Unity for Elderlies

JANUARY 08, 2021

- Learning Objectives: Pluggable FSM
- Introduction
- PlayerTank
- Finite State Machine Scriptable Object
  - StateController
  - State
  - Action
  - Transition and Decision
  - Creating States, Actions, and Decisions Scriptable Objects
  - AITankChaser
  - AITankScanner
- Further Details
  - TransitionToState in StateController.cs
  - Skipping Remaining Transitions
- Summary

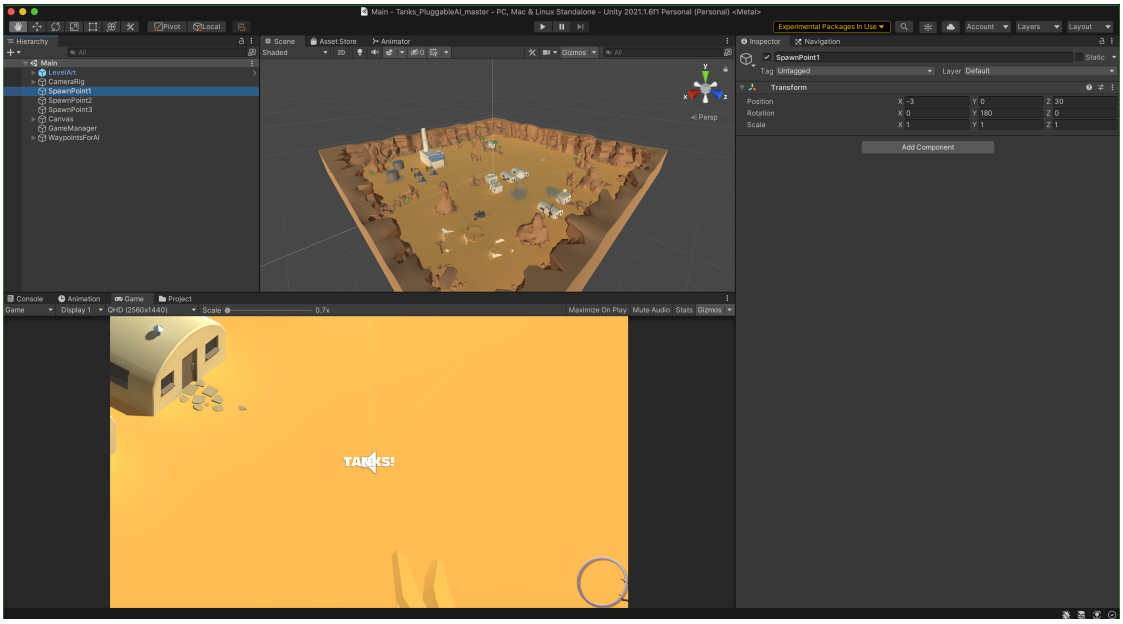
Learning Objectives: Pluggable FSM

- Using Scriptable objects as FSM
- Create state, transition, and action

- Trigger behaviors on certain states

Introduction

Create a new 3D project and import the asset `TankProject` uploaded at the course handout. Then, replace the Project Settings folder with the one provided at the course handout as well. Open Scenes » Main and you shall see this setup:

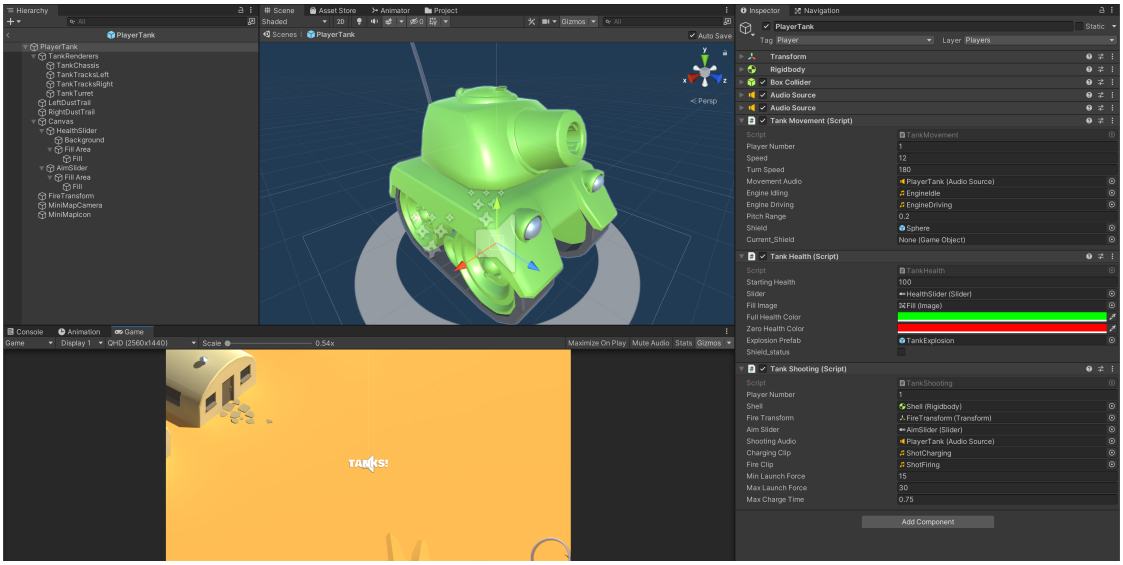


The goal of this tutorial is to **understand** how the game utilises a state machine such that the two tanks: green (scanner) and red (chaser) are able to identify you in the map and perform actions accordingly.

We will use ScriptableObjects that we have learned in the previous part to implement the state machine. The materials and contents in this tutorial is taken from this Official Unity Tutorial.

