Artificial Intelligence for Games

Unity Engine 2019.3

June 2020

Digital Learning Games
Tallinn University

Introduction

This project implements AI Behaviour in Unity game engine. This is made for the 'Artificial Intelligence for Games' course in Tallinn University.

Prerequisites

- Unity Engine version 2019.3 and higher
- GitHub Desktop (optional)

Built With

- All functionality and behaviour were created using Unity scripts in C#.

Access to Project

The project is available on GitHub for download. The GitHub repository can be accessed through this link.

AI Behaviour

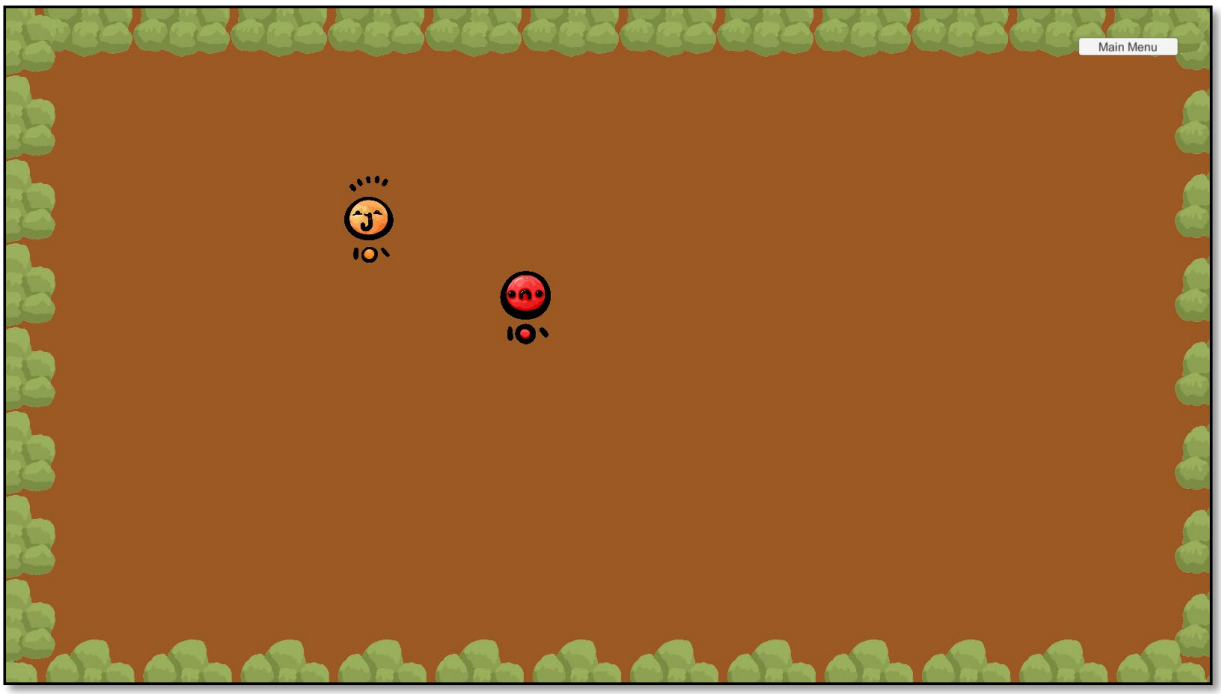
In this project, there are 5 game scenes/levels. Each scene presents an AI behaviour as follows:

1- AI character follows player character.
2- AI character patrols area.
3- AI character chases and shoots at player.
4- Mini-game where AI patrols the level and chases after player.
5- Finite State Machine where AI patrol, pursues and attacks player on sight.

Behaviour Description

The following describes how the AI behaves in each of the game scenes mentioned above.

Scene 1: Enemy Follow



In this scene, the AI character (red) acts as an enemy; by constantly keeping track of the player character's (orange) location. The AI follows the player but keeps a safe distance from them as well. Also, if the player gets too close to the AI, it moves away from the player. This can also be used to simulate AI player companions, where they always follow the player character around a level, and also give them space by backing off from them.

