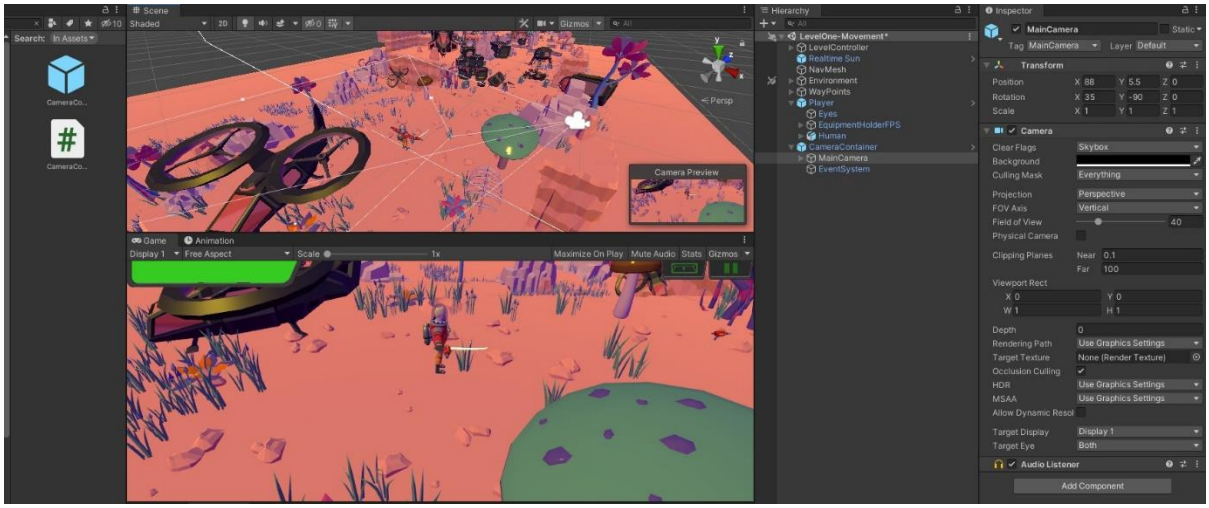
Sprint 02

Assignment 01 : POV

Part I

Step 01: Setup

- We need setup the Third-Person Camera for the Player.
- So first we add our Human 3D Model ("HughMann") under the "Player" Prefab.
- Then we set a proper Camera Angle for the Third-Person View.
- For this, we drag & drop the "*CameraController*" Prefab into the Scene Hierarchy, and adjust the Camera Angle accordingly our requirement.



- Then we Remove the Camera Look Around functionality from the "CameraController" script and Add an Offset to the Camera.

Step 02: Script & Workflow

- The Scripts workflow is like this:
  - LevelController > PlayerController & CameraController.
  - PlayerController > PlayerMovementController & PlayerAnimationController.

❖ **CameraController.**

- We simply delete some lines (in this case I have commented it) of code and Disable the Player's Input Callbacks.
- We also add a Camera Offset for the Camera.
- We achieve it by subtracting the Player's position from Camera.

```
C# CameraController.cs M ×   C# PlayerAnimationController.cs U   C# PlayerController.cs M   C# PlayerMovementController.cs M
Assets > Scripts > Camera > C# CameraController.cs > CameraController > UpdateCameraFollow()
0 references
public GameObject pauseMenu;
1 reference
public CameraController(GameObject cameraObjectPrefab, Transform parent, Transform player)
{
    this.player = player;
    //this.inputCallbacks = inputCallbacks;
    cameraObject = GameObject.Instantiate(cameraObjectPrefab, null);
    cameraObject.name = "Camera";

    eyes = player.Find("Eyes");

    /*inputCallbacks.OnPlayerLookFired += (lookDelta) =>
    {
        UpdateLookDirection(lookDelta);
    };*/

    mainCameraTransform = cameraObject.transform.Find("MainCamera").gameObject;

    cameraOffset = mainCameraTransform.transform.position - player.position;    //S2 - Assignment 01
}

1 reference
private void UpdateCameraFollow()
{
    //mainCameraTransform.transform.position = eyes.position;

    mainCameraTransform.transform.position = player.position + cameraOffset;    //S2 - Assignment 01
}

/*private void UpdateLookDirection(Vector2 lookDelta)
{
    mainCameraTransform.transform.Rotate(Vector3.up, lookDelta.x * sensitivityX, Space.World);
    mainCameraTransform.transform.Rotate(Vector3.left, lookDelta.y * sensitivityY, Space.Self);
}*/
```

❖ PlayerMovementController.

