Simple Sukoba Artificial Intelligence (SSAI) - Movement

Arthur William Dórea Melo[1], **Victor Flávio de Andrade Araujo**[1,2]

[1]Universidade Tiradentes (UNIT), Aracaju, SE – Brasil
[2] National Institute of Science and Technology Social and Affective Neuroscience (INCT-SANI)

arthur.william@souunit.com.br, victor.flavio93@souunit.com.br

Abstract. Introduction: *With the need of a new simple navigation and combat system for agents, the SSAI was created as a way to create a simple AI that can perform maneuvers procedurally such as ducking into cover, exposing to shoot, shooting and exploring.* ***Objective:*** *Create an in-house, customizable and simple AI system for shooting agent movement and shooting.* ***Methodology or Steps:*** *This article uses C#, Unity, raycasting and mathematics.* ***Results:*** *A functional, in-house, customizable and simple AI system for shooting agents in video games.*
Keywords *AI, raycasting, Unity, C#, Agents*

1. Introduction

Intelligent agents are sometimes essential for games to work properly, from being a simple opponent for the player to face, to being a fundamental side character that guides the player through important points in the map, but, in order for these agents to work, they typically need some sort of logic to move [Nareyek 2000]. During a point in the development of the shooter game "Sukoba", there were scripts that handled movement and shooting of entities, but the agents were completely static, because there was no controller for agents to allow them to use such scripts. Meaning there was a clear need for an AI navigation system for the agents that could also perform exploration and simple combat maneuvers, such as ducking into cover, exposing to shoot and shooting.

It was also important for the scripts to be as in-house as possible, trading the efficiency of using external scripts for higher customization and learning potential, perhaps using mostly raycasts, as they are simple to manipulate [Roth 1982], robust [Dodin et al. 2011] and critical for game development [Coutinho 2023]. This led to the development of the Simple Sukoba Artifical Intelligence (SSAI) with the objective of making a brand new simple AI system to be used in video games that require hiding, exposing, shooting or exploring.

2. Related Work

Russel's works were utilized extensively to understand the basics of AI agents and implement this knowledge into the game, particularly his agent model [RUSSEL e NORVIG 2013]. The design and philosophy processes were inspired by Unity 4.x Game AI Programming by Aung Sithu Kyaw and Thet Naing Swe, particularly this specific phrase from their work: "the objective here is not to replicate the whole thought process of humans or animals, but to make the NPCs seem intelligent by reacting to the changing situations inside the game world in a way that makes sense to the player"

[Kyaw et al. 2013]. Knowledge on the topic was also reeinforced by Intelligent agents for computer games [Nareyek 2000].

The NavMesh was chosen for its simplicity and performance, as reinforced by Zikky "Due to its simplicity and high efficiency in representing the 3D environment, navigation mesh has become a mainstream choice for 3D games" [Zikky 2016].

The knowledge to create simple formulas for generating sphere shapes using trigonometrical functions was reinforced by Renata Altman & Ivy Kidron's Constructing knowledge about the trigonometric functions and their geometric meaning on the unit circle, "In his effort to perform the different tasks, he has the opportunity to understand the process used to create unit circle representations of trigonometric expressions" [Altman e Kidron 2016].

Given the simplicity requirement, an equally simple tool was needed for the development, and, reinforced by Scott, ray casting seemed perfectly suitable for this, "By virtue of its simplicity, ray casting is reliable and extensible"[Roth 1982]. It was also reinforced that "the simplicity and robustness provided by Ray Casting is a great advantage" [Dodin et al. 2011], and also that "raycasting is a critical concept in game development" [Coutinho 2023].

3. Methodology

3.1. General movement

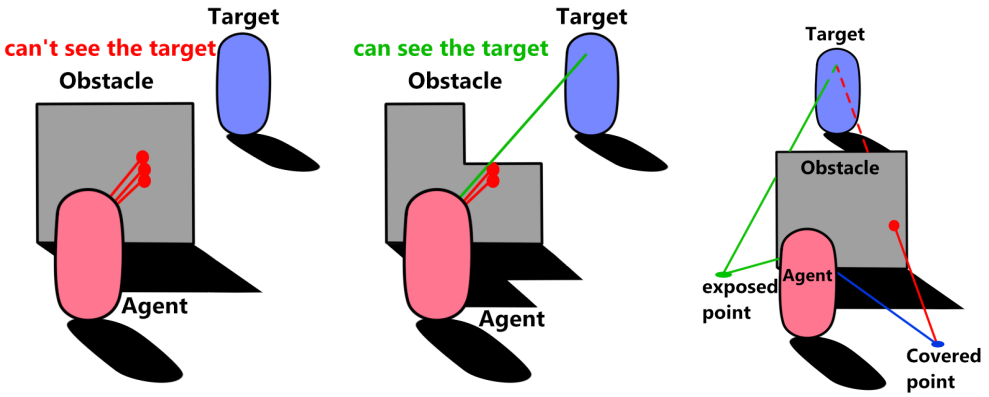
The general movement, that is, the ability to reach positions, is handled by both Unity's native package "NavMesh", and the "AICombatMovement", that inherits from "AIMovement", that inherits from "Movement", all being custom-built classes.

