AI Actions, Formations, FSM & Notification center

FSM Documentation:

1.- Unity3D – Easy Finite State Machine (C#)

State machines are a very effective way to manage game state, either on your main game play object (Game Over, Restart, Continue etc) or on individual actors and NPCs (AI behaviours, Animations, etc). The following is a simple state machine that should work well within any Unity context.

Designed with simplicity in mind

Most state machines come from the world of C# enterprise, and are wonderfully complicated or require a lot of boilerplate code. State Machines however are an incredibly useful pattern in game development, administrative overhead should never be a burden that discourages you from writing good code.

- Simple use of Enums as state definition.
- Minimal initialization – one line of code.
- Incredibly easy to add/remove states
- Uses reflection to avoid boiler plate code – only write the methods you actually need.
- Compatible with Coroutines.
- Tested on iOS and Android

Usage

An example project is included (Unity 5.0) to show the State Machine in action.

To use the state machine you need a few simple steps

Include the StateMachine package

```
using AxlPlay; //Remember the using statement before the class declaration

public class MyManagedComponent : MonoBehaviour{

}
```

Define your states using an Enum

```
public enum States{

    Init,

    Play,

    Win,

    Lose

}
```

Create a variable to store a reference to the State Machine

```
StateMachine<States> fsm;
```

Get a valid state machine for your MonoBehaviour

```
fsm = StateMachine<States>.Initialize(this);
```

This is where all of the magic in the StateMachine happens: in the background it inspects your MonoBehaviour (`this`) and looks for any methods described by the convention shown below.

You can call this at any time, but generally `Awake()` is a safe choice.

You are now ready to manage state by simply calling `ChangeState()`

```
fsm.ChangeState(States.Init);
```

State callbacks are defined by underscore convention (`StateName_Method`)

```
void Init_Enter(){

    Debug.Log("We are now ready");

}

//Coroutines are supported, simply return IEnumerator

IEnumerator Play_Enter(){

    Debug.Log("Game Starting in 3");
```

```
    yield return new WaitForSeconds(1);



    Debug.Log("Game Starting in 2");

    yield return new WaitForSeconds(1);

     Debug.Log("Game Starting in 1");

    yield return new WaitForSeconds(1);

    Debug.Log("Start");

}

void Play_Update(){

    Debug.Log("Game Playing");

}

void Play_Exit(){

    Debug.Log("Game Over");

}
```

Currently supported methods are:

- `Enter`
- `Exit`
- `FixedUpdate`
- `Update`
- `LateUpdate`
- `Finally`
- `OnCollisionEnter`
- `OnTriggerEnter`

These methods can be private or public. The methods themselves are all optional, so you only need to provide the ones you actually intend on using.

Couroutines are supported on Enter and Exit, simply return `IEnumerator`. This can be great way to accommodate animations.
Note:`FixedUpdate`, `Update` and `LateUpdate` calls won't execute while an Enter or Exit routine is running.

Finally is a special method guaranteed to be called after a state has exited. This is a good place to perform any hygiene operations such as removing event listeners. Note: Finally does not support coroutines.

Transitions

There is simple support for managing asynchronous state changes with long enter or exit coroutines.

```
fsm.ChangeState(States.MyNextState, StateTransition.Safe);
```

The default is `StateTransition.Safe`. This will always allows the current state to finish both it's enter and exit functions before transitioning to any new states.

```
fsm.ChangeState(States.MyNextState, StateTransition.Overwrite);
```

`StateMahcine.Overwrite` will cancel any current transitions, and call the next state immediately. This means any code which has yet to run in enter and exit routines will be skipped. If you need to ensure you end with a particular configuration, the finally function will always be called:

```
void MyCurrentState_Finally(){

    //Reset object to desired configuration

}
```

EXAMPLE:

```
using AxlPlay; //Remember the using statement before the class
declaration

 // init FSM

public enum States {

      Idle,

      Pursue,

      ArrivedEvent
```

```
}

public StateMachine<States> fsm;

void Start(){

        //Initialize State Machine Engine

        fsm = StateMachine<States>.Initialize(this, States.Idle);

 }

void Idle_Enter(){

    fsm.ChangeState(States.Pursue)

}

void Pursue_Enter(){

}

void Pursue_Update(){

}
```

