Artificial Intelligence for Video Games Project

Davide Caldara - 973885

February 2022

1 Introduction

The goal of this project is to create an interactive application with Unity in which are exploited algorithms and techniques of Artificial Intelligence for video games. This application tries to replicate a 3D version of the famous *Pacman* video game with some adjustments. The main focus is to show how to manage basic concepts and algorithms such as final state machines and path-finding to describe agents behaviors. In this project I used *Unity version 2020.3.20f1* and C# language. All project material can be found on GitHub at `https://github.com/DavideCaldara/AI4VGProject.git`

2 Game scene and gameplay

The game main scene consists of a labyrinth, a yellow cube representing Pacman, four bigger cubes representing the ghosts, each one with its representative color, a set of collectibles placed along the labyrinth and four power-ups represented by a blue rotating capsule. The goal of the game is to gather all the collectibles present in the labyrinth without getting caught by enemies. The ghosts will indeed try to chase for the player in different ways or anticipate his movements. Along the way the player will find some power ups. Picking up one of them allows the player to get in contact with a ghost to eliminate him, but they will obviously try to run away from him. This power-up state lasts 20 seconds. During this period each ghost will head to a different corner of the maze and will start patrolling following a predefined path, specific for each enemy. This path is implemented through a set of way-points. When the power-up time expires all ghosts will go back to their normal chase behavior.

3 Final State Machine

The final state machine implementation of this project exploits delegated condition and action functions.

```
1 // Defer function to trigger activation condition
2 // Returns true when transition can fire
3 public delegate bool FSMCondition();
4
5 // Defer function to perform action
6 public delegate void FSMAction();
7
```

```
8  public class FSMTransition {
9
10    // The method to evaluate if the transition is ready to fire
11    public FSMCondition myCondition;
12
13    // A list of actions to perform when this transition fires
14    private List<FSMAction> myActions = new List<FSMAction>();
15
16    public FSMTransition(FSMCondition condition, FSMAction[] actions
        = null) {
17       myCondition = condition;
18       if (actions != null) myActions.AddRange(actions);
19    }
20
21    // Call all  actions
22    public void Fire() {
23       foreach (FSMAction action in myActions) action();
24    }
25 }
```

A transition (lines 8-25) is made of a triggering condition and a list of actions to be performed. The method *myCondition()* (line 11) evaluates if the transition is ready to fire. If it returns true we just call all the actions in the list (lines 21-14).

```
1  public class FSMState {
2
3     // Arrays of actions to perform based on transitions fire (or
       not)
4     // Getters and setters are preferable, but we want to keep the
       source clean
5     public List<FSMAction> enterActions = new List<FSMAction> ();
6     public List<FSMAction> stayActions = new List<FSMAction> ();
7     public List<FSMAction> exitActions = new List<FSMAction> ();
8
9     // A dictionary of transitions and the states they are leading
       to
10    private Dictionary<FSMTransition, FSMState> links;
11
12    public FSMState() {
13       links = new Dictionary<FSMTransition, FSMState>();
14    }
15
16    public void AddTransition(FSMTransition transition, FSMState
       target) {
17       links [transition] = target;
18    }
19
20    public FSMTransition VerifyTransitions() {
21       foreach (FSMTransition t in links.Keys) {
22          if (t.myCondition()) return t;
23       }
24       return null;
25    }
26
27    public FSMState NextState(FSMTransition t) {
28       return links [t];
29    }
30
31    // These methods will perform the actions in each list
32    public void Enter() { foreach (FSMAction a in enterActions) a();
        }
33    public void Stay() { foreach (FSMAction a in stayActions) a(); }
```

```
34    public void Exit() { foreach (FSMAction a in exitActions) a(); }
35
36 }
```

A state has three list of actions: enter, stay and exit. They are performed based on the firing transition (lines 5-7). A dictionary (line 10) is created to link all transitions to the corresponding states. The method *VerifyTransitions()* (lines 20-25) verifies if there are transitions ready to fire while *NextState()* (lines 27-29) returns the next state given a transition. *Enter()*, *Stay()* and *Exit()* functions at last (lines 32-34), execute all actions in the various situations.

```
1  public class FSM {
2
3     // Current state
4     public FSMState current;
5
6     public FSM(FSMState state) {
7        current = state;
8        current.Enter();
9     }
10
11    public void Update() {
12       FSMTransition transition = current.VerifyTransitions ();
13       if (transition != null) {
14          current.Exit();
15          transition.Fire();
16          current = current.NextState(transition);
17          current.Enter();
18       } else {
19          current.Stay();
20       }
21    }
22 }
```

The main FSM class, given a current state, examine all leading out transitions. If a condition is activated, then:

1. Execute actions associated to exit from the current state (line 14);
2. Execute actions associated to the firing transition (line 15);
3. Retrieve the new state and set is as the current one (line 16);
4. Execute actions associated to entering the new current state (line 17);
5. Otherwise, if no condition is activated, Execute actions associated to staying into the current state (line 19).