NavMesh is responsible not only for calculating the route, but also for outputting the direction of the route relative to the agent's position. On the other hand, AICombatMovement is responsible for making the agent physically move, getting the NavMesh direction as an input to do that. While the agent does not detect any targets, it will walk to random predefined interest positions indefinitely, it's also important to mention that the agent has a field of view, and any target outside of its field of view will not be detected as long as it does not emit a detectable sound, which will result in the agent turning onto the sound's direction and, if there is a visible target, detect the target.

3.2. Target detection

The ability to detect targets is given by simple ray casts, these ray casts have the agent view point as the origin, the target detection points as targets, and only detect the "ground" layer, which is the layer obstacles are set to. Every detectable entity in the game has the internally developed "Detection" script, this script holds keypoints of detection, such as feet, abdomen, and head (view point). AICombatMovement get these points from the Detection class from a predetermined target and then cast a ray for each keypoint from the AICombatMovement owner's view point, if a ray cast had any interference (hit an obstacle), then it means this point is not visible. If any ray cast is able to hit a keypoint, then the target is at least somewhat visible and will be shot at. Figure 1(a) and Figure 1(b) display how the vision works.

The shooting is handled by another in-house script that takes target exposition information from AIMovement to decide whether to shoot or not.



(a) Target hidden by obstacle. (b) Target exposed. (c) Exposed and covered points.

Figure 1. Target detection and obstacle interference.

3.3. Cover detection

The cover detection functions similarly to the target detection, but in a slightly different way. Instead of casting a ray from the agent view point to the target detection point and returning true if there is no obstacle in-between as it is in target detection, cover detection is done by casting a ray from a certain point onto the target's view point and returning true if there is a obstacle in-between. Figure 1(c) shows how exposed and covered positions are classified.

But, in order for the agent to be able to navigate and get into cover in a random scenario, the specific points used for cover detection have to be calculated around the agent. That is handled by the AIMovement function "GetNearPointsBasedOnVision()".

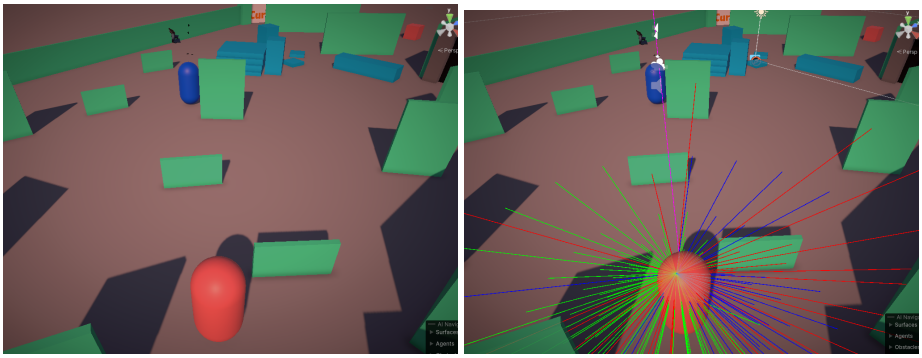
3.3.1. Get near points based on vision function

The "GerNearPointsBasedOnVision()" function creates 24 horizontal slices and 12 vertical slices of a sphere, casting 288 evenly spaced rays that detect the "ground" layer, with a 15 meter range each, equation 1 shows how this specific process is calculated to find the direction of each ray. This creates a spherical explosion-like pattern that originates from the agent's point of view, then returns all the points hit by ray casts. Figure 2 shows how this system works in-game.

$$direction = \frac{(Cos(h/24), Cos(v/12), Sin(h/24))}{|(Cos(h/24), Cos(v/12), Sin(h/24))|} \quad (1)$$

In which:
h = Current horizontal split
v = Current vertical split

This array of points is then divided into 2 types: Covered points and exposed points. Covered points are those who have an obstacle in-between the point itself and the target's view point, while also verifying if any other part of the body will be exposed to



(a) With no visible ray casts (the pink capsule is the agent and the blue capsule is its target, the player).

(b) With visible ray casts (blue denotes covered, green denotes exposed, red denotes unreachable, ray casts that did not hit a surface were not displayed).

Figure 2. In-game nearby points detection (red capsule is the agent and the blue capsule is its target).

that target, while the exposed point is the complete opposite. For a point to be considered reachable, its plane's normal vector has to have 30 degrees of incline at most. With covered and exposed points calculated, the agent is now expected to perform two different movements: Move into cover and move into a shooting position.

3.3.2. Moving into cover

The agent moves into cover if it is being shot at or if its shooting timer has run out. After ducking into cover, a cover timer starts, and once it runs out, it switches into a shooting position. The agent will only move into cover if there is available cover.

