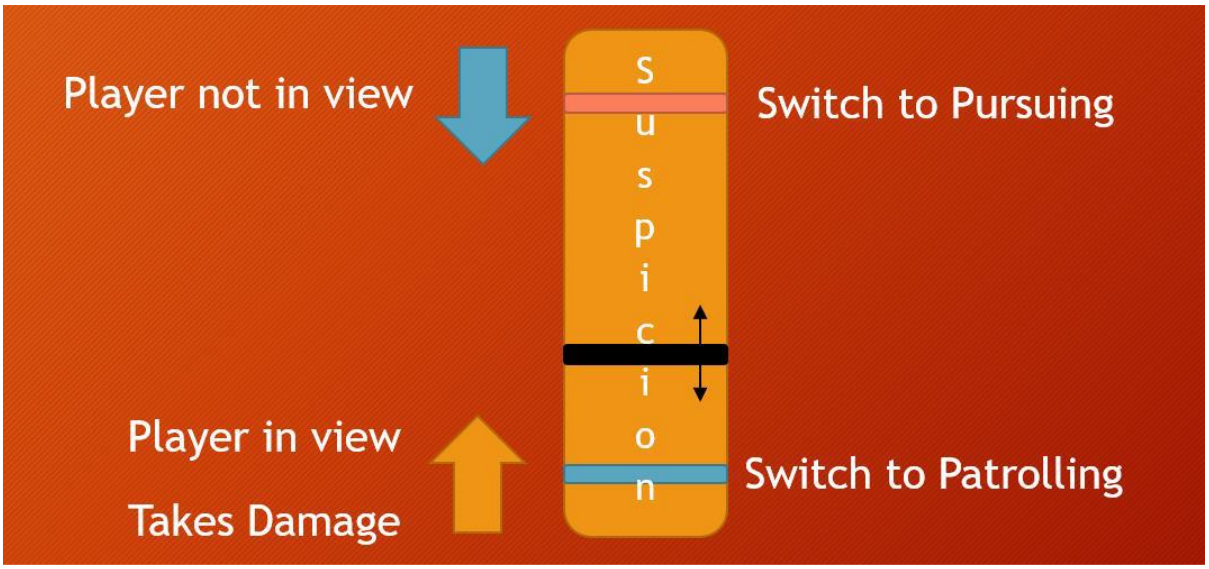
Sprint 01

Assignment 03 : AI Part II

Part I: Guard Suspicion

Step 01: Setup

- What we want to achieve in this assignment is to create a dynamic guard suspicion system which switches between states: Patrolling & Pursuing depending when the Guard sees the Player & when not.



- So first need to set up a "*GuardSuspicion*" data structure that is available in the Guard Script.
- For that we would need to define some structures in the "Guards" script.

Step 02: Script & Workflow

- The Scripts workflow for **Guard Suspicion** is like this:
  - Guards > SuspicionData (*class*) > GuardSuspicion > GuardController
    - Where, SuspicionData is just another Class in "Guards" script itself.

❖ **SuspicionData (inside Guards).**

Note: *[System.Serializable] is used because we don't Inherit from any class or structure (like Monobehviour, etc.). So, in order to make that class appear in the "Inspector" tab, use the above.*

```
C# Guards.cs M ×
Assets > Scripts > Guards > C# Guards.cs > ...
66
67  [System.Serializable]
    3 references
68  public class SuspicionData
69  {
        0 references
70      public float currentSuspicion;
71
72      [Range(0.0f, 1.0f)]
        1 reference
73      public float patrollingThresold = 0.1f; //Blue Bar in the Powerpoint document
74
75      [Range(0.0f, 1.0f)]
        1 reference
76      public float pursuingThresold = 0.6f;   //Red Bar in the Powerpoint document
77
        1 reference
78      public float suspicionBuildRate = 0.6f;
        1 reference
79      public float suspicionDecayRate = 0.2f;
80
81  }
82
```

- We first create a class called "*SuspicionData*" in the "*Guards*" script.
  - **currentSuspicion**: I don't know what this does. We haven't used it anywhere I believe.

- **patrollingThresold**: It is the thresold bewteen 0 to 1 where the Guard performs Patrolling Action. (Guard will perform Patrolling action until it reaches 0.5f thresold)
- **pursuingThresold**: It is the thresold bewteen 0 to 1 where the Guard performs Pursuing Action. (Guard will start performing Puruing action when Puruing Thresold reaches 0.6f or above)
- **suspicionBuildRate**: It is the thresold that "*Builds Suspicion*" in the Guard when it detects the Player. This value defines the amount of time that the Guard will take to perform the "*Attack*" action after the Player has been detected. The lower the value, the more time the Guard will take to detect the Player. And vice versa.
- **suspicionDecayRate**: It is the thresold that "*Decays Suspicion*" in the Guard when it detects the Player. This value defines the amount of time that the Guard will take to perform the "*Loose Player*" action after the Player has been detected. The lower the value, the more time the Guard will take to loose the Player. And vice versa.

  Note: *We would want to keep the "suspicionDecayRate" as low as possible because after the Guard detects the Player, we want the Guard to take some time to lose track of the Player.*

- Then after creating the "*SuspicionData*" class, we set a reference to it in the "*Guards*" class, for 2 reasons:
  - We would want this class to appear in the the "*Inspector*" property if the Guard. (for that we add *"[System.Serializable]*" above the class)
  - We would also want to access this class in various other scripts.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

25 references
public class Guards : MonoBehaviour     //Assignment - 03
{
    7 references
    public GeneralData generalData;
    2 references
    public WayPoints waypointsList;
    3 references
    public VisionData visionData;
    1 reference
    public SuspicionData suspicionData;

    3 references
    public Action<float, Transform> OnDamageTaken = delegate { };