- This is the Main script that controls the Movement of the Player.
- We control the Player with the User inputs and also we want our Player to Rotate the Player a Specific Direction when User inputs certain Keys.

```
C# CameraController.cs M   C# PlayerMovementController.cs M ×   C# PlayerAnimationController.cs U   C# PlayerController.cs M
Assets > Scripts > Player > Movement > C# PlayerMovementController.cs > PlayerMovementController > PlayerMovementController(Tran
using System;
using UnityEngine;
4 references
public class PlayerMovementController
{
    5 references
    private Transform transform;
    //private PlayerInputCallbacks inputCallbacks;
    //private float sensitivityX = 0.05f;
    //private float sensitivityY = 0.05f;

    //S2 - Assignment 01
    6 references
    private Animator animator;
    1 reference
    private float movementSpeed = 5f;
    1 reference
    private float runToIdelSpeed = 0.5f;
    1 reference
    private float idleToRunSpeed = 3f;
    8 references
    private Target target;
    3 references
    private float turnAmount;
    4 references
    private float forwardAmount;
    0 references
    private float turnSpeed;

    1 reference
    public PlayerMovementController(Transform transform, Animator animator) //S2 - Assignment 01
    {
        this.transform = transform;
        this.animator = animator;
        //this.inputCallbacks = inputCallbacks;

        /*inputCallbacks.OnPlayerLookFired += (lookDelta) =>
        {
            UpdateLookDirection(lookDelta);
        };*/
    }

```

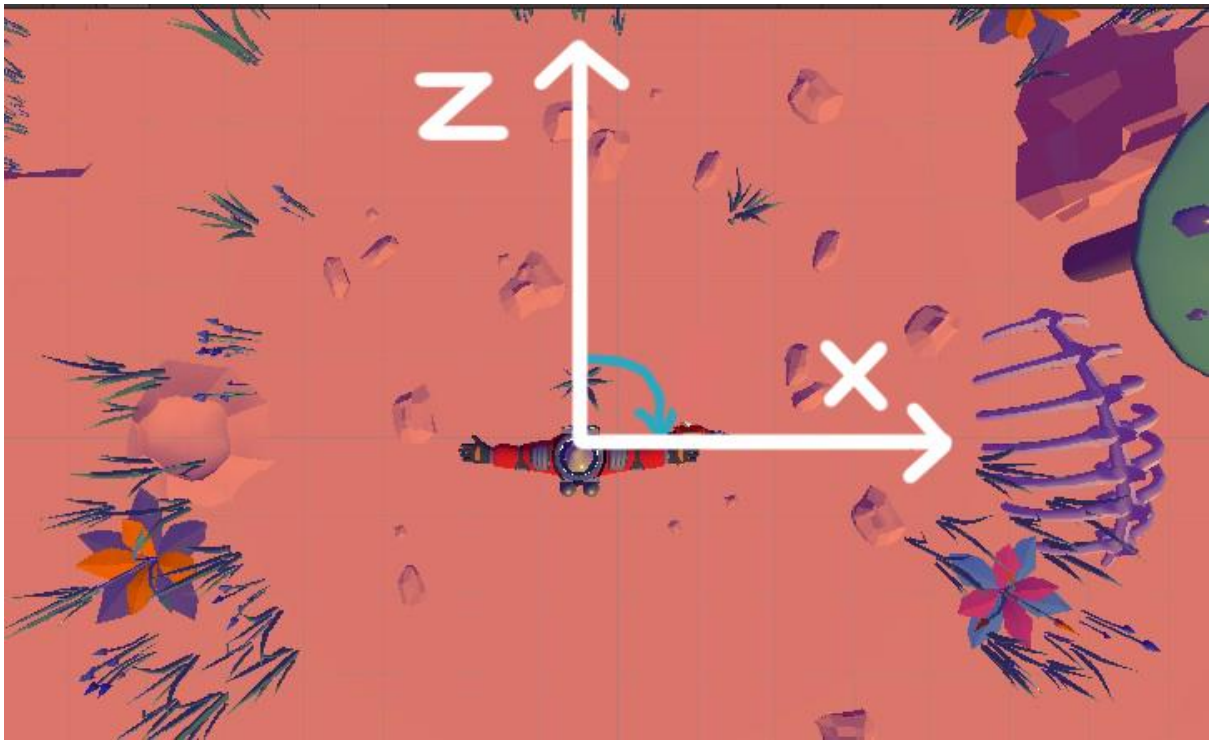
- For that, we create two Classes namely, "*FaceTarget*" & "*MoveTarget*".
- Both the classes implement an Abstract Class called "*Target*".

- FaceTarget Class is used to make the Player look at Specific Direction or Turn/Rotate the Player at Specific Directions.
- MoveTarget Class is used to actually Move the Player, even while the Player is Rotating.

```
4 references
private abstract class Target   //S2 - Assignment 01
{
    3 references
    public Action OnComplete = delegate { };
    3 references
    public Action<float, float, float> OnAnimatorPropertiesUpdated = delegate { };

    1 reference
    public abstract void Start();
    1 reference
    public abstract void Update();
}

2 references
private class FaceTarget : Target   //S2 - Assignment 01
{
    4 references
    private Transform transform;
    2 references
    private Vector3 targetDirection;

    1 reference
    public FaceTarget(Transform transform, Vector3 targetDirection)
    {
        this.transform = transform;
        this.targetDirection = targetDirection;
    }

    1 reference
    public override void Start()
    {

    }