4 Ghosts AI

Blinky, Pinky, Inky and Clyde artificial intelligence is modelled around the final state machine described in *section 3*. Every state of the FSM represents a behavior and transitions between states are fired checking for particular conditions. Even though some features are shared between ghosts every enemy has its own behavior. The structure of the state machine is overall the same. What changes between the different ghosts is the behavior coded within the state. The movement of the enemy inside the labyrinth is managed directly by unity with the use of a NavMesh. Updating the destination of the agent inside the script attached to it makes it move towards it.

4.1 Blinky

Blinky is the red ghost. His state machine is made up of two states and it is similar to Pinky's and Inky's.

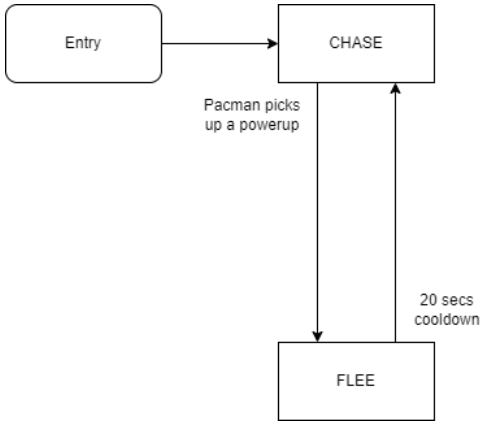

Figure 1: Blinky FSM

The first state is *chase*. Entering the state launches the coroutine *GoChase()* which keeps updating the destination of Blinky every *resampleTime* seconds.

```
1     private IEnumerator GoChase() // chasing player state
2     {
3         while (true)
4         {
5             GetComponent<NavMeshAgent>().destination = destination.
      position;
6             yield return new WaitForSeconds(PlayerController.
      resampleTime);
7         }
8     }
```

In the code above we can see how we can simply periodically update the destination of the agent inside the NavMesh assigning him the actual position of the player to make the ghost following him (line 5). Once in a state, the main FSM coroutine periodically checks for a transition condition from that state to any other to be verified. In the case of Blinky, from chase state I can only go to flee state, so I only need to check for the *PowerUp()* condition.

```
1     public bool PowerUp()
2     {
3         // I check for the powerup to be collected, if one is
      missing fire transition
4         int count = 0;
5         foreach (GameObject go in GameObject.FindGameObjectsWithTag
      (PlayerController.poweruptag)) {
6             if (go.activeSelf) { // count how many powerups are
      still active
7                 count++;
8             }
9         }
10        if (totalActivePowerUps == count) { // stay in chase state
11            print("no powers up collected, keep chasing");
12            return false;
```

```
13          }
14          else { // one has been collected
15              print("power up collected, state transition to flee");
16              totalActivePowerUps--;
17              StopCoroutine(coroutine);
18              coroutine = null;
19              return true;
20          }
21      }
```

The *PowerUp()* function is responsible for the change of state. It checks for a power-up to be collected (lines 107-112) and fires the transition if one is missing (lines 116-121). If none has been collected I stay in chase state waiting for the next update of the main FSM. Firing the transition means also stopping the current active coroutine *GoChase()* (line 119). The new one will be launched inside the next state.

```
1  private IEnumerator GoFlee() // run toward labyrinth corner
2      {
3          while (true)
4          {
5              Vector3 nextWaypointPosition = waypoints[
       nextWaypointIndex].position;
6              GetComponent<NavMeshAgent>().destination =
       nextWaypointPosition;
7              CycleWaypointWhenClose(nextWaypointPosition);
8              yield return new WaitForSeconds(PlayerController.
       fleeResampleTime);
9          }
10     }
```

The coroutine *GoFlee()* is shared between ghosts with different inputs. Indeed, when a power-up has been collected I want all ghosts to head for a specific corner of the maze and follow a specific path. I feed the function of each ghost with a different set of way-points, stored inside an array as positions.