2.- FSM Example:

```
// init FSM
    public enum States{
        Idle,
        Flee,
        Finish
    }
    public StateMachine<States> fsm;
    void Awake() {
      navMeshAgent = GetComponent<NavMeshAgent>();
    }
    void Start() {
        //Initialize State Machine Engine
        fsm = StateMachine<States>.Initialize(this, States.Idle);
    }
   void Idle_Enter(){
        Debug.Log("Flee => Idle_Enter");
    }
```

Actions Documentation:

3.- Actions:

Flee:

Flee will move the agent away from the target with pathfinding.

fleedDistance

The agent has fleed when the magnitude is greater than this value

lookAheadDistance

The distance to look ahead when fleeing

target

The GameObject that the agent is fleeing from

Pursue:

Pursue predicts where the target is going to be in the future. This allows the agent to arrive at the target earlier than it would have with the Set Destination Action.

targetDistPrediction

How far to predict the distance ahead of the target. Lower values indicate less distance should be predicated

targetDistPredictionMult

Multiplier for predicting the look ahead distance

target

The GameObject that the agent is pursuing

EVADE:

Evade is similar to the Flee except the Evade predicts where the target is going to be in the future. This allows the agent to flee from the target earlier than it would have with the Flee.

evadeDistance

The agent has evaded when the magnitude is greater than this value.

lookAheadDistance

The distance to look ahead when evading.

targetDistPrediction

How far to predict the distance ahead of the target. Lower values indicate less distance should be predicated.

targetDistPredictionMult

Multiplier for predicting the look ahead distance.

target

The GameObject that the agent is evading.

PATROL:

Patrol moves from waypoint to waypoint.

randomPatrol

Should the agent patrol the waypoints randomly?

waypointPauseDuration

The length of time that the agent should pause when arriving at a waypoint

waypoints

The waypoints to move to

CAN SEE OBJECT:

Can See Object returns the object when it sees an object in front of the current agent.

usePhysics2D

Should the 2D version be used?

targetObject

The object that we are searching for. If this value is null then the objectLayerMask will be used

objectLayerMask

The LayerMask of the objects that we are searching for

ignoreLayerMask

The LayerMask of the objects to ignore when performing the line of sight check

fieldOfViewAngle

The field of view angle of the agent (in degrees)

viewDistance

The distance that the agent can see

offset

The offset relative to the pivot position

targetOffset

The target offset relative to the pivot position

returnedObject

The object that is within sight

CAN HEAR OBJECT:

The Can Hear Object return the object when it hears another object.

usePhysics2D

Should the 2D version be used?

targetObject

The object that we are searching for. If this value is null then the objectLayerMask will be used

objectLayerMask

The LayerMask of the objects that we are searching for

hearingRadius

How far away the unit can hear

audibilityThreshold

The further away a sound source is the less likely the agent will be able to hear it. Set a threshold for the the minimum audibility level that the agent can hear

offset

The offset relative to the pivot position

returnedObject

The returned object that is heard

WANDER:

Wander moves the agent randomly throughout the map.

wanderDistance

How far ahead of the current position to look ahead for a wander

wanderRate

The amount that the agent rotates direction

SEARCH:

Search will search the map by wandering until it finds the target. It can find the target by seeing or hearing the target.

wanderDistance

How far ahead of the current position to look ahead for a wander

wanderRate

The amount that the agent rotates direction

fieldOfViewAngle

The field of view angle of the agent (in degrees)

viewDistance

The distance that the agent can see

ignoreLayerMask

The LayerMask of the objects to ignore when performing the line of sight check

senseAudio

Should the search end if audio was heard?

hearingRadius

How far away the unit can hear

offset

The offset relative to the pivot position

targetOffset

The target offset relative to the pivot position

objectLayerMask

The LayerMask of the objects that we are searching for

linearAudibilityThreshold

The further away a sound source is the less likely the agent will be able to hear it. Set a threshold for the the minimum audibility level that the agent can hear

returnedObject

The object that is within sight.