```

```
145         public override void Update()
146         {
147             //Unit Vector = Normalized Magnitude of a Vector.
148             //Normalized Value is always 1.
149             //Magnitude can be any Integer value.
150             //When you Transform a Vector with a Unit Vector (Normalized Vector), then the Unit Vector remains same,
151             //and the other Non-Unit Vector affects in terms or Rotation, Scale or Translation.
152             //But if you Transform a Vector with another Vector, both the Vectors change their values.
153            Vector3 faceDirection = targetDirection;
154            if(faceDirection.magnitude > 1)
155            {
156                faceDirection.Normalize();   //Unit Vector
157            }
158
159             //Transforms a direction from world space to local space.
160             faceDirection = transform.InverseTransformDirection(faceDirection);
161
162             //Projects the Player onto the Plane. (Since in our case, we have Plan Surfaces for each levels, so this works)
163             //We might would have want to use different logic or function when we are having a Non-Planar Terrain.
164             faceDirection = Vector3.ProjectOnPlane(faceDirection, Vector3.zero);
165
166             //Returns the angle in radians whose Tan is y/x.
167             //Z = Player's Forward Direction where Player is Facing.
168             //X = The Left(-ve) and Right(+ve) Direction of the Player, where he moves or turns Left or Right.
169             //Y = Player's Vector Upwards Direction. This is used to check Player on Ground or to Physcially Rotate the Player in the Scene.
170             //In this Case, we use X with Z becoz we want to rotate Left or Right with respective to the Player
171             //always Facing Forward Direction while rotating.
172             float turnAmount = Mathf.Atan2(faceDirection.x, faceDirection.z);
173
174             //Returns the signed angle in degrees between from and to.
175             float angleToDirection = Vector3.SignedAngle(faceDirection, transform.forward, Vector3.up);
176
177             float maxTurn = Math.Sign(angleToDirection) * 360 * Time.deltaTime;
178             float turnSpeed = Math.Min(maxTurn, angleToDirection);
179
180             //This will set below Values in the "SwitchTarget" function.
181             OnAnimatorPropertiesUpdated(turnAmount, 0, turnSpeed);
182
183             if(Vector3.Angle(faceDirection, transform.forward) < 1f)
184             {
185                 OnComplete();
186             }
187         }
188     }
189
```

- We implement the Abstract functions in the FaceTarget class.
- We do most of the logical part in the "*Update()*" function, and call it in the "*Start()*" function.
- Then we implement the Rotation logic as below.
    - We assign the Target as "*FaceTarget*" and convert it into an "*Unit Vector*" by "*Normalizing*" its magnitude value.
    - Then we convert the Target's World Space Position into Local Space Position, because with that, we would easily move its Physical Model.
    - Next, we make the Player Stick to the gorund.
    - Next, we calculate the "*TurnAmount*" i.e., the Angle of Rotation in Radians, by using the "*Atan2*" function which takes the X & Z values as to calculate the Angle formed between the Forward Vector(Z) and its Adjacent Vector(X).



- Then we convert that Angle from Radian into Degrees by using the "*SignedAngle()"* function. (We can also use the "*Rad2Deg*" function by multiplying it with the rest variables.)
- Then I don't know what these two Functions really do and why are they used?

```
float maxTurn = Math.Sign(angleToDirection) * 360 * Time.deltaTime;
float turnSpeed = Math.Min(maxTurn, angleToDirection);
```

- Then we Update these values in the "*OnAnimatorPropertiesUpdated()*".
- And lastly, we pass an "If-Statement" which Completes the function, when the Angle between the two axes of the Player returns than One value.
- Here, we are passing Zero value for the "*ForwardAmount*", as in this function, we do not want to Move the Player, and only make Player Rotate.

- Then we implement the Abstract functions in the MoveTarget class.
- We do the same thing as above in the "*FaceTarget*" class, but only we do it in the "*Start()"* function and we call "*OnComplete*" delegate function in the "*Update()"* function.

- And the only change we do in the Logic is, we Add a Movement Vector (along Z-axis) and store it in "*MovementTarget*".
- Then we use the "*Lerp()*" function to lerp between two Vector values.
- Then we update the respective values by passing them in the "*OnAnimatorPropertiesUpdated()*" function.
- Here, we are passing the "ForwardAmount" value, as in this function, we want to Move the Player.
- And does this "*Lerp()*" function also give the functionality to move Diagonally?

```
C# PlayerMovementController.cs M ×
Assets > Scripts > Player > Movement > C# PlayerMovementController.cs > PlayerMovementController > PlayerMoveme
189
        1 reference
190     private class MoveTarget : Target
191     {
            2 references
192         private Transform transform;
            11 references
193         private Vector3 movementVector;
            1 reference
194         private float stationaryTurnSpeed = 720f;
            0 references
195         private float movingTurnSpeed = 360f;
196
            1 reference
197         public MoveTarget(Transform transform, Vector3 movementVector)
198         {
199             this.transform = transform;
200             this.movementVector = movementVector;
201         }
202
            1 reference
203         public override void Start()
204         {
205             if(movementVector.magnitude > 1f)
206             {
207                 movementVector.Normalize();
208             }
209
```

```
            movementVector = transform.InverseTransformDirection(movementVector);
            movementVector = Vector3.ProjectOnPlane(movementVector, Vector3.zero);
            float turnAmount = Mathf.Atan2(movementVector.x, movementVector.z);

            float forwardAmount = 0f;
            float movementTarget = movementVector.z;

            if(movementTarget != 0)
            {
                forwardAmount = movementVector.z;
            }
            else
            {
                forwardAmount = Mathf.Lerp(forwardAmount, movementTarget, Time.deltaTime);
            }

            float turnSpeed = Mathf.Lerp(stationaryTurnSpeed, movementTarget, turnAmount);
            float extraRotation = turnAmount * turnSpeed * Time.deltaTime;

            OnAnimatorPropertiesUpdated(turnAmount, forwardAmount, extraRotation);
        }

        2 references
        public override void Update()
        {
            OnComplete();
        }
    }
}
```