PlayerTank

Go to Prefab »> `PlayerTank` and open the prefab:



Study these 3 scripts attached to `PlayerTank`:

1. The first script is `TankMovement.cs`, as you know it, it get the keyboard input and perform update of tank location under the FixedUpdate() method
2. The second script is called `TankHealth.cs`. It contains **all** the methods responsible to manage the amount of health this instance of player tank has

> Find out which script calls these **health update** method:
> `TakeDamage`.

3. The third script is `TankShooting.cs`. This is the script that controls how fast a bullet is shot out from the tank, i.e: the longer u press space, the more force the bullet has. Take a look at **line 73 - 75**:
   - It creates a **new** instance of a bullet, and gives it an **initial** velocity in the "forward" direction (local to the tank).
   - Physics engine took care of the rest.
   - Remember you need to **clean** the unused bullet in the end.

> Find out **which script cleans up or Destroy fired bullets**.

Finite State Machine Scriptable Object

Test the game and notice that there's two AI Tanks spawned on the scene:

- **AITankChaser**: the red tank. It will patrol around and will chase the player to oblivion once it has the player once, and shoot once in range until the player dies. Quite unrealistic.
- **AITankScanner**: the green tank. A slightly better Tank that will patrol around and attempt to shoot the Player when seen. Player can "escape" and if the Player is out of sight it will *Scan* for a bit before going back into patrolling.

Both Tanks utilise the FSM architecture works as follows:

1. The enemy tank begin at a start `State`
2. Upon entering any state, you can perform `Action` (s).
3. Then afterwards, make a `Decision` : to compute what should be the next `State` (s) to go to.

`Decision` , `Action` , and `State` are all made using **Scriptable objects**.

All scripts for this FSM can be found under Scripts » PluggableAI

StateController

The Script `StateController.cs` is attached to both AI Tanks. As the name suggests, it dictates the behavior of the AI Tanks, and it receives reference to `CurrentState` (which is starting state when set in the Inspector before the game is run) and `RemainState` (if the AI doesn't transit to other `State`).

State

`State.cs` script contains references to `Action` and `Transition` Scriptable Object (will be explained later):

```
using UnityEngine;

[CreateAssetMenu(menuName = "PluggableAI/State")]
public class State : ScriptableObject
{
        public Action[] actions;
        public Transition[] transitions;
        public Color sceneGizmoColor = Color.grey;
}
```

There are also **three methods** in `State.cs` :

1. `UpdateState` : to be called by StateController during each `Update()` call. This will trigger two other methods: DoActions and CheckTransitions.
2. `DoActions` : we perform the effect of being in this state. The exact instruction is written in the Scriptable Object referred to inside `actions` .
3. `CheckTransitions` : we find out what the **next** state should be, and tell the StateController to *switch* to another state.

Action

There are three kinds of actions: Attack, Chase, and Patrol. All three actions inherit from the same abstract class `Action.cs` :

```
using UnityEngine;
public abstract class Action : ScriptableObject
{
        public abstract void Act(StateController controller);
}
```

It receives `StateController` instance as an input, which is a generic “brain” for each AI Tank. The implementation of each Action is clear cut, for example the

`PatrolAction` tells the `StateController` attached to an Enemy Tank to navigate between a list of WayPoints in the scene (patrol):

```
using UnityEngine;

[CreateAssetMenu(menuName = "PluggableAI/Actions/Patrol")]
public class PatrolAction : Action
{

    public override void Act(StateController controller)
    {
                Patrol(controller);
    }

        void Patrol(StateController controller)
        {
                controller.navMeshAgent.destination = controller.wayPointList[contro
                controller.navMeshAgent.isStopped = false;
                controller.navMeshAgent.speed = controller.enemyStats.moveSpeed;
                if (controller.navMeshAgent.remainingDistance <= controller.navMeshA
                {
                        controller.nextWayPoint = (controller.nextWayPoint + 1) % co
                }
        }
}
```

Please briefly read the functionality of `ChaseAction` and `AttackAction` before proceeding.

Transition and Decision

`Transition.cs` is a simple class that serves as a data structure holding a `Decision` scriptable object reference, and two `States` that a `StateController` can transit to depending on whether the `Decision` 's output is `true` or `false` :

```
[System.Serializable]
public class Transition
```

```
{
        public Decision decision;
        public State trueState;
        public State falseState;
}
```

Then, similar to `Action.cs`, `Decision.cs` is an **abstract class** defining what method should exist:

```
using UnityEngine;
public abstract class Decision : ScriptableObject
{
        public abstract bool Decide(StateController controller);
}
```

The call to `Decide` is called at each `Update()` as well, as result of `CheckTransitions()` in `State.cs` that's called at each `Update()` in `StateController.cs`.