TARGET RECHEABLE:

Target Reachable detect if a either a complete or partial path is found.

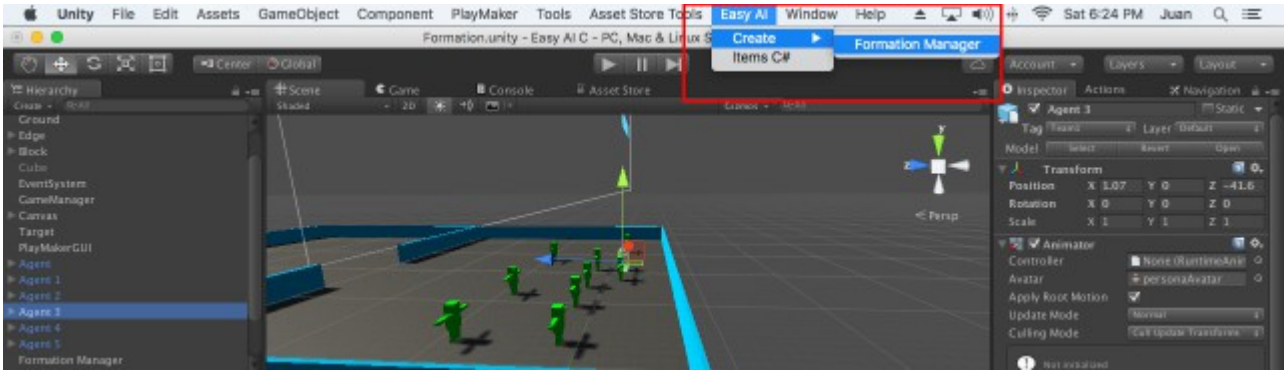
Target Reachable => Target GameObject.

isReachable => True if a either a complete or partial path is found and false otherwise.

Formations Documentation:

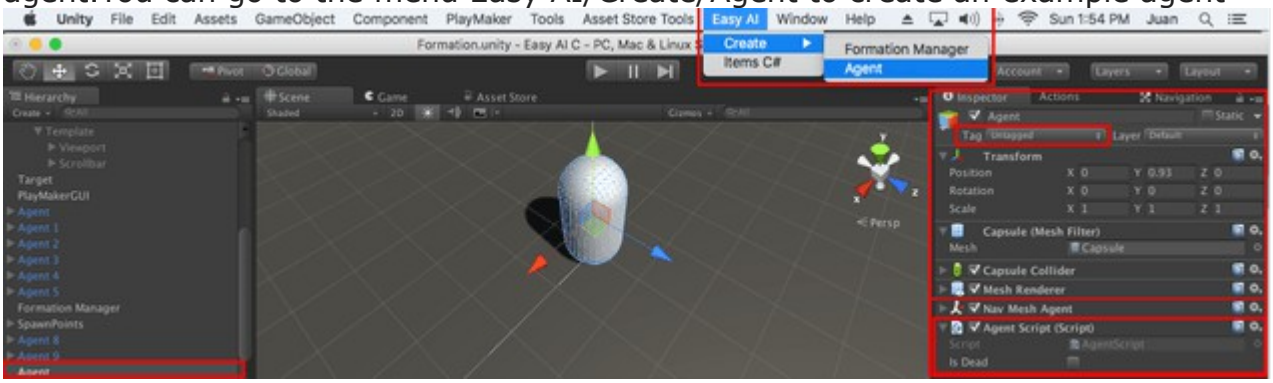
To use a formation:

1- Create the formation manager (Go to the menu Easy AI/Create/Formation Manager).This will create a gameobject named Formation Manager with the script Formation Manager