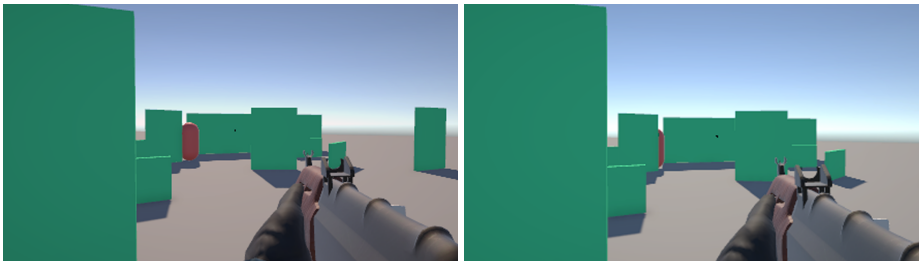
3.3.3. Moving into shooting position

The agent moves into a shooting position once the cover timer has run out, it will then start shooting the target until the shooting timer runs out, in which case it ducks into cover once again. If there is no cover available, the agent will keep shooting and moving into other random shooting positions until it finds available cover. Both moving and staying in cover or shooting positions define the agent combat mode. The exposing to shoot and ducking into cover mechanic can be seen in Figure 3.

3.4. Investigating last target position

There is also the condition where the target contact is lost. This condition does nothing while in cover, but when in position to shoot, losing target contact leads the agent to switch its mode to search mode.

When the agent enter search mode, it creates an array of points in a sphere-like shape around the last seen target position, the radius of the sphere is directly proportional to the distance of the last seen target position.



(a) Agent exposing to shoot. (b) Agent ducking into cover.

Figure 3. Agent combat behavior.

For each of these points, a 1.5 meter long "ground" layer detecting ray is casted downwards, if it hits something and the point is not visible to the agent's view point, it becomes a suspicious point, all suspicious points are then stored in an array and the agentrandomly selects one of them and set it as its destination, returning to exploration mode if it doesn't reach the point in time.

4. Results

The SSAI achieved the requirements, namely: Simplicity, in-house production, capacity of performing exploration movement, capacity of performing combat movement and customizability. The exploration aspect of it is a little too simplistic, due to its lack of procedurality, it requires the developer to choose specific exploration points for the agent to travel through, which may not be ideal for big maps. However, for Sukoba's needs it is enough already in that given aspect.

In search mode, the agent is able to set rooftops or other inaccessible points as its destination, leading it to be stuck until it gives up on looking for a target and then goes into exploration mode again. But, what SSAI really excels at is in its simple, yet robust, procedural combat movement, making the agent capable of reliably ducking into cover when needed (when there is available cover) and also exposing itself when it has to shoot, as visible in Figure 1(a) and Figure 1(b), Achieving the desired dynamic with minimal effort on setting up covers for it, due to its ability to find cover by itself, dramatically improving development efficiency.

5. Final consideration

The SSAI movement is effective at completing its tasks, it works consistently, is customizable and procedural (excluding exploration mode), meaning it works well at most situations as long as they are not too extreme, such as lack of cover.

The combat system is particularly successful, and it works well for the game's requirements. However, there are some inconsistencies in the search mode, particularly in the point gathering aspect of it: It can and will select unreachable spots sometimes, which is still an issue and presents an opportunity for future improvement.

Acknowledgements

This study was financed by Conselho Nacional de Desenvolvimento Científico e Tecnológico – Brasil (CNPq), through processes nº 309228/2021-2; 406463/2022-0; 153641/2024-0

References

Altman, R. e Kidron, I. (2016). Constructing knowledge about the trigonometric functions and their geometric meaning on the unit circle. *International Journal of Mathematical Education in Science and Technology*, 47(7):1048–1060.

Coutinho, C. (2023). Raycasting. In *Roblox Lua Scripting Essentials: A Step-by-Step Guide*, pages 279–304. Springer.

Dodin, P., Martel-Pelletier, J., Pelletier, J.-P., e Abram, F. (2011). A fully automated human knee 3d mri bone segmentation using the ray casting technique. *Medical & biological engineering & computing*, 49:1413–1424.

Kyaw, A. S., Peters, C., e Swe, T. N. (2013). *Unity 4. x Game AI Programming*. Packt Publishing.

Nareyek, A. (2000). Intelligent agents for computer games. In *International Conference on Computers and Games*, pages 414–422. Springer.

Roth, S. D. (1982). Ray casting for modeling solids. *Computer graphics and image processing*, 18(2):109–144.

RUSSEL, S. J. e NORVIG, P. (2013). Inteligência artificial. [sl].

Zikky, M. (2016). Review of a*(a star) navigation mesh pathfinding as the alternative of artificial intelligent for ghosts agent on the pacman game. *EMITTER International journal of engineering technology*, 4(1):141–149.