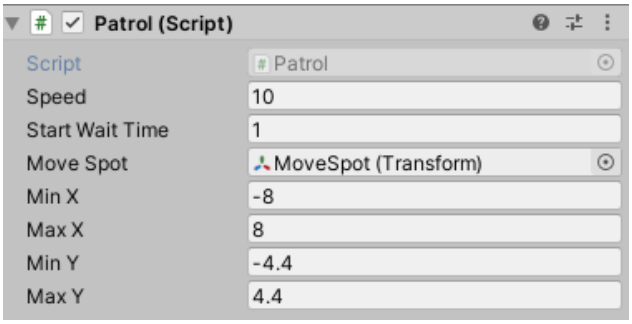
```
}

// Update is called once per frame
Unity Message | 0 references
void Update()
{
    // If target distance to enemy is more then stopping distance, start following target
    if (Vector2.Distance(transform.position, target.position) > stoppingDistance)
    {
        transform.position = Vector2.MoveTowards(transform.position, target.position, speed * Time.deltaTime);
    }
    // if distance between enemy and target is less than stopping distance, but still more than retreat distance, then stand still
    else if (Vector2.Distance(transform.position, target.position) < stoppingDistance && (Vector2.Distance(transform.position, target.position) > ret
    {
        transform.position = this.transform.position;
    }
    // if distance between enemy and target is less than retreat distance, back away from target
    else if (Vector2.Distance(transform.position, transform.position) < retreatDistance)
    {
        transform.position = Vector2.MoveTowards(transform.position, target.position, -speed * Time.deltaTime);
    }

}
```

Sample Code from 'Enemy Follow' Script

Scene 2: Enemy Patrol

In this scene, the AI character (red man) patrols within an area whose boundaries can be set in the inspector window. A target (red diamond) is positioned randomly within the patrol area, and the AI moves towards the target's location. After standing there for a set amount of time, the target is again repositioned to a new random location, triggering the AI to move to it. This simulates the scenario where a guard is patrolling a location, but instead of a static patrol path, the patrol routine is always random and more human-like.

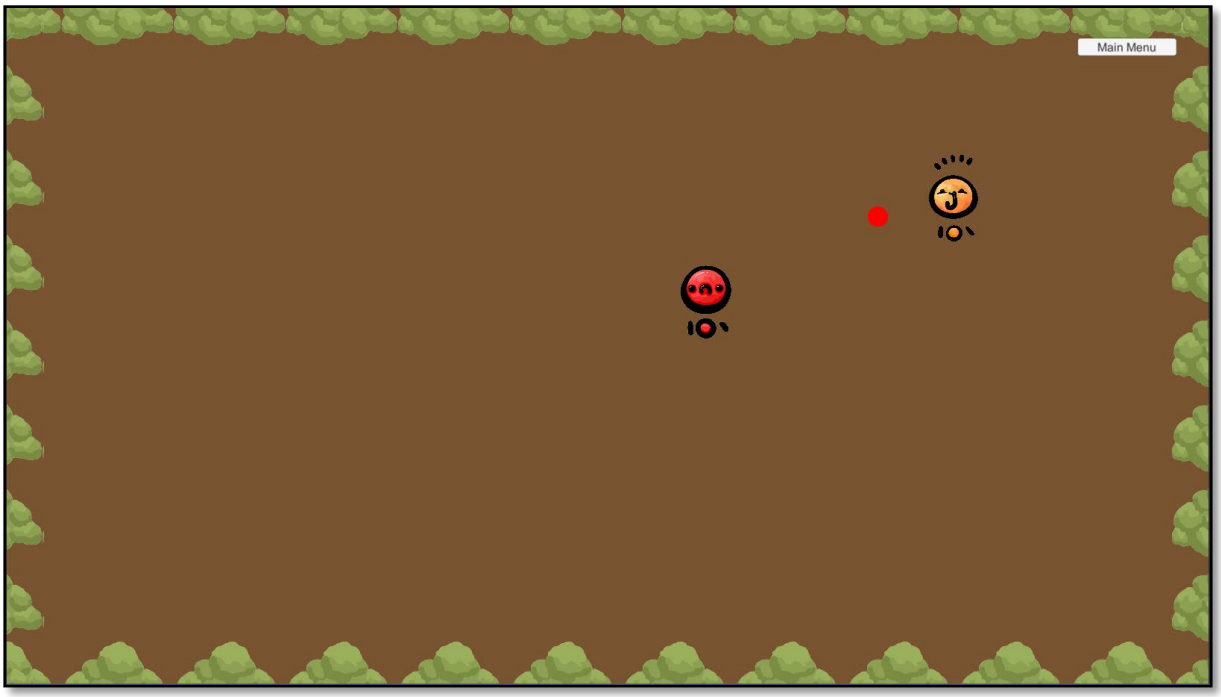
The Patrol script, which controls the AI behaviour, in Unity's inspector window. **Start Wait Time** sets how long should the AI character stand at a patrol point, while the **Min/Max X and Y** values determine the boundaries of the patrol zone.



```
// Update is called once per frame
Unity Message | 0 references
void Update()
{
    // Move the enemy towards the patrol spot location
    transform.position = Vector2.MoveTowards(transform.position, moveSpot.position, speed * Time.deltaTime);

    // Check whether enemy made it (close) to the patrol spot
    if (Vector2.Distance(transform.position, moveSpot.position) < 0.2f)
    {
        // if enemy has waited the set waiting time at patrol location, move the patrol spot
        // to a new random location and reset the waiting countdown timer
        if (waitTime <= 0)
        {
            moveSpot.position = new Vector2(Random.Range(minX, maxX), Random.Range(minY,maxY));
            waitTime = startWaitTime;
        }
        // if the waiting time hasn't passed yet, don't move the enemy from patrol location and
        // continue counting down time waited at patrol location
        else
        {
            waitTime -= Time.deltaTime;
        }
    }

}
```