    1 reference
    public void TakeDamage(float damageTaken, Transform instigator)
```

❖ **GuardSuspicion**

- Then we create a new script called "*GuardSuspicion*" which would be responsible for "*Building*" and "*Decaying*" Suspicion with respective to the thresolds.
- It is also responsible for playing certains "*Animations*" when the Player gets detected or gets lost by the guard (i.e. Patrol & Pursuing animtions) and setting the respective states of Patrolling & Pursuing.
- In this script the "*GuardCurrentState*" Enum is located.
- This sciprt has an additional function which sets the Guard to Pursue Mode when it takes Damage from the Player.

C# GuardPursueBehavior.cs M | C# GuardSuspicion.cs U × | C# Guards.cs M | C# GuardController.cs M

Assets > Scripts > Guards > Detection > C# GuardSuspicion.cs > ...

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

4 references
public class GuardSuspicion     //Part II
{
    10 references
    private float suspicionThresold = 0.0f;
    12 references
    private CurrentGuardState currentState;

    //CurrentGuardState - These is used 2 time bcoz, the 1st one represents "CurrentState" and the 2nd one represents "NextState".
    //Transform - is used to get the Transform of the Object that the Guard just saw or made contact with.
    //("GameObject" is not used here for Optimization purpose)
    6 references
    public Action<CurrentGuardState, CurrentGuardState, Transform> OnSuspicionStateUpdated = delegate { };
    2 references
    private Transform firstObjectInView;
    2 references
    private GuardHealth healthController;
    6 references
    private SuspicionData suspectData;
    1 reference
    private GuardVision vision;

    1 reference
    public GuardSuspicion(GuardVision vision, GuardHealth healthController, SuspicionData suspicionData, Guards guard)
    {
        this.vision = vision;
        this.healthController = healthController;
        this.suspectData = suspicionData;
        vision.OnObjectsInView += BuildSuspicion;
        vision.OnNoObjectsInView += DecaySuspicion;
        currentState = CurrentGuardState.Patrolling;
        guard.OnDamageTaken += (damage, damageSource) => DamageTaken(damageSource, damage);
    }

```

```
    private void DamageTaken(Transform damageSource, float damage)
    {
        if(healthController.IsAlive)
        {
            suspicionThresold = 1.0f;
            OnSuspicionStateUpdated(CurrentGuardState.Pursuing, currentState, damageSource);
            currentState = CurrentGuardState.Pursuing;
        }
    }

    1 reference
    private void BuildSuspicion(List<GameObject> obj)
    {
        firstObjectInView = obj[0].transform;
        suspicionThresold += suspectData.suspicionBuildRate * Time.deltaTime;   //As it says "Build" we Increment.
        suspicionThresold = Mathf.Clamp(suspicionThresold, 0.0f, 1.0f);
        CheckThresold();
    }

    1 reference
    private void DecaySuspicion()
    {
        suspicionThresold -= suspectData.suspicionDecayRate * Time.deltaTime;   //As it says "Decay" we Decrement.
        suspicionThresold = Mathf.Clamp(suspicionThresold, 0.0f, 1.0f);
        CheckThresold();
    }


```

```
2 references
private void CheckThresold()
{
    if(     currentState != CurrentGuardState.Pursuing &&       //currentState != CurrentGuardState.Pursuing - is used to make sure that Guard is already not in that state
            suspicionThresold >= suspectData.pursuingThresold)
        {
                OnSuspicionStateUpdated(CurrentGuardState.Pursuing, currentState, firstObjectInView);
                currentState = CurrentGuardState.Pursuing;
        }
    else if(currentState != CurrentGuardState.Patrolling &&     //Same reason as of above
            suspicionThresold < suspectData.patrollingThresold)
        {
                OnSuspicionStateUpdated(CurrentGuardState.Patrolling, currentState, null);
                currentState = CurrentGuardState.Patrolling;
        }
    else if    (currentState != CurrentGuardState.Annoyed &&
            suspicionThresold > suspectData.lookingThresold)
            {
                OnSuspicionStateUpdated(CurrentGuardState.Annoyed, currentState, null);
                currentState = CurrentGuardState.Annoyed;
            }
    }
}

27 references
public enum CurrentGuardState
{
    7 references
    Patrolling,
    8 references
    Pursuing,
    4 references
    Annoyed
}
```

❖ GuardController

- Then we do some changes in the "*GuardController*" scirpt.
- Basically we set off some functions and add some extra ones.
- We set off the "*SetNewState*" function as we do not need this function to tell the Guard perform certain states or actions.
- Instead of that we create a new function called "*UpdateSuspicionState*" which will be responsible to set our guard to various States.
- Also, note that we delete or diable the "*ObjectsInVision*" & "*NoObjectsInVision*" functions too, as we don't need them to excuslively set some action states like Patrolling or Pursuing, as they are already been set in the "*UpdateSuspicionState*" and we don't want them to perform any state at the same time.

Functions we turn off

GuardPursueBehavior.cs M | GuardController.cs M | GuardSuspicion.cs U | Guards.cs M | GuardVisio

Assets > Scripts > Guards > GuardController.cs > GuardController

```
95
96      /*private void ObjectsInVision(List<GameObject> objectsInView)
97      {
98          SetNewState(CurrentGuardState.Pursuing);    //Sets the "newState" to Pursuing.
99      }
100
101     private void NoObjectsInVision()
102     {
103         SetNewState(CurrentGuardState.Patrolling);  //Sets the "newState" to Patrolling.
104     }
105
106     private void SetNewState(CurrentGuardState newState)
107     {
108         if(currentState == newState)    //IMP: This line loops the Patrolling action.
109         {
110             return;
111         }
112         switch(newState)
113         {
114             case CurrentGuardState.Patrolling:  //Called from the above line.
115
116                 //The below line is IMP bcozin here we set the currentState to some state,
117                 //so as it can execute-break-execute seamlessly
118                 currentState = CurrentGuardState.Patrolling;
119                 SetPatrolBehavior();
120             break;
121
122             case CurrentGuardState.Pursuing:
123                 //The below Line...  same as the above.
124                 currentState = CurrentGuardState.Pursuing;
125                 SetPursueBehavior();
126             break;
127
128         }
129     }*/
130
```

```
31
32          //vision.OnObjectsInView += ObjectsInVision;
33          //vision.OnNoObjectsInView += NoObjectsInVision;
34
```

Function we introduce

```
13      private GuardSuspicion guardSuspicion;  //Part II
        4 references
```