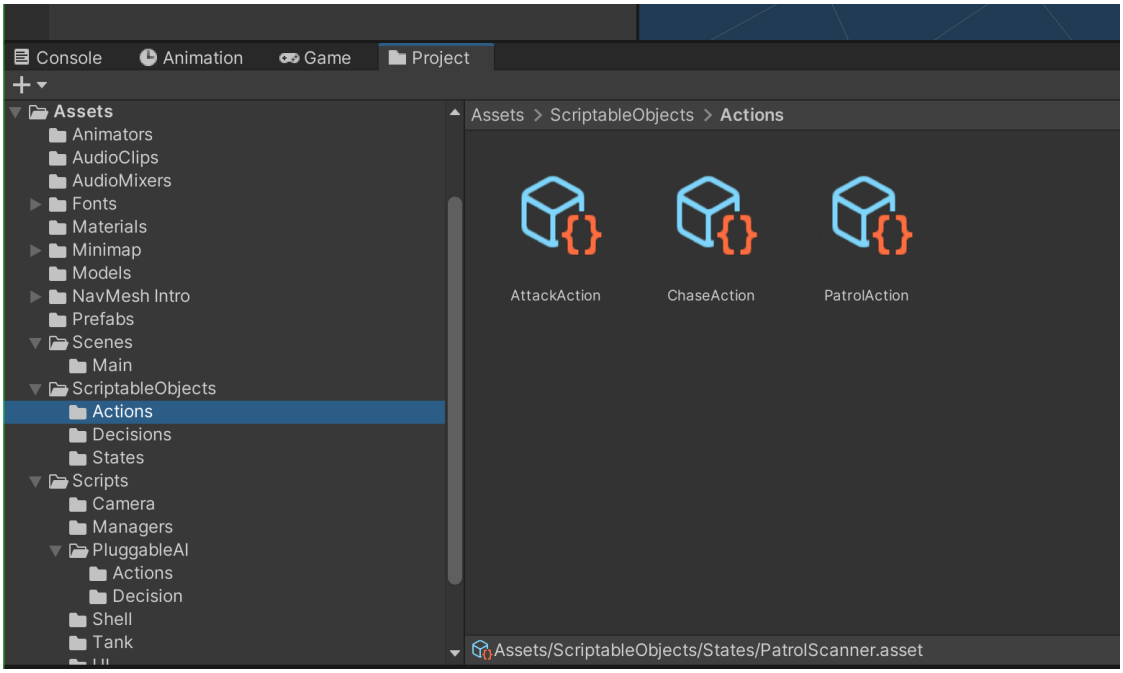
One of the decisions that is made all the time is `LookDecision.cs`; this is where the EnemyTank compute whether or not it has seen the PlayerTank using `RaycastHit`:

```
using UnityEngine;

[CreateAssetMenu(menuName = "PluggableAI/Decisions/Look")]
public class LookDecision : Decision
{

    public override bool Decide(StateController controller)
    {
        bool targetVisible = Look(controller);
        return targetVisible;
    }

    private bool Look(StateController controller)
    {
        RaycastHit hit;
                Vector3 position = controller.eyes.position;
```

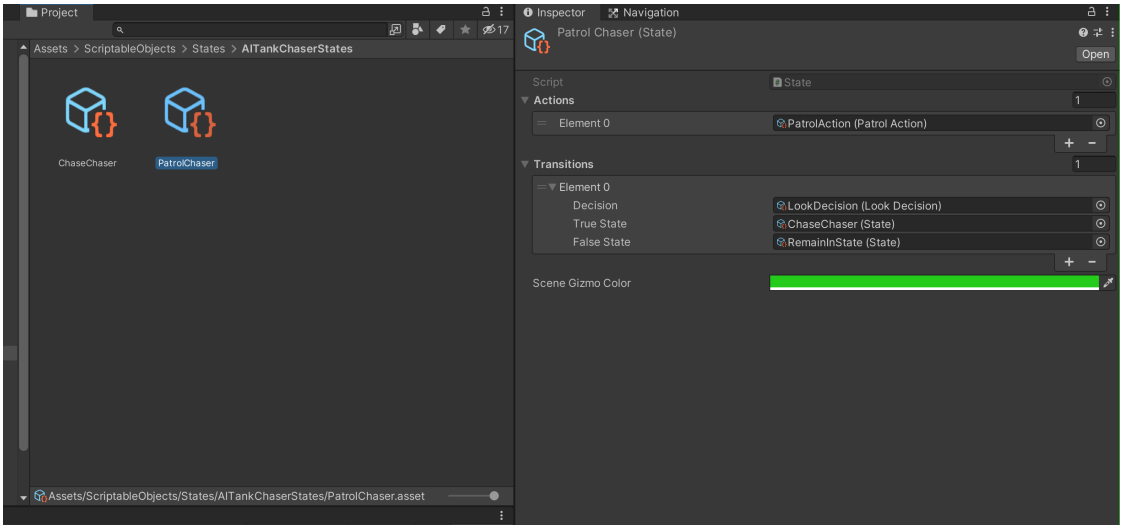
```
                float radius = controller.enemyStats.lookSphereCastRadius;
                Vector3 direction = controller.eyes.forward;
                float lookRange = controller.enemyStats.lookRange;

                Debug.DrawRay(position, direction.normalized * lookRange, Color.gree

        if (Physics.SphereCast(position, radius, direction, out hit, lookRange) && h
        {
                        controller.chaseTarget = hit.transform;
                        return true;
        }
                else
                {
                        return false;
                }
    }
}
```

Creating States, Actions, and Decisions Scriptable Objects

All Scriptable Object instances can be found under ScriptableObjects folder. For Actions and Decisions, they're pretty straightforward: one instance per script type with no other setup required:



The States instances however require more setup. For instance, click on ScriptableObjects » States » AITankChaserStates » **PatrolChaser**, and you shall see this at the inspector:



RemainInstate is a special state with empty Actions and Transitions. It's effect is to stay at the same state and perform `DoActions` and `CheckTransitions` of the **same state** again at the **next** `Update()` call.

PatrolChaser is the one of the States to be plugged into AITankChaser (the not-so-fun one). Click on other States and study its structure: a state can have multiple Actions and Transitions – to be computed in *sequential* order as per `DoActions` and `CheckTransitions` methods in `State.cs` .