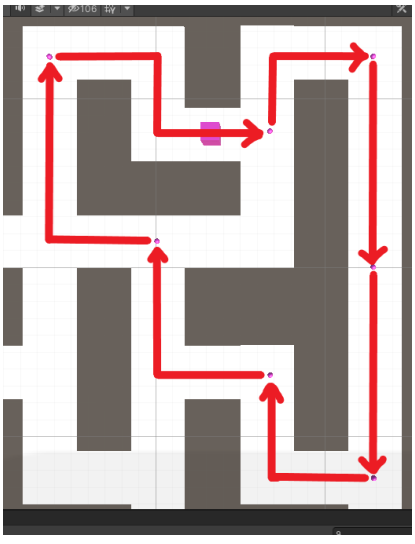

Figure 2: Waypoint Path Example

The coroutine is in charge to update the destination of the NavMesh agent (i.e. the ghost) with the next way-point position (line 135) as soon as the ghost is approaching the current target (line 134).

```
1  public bool Timer()
2      {
3          //print("condizione verificata, torno a stato chase");
4          timeLeft -= (Time.deltaTime + PlayerController.reactionTime
       );
5          print(timeLeft);
6          if (timeLeft < 0) {
7              StopCoroutine(coroutine);
8              coroutine = null;
9              return true;
10         }
11         return false;
12     }
```

The *Timer()* function is responsible for the transition from flee state to chase state. In works as a countdown. After being in the state for 20 seconds it fires back the transition automatically.

```
1  public void BlinkyChase()
2      {
3          //print("entered blinkchase state");
4          activePowerUp = false;
5          coroutine = GoChase();
6          StartCoroutine(coroutine);
7      }
8
9  public void BlinkyFlee()
10     {
11         //print("entered blinkflee state");
12         activePowerUp = true;
13         coroutine = GoFlee();
14         StartCoroutine(coroutine);
15         timeLeft = PlayerController.powerUpDuration; // powerup 20
       seconds
16     }
```

These lines of code represent the states themselves. When I enter *Blinky-Chase()* state I set the activePowerUp variable to false, that means I have no active power-ups at the moment and I start the *GoChase()* coroutine. When I enter *BlinkyFlee()* state instead, I set the *activePowerUp* variable to true, I start the *GoFlee()* coroutine, and I set the countdown.

4.2 Pinky

Pinky is the pink ghost. His state machine has two states and is similar to Blinky's. The only difference lies in the behavior coded inside chase state. We want Pinky to chase after Pacman not directly. This ghost tries to anticipate Pacman movement heading for a way-point placed some units in front of him.

```
1  private IEnumerator GoChase() {
2          while (true) {
3              GetComponent<NavMeshAgent>().destination = GameObject.
       Find("FrontWaypoint").transform.position;
4              yield return new WaitForSeconds(PlayerController.
       resampleTime);
5          }
6      }
```

The agent destination is updated with the position of the GameObject *Front-Waypoint* (line 3), that is directly attached to the player GameObject following his transformations inside the game scene.

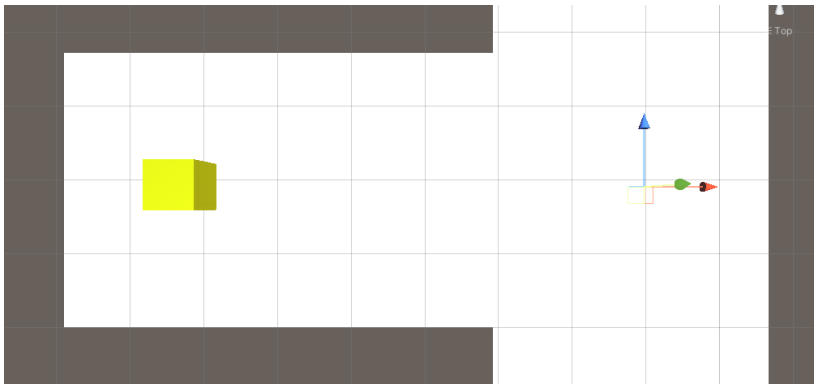

Figure 3: Pacman FrontWaypoint

4.3 Inky

Pinky is the cyan ghost. Once again his state machine has two states and is similar to Blinky's. The only difference lies in the behavior coded inside chase state. His target is a bit complex. It is relative to both Blinky and Pac-Man, where the distance Blinky is from Pinky's target is doubled to get Inky's target.