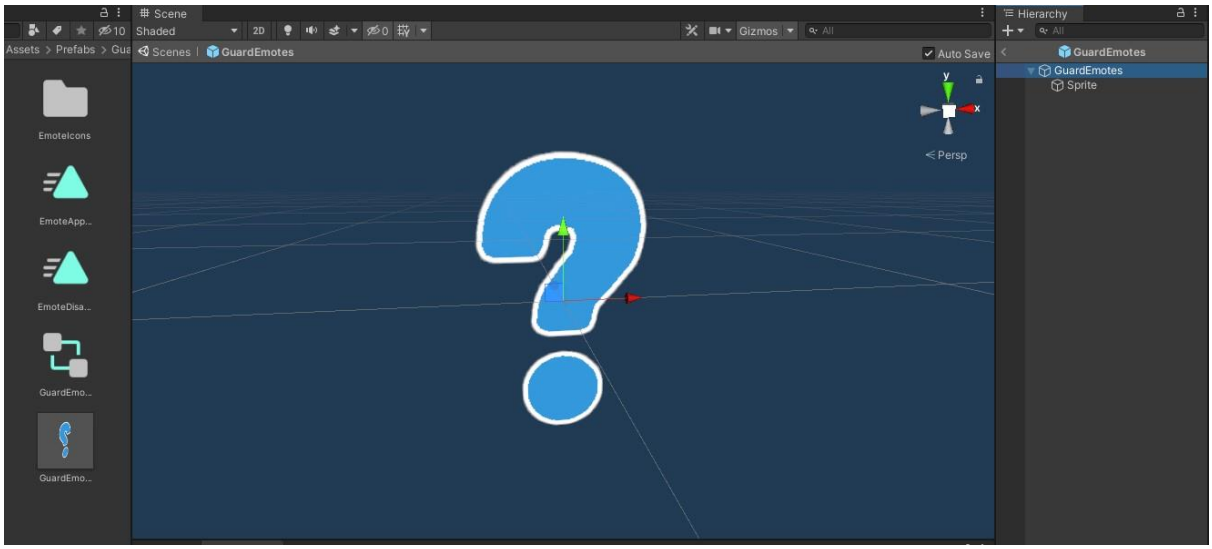
```
42
43          guardSuspicion = new GuardSuspicion(vision, guardHealth,guards.suspicionData, guard);  //Part II
44          guardSuspicion.OnSuspicionStateUpdated += UpdateSuspicionState; //Part II
45
```

```
        1 reference
50      private void UpdateSuspicionState(CurrentGuardState newState, CurrentGuardState oldState, Transform instigator)   //Part II
51      {
52          currentBehavior.End();
53          if(guardHealth.IsAlive)
54          {
55              switch(newState)
56              {
57                  case CurrentGuardState.Patrolling:
58                      currentBehavior = new GuardPatrolBehavior(guards, guards.generalData.patrolMoveSpeed, guardAnimator);
59                  break;
60
61                  case CurrentGuardState.Pursuing:
62                      currentBehavior = new GuardPursueBehavior(guards, player.ObjectData.transform, guards.generalData.pursuitMoveSpeed);
63                  break;
64
65                  case CurrentGuardState.Annoyed:
66                      //currentBehavior = new GuardAnnoyedBehvior(guards, meshAgent, animator);
67                      SetLookAroundBehavior();
68                      Debug.Log("Annoying! in Switch Case");
69
70                  break;
71
72                  default:
73                      Debug.Log("Missing case in GuardController UpdateSuspicion");
74                  break;
75              }
76
77              currentBehavior.Start();
78          }
79      }
80
```

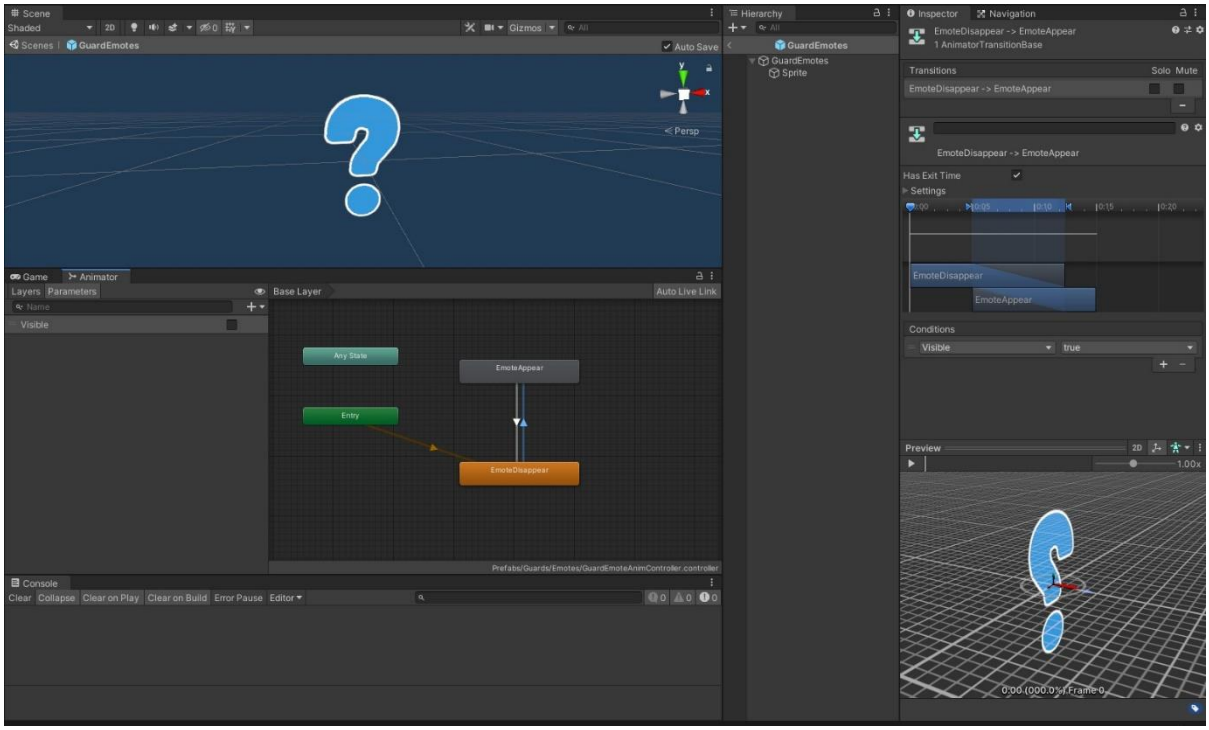
Part II: Emotes & Animations

Step 01: Setup

- We open the “GuardEmotes” prefab from the library or if not create a new prefab with any emote icon.