- Next we Add this functions references in the "*SwitchTarget()*" function.
- This function simply passes a New Target.
- This New Target can be either of these twos: FaceTarget or MoveTarget.
- It would pass any one of those Targets at a Time, and execute their functionalities respectively.

```
    private void SwitchTarget(Target newTarget, Action OnActionComplete)
    {
        target = newTarget;

        target.OnAnimatorPropertiesUpdated += (turn, forward, turnSpeed) =>
        {
            turnAmount = turn;  //How much to turn (Angle of Rotation)
            forwardAmount = forward;    //How many meter or unit to move forward.
            ApplyRotation(turnSpeed);   //Speed of ROtation.
            UpdateAnimator();
        };

        target.OnComplete += () =>
        {
            target = null;
            OnActionComplete?.Invoke();
        };

        target.Start();
    }
```

- It also has two additional functions namely: "*ApplyRotation()*" & "*UpdateAnimator()*".
- "*ApplyRotation()*" simply rotates the Player along its Y-axis Component, by passing a parameter value to its Y component.

```
1 reference
private void ApplyRotation(float extraRotation) //Actual Rotation of Player along Y-axis.
{
    transform.Rotate(0, extraRotation, 0);
}
```

- "*UpdateAnimator()*" is responsible to set the Player' Animator States into Action. The "*Forward*" & the "*Turn*" States.

```
1 reference
private void UpdateAnimator()   //Updates Animation States & its Variables.
{
    float animForward = animator.GetFloat("Forward");
    //forwardAmount = Player's Movement Input.
    float transitionSpeed = animForward > forwardAmount ? runToIdelSpeed : idleToRunSpeed;

    animator.SetFloat("Forward", forwardAmount, 0.1f, Time.deltaTime * transitionSpeed);
    animator.SetFloat("Turn", turnAmount, 0.1f, Time.deltaTime);
}
```

- Then finally we pass the "*SwitchTarget()*" & the rest of the functionalities in the "*Face()*" & "*MoveTo()*" function.
- And also, we get rid of some of its Old Functionalities.

```
1 reference
public void MoveTo(Vector3 movementVector)
{
    transform.position += movementVector * Time.deltaTime * movementSpeed;

    //We check if the Face is in Progress, then simply return becoz we do not want the Player to Move when the Face Rotation is in process.
    if(IsFaceInProgress())
    {
        return;
    }

    //Move the Player
    SwitchTarget(new MoveTarget(transform, movementVector), null);
}

1 reference
private bool IsFaceInProgress() //Returns "True" if target is FaceTarget. Else returns False.
{

    return target is FaceTarget;
}

/*private void UpdateLookDirection(Vector2 lookDelta)
{
    transform.Rotate(Vector3.up, lookDelta.x * sensitivityX, Space.World);
    transform.Rotate(Vector3.left, lookDelta.y * sensitivityY, Space.Self);
}*/

1 reference
public void Face(Vector3 faceVector, Action OnComplete) //Creates FaceTarget in SwitchTarget function.
{
    //transform.forward = faceVector;
    //OnComplete();

    //We Set newTarget here - Face Setup
    SwitchTarget(new FaceTarget(transform, faceVector), OnComplete);    //FaceTarget faceTarget = new FaceTarget(transform, faceVector);
}
```

- At last, we create the "*Update()*" function, and pass in the "*TargetUpdate*" function.
- Also, we add the If-Else statement for safety purpose, that it the "*TargetUpdate*" doesn't get execute, return every value as Zero.

```
1 reference
public void Update()
{
    if(target != null)
    {
        target.Update();    //Calls the "Abstract Update()". So all "Override Updates()" get called.
    }
    else
    {
        turnAmount = 0;
        forwardAmount = 0;
        animator.SetFloat("Forward", 0);
        animator.SetFloat("Turn", 0);
    }
}
```

❖ **PlayerAnimationController.**

- This script is responsible to Move the Animator Component as well as the Rigidbody Component of the Player, since we are using the "*RootAnimation*" and the "*Rigidbidy*" properties.
- So, we implement a logic of Movement & Rotation of the Rigidbody, whenever the "*Animator*" moves.
- This is done inside the "*OnAnimatorMove()*" function.