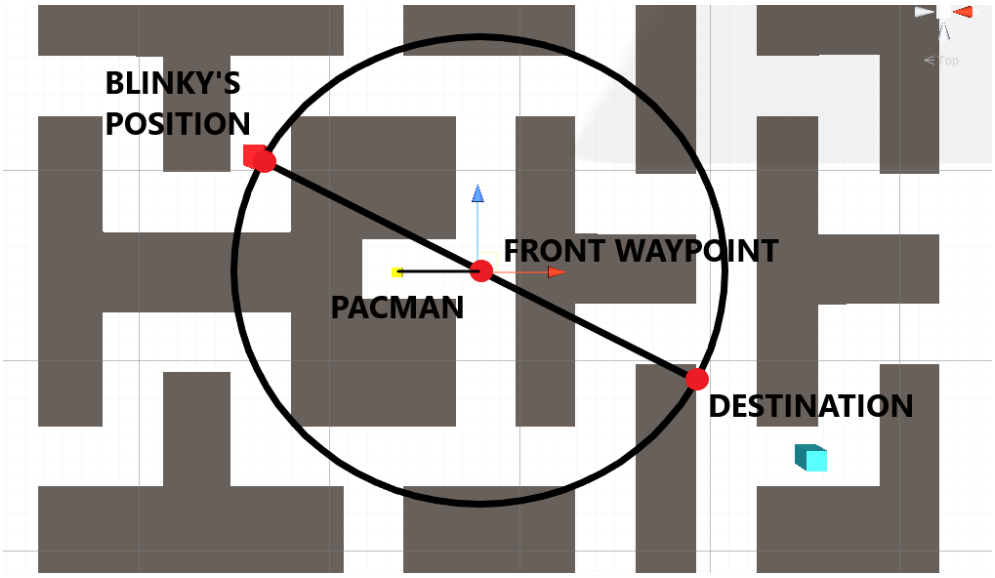

Figure 4: Inky Destination Calculation

```
1 private IEnumerator GoChase()
2     {
```

```
3         while (true)
4         {
5             BlinkyPos = GameObject.Find("Blinky").transform.
    position;
6             WaypointPos = GameObject.Find("FrontWaypoint").
    transform.position;
7             GetComponent<NavMeshAgent>().destination =
    CalculateInkyDestination(BlinkyPos, WaypointPos);
8             yield return new WaitForSeconds(PlayerController.
    resampleTime);
9         }
10    }
11
12    // calculate destination of Inky based on pacman front waypoint
       and blinky position
13    private Vector3 CalculateInkyDestination(Vector3 blinkyPos,
    Vector3 waypointPos)
14    {
15        // switch to 2D (x, z) cause player and ghosts are on the
    same plane
16        Vector2 BlinkyPos = new Vector2(blinkyPos.x, blinkyPos.z);
17        Vector2 WaypointPos = new Vector2(waypointPos.x,
    waypointPos.z); // center of mirroring
18
19        float d = Vector2.Distance(BlinkyPos, WaypointPos);
20        Vector2 versor = ((BlinkyPos - WaypointPos) / d).normalized
    ;
21        Vector2 result = WaypointPos - (versor * d);
22
23        return new Vector3(result.x, GameObject.Find("Blinky").
    transform.position.y, result.y);
24    }
```

The calculation of the destination is delegated to *CalculateInkyDestination()* function which takes in input the position of Blinky and the FrontWaypoint and returns the target position for Inky. To get the right position I have to calculate first the vector from Blinky to FrontWaypoint (lines 19-20). Then I need to calculate the reverse vector and apply it from the FrontWaypoint to find my destination (lines 21). I can simplify the overall computation working in two dimensions because all my positions can be considered lying on the same plane (lines 16-17).

4.4 Clyde

Clyde is the orange ghost. His state machine is a bit more complex than the previous ones because of its additional state. I want Clyde to chase directly for the player always keeping a minimum distance from him. If the ghost gets too close to the player it enters a new state that makes it run away until the minimum distance is complied.

The state machine is exactly the same as Blinky's, with the addition of the Run state. Because of that I also needed to implement two new conditions for my transitions. The *TooClose()* condition for the transition from Chase to Run state and *FarEnough()* condition for the transition from Run to Chase state. I will never go from Flee to Run state, cause I do not want Clyde to run away from Pacman while a power-up is active, so I only need the inverse transition, which I already have. It will fire when the *Timer()* function returns true, that is at the end of the countdown.