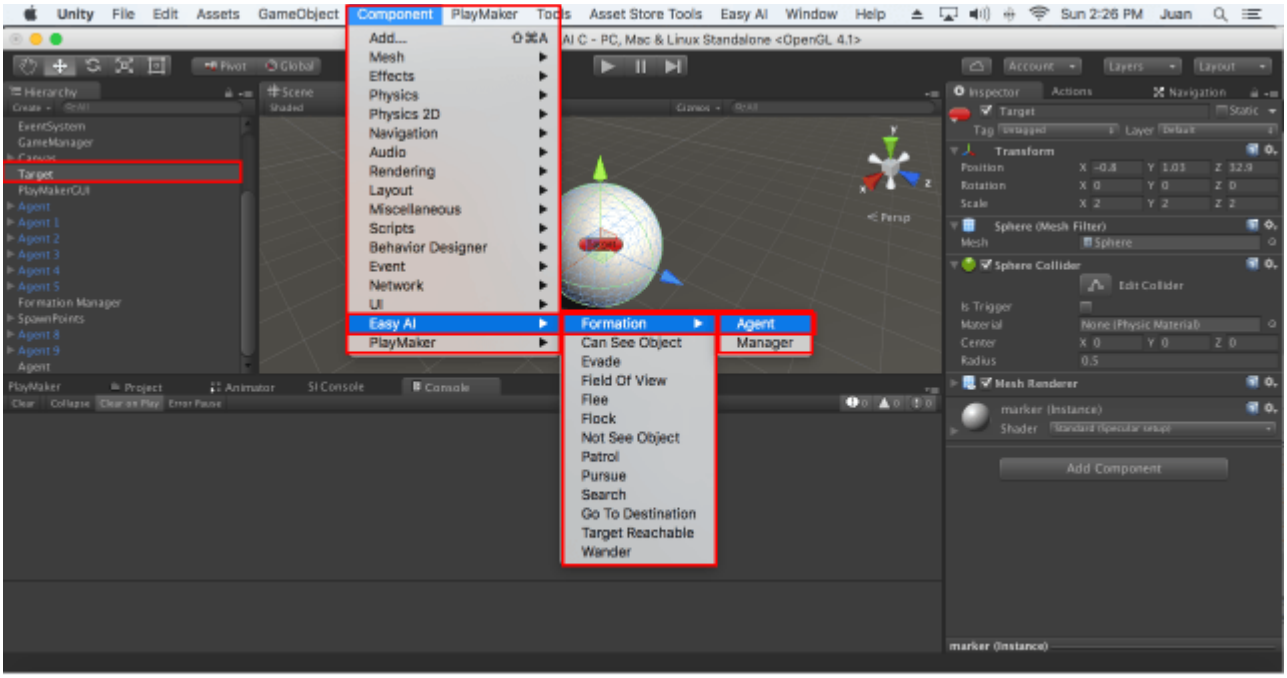


2-Select the gameobject Formation Manager and select the type of formation that you want. For example:

3-Now the next step is to create an agent. There are several ways to create an agent.You can go to the menu Easy AI/Create/Agent to create an example agent