Note that CheckTransitions will always call StateController's TransitionToState based on the value of decisionSucceeded:

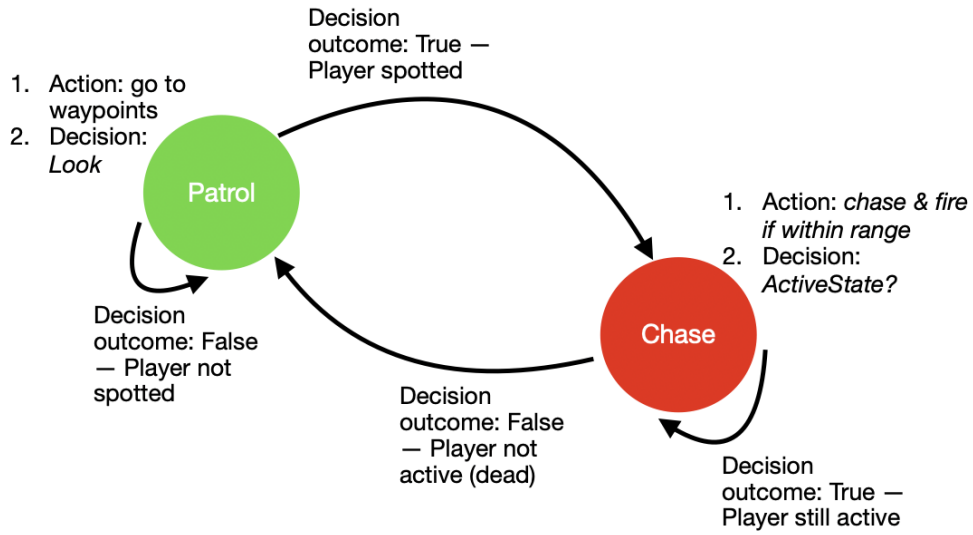
```
// In State.cs
        private void CheckTransitions(StateController controller)
        {
                controller.transitionStateChanged = false; //reset
                for (int i = 0; i < transitions.Length; ++i)
                {
                        //check if the previous transition has caused a change. If y
                        if (controller.transitionStateChanged){
                                break;
                        }
                        bool decisionSucceded = transitions[i].decision.Decide(contr
                        if (decisionSucceded)
                        {
                                controller.TransitionToState(transitions[i].trueStat
                        }
                        else
                        {
                                controller.TransitionToState(transitions[i].falseSta
                        }
                }
        }
```

It will only proceed to the next `i` **if** `transitions[i]` true or false state is `RemainInState`:

```
// in StateController.cs
        public void TransitionToState(State nextState)
        {
                if (nextState == remainState) return;
                currentState = nextState;
                transitionStateChanged = true;

                if(currentState.name == "ChaseScanner" || currentState.name == "Chas

                OnExitState();
        }
```

AITankChaser

The Red AI Tank's (chaser tank) state transition diagram can be drawn such:

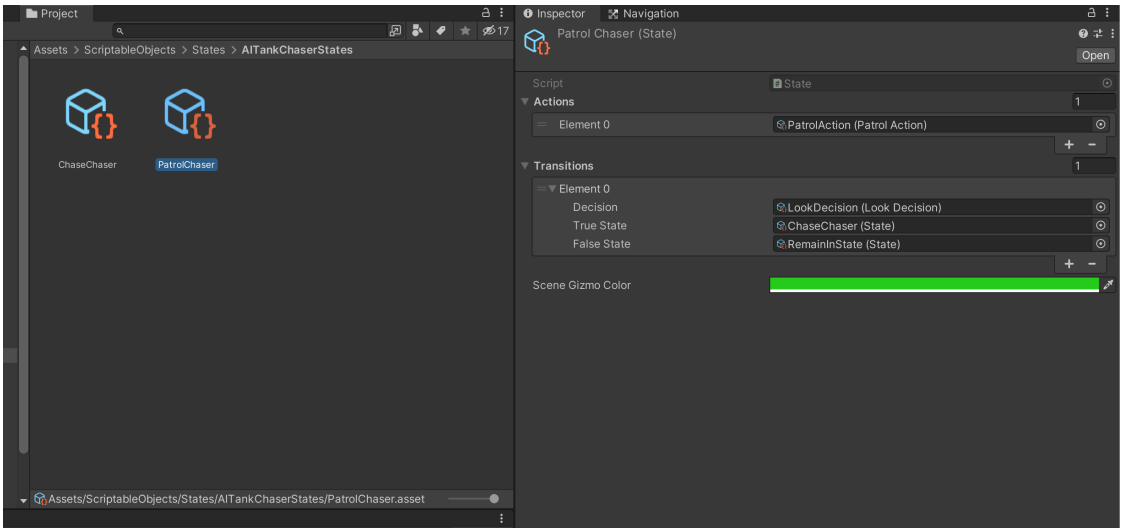


What this Bot does in a nutshell:

