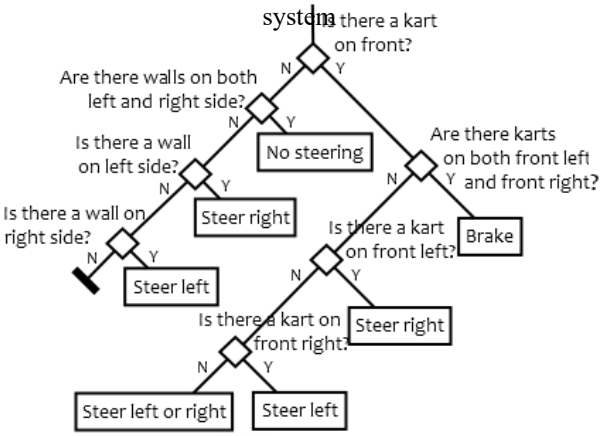


Figure 8. Decision tree diagram of raycasting

Figure 8 shows decision tree diagram of raycasting design. It starts with checking whether there is a kart in front of NPC. If there is none, then proceeds to check whether there are walls on left and right side. If on both, then NPC does not turn. If only on left side, then NPC accelerates and turns right. If only right side, then NPC accelerates and turns left. If in neither side, then do nothing. However, if there is a kart in front, then checks whether there are karts on left side and right side. If on both, then NPC stops by braking. If only on left side, then NPC accelerates and turns right. If only on right side, then NPC accelerates and turns left. If there is a kart in front but no karts on left side and right side, then NPC accelerates and can turn left or right.

The result of both diagram in Figure 7 and Figure 8 is used by the NPC for making decision in choosing whether to accelerate, brake, or turn. First decision is obtained from decision tree of waypoint system. Second one is obtained from decision tree of raycasting.

The first decision will be executed by NPC as long as NPC does not detect obstacles (karts or walls). If an obstacle is detected, then the second decision is executed instead. If the result of the second decision is a null, when NPC does not detect kart in front and does not detect walls, then NPC executes the first decision.

Based on decision tree diagram in Figure 7 and Figure 8, a class diagram will be designed according to available decisions and leaves. Class diagram begins with designing parent class `DTNode` which will be inherited to pseudo-abstract class `Decision` and `Action`. Next is designing the inherited class from `Decision` and `Action`. Figure 9 shows class diagram of decision tree. Figure 10 shows pseudocode for decision tree.