Sample Code from 'Patrol' Script

Scene 3: Enemy Shoot



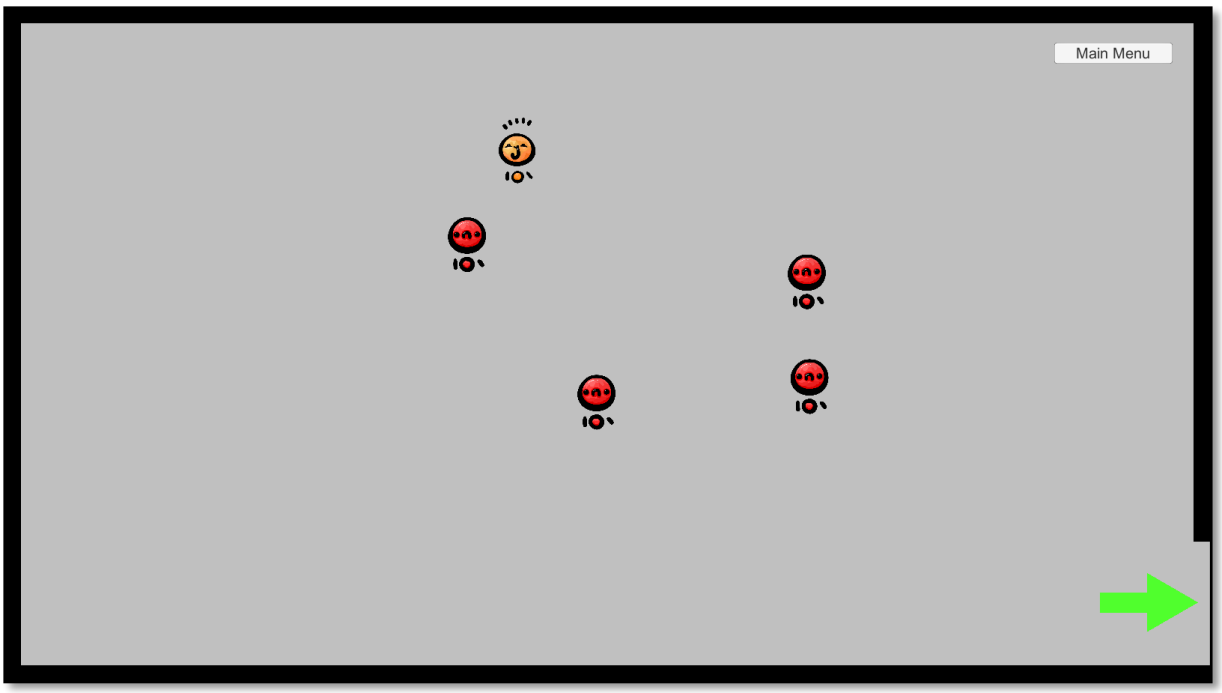
In this scene, the AI character (red) follows the player character (orange) similar to the behaviour explained in Scene 1: Enemy Follow, with the addition of shooting a projectile (red circle) at the player character at equal time intervals. The moment the AI shoots at the player, the player's location at that instant is saved and set as the projectiles target