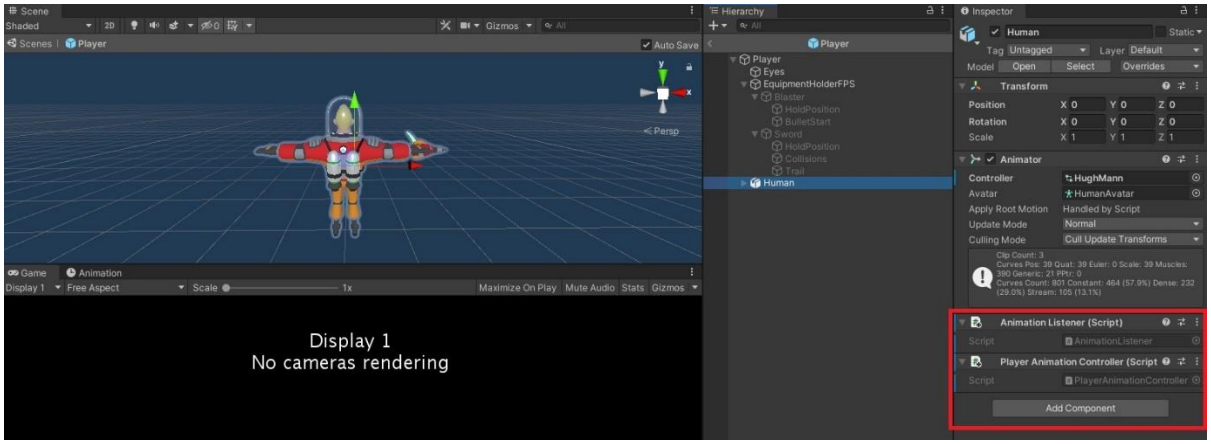
PlayerMovementController.cs M | PlayerAnimationController.cs U

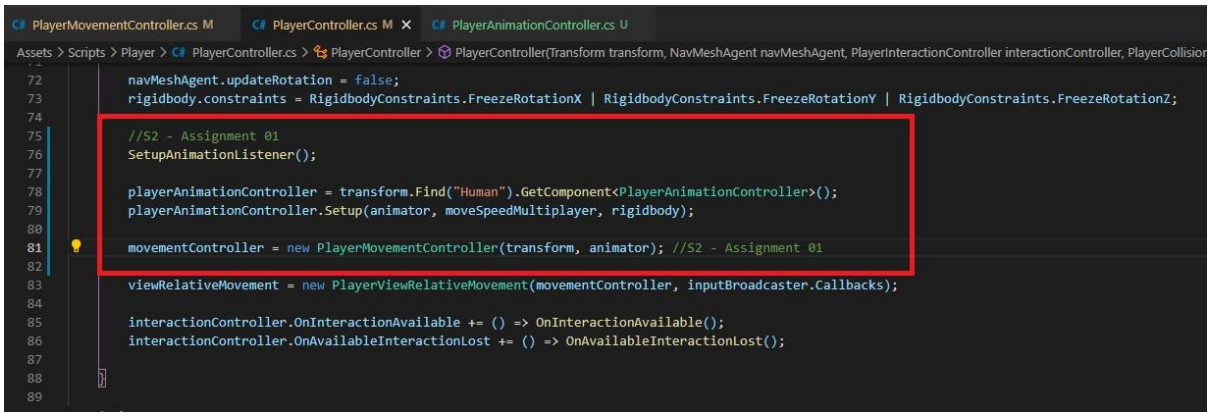
Assets > Scripts > Player > Animation > PlayerAnimationController.cs > ...

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

//  This Script moves the Rigidbody along the Animator values. As we are Using "Root Motion".
2 references
public class PlayerAnimationController : MonoBehaviour
{
    2 references
    private float moveSpeedMultiplayer;
    3 references
    private Animator animator;
    4 references
    private Rigidbody rb;

    1 reference
    public void Setup(Animator animator, float moveSpeedMultiplayer, Rigidbody rb)
    {
        this.animator = animator;
        this.moveSpeedMultiplayer = moveSpeedMultiplayer;
        this.rb = rb;
    }

    //OnAnimatorMove = Callback for processing animation movements for modifying root motion.
    0 references
    private void OnAnimatorMove()
    {
        if(Time.deltaTime > 0)
        {
            //animator.deltaPosition = The Position value from Last Frame to Current Frame.
            Vector3 velocity = (animator.deltaPosition * moveSpeedMultiplayer) / Time.deltaTime;

            //We want the animator & the Player stuck to the gorund. So we do not modify the Animator's Y value.
            //We keep the Animator's Y value equals to Rigidbody's Y value.
            velocity.y = rb.velocity.y;

            //In this way, we are just moving the Rigidbody along X & Z axes.
            rb.velocity = velocity;

            //animator.deltaRotation = The Rotation value from Last Frame to Current Frame.
            rb.rotation *= animator.deltaRotation;
        }

    }
}
```

❖ PlayerController.

- We simply pass these Functions in the "*PlayerController*" script.
- For the "*PlayerAnimationController*" script, we create a reference of it in the Player's Constructor, and pass in the required components respectively.
- We also Add this script to the "*Human*" ChildObject of the Player Prefab.
- We also Add the "*AnimationListener*" script to the "*Human*" ChildObject.
- Then we get the "*AnimationListener*" script component, in this script and pass in the "Death" functionality.