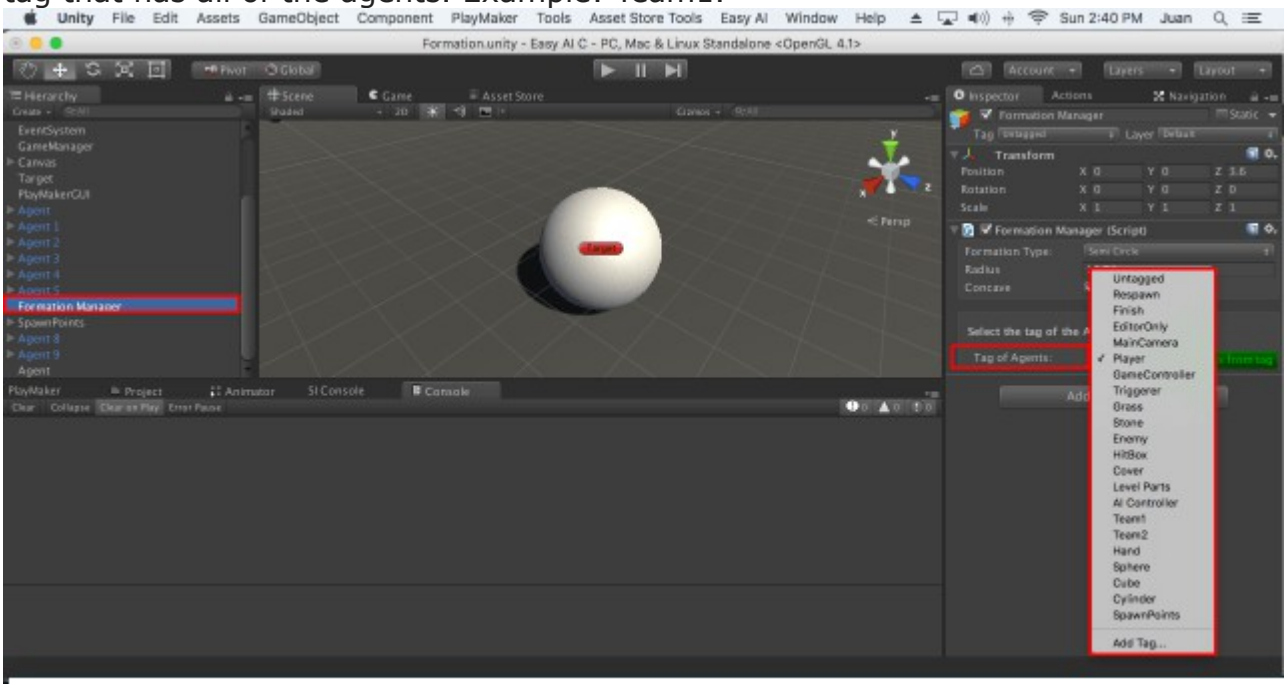
If you don’t want to create an example agent, you only must select your gameobject and go to the menu Component/Easy AI/Formation/Agent



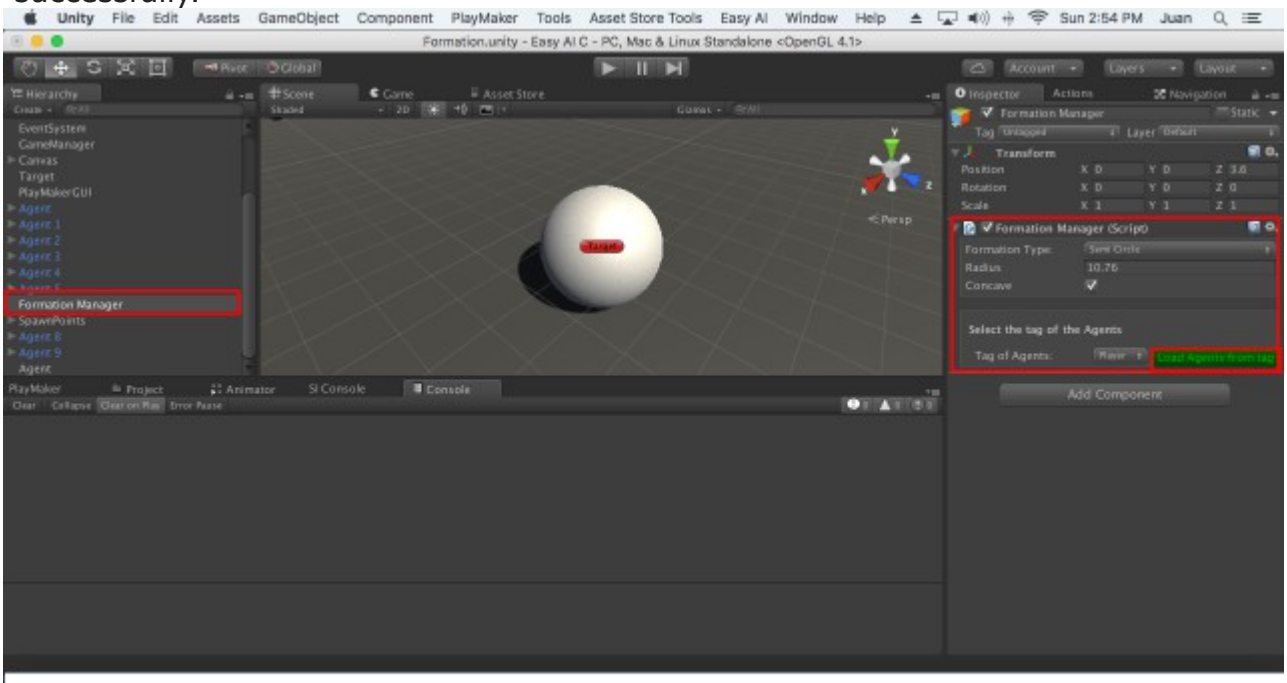
And this will add the components that you gameobject need to be an Agent.