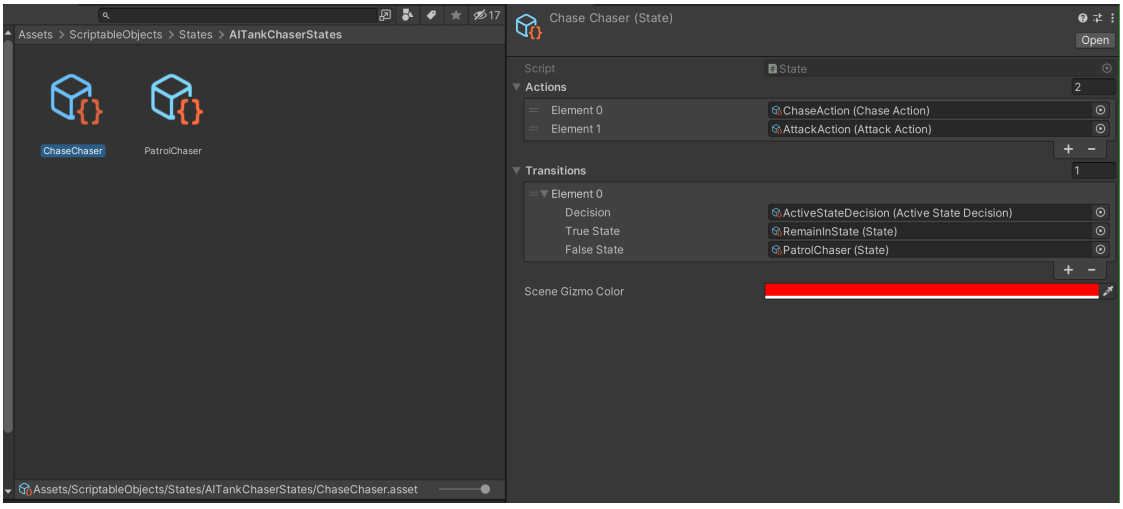
1. Begin at state `PatrolChaser` and perform `PatrolAction` of circulating around waypoint 1, 2, then 3 in that order. Afterwards, always perform `LookDecision` : cast a ray and see if it hits anybody with a tag of Player.
   - If `true` (yes), set the chase target as the Player object and store its location, and go to `ChaseChaser` State
   - If `false` (no), `RemainInState` : `PatrolState` and **repeat** `PatrolAction` and `LookDecision` in the next update call
2. If the bot arrives at `ChaseChaser` State, perform **two** actions.
   - Action 1: `ChaseAction` Move to the last chase target location of Player, and then
   - Action 2: `AttackAction` — fire bullet **if** player is within range, otherwise this attack motion does nothing.
   - Then, to perform Decision called `ActiveStateDecision` that checks if PlayerTank gameobject is still **Active**:

- If PlayerTank is **still active**: The tank will `RemainInState : ChaseChaser` and **repeat** `ChaseAction` and `AttackAction` , followed by `ActiveStateDecision` until the PlayerTank *dies* (inactive in Hierarchy).
- If PlayerTank is **no longer active**: — means *dead* — The bot will go back to `PatrolChaser` State.

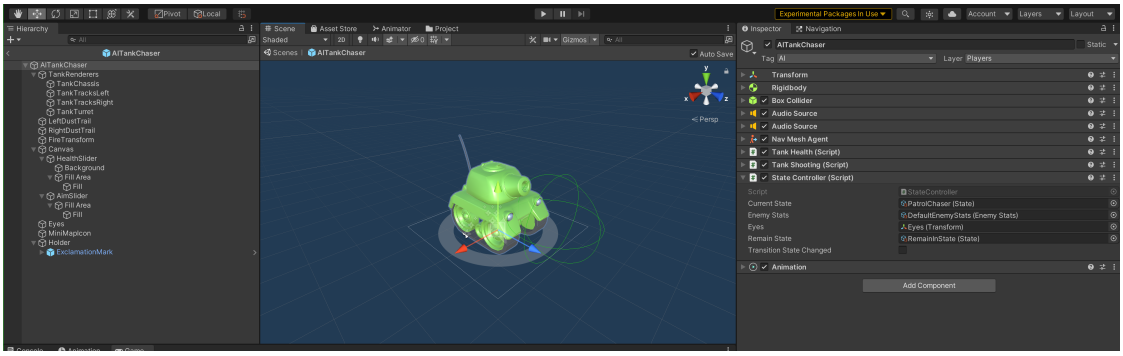
The green state “Patrol” above corresponds to the Scriptable Object `PatrolChaser` :



and the red state “Chase” in the diagram above corresponds to the Scriptable Object `ChaseChaser` in the Project.