```
1 reference
private void SetupAnimationListener()   //S2 - Assignment 01
{
    animationListener = transform.Find("Human").GetComponent<AnimationListener>();

    animationListener.OnWeightedAnimationEvent += (arguent, weight) =>
    {
        switch(arguent)
        {
            case "DeathAnimationComplete":
                DeathAnimationComplete();
                break;
        }
    };
}
```

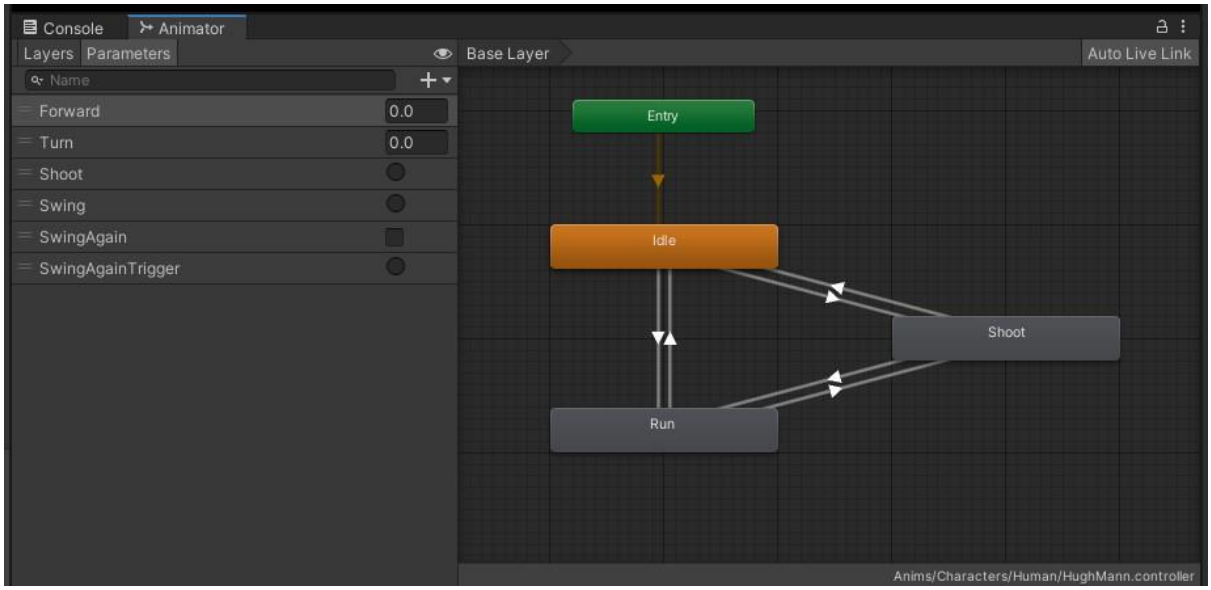
❖ LevelController.

- Then finally, we simply need to Add a reference to the "*Animator*" component of the "*Human*" ChildObject.
- We get that using the "*PlayerObject*" component, as it stores all the Player Components data in the "*Inspector*".

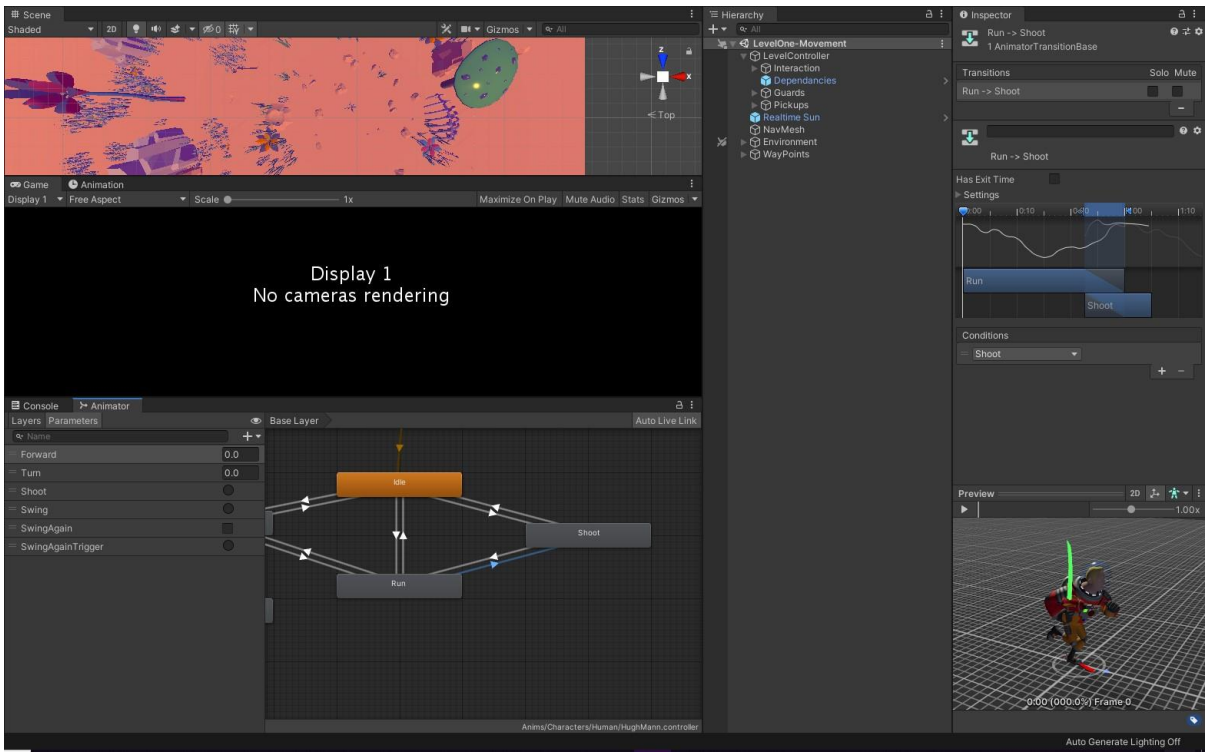
```
C# PlayerMovementController.cs M   C# LevelController.cs M ×   C# PlayerController.cs M   C# PlayerAnimationController.cs U
Assets > Scripts > Level > C# LevelController.cs > LevelController > CreatePlayer(GameObject playerObject, ProjectilePool projectilePool)