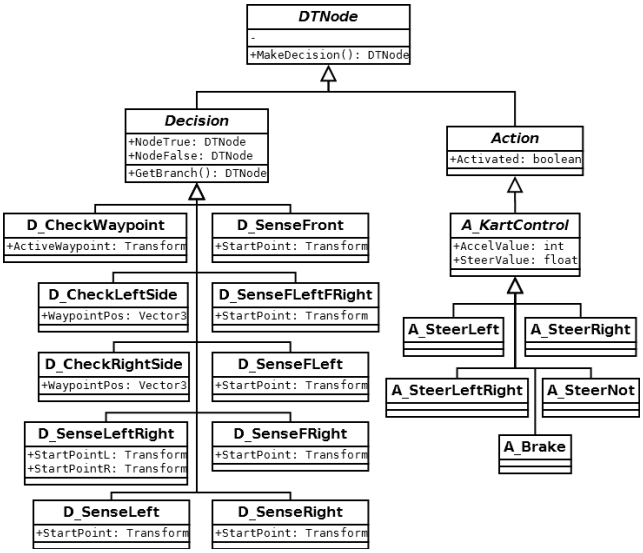


Figure 9. Class diagram of decision tree

```
Pseudocode 3: Decision Tree
BEGIN
 Decision waypointOnRight = new
D_CheckRightSide {
   WaypointPos = waypoint position
relative to NPC,
   NodeTrue = steer right,
   NodeFalse = don't steer
 };
 Decision waypointOnLeft = new
D_CheckLeftSide {
   WaypointPos = waypoint position
relative to NPC,
   NodeTrue = steer left,
   NodeFalse = waypointOnRight
 };
 Decision rootWaypoint = new
D_CheckWaypoint {
   ActiveWaypoint = activeWaypoint,
   NodeTrue = waypointOnLeft,
   NodeFalse = don't steer
 };
 A_KartControl kControl =
rootWaypoint.MakeDecision();
 m_Acceleration = kControl.AccelValue;
 m_Steering = kControl.SteerValue;
 Decision senseFR = new D_SenseFRight {
   StartPoint = start point of the ray,
   NodeTrue = steer left,
   NodeFalse = steer left OR right
 };
 Decision senseFL = new D_SenseFLeft {
   StartPoint = start point of the ray,
   NodeTrue = steer right,
   NodeFalse = senseFR
 };
 Decision senseFLFR = new
D_SenseFLeftFRight {
   StartPoint = start point of the ray,
   NodeTrue = brake,
   NodeFalse = senseFL
 };
 Decision senseR = new D_SenseRight {
   StartPoint = start point of the ray,
   NodeTrue = steer left,
   NodeFalse = null
 };
 Decision senseL = new D_SenseLeft {
   StartPoint = start point of the ray,
   NodeTrue = steer right,
   NodeFalse = senseR
 };
 Decision senseLR = new D_SenseLeftRight {
   StartPoint = start point of the ray,
   NodeTrue = don't steer,
   NodeFalse = senseL
 };
 Decision rootSensor = new D_SenseFront {
   StartPoint = start point of the ray,
   NodeTrue = senseFLFR,
   NodeFalse = senseLR
 };
 A_KartControl kControl =
rootSensor.MakeDecision();
 m_Acceleration = kControl.AccelValue;
 m_Steering = kControl.SteerValue;
END
```

Figure 10. Pseudocode for decision tree

3. RESULTS AND DISCUSSION

The developed AI of NPC will be tested by using black box dan white box. FPS test, lap time test, and driving test will be used as well.

3.1. BLACK BOX TEST

Black box test is a software test method which is done without looking at inner structure, design, or implementation of an application. This test aims to know the validity of all designed behaviors with their implementation. Table 2 shows black box test result.

Table 2. Black box test result

No	Test Case	Expected Result	Status
1.	NPC can accelerate	NPC is expected to accelerate	Valid
2.	NPC can brake	NPC is expected to brake	Valid
3.	NPC can turn left	NPC is expected to turn left	Valid
4.	NPC can turn right	NPC is expected to turn right	Valid
5.	NPC can look at front to check for kart	NPC is expected to look at front to check for kart	Valid
6.	NPC can look at front left to check for kart	NPC is expected to look at front left to check for kart	Valid
7.	NPC can look at front right to check for kart	NPC is expected to look at front right to check for kart	Valid
8.	NPC can look at left to check for wall	NPC is expected to look at left to check for wall	Valid

No	Test Case	Expected Result	Status
9.	NPC can look at right to check for wall	NPC is expected to look at right to check for wall	Valid

Based on the validity test of designed and implemented behaviors by using black box, NPC could do all of its expected behaviors, that is: NPC can accelerate, brake, turn left, turn right, look at front to check for kart, look at front left to check for kart, look at front right to check for kart, look at left to check for wall, and look at right to check for wall.

3.2. WHITE BOX TEST

White box test is done by testing the implemented pseudocode. This test will use basis path testing. Basis path testing begins with the pseudocode modeled into flow graph which has a collection of node and edge. Node represents process done in a function. As for edge, it represents the relation between nodes in a flow graph. From the created flow graph, the calculation of cyclomatic complexity can be done by using Equation (1), (2), and (3) [13].

$$V(G) = R \quad (1)$$
$$V(G) = E - N + 2 \quad (2)$$
$$V(G) = P + 1 \quad (3)$$

where,
$V(G)$ = cyclomatic complexity
R = total regions
E = total edges
N = total nodes
P = total predicate nodes

Lastly, determine the independent paths based on the paths available in flow graph. Each independent path then will be tested.

3.2.1. WHITE BOX TEST FOR WAYPOINT SYSTEM

Pseudocode from waypoint system (Figure 3) will be modeled into flow graph, its complexity cyclomatic calculated, and its independent paths determined. Figure 11 shows flow graph of waypoint system.