Don’t forget that all the agents must have the same tag. Example: Team1.

4- Select the formation manager gameobject and in the field Tag Of Agents select the tag that has all of the agents. Example: Team1.



5- If you click the Load Agents From Tag button and turn green all agents was loaded successfully.



Variables:

zLookAhead

The distance to look ahead to the destination. The higher the value the better the agent will avoid obstacles and less keep formation

formationSpeed

The agent move speed as the group is forming and advancing

target

The target that the formation must go to.

TRIANGLE

Length

The length of the triangle

CIRCLE & SEMICIRCLE

concave

(ONLY FOR SEMICIRCLE) If you want that the semicircle will be concave.

radius

The radius of the circle or semicircle.

V & WEDGE

_separation

The separation between agents in the formation.

fill

If you want that the agents fill the formation.

TagOfAgents

The tag of the agents that will be formed.

Load Agents From tag

If this button is of green color all of the agents was loaded successfully.

Notification Center Documentation:

Notification Center allows for easy and powerful messaging between objects in Unity. An object can register as an observer and receive notifications of a certain type when they occur. When that notification type is posted elsewhere, all objects that registered as an observer for that notification type will receive a message that it has occurred, along with the associated data.

This system allows for objects to communicate their status and events with one another, without having to directly reference each object that receives the notification. An object can simply post an event when it occurs, and any interested party can register to receive notification when it happens.

Using a notification center will allow for dynamic and flexible coding practices. For example, a pinball game can have a bumper that posts a notification when the ball collides with it. An object that keeps track of score can register to receive a notification when this event occurs, and increment the score accordingly without having to communicate directly with or keep track of the bumper. Another object can play a

sound when it receives notification of a bumper impact. Another object can display a particle effect when it receives notification of a bumper impact. Other features can be added to respond to this event later on without having to modify the existing objects, because they independently choose to receive and act on the notification event.

Notification Example:

```
//Register to receive notifications by calling the AddObserver method:
NotificationCenter.DefaultCenter.AddObserver (this, "AgentInPosition");

//Remove an observer with the RemoveObserver() method:
NotificationCenter.DefaultCenter.RemoveObserver (this, "AgentInPosition");

//Post a notification by calling the PostNotification method:
// Post Notification with data:
Hashtable _hashtable = new Hashtable();
_hashtable.Add("test1", "Hello!");
_hashtable.Add("test2", 123);
NotificationCenter.DefaultCenter.PostNotification(this, "AgentInPosition", _hashtable);
// Post Notification without data:
NotificationCenter.DefaultCenter.PostNotification(this, "AgentInPosition");
```

To receive notification, simply implement a method with the notification name:

```
 void AgentInPosition (NotificationCenter.Notification note)
   {
      // reuse data if it exists
      if (note.data != null) {
         Debug.Log ("Sender=" + note.sender.name);
         Debug.Log ("Data:");
         foreach (DictionaryEntry entry in note.data) {
            Debug.Log (" " + entry.Key + "=" + entry.Value);
         }
      } else {
         Debug.Log ("data is null.");
      }
```

```
}
```

If you need any help with the implementation or if you think we should add some other new features, please don't hesitate to contact me

oliver@axlplay.com

https://axlplay.com