1 reference
private GameObject CreatePlayerObject(GameObject playerObjectPrefab)
{
    GameObject playerObject = GameObject.Instantiate(playerObjectPrefab, transform);
    playerObject.name = "Player";

    return playerObject;
}

1 reference
private Player CreatePlayer(GameObject playerObject, ProjectilePool projectilePool)
{
    NavMeshAgent navMeshAgent = playerObject.GetComponent<NavMeshAgent>();

    PlayerObjectData objectData = playerObject.GetComponent<PlayerObjectData>();

    //Bcoz "Human" is One of the Child Object to "Player".
    Animator animator = playerObject.transform.Find("Human").GetComponent<Animator>();  //S2 - Assignment 01

    PlayerCollision collision = new PlayerCollision(playerObject.transform);

    PlayerInteractionController interaction = new PlayerInteractionController(transform, playerObject.transform, collision, objectData);

    PlayerInputBroadcaster broadcaster = new PlayerInputBroadcaster();

    inventory = new InventoryController();  //Assignment 04 - Part I
```

❖ Shoot.

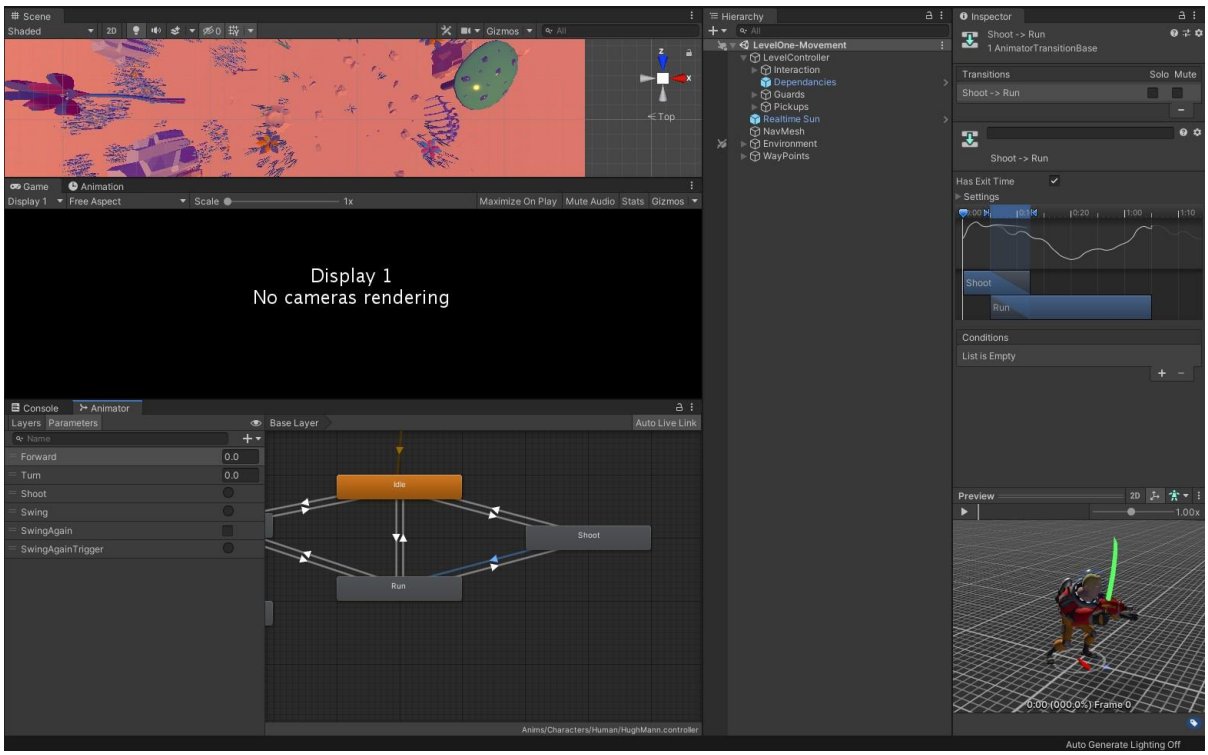
- First we create a "*Trigger*" called "*Shoot*" and also added functionality in the code where if we press "*SpaceBar*" the Shoot Animation is played.
- Then we add the Shoot State in the Animator and create Transitions with the "Idle" and "Run" states.



- Then, we Add the "*Shoot*" trigger to the Transition that are going from the **Idle > Shoot** & **Run > Shoot** and turn off the "*HasExitTime*".

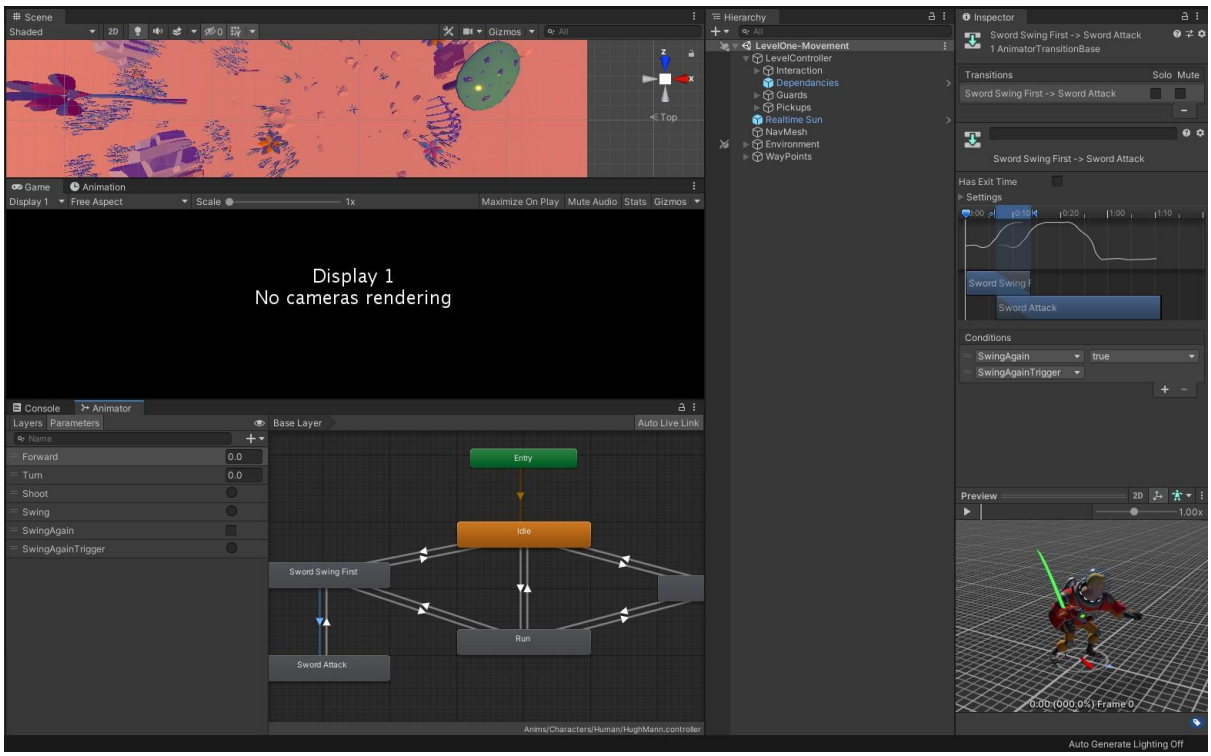


- For Then, for the return Transition, i.e., from **Shoot > Idle** or **Shoot > Run**, we leave it without any condition, and with "*HasExitTime*" set to On.