Based on Figure 11, the complexity cyclomatic can be calculated with Equation (1), (2), and (3) as follows.

Cyclomatic Complexity:

$V(G) = R$ $V(G) = E - N + 2$ $V(G) = P + 1$
$= 7$ $= 29 - 24 + 2$ $= 6 + 1$
$= 5 + 2$ $= 7$
$= 7$

From the calculated complexity cyclomatic, total test which must be done to ensure all codes in pseudocode is run at least once is 7 times.

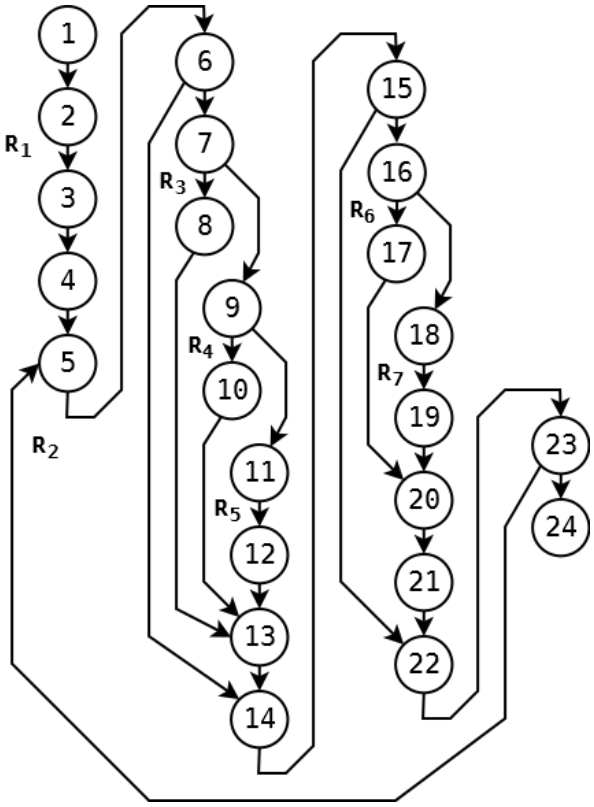


Figure 11. Flow graph of waypoint system

The following are 7 independent paths obtained based on the complexity cyclomatic result.

Independent Path:

Path #1: 1–2–3–4–5–6–7–8–13–14–15–16–17–20–21–22–23–24
Path #2: 1–2–3–4–5–6–7–9–10–13–14–15–16–17–20–21–22–23–24
Path #3: 1–2–3–4–5–6–7–9–11–12–13–14–15–16–17–20–21–22–23–24
Path #4: 1–2–3–4–5–6–7–8–13–14–15–16–18–19–20–21–22–23–24
Path #5: 1–2–3–4–5–6–7–9–10–13–14–15–16–18–19–20–21–22–23–24
Path #6: 1–2–3–4–5–6–7–9–11–12–13–14–15–16–18–19–20–21–22–23–24
Path #7: 1–2–3–4–5–6–14–15–22–23–5–6–14–15–22–23–24

Table 3 shows white box test result for waypoint system with test cases based on independent paths.

Table 3. White box test result for waypoint system

Path #	Test Case	Expected Result	Status
1.	Current waypoint is on left side; NPC is close to it; it is last in the list so it goes back to first waypoint; and player or NPC reached finish line on last lap	NPC is expected to turn left when heading to last waypoint in the list then proceed to head to first one in the list	Valid

Path #	Test Case	Expected Result	Status
2.	Current waypoint is on right side; NPC is close to it; it is last in the list so it goes back to first waypoint; and player or NPC reached finish line on last lap	NPC is expected to turn right when heading to last waypoint in the list then proceed to head to first one in the list	Valid
3.	Current waypoint is in neither side; NPC is close to it; it is last in the list so it goes back to first waypoint; and player or NPC reached finish line on last lap	NPC is expected to not turn when heading to last waypoint in the list then proceed to head to first one in the list	Valid
4.	Current waypoint is on left side; NPC is close to it; it is not last in the list so it goes to next waypoint; and player or NPC reached finish line on last lap	NPC is expected to turn left when heading to waypoint other than the last one in the list then proceed to head to next one in the list	Valid
5.	Current waypoint is on right side; NPC is close to it; it is not last in the list so it goes to next waypoint; and player or NPC reached finish line on last lap	NPC is expected to turn right when heading to waypoint other than the last one in the list then proceed to head to next one in the list	Valid
6.	Current waypoint is in neither side; NPC is close to it; it is not last in the list so it goes to next waypoint; and player or NPC reached finish line on last lap	NPC is expected to not turn when heading to waypoint other than the last one in the list then proceed to head to next one in the list	Valid
7.	Current waypoint does not exist; NPC is not close to it; no waypoint means no list; and player or NPC has not reached finish line on last lap	NPC is expected to do nothing	Valid