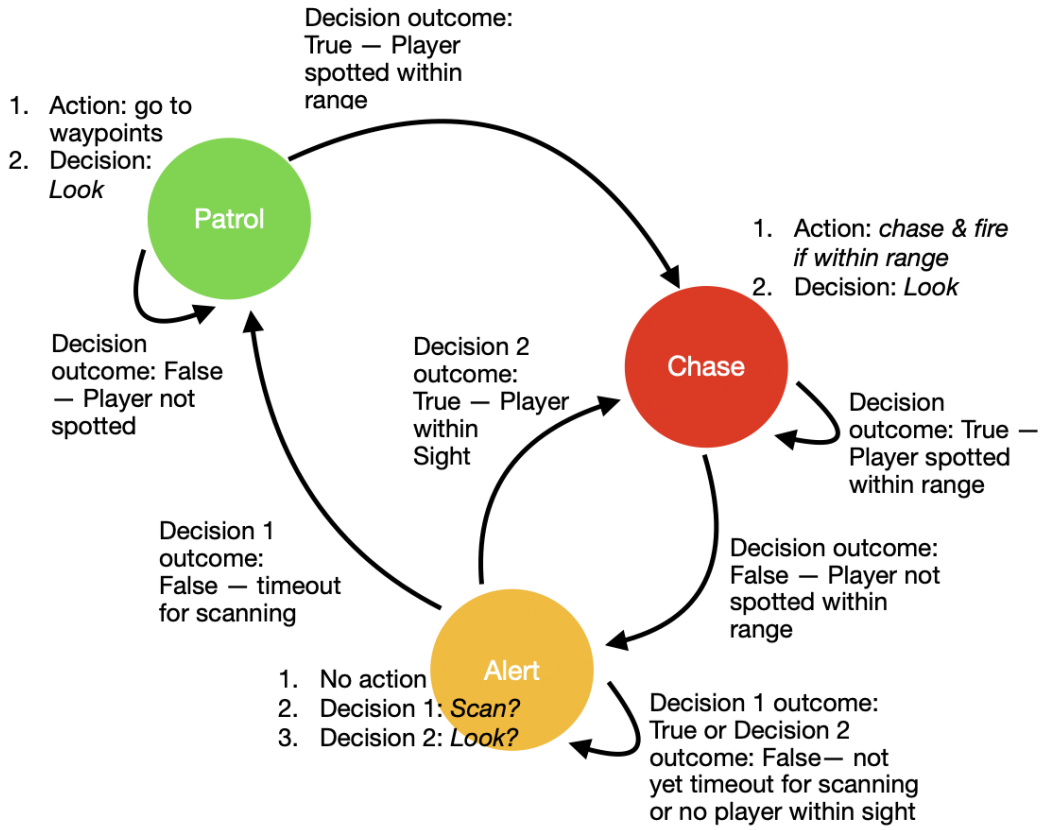


Hence the StateController component in AITankChaser is set to start at `PatrolChaser` as it’s **starting state**:



AITankScanner

The Green AI Tank's (scanner tank) state transition diagram can be drawn such:



What this Bot does in in a nutshell:

1. An AITankScanner will start in the `PatrolScanner` State,
2. Similarly, it performs the action of going around waypoints if there's no player in sight (`PatrolAction`).
3. It will also perform the `LookDecision` :
   - If there's a player in sight, it will go to `ChaseScanner` state (chase and fire).

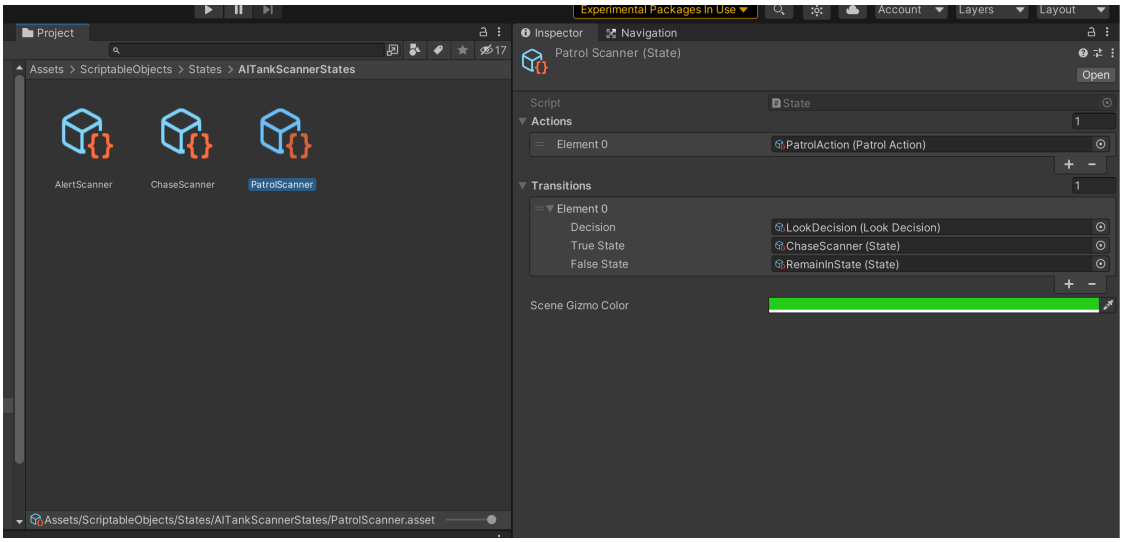
- If there's no player in sight, `RemainInState : PatrolScanner` state and repeat.

4. If it arrives at `ChaseScanner` State, it will perform `AttackAction` similar to the Chaser Tank and then perform another `LookDecision` :
   - If player's no longer within sight then it will go to an `AlertScanner` state where it will perform **two** Decisions — one after another:
     - `ScanDecision` (rotate in place to search for player) and then,
     - `LookDecision` .
     - In `ScanDecision` , there's a *timeout* involved.

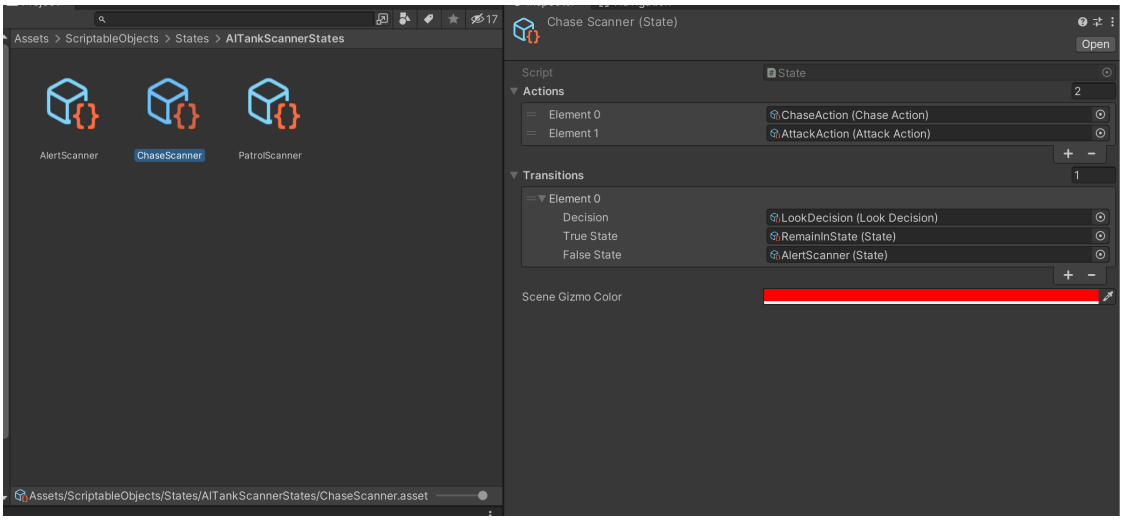
> Find which script keeps track of the time elapsed while Scanning?