- This “*GuardEmotes*” prefab has an “*Animation Controller*” already set up.
- It is nothing but just witch between two states: Appear & Disappear.
- These two states are connected with two “*Transitions*” which simply pass a Boolean called “*Visible*” which is set “*true*” when it appears, and “*false*” when it disappears.



- Then, we create a script called "*GuardEmote*" and attach it to the "*GuardEmotes*" prefab.

Step 02: Script & Workflow

- The Scripts workflow for **Guard Emote** is like this:
    - GuardEmote > SpriteBillboarder > EmoteController > GuardController

- GuardEmote is a "*MonoBehviour*" script, as it is attached to the GameObject "***GuardEmotes***".

- The Scripts workflow for **Guard Animator** is like this:
    - GuardAnimator > PatrolRotate > GuardPatrolBehavior > GuardController

❖ GuardEmote

- This script is responsible for "*Showing*" & "*Hiding*" particular Emotes.
- It has different functon for each emote, and a common "*DisplayEmote()*" function that displays the respective emote.
- Same with the "*HideEmote()*" function.
- Also, it has an "*EmoteDisplayTime*" value that decides for how long is the emote to be displayed.

```
C# GuardPursueBehavior.cs M    C# GuardEmote.cs U ×    C# GuardControl
Assets > Scripts > Guards > Emotes > C# GuardEmote.cs > ...
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

2 references
public class GuardEmote : MonoBehaviour //Part II
{
    1 reference
    public Sprite alertedEmote;
    1 reference
    public Sprite lostPlayerEmote;
    1 reference
    public Sprite suspiciousEmote;

    1 reference
    public float emoteDisplayTime = 3.0f;
    4 references
    private float currentEmoteDisplayTime;
    3 references
    private Animator animator;
    3 references
    public SpriteRenderer spriteRenderer;

    0 references
    private void OnEnable()
    {
        animator = GetComponent<Animator>();
    }

```

```
0 references
void Update()
{
    //Track time emote shown for
    currentEmoteDisplayTime -= Time.deltaTime;
    if(currentEmoteDisplayTime <= 0)
    {
        HideEmote();
    }
}

```

```
    public void ShowAlertedEmote()
    {
        spriteRenderer.sprite = alertedEmote;
        DisplayEmote();
    }

    1 reference
    public void ShowPlayerLostEmote()
    {
        spriteRenderer.sprite = lostPlayerEmote;
        DisplayEmote();
    }

    0 references
    public void ShowSuspiciousEmote()
    {
        spriteRenderer.sprite = suspiciousEmote;
        DisplayEmote();
    }

    3 references
    public void DisplayEmote()
    {
        animator.SetBool("Visible", true);
        currentEmoteDisplayTime = emoteDisplayTime;
    }

    1 reference
    public void HideEmote()
    {
        animator.SetBool("Visible", false);
        currentEmoteDisplayTime = 0;
    }
}

```

❖ EmoteController

- Next, we create a "*EmoteController*" that displays the emote in respective Guard States.
- That is it displays the "*Alerted*" emote when it detects the Player, and "*QuestionMark*" emote when it looses the Player.
- This is achieved by implementing a function called "*EmoteUpdate()*" which displays emotes using the Swtich cases which set repsective emotes according to the states.
- It also does an additional check, to see it the "*Current State*" is not similar as the "*Old State*" in order to avoid unwanted glitches.

```
C# GuardPursueBehavior.cs M   C# EmoteController.cs U ×   C# GuardEmote.cs U   C# GuardController.cs M   C# GuardAnimator.cs U   C# GuardSu
Assets > Scripts > Guards > Emotes > C# EmoteController.cs > EmoteController
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

2 references
public class EmoteController    //Part II
{
    1 reference
    GuardSuspicion guarSuspicion;
    3 references
    GuardEmote emotes;
    //Guards[] guards;
    //Player player;

    1 reference
    public EmoteController(GuardSuspicion guardSuspicion, GameObject emotesObject)
    {
        this.guarSuspicion = guardSuspicion;
        guardSuspicion.OnSuspicionStateUpdated += (newState, oldState, intigator) => EmoteUpdated(newState, oldState);
        this.emotes = emotesObject.GetComponent<GuardEmote>();

        //for(int i = 0; i < guards.Length; i++)
            //emotesObject.transform.LookAt(player.Controller.Transform.forward, Vector3.up);
    }
```

```
    1 reference
    private void EmoteUpdated(CurrentGuardState newState, CurrentGuardState oldState)
    {
        switch(newState)
        {
            case CurrentGuardState.Patrolling:
            //Check whether if we are not in the Old State or our previous state was not the same as New State(i.e., already not in Patrolling State)
            if(oldState != CurrentGuardState.Patrolling)
            {
                emotes.ShowPlayerLostEmote();
            }
            break;

            case CurrentGuardState.Pursuing:
            //Same as above
            if(oldState != CurrentGuardState.Pursuing)
            {
                emotes.ShowAlertedEmote();
            }
            break;

            /*case CurrentGuardState.Annoyed:
            for(int i = 0; i < guards.Length; i++)
            {
                guards[i].OnDamageTaken += (damageAmount, instigator) => emotes.ShowSuspiciousEmote();
            }
            break;*/

            default:
                Debug.LogError("Missing Case in EmoteController > EmoteUpdated");
            break;
        }
    }
}
```