destination; this prevents the projectile from constantly following the player character as they move (like in homing missiles). If the projectile collides with the player, or if it reaches it's set target without hitting the player, it will vanish.

```
void Update()
{
    // move the projectile towards its target position
    transform.position = Vector2.MoveTowards(transform.position, target, speed * Time.deltaTime);

    // if projectile reaches the target location (without hitting player), destroy it
    if (transform.position.x == target.x && transform.position.y == target.y)
    {
        DestroyProjectile();
    }
}

// if projectile collided with player character, destroy it
Unity Message | 0 references
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.CompareTag("Player"))
    {
        DestroyProjectile();
    }
}

// Destroys the projectile game object
2 references
void DestroyProjectile()
{
    Destroy(gameObject);
}
```

Sample Code from 'Projectile' Script

Scene 4: Spy Game



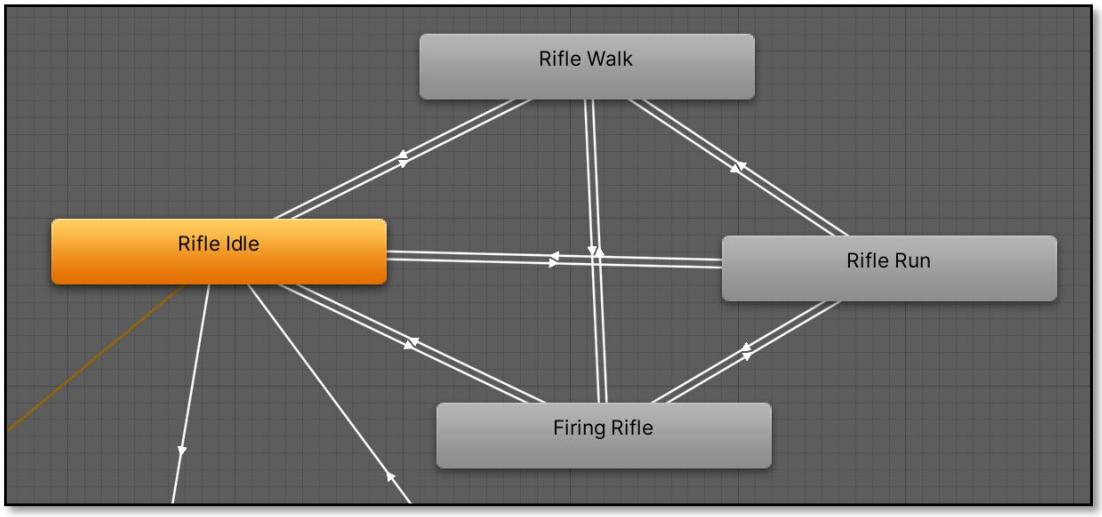
In this scene, a mini-game can be played which implements the patrol and follow AI behaviours. The scene puts the player character (orange) in a room with 4 other AI characters (red) acting as enemies. The goal is for the player to navigate the space to reach the exit on the other end of the room (green arrow) without being captured by the enemy AI. As in the patrol scene, the AI characters patrol the room randomly and if the player gets close enough, they will 'see' the player and start chasing after them. If the player manages to get far away enough from their chasers, they will stop chasing and return to patrolling behaviour.

```
1 reference
void ChaseThePlayer()
{
    // make the enemy run after player character
    transform.position = Vector2.MoveTowards(transform.position, player.position, speed * Time.deltaTime);
}

Unity Message | 0 references
private void OnTriggerEnter2D(Collider2D target)
{
    // if character tagged player enters enemy trigger radius, enemy sets the player as their chase target,
    // sets chasing to true, and then disables the patrol script to stop patrolling
    if (target.CompareTag("Player"))
    {
        player = target.transform;
        isChasing = true;
        patrolScript.enabled = false;
    }
}

Unity Message | 0 references
private void OnTriggerExit2D(Collider2D target)
{
    // if player exits the trigger radius of enemy, enemy stops chasing player and
    // reenables patrolling script
    if (target.CompareTag("Player"))
    {
        isChasing = false;
        patrolScript.enabled = true;
    }
}

Unity Message | 0 references
private void OnCollisionEnter2D(Collision2D collision)
{
    // if enemy collides with player character, freeze game and activate game over UI panel (loss)
    if (collision.transform.CompareTag("Player"))
    {
        Time.timeScale = 0.00f;
        gameOverPanel.SetActive(true);
    }
}
```