   - Else, if Player is still spotted within range then `RemainInState : ChaseScanner` .
   - The tank will go either to back to `PatrolScanner` , remain in `ScanScanner` state, or go `ChaseScanner` State accordingly depending on the outcome of these two decisions.

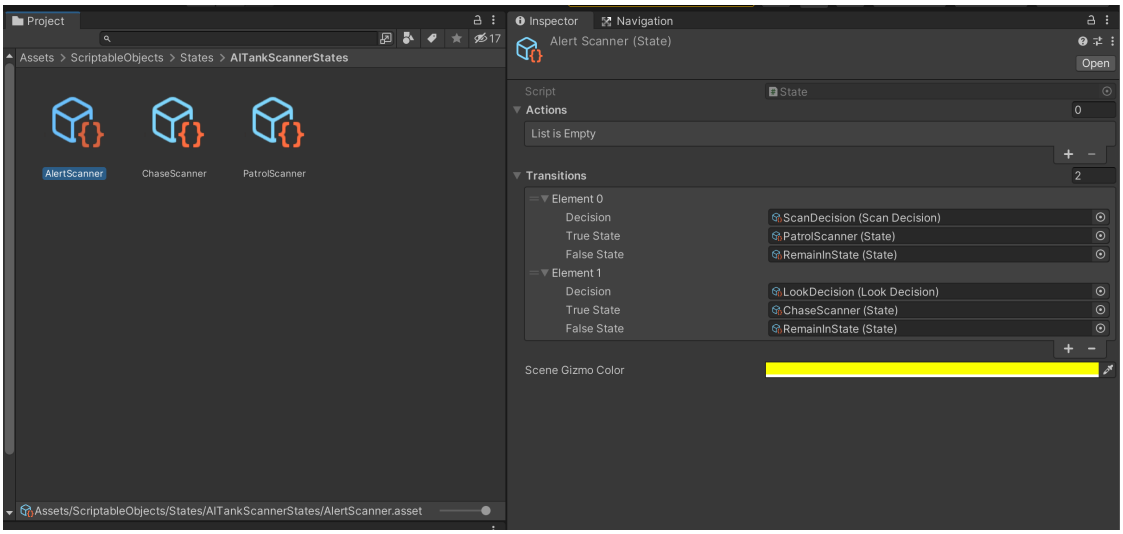
The green state "Patrol" above corresponds to the Scriptable Object `PatrolScanner :`



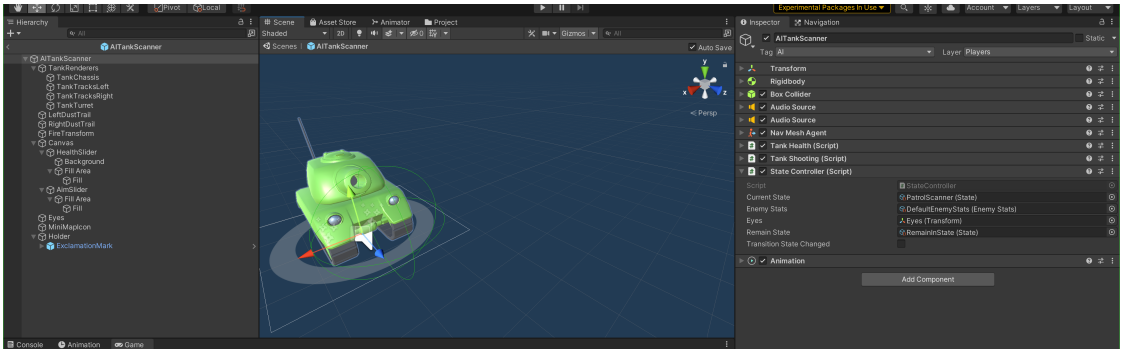
The red state “Chase” in the diagram above corresponds to the Scriptable Object `ChaseScanner`:



The yellow state “Alert” in the diagram above corresponds to the Scriptable Object `AlertScanner` in the Project:



Hence the StateController component in AITankScanner is set to start at `PatrolScanner` as it’s **starting state**:



Further Details

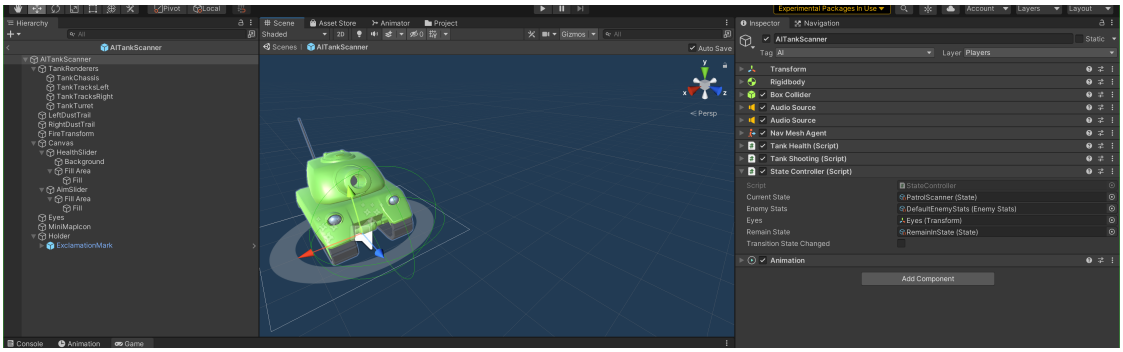
This section explains further on the details between state transitions: `Decision` that results in `RemainInState` or some new State.