❖ SpriteBillboarder

- Then, after that we create a "*Spritebillboarder*" script that forces the "GuardEmote" GameObject always towards the Player, as it is a 2D Image.
- This scirpt is dynamic, i.e., we can set any Axis Plane that we want the prefab to look towards.

```
C# GuardPursueBehavior.cs M   C# SpriteBillboarder.cs U ×   C# EmoteController.cs U   C# GuardEmote.cs U   C#
Assets > Scripts > Guards > Emotes > C# SpriteBillboarder.cs > ...
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

0 references
public class SpriteBillboarder : MonoBehaviour
{
    1 reference
    public PivotAxis pivotAxis = PivotAxis.XY;


    [Tooltip("The target the object orients towards. Defaults to camera if none is set")]
    4 references
    public Transform targetTransform;

    /// <summary>
    /// Keeps the object facing the camera.
    /// </summary>
    0 references
    private void Update()
    {
        // If we've switched cameras and destroyed our old one, find our new main camera
        if (targetTransform == null)
        {
            if (Camera.main.transform != null)
            {
                targetTransform = Camera.main.transform;
            }
            else
            {
                return;
            }
        }

```

```
31          // Get a Vector that points from the target to the main camera.
32          Vector3 directionToTarget = targetTransform.position - transform.position;
33
34          bool useCameraAsUpVector = true;
35
36          // Adjust for the pivot axis.
37          switch (pivotAxis)
38          {
39              case PivotAxis.X:
40                  directionToTarget.x = 0.0f;
41                  useCameraAsUpVector = false;
42                  break;
43
44              case PivotAxis.Y:
45                  directionToTarget.y = 0.0f;
46                  useCameraAsUpVector = false;
47                  break;
48
49              case PivotAxis.Z:
50                  directionToTarget.x = 0.0f;
51                  directionToTarget.y = 0.0f;
52                  break;
53
54              case PivotAxis.XY:
55                  useCameraAsUpVector = false;
56                  break;
57
58              case PivotAxis.XZ:
59                  directionToTarget.x = 0.0f;
60                  break;
61
62              case PivotAxis.YZ:
63                  directionToTarget.y = 0.0f;
64                  break;
65
66              case PivotAxis.Free:
67              default:
68                  // No changes needed.
69                  break;
```

```
        // If we are right next to the camera the rotation is undefined.
        if (directionToTarget.sqrMagnitude < 0.001f)
        {
            return;
        }

        // Calculate and apply the rotation required to reorient the object
        if (useCameraAsUpVector)
        {
            transform.rotation = Quaternion.LookRotation(-directionToTarget, targetTransform.transform.up);
        }
        else
        {
            transform.rotation = Quaternion.LookRotation(-directionToTarget);
        }
    }
}

9 references
public enum PivotAxis
{
    // Most common options
    2 references
    XY,
    1 reference
    Y,
    // Rotate about an individual axis.
    1 reference
    X,
    1 reference
    Z,
    // Rotate about a pair of axes.
    1 reference
    XZ,
    1 reference
    YZ,
    // Rotate about all axes.
    1 reference
    Free
}
```

❖ GuardController

- Then, finally after setting up all these things we create an insctance of the "*EmoteController*" script in the "*GuardController*" script.
- We do this in the "*GuardController*" script only because it is the one that controls all the Guard Behaviors and is the one that fires all events and scripts.

```
GuardPursueBehavior.cs M   GuardController.cs M ×   EmoteController.cs U   GuardEmote.cs U   GuardAnimator.cs U   GuardSuspicion.c
Assets > Scripts > Guards > GuardController.cs > GuardController

    1 reference
19  private EmoteController guardEmoteController;   //Part II
20
    1 reference
21  public GuardController(Guards guard, Player player)  //Bcoz this will be called in GuardManager.
22  {
23      this.guards = guard;
24      this.player = player;
25      meshAgent = guard.GetComponent<NavMeshAgent>();
26      animator = guards.GetComponent<Animator>();
27
28      guardAnimator = new GuardAnimator(meshAgent, animator); //Part II
29
30      vision = new GuardVision(guard, guard.visionData, player);
31
32      //vision.OnObjectsInView += ObjectsInVision;
33      //vision.OnNoObjectsInView += NoObjectsInVision;
34
35      SetPatrolBehavior();
36
37      guardHealth = new GuardHealth(guards.generalData.maxHealth);
38      guardHealth.OnDamageTaken += GuardDamaged;
39      guardHealth.OnKilled += GuardKilled;
40
41      guard.OnDamageTaken += (damageAmount,damageSource) => guardHealth.TakeDamage(damageAmount);
42
43      guardSuspicion = new GuardSuspicion(vision, guardHealth,guards.suspicionData, guard);  //Part II
44      guardSuspicion.OnSuspicionStateUpdated += UpdateSuspicionState; //Part II
45
46      guardEmoteController = new EmoteController(guardSuspicion, guard.transform.Find("GuardEmotes").gameObject); //Part II
47
48  }
49
```