Sample Code from ‘Chase’ Script

Scene 5: Finite State Machine

This final scene presents the Finite State Machine (FSM). When AI behaviour becomes more complex, writing it all down in one class/script can become tangled and messy very quickly. The FSM separates each state the AI is in (idle, patrol, pursue, attack, sleep, etc.) into separate classes, but all derived from same base class. Each state then acts as a node with links between it and the other states. The AI then moves between these nodes based on checks and current situations.



Example how FSM states can be connected

Each state contains within itself 3 stages: Enter, Update, and Exit. When the AI decides to move onto and calls a new state, the state first runs through the Enter stage. In this stage, all preliminary functionality takes place to prepare for this new state (e.g. set the character's animation for this state). Afterwards, the Update stage takes place. This state stays as current

and the main AI behaviour happens during it (e.g. run after player character). The AI remains within this state until an event happens which forces it to move to another state. Before proceeding to the next state, the AI passes through the Exit stage of the current state. In this final stage, similar but opposite to Enter stage, all the necessary functions run to clear out any hanging behaviour which might either interrupt or conflict with the new state (e.g. stop playback of walking sound before entering standing state). Looking at the code, the FSM allows for flexibility to expand the AI's behaviour, while still practically wrapping each part of it within itself. This makes it easy to understand how each separate aspect functions and modify it without adversely impacting other unrelated behaviour.

In this project's example, the AI is an armed guard which starts in **Idle** state. With a 10% chance that runs every frame, the guard may move to the **Patrol** state. This state has the guard walking from one point to the next in order. When the player enters the guard's field of vision (i.e. has to be in front of the guard), the guard then enters the **Pursue** state, chasing after the player. If the guard gets close enough to the player, it will then move to an **Attack** state, which makes the AI guard start shooting the player. Should the player run away from the guard, it will start chasing after the player again; meaning it goes back to the Pursue state. If the player manages to completely escape the guard's sight distance (or angle), the AI will then go into an Idle state and then back to Patrol again, going back to the nearest patrol point on its path – until the player comes into contact again.

```
4 references
public class StateIdle : State
{
    // setup inherited constructor
    3 references
    public StateIdle(GameObject _npc, NavMeshAgent _agent, Animator _anim, Transform _player)
            : base(_npc, _agent, _anim, _player)
    {
        name = STATE.IDLE;  // set name of this state
    }

    9 references
    public override void Enter()
    {
        anim.SetTrigger("isIdle");  // trigger the idle animation
        base.Enter();
    }

    5 references
    public override void Update()
    {
        // if AI saw the player, then exit idle state and start chasing
        if (CanSeePlayer())
        {
            nextState = new StatePursue(npc, agent, anim, player);
            stage = EVENT.EXIT;
        }
        // Set a check to exit the Update stage and go to next one (Exit), otherwise will be stuc
        // This condition creates 10% chance to exit Update stage
        else if(Random.Range(0, 100) < 10)
        {
            nextState = new StatePatrol(npc, agent, anim, player);
            stage = EVENT.EXIT;
        }
    }

    9 references
    public override void Exit()
    {
        anim.ResetTrigger("isIdle");    // Reset the triggert set to make sure it's clear before
        base.Exit();
    }
```

An Idle state class – showing the 3 stages (Enter, Update & Exit)

Team Members

- Alper Üste
- Ahmed Mohamed Said Anwar ElShenawy
- Abdelsalam Mohamed Abdelwahab Megahed

Acknowledgments

- AI video tutorials by Noa (aka Blackthornprod) - link
- Finite State Machines by Penny de Byl PHD - link