3.2.2. WHITE BOX TEST FOR RAYCASTING

Flow graph of raycasting based on Figure 6 is shown in Figure 12. Based on Figure 12, the complexity cyclomatic can be calculated with Equation (1), (2), and (3) as follows.

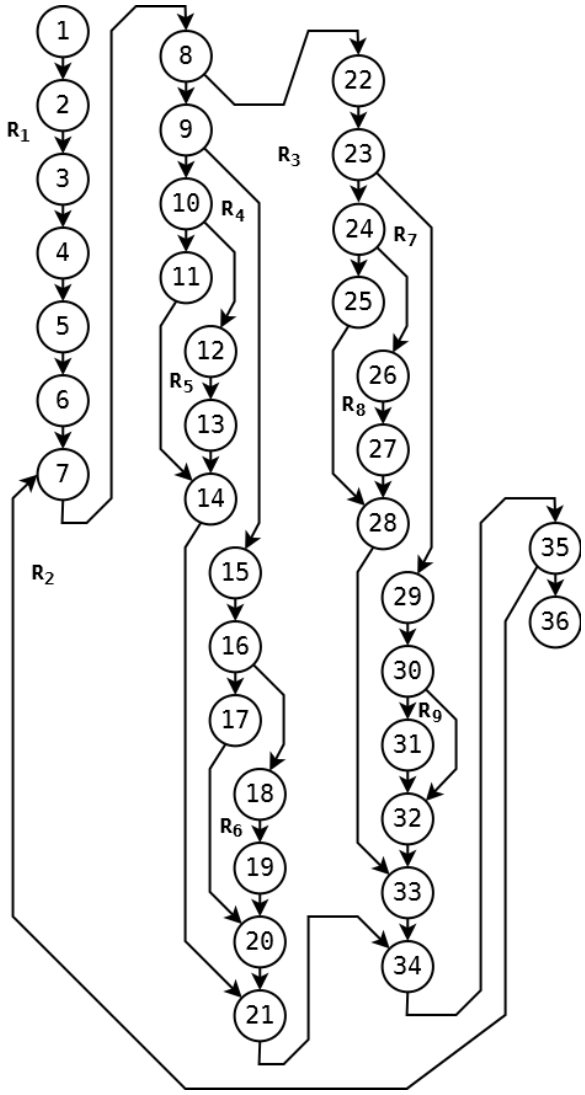
Cyclomatic Complexity:

$$V(G) = R = 9$$

$$V(G) = E - N + 2 = 43 - 36 + 2 = 7 + 2 = 9$$

$$V(G) = P + 1 = 8 + 1 = 9$$

From the calculated complexity cyclomatic, total test which must be done to ensure all codes in pseudocode is run at least once is 9 times.



Gambar 12. *Flow graph* pada *raycasting*

The following are 9 independent paths obtained based on the complexity cyclomatic result.

Independent Path:

Path #1: 1–2–3–4–5–6–7–8–9–10–11–14–21–34–35–36

Path #2: 1–2–3–4–5–6–7–8–9–10–12–13–14–21–34–35–36

Path #3: 1–2–3–4–5–6–7–8–9–15–16–17–20–21–34–35–36

Path #4: 1–2–3–4–5–6–7–8–9–15–16–18–19–20–21–34–35–36

Path #5: 1–2–3–4–5–6–7–8–22–23–24–25–28–33–34–35–36

Path #6: 1–2–3–4–5–6–7–8–22–23–24–26–27–28–33–34–35–36

Path #7: 1–2–3–4–5–6–7–8–22–23–29–30–31–32–33–34–35–36

Path #8: 1–2–3–4–5–6–7–8–22–23–29–30–32–33–34–35–36

Path #9: 1–2–3–4–5–6–7–8–9–10–11–14–21–34–35–7–8–9–10–11–14–21–34–35–36

Table 4 shows white box test result for raycasting with test cases based on independent paths.