❖ GuardAnimator

- This script is responsible for the Guard "*Movement*" and "*Rotation*" animations.
- This script plays the movement animation when the character moves and rotate animation when the guard rotates.
- We access the Animation Clips by using their "*Names*" or "*Parameters*" defined.
- But we also use the "*Hash*" value of the animation clip.
- We have done this to get the "*Locomotion BlendTree*" blend tree form the Guard's Animation Controller.

```
C# GuardPursueBehavior.cs M   C# GuardAnimator.cs U ×   C# GuardController.cs M   C# EmoteController.cs U   C# GuardEmote.cs U   C# GuardSuspicion.cs U   C# Guards.cs M
Assets > Scripts > Guards > C# GuardAnimator.cs > ...
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;
    //IMP: Tick the "Apply Root Motion" checkbox.
    //IMP: HashID - Every Unity Component has an unique "HashID" or an "InstanceID" which Unity uses as a reference. (particularly to get Components)
6 references
public class GuardAnimator  //Part II
{
    4 references
    private NavMeshAgent meshAgent;
    8 references
    private Animator animator;
    4 references
    private float turnAmount;
    4 references
    private float forwardAmount;
    3 references
    private bool UpdatedTurnThisFrame;  //Default Value: False
    3 references
    private bool UpdatedMoveThisFrame;  //Default Value: False
    3 references
    private bool LocomotionStateExists;
    3 references
    private string LocomotionStateName = "Locomotion BlendTree";
    2 references
    public bool isInLocomotion;


    1 reference
    public GuardAnimator(NavMeshAgent meshAgent, Animator animator)
    {
        this.meshAgent = meshAgent;
        this.animator = animator;

        //HasState - Retuns has State: True or False. (same as Boolean)
        //LayerIndex - The Index in the Animator > Layer. In our case it is "0" as we only have only one layer "BaseLayer".
        //StateID - Every State has an ID. (created by Untiy)
        //Animator.StringToHash - Converts a "String" to "Hash". We use this bcoz we use the name of Blend Tree "Locomotion BlendTree".
        //So we need to convert it into a Hash value, which can be used as a "StateID".
        LocomotionStateExists = animator.HasState(0, Animator.StringToHash(LocomotionStateName));

        if(!LocomotionStateExists)
        {
            //$ - An alternate to "True for + {animator.gameObject.name} + because we can't find a state named + {LocomotionStateName}");
            //Instead of using "+" multiple times, we can simply use "$" in the begining and Angular Brackets "{ }" to define a component.
            Debug.LogWarning($"isInLocomotion will always be True for {animator.gameObject.name} because we can't find a state named {LocomotionStateName}");
        }
    }

    // Update is called once per frame
    1 reference
    public void Update()
    {
        DecayMovementateToZero();
        DecayTurnRateToZero();

        if(!meshAgent.hasPath)
        {
            return;
        }
        else
        {
            //desiredVelocity - Tells us where the NavMeshAgent(Guard) wants to move immediately.
            Move(meshAgent.desiredVelocity);

            if(LocomotionStateExists)
            {
                isInLocomotion = animator.GetCurrentAnimatorStateInfo(0).IsName(LocomotionStateName);
            }
        }
    }
```

- In the "*Update()*" function, we have set up two functions, that are responsible for slowing down the guard's movement speed as when the guard tries to make a "*Rotation*" as well as the guard's "*Rotation Turn Rate*".
- The line:

```
LocomotionStateExists = animator.HasState(0, Animator.StringToHash(LocomotionS
tateName));
```

Checks if the "*Animator*" has any Animation State or not.

- It (**HasState**) simply is a boolean that returns "*true*" if an "*Animation State*" found, and "*false*" if it is not found.

```
private void Move(Vector3 move)
{
    if(move.magnitude > 1f)
    {
        //Bcoz we set up in Blend Tree values between 0 to 1 and if the Guard is running
        //at a speed greater than 1 (suppose 4), then we would want to bring that speed
        //amount back to 1.
        move.Normalize();   //Always return value "1".
    }

    //TransformDirection - Transforms a direction from 'local space' to 'world space'.
    //InverseTransformDirection - Transforms a direction from 'world space' to 'local space'.
    //We need to use this function as we want our Guard to move around the Local Axis.
    move = meshAgent.transform.InverseTransformDirection(move);

    //To keep the Guard enatcted to the Plane(Floor GameObject in the scene).
    move = Vector3.ProjectOnPlane(move, Vector3.up);

    //Returns radian value between X & Z.
    //Rotates the Guard.
    turnAmount = Mathf.Atan2(move.x, move.z);

    //Simply meshAgent's Forward Direction.
    forwardAmount = move.z;

    turnAmount = Mathf.Clamp(turnAmount, -1.0f, 1.0f);  //We use -1 & 1 as our Guard can rotate Left or Right i.e., in Opposite Directions.
    forwardAmount = Mathf.Clamp01(forwardAmount); //bcoz Guard's Animator Blend Tree has values between 0  to 1.

    //Damping - Slwoly moving the values from Initila value to Final value. Same as Linear Interpolation.
    animator.SetFloat("TurnRate", turnAmount, 0.15f, Time.deltaTime);
    animator.SetFloat("MoveSpeed", forwardAmount, 0.35f, Time.deltaTime);

    UpdatedMoveThisFrame = true;
}
```

- **InverseTransfromDirection**: simplys converts guard's "*World Space*" forward into "*Local Space*" forward.
- We do this because we want play the animation in the Local Space forward. As Local Space is the direction that updtaes respectively with the change in a GameObject.
- **ProjectOnPlane**: simply checks wether the Guard is on the plane or not. (not floationg above)
- **Atan2**: is a Maths function that returns radian values between two axes. It is usually used for Rotation.

```
1 reference
public void TurnOnSpot(float turnRate)  //From PatrolRotate > TurnOnSpot(rotateScaleSpeed * rotateSpeed)
{
    //Slowly lerp from 0.15f towards turnRate.
    animator.SetFloat("TurnRate", turnRate, 0.15f, Time.deltaTime);

    //Slowly lerp from 0.15f towards 0.0f.
    animator.SetFloat("MoveSpeed", 0.0f, 0.5f, Time.deltaTime);

    //We set it "Ture" bcoz we want to Rotate the Guard only on this frame, and not other.
    //Then after this function it goes to "DecayTurnRateToZero()" function, which sets it value again to "False" and so the animation is performed only once.
    UpdatedTurnThisFrame = true;
}
```

- This function is responsible to make the Guard "Rotate".

- It is been called in the "*PatrolRotate*" script form the "*GuardBehavior*", and passes a parameter.

```
1 reference
private void DecayTurnRateToZero()
{
    //If we have set Turn Rate this frame, switch the flag to False so we don't adjust Turn Rate
    //Simply means: If(UpdatedTurnThisFrame == true)
    if(UpdatedTurnThisFrame)    //Default Value: false
    {
        UpdatedTurnThisFrame = false;
        return;
    }

    animator?.SetFloat("TurnRate", 0.0f, 0.15f, Time.deltaTime);
}

1 reference
private void DecayMovementateToZero()
{
    //If our Pathing's set to a Move Speed  this frame, switch the flag to False so we don't adjust it
    //Simply means: If(UpdatedMoveThisFrame == true)
    if(UpdatedMoveThisFrame)    //Default Value: false (Changes after the First Frame Update: True)
    {
        UpdatedMoveThisFrame = false;
        return;
    }

    animator?.SetFloat("MoveSpeed", 0.0f, 0.15f, Time.deltaTime);
}

}
```

- The "*DecayTurnRate()"* simplys decays the speed of rotation.
- The "*DecayMovementToZero()*" simply decays the guard's movement speed to zero as when the Rotation function is called.