TransitionToState in StateController.cs

In `StateController.cs` script, we have this method:

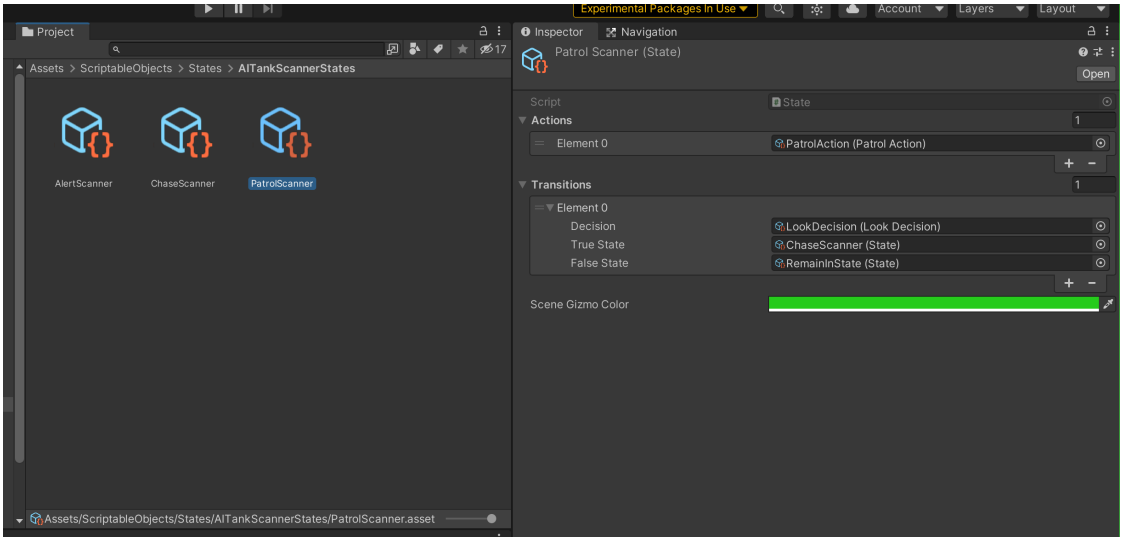
```
public void TransitionToState(State nextState)
{
    if (nextState == remainState) return;
    currentState = nextState;
    if(currentState.name == "ChaseScanner" || currentState.name == "ChaseChaser")
    OnExitState();
}
```

It is clear that if `nextState` equals to `remainState` (which is a **public** variable with `RemainInState` Scriptable Object Linked Up in the Inspector) as shown for AITankScanner as example:



..then we don't need to execute the rest of the instructions anymore.

But why can't we simply put the **same** state, for example `PatrolScanner` as the `FalseState` instead? Why do we have to use `RemainInState` ?



The difference between using `RemainInState` and just setting the `False State` as itself (e.g: `PatrolScanner` for above's example) depends on whether `OnExitState()` is called in the last instruction of `TransitionToState` in `StateController.cs`.

This design of **not calling** `OnExitState` when `RemainInState` is handy if we want to keep track of some kind of **countdown** (`timeElapsedInState`) while being in a State for more than a frame, or if we need to clean up some variables while remaining in the State for more than one Update frame. In `StateController.cs`, `OnExitState` is used to cleanup the timer to keep track how long exactly we have been in the `AlertScanner` state:

```
void OnExitState()
{
        stateTimeElapsed = 0;
}
```

Skipping Remaining Transitions

In `State.cs`, we have the following check transition method where we check the `bool transitionStateChanged` inside `controller`, due to the effect of calling `TransitionToState` at the end of the `for` -loop.

```
    private void CheckTransitions(StateController controller)
    {
        controller.transitionStateChanged = false; //reset
        for (int i = 0; i < transitions.Length; ++i)
        {
            //check if the previous transition has caused a change. If yes, stop. Let
            if (controller.transitionStateChanged){
                break;
            }

            bool decisionSucceeded = transitions[i].decision.Decide(controller);
            if (decisionSucceeded)
            {
                controller.TransitionToState(transitions[i].trueState);
            }
            else
            {
                controller.TransitionToState(transitions[i].falseState);
            }
        }
    }
```

This instruction: `if (controller.transitionStateChanged)` is used to **skip** the need for going through other `Decisions` once a previous `Decision` leads to a **NEW** next state. Otherwise if the previous Decisions always ends up with `RemainInState`, then we continue iterating through other Decisions to check if any of them leads to a **new next state**.

Summary

There's no checkoff with this lab. In a nutshell, we have learned how to use ScriptableObject to create a simple Finite State Machine:

1. We can create various types of Abstract classes inheriting ScriptableObject: Action, Decision, and State
2. Create new scripts inheriting either of the above for **specific** usage

3. How to use them along with a generic `StateController` specific to the tank
4. How to trace back the usage of Action, Decision, and State Scriptable object instances during each `Update` call of `StateController`

NEXT POST
Unity for Midlifes