❖ SwordSwing.

- For this, I have created a "*Swing*" Trigger and added functionality such that when "*Enter*" key is pressed, it performs the Sword Swing animation.
- Then I added the "*Swing*" trigger to state that go from **Idle > SwordSwingFirst** & **Run > SwordSwingFirst**.
- Then I added the Second Swing Animation and created a "*SwingAgainTrigger*" Trigger & "*SwingAgain*" Boolean.
- Then, for the state from **SwordSwingFirst > SwordAttack**, I added *SwingAgain = true* & *SwingAgainTrigger*.
- Then for the state from **SwordAttack > SwordSwingFirst**, I added *SwingAgain = false*.
- Then from the **SwordSwingFirst > Run** & **SwordSwingFirst > Idle** transitions, I added *SwingAgain = false* to both respectively.




```
    private void UpdateAnimator()   //Updates Animation States & its Variables.
    {
        float animForward = animator.GetFloat("Forward");
        //forwardAmount = Player's Movement Input.
        float transitionSpeed = animForward > forwardAmount ? runToIdelSpeed : idleToRunSpeed;

        animator.SetFloat("Forward", forwardAmount, 0.1f, Time.deltaTime * transitionSpeed);
        animator.SetFloat("Turn", turnAmount, 0.1f, Time.deltaTime);

        if(Input.GetKeyDown(KeyCode.Space))
        {
            animator.SetTrigger("Shoot");
            //transform.position += Vector3.zero * Time.deltaTime * 0;
        }

        if(Input.GetKeyDown(KeyCode.Return))
        {
            animator.SetTrigger("Swing");
            //transform.position += Vector3.zero * Time.deltaTime * 0;
        }
    }

```

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