❖ PatrolRotate

- This script is responsible to make the Guard Rotate on the Spot.
- It uses the "*TurnOnSpot*" function from the "*GuardAnimator*" script to rotate the guard.

C# GuardPursueBehavior.cs M | C# PatrolRotate.cs M × | C# GuardAnimator.cs U | C# GuardController.cs M | C# EmoteController.cs U | C# GuardEmote.cs U | C#

Assets > Scripts > Guards > Patrolling > C# PatrolRotate.cs > ...

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;

1 reference
public class PatrolRotate : PatrolCommand
{
    7 references
    private NavMeshAgent meshAgent;
    4 references
    private Vector3 rotateGoal;
    4 references
    private float rotateSpeed;
    1 reference
    private Animator animator;
    2 references
    private GuardAnimator guardAnimator;
    0 references
    private Guards guards;

    1 reference
    public PatrolRotate(NavMeshAgent meshAgent, Vector3 rotateGoal, float rotateSpeed, Animator animator, GuardAnimator guardAnimator)
    {
        this.meshAgent = meshAgent;
        this.rotateGoal = Quaternion.Euler(rotateGoal) * Vector3.forward;   //forward - Always gives a magnitude value of 1.
        this.rotateSpeed = rotateSpeed;
        this.animator = animator;
        this.guardAnimator = guardAnimator;
    }
    4 references
```

```
    public override void Start()
    {
        //Debug.Log("Rotate Started!");
        //meshAgent.speed = 0;
        meshAgent.updateRotation = false;
    }

    1 reference
    public override void Update()
    {
        if(rotateSpeed == 0)
        {
            Debug.Log("Did not rotate Properly!");
            CommandComplete();
        }

        float stepAmount= rotateSpeed * Time.deltaTime;
        Vector3 newDirection = Vector3.RotateTowards(meshAgent.transform.forward, rotateGoal, stepAmount, 0.0f);
        meshAgent.transform.forward = newDirection;
        //animator.SetFloat("StrafeSpeed", 1);

        //Returns the Signed Angle in Degrees between From and To.
        var signedAngle = Vector3.SignedAngle(rotateGoal, meshAgent.transform.forward, Vector3.up);

        //We divide it with 45 Degress to get a Scalar Amount. And aslo bcoz the Angles are in Higher values which if used anywhere
        //will give wierd results like Rotating too Fast due to higher values. SO we need to put them down to Single Digits Lower Values
        float rotateScaleSpeed = signedAngle / 45.0f;

        guardAnimator.TurnOnSpot(rotateScaleSpeed * rotateSpeed);

```

```

        //Used to check if the Guard is Perpendiculat to the Waypoint.
        //where, 0 means Perpendicular & 1 or above means Not Perpendicular.
        //So, in this case we check if the Dot Product is 1 or above, then it is not perpendicular and thus, end the Action by calling Complete function.
        if( Vector3.Dot(meshAgent.transform.forward, rotateGoal) > 0.99f)
        {
            CommandComplete();
            return;
        }

        //navMesh.transform.Rotate(0, new Vector3(0, rotateGoal.transform.forward, 0), 0);
        //navMesh.transform.localRotation = Vector3.RotateTowards(navMesh.transform.rotation, rotateGoal.transform.rotation);
    }