Table 4. White box test result for raycasting

Path #	Test Case	Expected Result	Status
1.	Raycast detects kart in front (F), front left (FL), and front right (FR); and player or NPC reached finish line on last lap	NPC is expected to brake	Valid
2.	Raycast detects kart in front (F), front left (FL), and not in front right (FR); and player or NPC reached finish line on last lap	NPC is expected to turn right	Valid
3.	Raycast detects kart in front (F), front right (FR), and not in front left (FL); and player or NPC reached finish line on last lap	NPC is expected to turn left	Valid
4.	Raycast detects kart in front (F), and neither in front left (FL) and front right (FR); and player or NPC reached finish line on last lap	NPC is expected to turn left or right	Valid
5.	Raycast does not detect kart in front (F); and detects wall on left side (L) and right side (R); and player or NPC reached finish line on last lap	NPC is expected to not turn	Valid
6.	Raycast does not detect kart in front (F); and detects wall on left side (L) and not on right side (R); and player or NPC reached finish line on last lap	NPC is expected to turn right	Valid
7.	Raycast does not detect kart in front (F); and detects wall on right side (R) and not on left side (L); and player or NPC reached finish line on last lap	NPC is expected to turn left	Valid
8.	Raycast does not detect kart in front (F); and does not detect wall on either left side (L) and right side (R); and player or NPC reached finish line on last lap	NPC is expected to do nothing	Valid
9.	Raycast detects kart in front (F), front left (FL), and front right (FR); and player or NPC has not reached finish line on last lap	NPC is expected to not turn	Valid

3.2.3. WHITE BOX TEST FOR DECISION TREE

Figure 13 shows flow graph of decision tree based on Figure 10. The complexity cyclomatic can be calculated with Equation (1), (2), and (3) as follows.

Cyclomatic Complexity:

$$V(G) = R = 1$$

$$V(G) = E - N + 2 = 17 - 18 + 2 = (-1) + 2 = 1$$

$$V(G) = P + 1 = 0 + 1 = 1$$

From the calculated complexity cyclomatic, total test which must be done to ensure all codes in pseudocode is run at least once is 1 time.

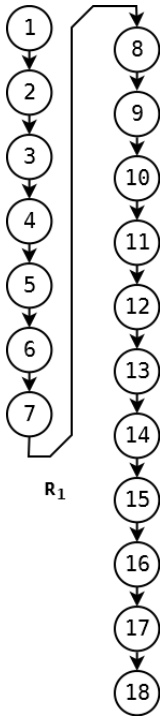


Figure 13. Flow graph of decision tree

The following are 1 independent path obtained based on the complexity cyclomatic result.

Independent Path:

Path #1: 1–2–3–4–5–6–7–8–9–10–11–12–13–14–15–16–17–18

Table 5 shows white box test result for decision tree with test cases based on independent paths.

Table 5. White box test result for decision tree

Path #	Test Case	Expected Result	Status
1.	Initialize decision tree of waypoint system and raycasting then proceed to execute both and the result is used for determining what the NPC will do	NPC is expected to accelerate or brake and turn based on the result of the two decision tree	Valid

Based on the white box test results, each created test cases have been in accordance with tested total paths and gave the expected results. Therefore, it can be concluded that NPC have met all of its designed behavior.

3.3. FPS TEST

FPS (frame per second) test aims to determine the game performance based on how many NPCs is added in *Micro-Game Karting*. It is done in phases starting with placing a few to many NPCs, then writing down the game performance by looking at the displayed FPS value. The system specification used is Intel Core i7 processor, 4096MB RAM, NVIDIA GeForce GT 635M 2GB graphic card, Windows 7 Ultimate SP1 64-bit operating system, and Unity 3D version 2018.4.6f1 64-bit. Table 6 and Figure 15 shows FPS test result and its chart.