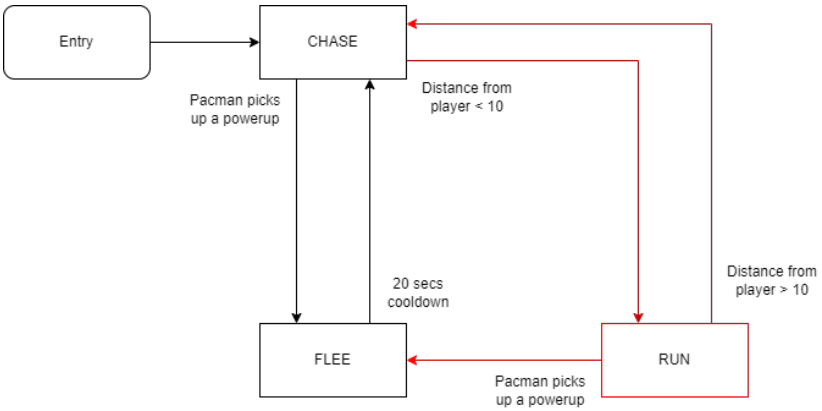

Figure 5: Clyde FSM

```
1  public bool TooClose() // when i am too close to the player
       transition to run state
2      {
3          //print(Vector3.Distance(transform.position, GameObject.
       Find("Player").transform.position));
4          if (Vector3.Distance(transform.position, GameObject.Find("
       PacMan").transform.position) < maxDistance)
5          {
6              StopCoroutine(coroutine);
7              coroutine = null;
8              return true;
9          }
10         return false;
11     }
```

The *TooClose()* condition controls periodically the distance between Clyde and the player (line 4). If the result is less than the minimum allowed distance we stop the current coroutine and fire the state transition to Run.

```
1      public bool FarEnough() // when i am far enough i can go back
       chasing the player
2      {
3          if (Vector3.Distance(transform.position, GameObject.Find("
       PacMan").transform.position) > maxDistance)
4          {
5              StopCoroutine(coroutine);
6              coroutine = null;
7              return true;
8          }
9          return false;
10
11     }
```

The *FarEnough()* conditions, on the contrary, triggers when the distance from Clyde to the player exceeds the maximum allowed (line 3), returning the state machine to the Chase state.

```
1  private IEnumerator GoRun() // run away from the player
2      {
3          while (true)
4          {
```

```
5             float distance = Vector3.Distance(transform.position,
    GameObject.Find("PacMan").transform.position);
6
7             if(distance < maxDistance)
8             {
9                 Vector3 dirToPlayer = transform.position -
    GameObject.Find("PacMan").transform.position;
10                Vector3 newPos = transform.position + dirToPlayer;
11                agent.destination = newPos;
12            }
13            yield return new WaitForSeconds(runResampleTime);
14        }
15    }
```

GoRun() is the coroutine that runs inside the new Run state. It checks for the distance from Clyde to the Player (line 5). If I am too close to the player assign a new agent destination to the opposite direction (lines 9-10).

5 Gameplay enhancements

To report the successful collection of a power up and the entry in the "activePowerUp" state, in addition to the change in behavior of ghosts, I also added a graphic enhancement.

```
1  private void Update()
2      {
3          if (Time.time > nextActionTime) {
4
5              nextActionTime += period;
6
7              if (activePowerUp) // alarm effect on ghosts when
    powerup is active
8              {
9                  if (GetComponent<Renderer>().material.color == temp
    )
10                     GetComponent<Renderer>().material.SetColor("
    _Color", Color.blue);
11                 else
12                     GetComponent<Renderer>().material.SetColor("
    _Color", temp);
13             }
14             else
15             {
16                 GetComponent<Renderer>().material.SetColor("_Color"
    , temp);
17             }
18         }
19     }
```

I saved the original color of my ghost in a temporary variable. Inside the *Update()* method of each ghost, if the power up is active, I keep switching material color between blue and my original color every *period* seconds.

6 Conclusions

The goal of the project was to create an interactive application that exploits methods and algorithms of artificial intelligence for video games. The final result is a fast, responsive and intuitive application as required by this kind of

games. The Entities AIs I developed are working as scheduled without apparent unexpected behaviours. Possible extensions of this project are an overall code optimization, the improvement of graphic elements like new asstes and meshes for the NPCs or for the game board.