    1 reference
    public override void End()
    {
        //Debug.Log("Rotate Ended!");
        //if(guards != null)
            //meshAgent.speed = guards.generalData.patrolMoveSpeed;
        meshAgent.updateRotation = true;
    }
}
```

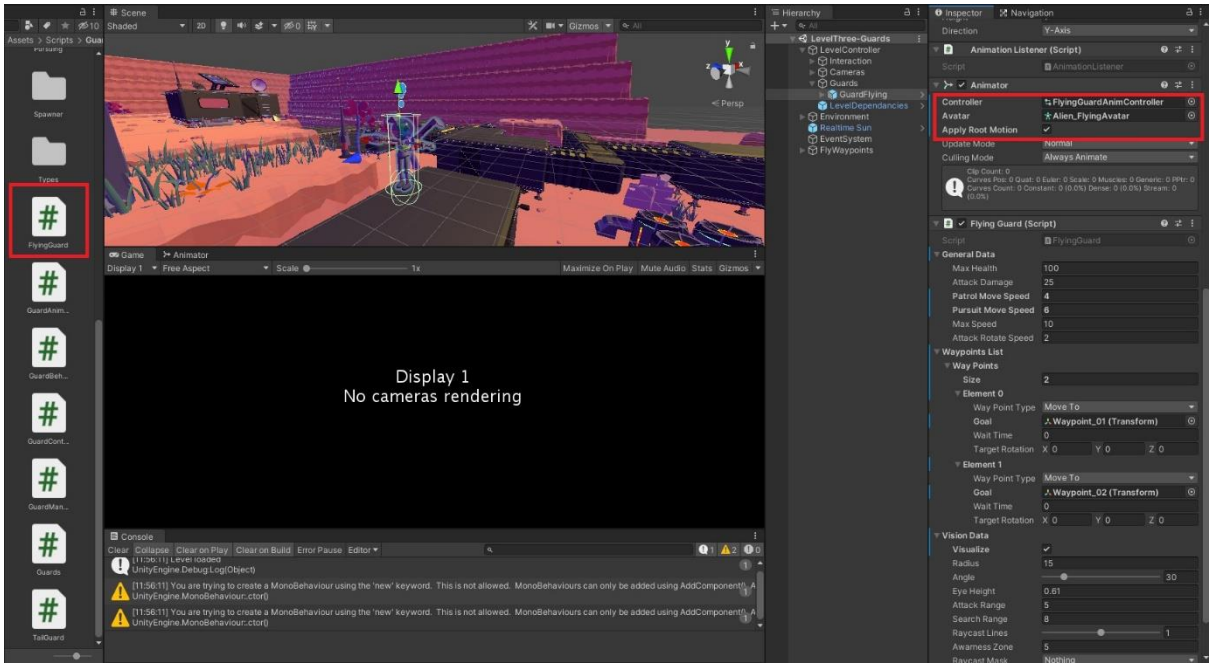
❖ GuardController

- Then, finally we set an instance of "*GuardAnimator*" in the "*GuardController*".

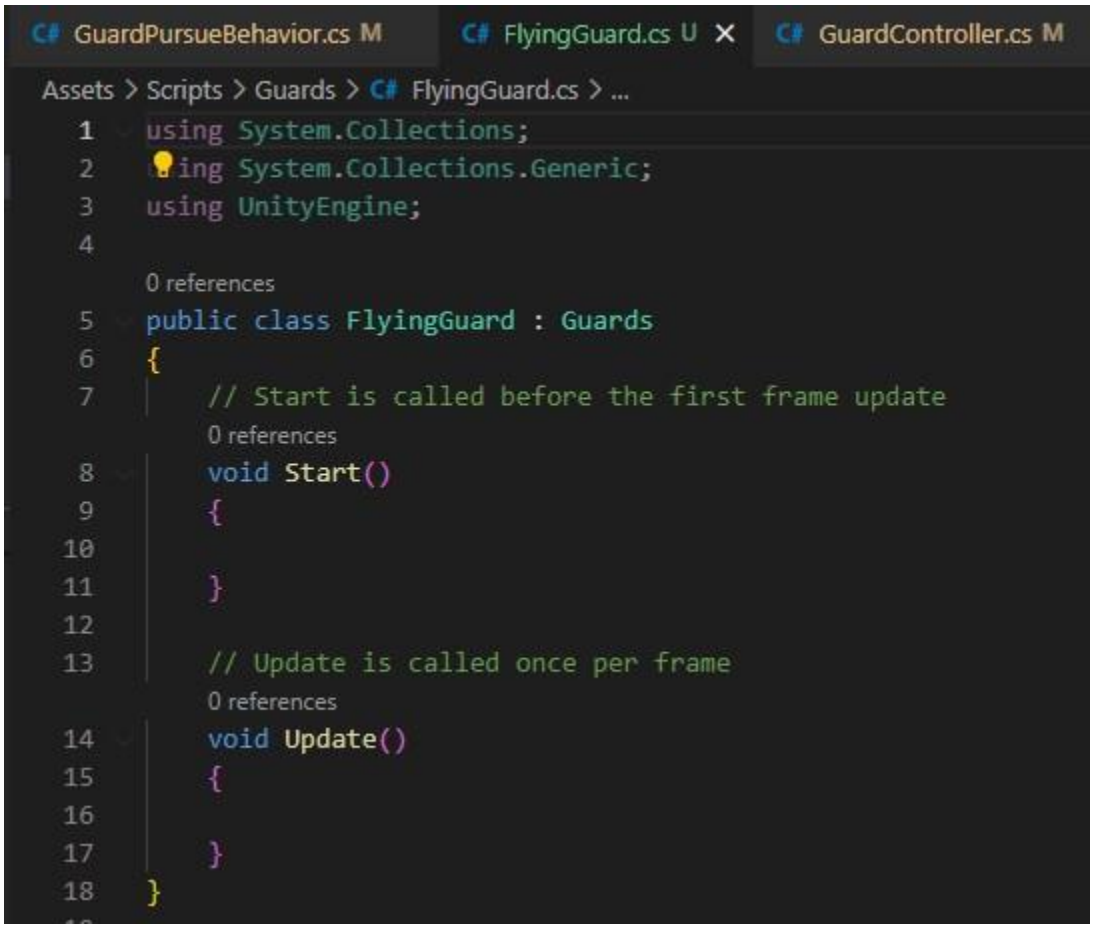
```
C# GuardPursueBehavior.cs M   C# GuardController.cs M ×   C# GuardAnimator.cs U   C# PatrolRotate.cs M   C# EmoteController.cs U   C# GuardEmote.cs
Assets > Scripts > Guards > C# GuardController.cs > GuardController > guardAnimator
    private Animator animator;
    4 references
    private GuardAnimator guardAnimator;    //Part II
    1 reference
    private EmoteController guardEmoteController;   //Part II

    1 reference
    public GuardController(Guards guard, Player player)  //Bcoz this will be called in GuardManager.
    {
        this.guards = guard;
        this.player = player;
        meshAgent = guard.GetComponent<NavMeshAgent>();
        animator = guards.GetComponent<Animator>();

        guardAnimator = new GuardAnimator(meshAgent, animator); //Part II

        vision = new GuardVision(guard, guard.visionData, player);

        //vision.OnObjectsInView += ObjectsInVision;
        //vision.OnNoObjectsInView += NoObjectsInVision;

        SetPatrolBehavior();

        guardHealth = new GuardHealth(guards.generalData.maxHealth);
        guardHealth.OnDamageTaken += GuardDamaged;
        guardHealth.OnKilled += GuardKilled;

        guard.OnDamageTaken += (damageAmount,damageSource) => guardHealth.TakeDamage(damageAmount);

        guardSuspicion = new GuardSuspicion(vision, guardHealth,guards.suspicionData, guard);  //Part II
        guardSuspicion.OnSuspicionStateUpdated += UpdateSuspicionState; //Part II

        guardEmoteController = new EmoteController(guardSuspicion, guard.transform.Find("GuardEmotes").gameObject); //Part II
```

❖ FlyingGuard

- For the "*Flying Guard*", we simply Drag & dorp its prefab into the scene, and create some Waypoints (Empty GameObjects) for it to move around.
- Then we attach its respective "*Animation Controller*" and "*Avatar*" to it.
- Then, we simply create a "*FlyingGuard*" script and simply make it inherit the "*Guards*" script.



- This gives us the whole functionality of the Guard which we just set uped. As we created a Dynamic System for the Guards.
- Same procedure can be followed for other Guards too.



Step 03: What have I learnt

All the above things mentioned.

----------THE END----------