Table 6. FPS test result

Test #	Total NPC	FPS
1.	3	46,0
2.	5	45,8
3.	7	45,5
4.	9	45,3
5.	11	45,0

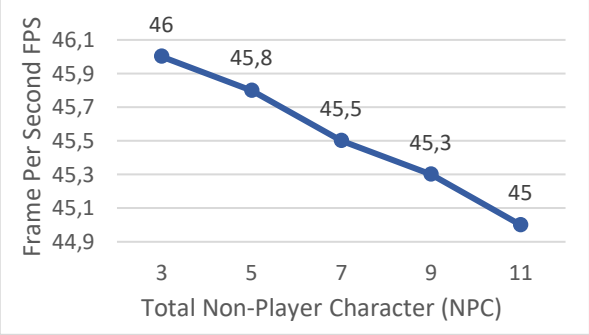


Figure 15. FPS test result chart

Based on FPS test result, the game performance in processing a lot of NPCs only suffers a tiny decrease, -0,3 to -0,2 FPS for every additon of 2 NPCs. Therefore, it can be concluded that the game performance when there are a lot of NPCs is still in good condition.

3.4. LAP TIME TEST

The aim of this test is to determine the lap time performance of the developed AI of NPC with proposed method compared to NPC with machine learning method in the new version of *Micro-Game Karting* with its name changed to *Karting Microgame*. Table 7 shows the result of the race in mm:ss.ms format for 10 laps with top speed of 10 m/s (36 km/h) for both of NPC. The first lap time is the highest due to both of them started with speed of 0 at the start line and have only begun to accelerate to top speed. The developed NPC won over machine learned NPC in all 10 laps with total time difference of 7.21 seconds and average time difference of 0.70 seconds. While the race is in progress, the ML NPC erratically steered a bit to left and right most of the time even in the straight path when it was not needed causing it to lost its velocity. The developed NPC only steered when the current waypoint is not in front of it resulting in maintained velocity, especially in straight path. This is the most likely cause of developed NPC winning over ML NPC. Figure 15 shows track for lap time testing.

Table 7. NPC lap time

Lap #	Developed NPC	ML NPC
1.	00:28.26	00:29.03
2.	00:27.98	00:28.80
3.	00:28.02	00:28.48
4.	00:28.02	00:28.75
5.	00:27.99	00:28.76
6.	00:28.00	00:28.67
7.	00:27.96	00:28.77
8.	00:28.02	00:28.65
9.	00:28.02	00:28.74
10.	00:27.98	00:28.81
Total	04:40.25	04:47.46
Average	00:28.00	00:28.70

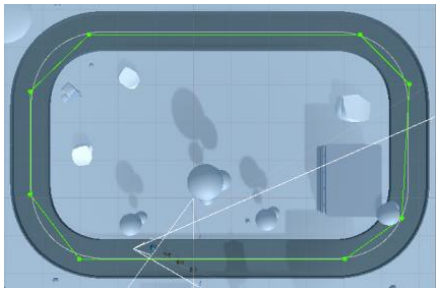


Figure 14. Track for Lap Time Test

3.5. DRIVING TEST

For this test, NPC races by itself for 10 laps with top speed of 20 m/s (72 km/h), then the movement path used by NPC is drawn as shown in Figure 14. When NPC is turning, NPC oftens slightly touch the wall, sometimes crash, and seldoms do not touch the wall in 8 circled areas as shown in Figure 15. The most likely cause for this is NPC could not measure the appropriate speed to be used yet when turning in the corner. Even so, NPC could finish all laps fast enough so it can still provide adequate difficulty for players.

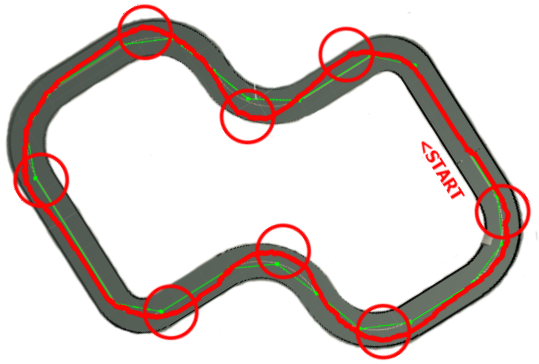


Figure 15. Movement path used by NPC

4. CONCLUSION

The proposed methods, pathfinding and decision tree, have been implemented and as shown in the result of black box and white box testing, NPC could do all of its designed behaviors. The result of FPS testing showed the game performance is still in good condition, it only suffers 0,2-0,3 FPS decrease for every addition of 2 NPCs. The result of lap time test showed the developed NPC is a bit faster than NPC with machine learning. Lastly, the result of driving test showed even though the NPC performance is slightly bad when turning in the corner, it could still provide an adequate challenge for players. From all test results, it can be concluded the NPC can act as opponents for players in *Micro-Game Karting* without causing frustation on them because of low FPS value which leads to increased fun factor for them. In addition, the NPC could prove to be a good challenge for average players. With the NPC acting as opponent, then *Micro-Game*

Karting can be played in player versus enemy (PvE) mode.

Even so, pathfinding with waypoint system is only effective if the track path is static. For dynamic track path, another method can be added to complement it or changing it to another more effecive method. With raycasting, NPC have been able to avoid another karts and walls in track courses. However, the raycast detection reach is only a straight line. For larger detection reach, it is better to use another more effective method. Decision tree have been effective in serving as the decision maker for NPC. This is because conditions used here is not that many. For the case of many conditions, another more effective method should be used.

5. REFERENCES

[1] J. Sirani, "Top 10 Best-Selling Video Games of All Time," *IGN*, 2019. https://www.ign.com/articles/2019/04/19/top-10-best-selling-video-games-of-all-time (accessed Aug. 30, 2019).

[2] E. Jones, "All Upcoming Racing Games of 2019 | Heavy.com," 2019. https://heavy.com/games/2019/01/top-best-upcoming-racing-games-2019/ (accessed Aug. 30, 2019).

[3] D. Sirlin, "Playing To Win – Becoming the Champion: Introduction — Sirlin.Net — Game Design," 2000. http://www.sirlin.net/ptw-book/introduction (accessed Jul. 10, 2020).

[4] I. Millington and J. Funge, *Artificial Intelligence for Games*, 2nd ed. Burlington: Morgan Kaufmann, 2009.

[5] M. T. Chan, C. W. Chan, and C. Gelowitz, "Development of a Car Racing Simulator Game Using Artificial Intelligence Techniques," *Int. J. Comput. Games Technol.*, vol. 2015, 2015, doi: 10.1155/2015/839721.

[6] J. Glover, "Learn and Understand Raycasting in Unity3D – Zenva | GameDev Academy," 2017, Accessed: Oct. 03, 2019. [Online]. Available: https://gamedevacademy.org/learn-and-understand-raycasting-in-unity3d/.

[7] T. B. Yoon, K. H. Park, J. H. Lee, and K. M. Lee, "User Adaptive Game Characters Using Decision Trees and FSMs," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 4496 LNAI, pp. 972–981, 2007.

[8] L. D. Pyeatt, "Reinforcement Learning with Decision Trees," *IASTED Int. Multi-Conference Appl. Informatics*, vol. 21, pp. 26–31, 2003.

[9] M. Claypool, K. Claypool, and F. Damaa, "The Effects of Frame Rate and Resolution on Users Playing First Person Shooter Games," *Multimed. Comput. Netw. 2006*, vol. 6071, p. 607101, 2006, doi: 10.1117/12.648609.

[10] C. Bennett and D. V. Sagmiller, *Unity AI Programming Essentials*, 1st ed. Birmingham: Packt Publishing, 2014.

[11] R. Graham, H. McCabe, and S. Sheridan, "Pathfinding in Computer Games," *ITB J.*, vol. 4, no. 2, 2003, doi: 10.21427/D7ZQ9J.

[12] M. Buckland, *Programming Game AI by Example*. Texas: Wordware Publishing, Inc., 2005.

[13] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 5th ed. New York: McGraw-Hill, 2001.