```
using System.Collections;
using System.Collections.Generic;
using Cinemachine;
using DG.Tweening;
using KBCore.Refs;

using Unity.VisualScripting.Antlr3.Runtime.Tree;

using UnityEngine;
using UnityEngine.Audio;


namespace teamFourFinalProject
{
    public class PlayerController : ValidatedMonoBehaviour
    {
        [Header("References")]
        //For some reason Animator is causing issues with jump. Has been omitted for now
        //[SerializeField, Self] Animator animator;
        [SerializeField, Self] Rigidbody rb;
        [SerializeField, Self] GroundChecker groundChecker;
        [SerializeField, Anywhere] CinemachineFreeLook freelookVCam;
        [SerializeField, Anywhere] InputReader input;


        [Header("Settings")]
        [SerializeField] float moveSpeed = 6f;
        [SerializeField] float rotationSpeed = 15f;
        [SerializeField] float smoothTime = 0.2f;

        [Header("Jump Settings")]
        [SerializeField] float jumpForce = 10f;
        [SerializeField] float jumpDuration = 0.5f;
        [SerializeField] float jumpCooldown = 0f;
        [SerializeField] float jumpMaxHeight = 5f;
        [SerializeField] float gravityMultiplier = 3f;
        [SerializeField] int maxJumpCount = 2;
        [SerializeField] float doubleJumpMultiplier = 2;
        int currentJumpCount = 0;
        private bool canDoubleJump = false;
        private bool doubleJumpRequested = false;
        private Animator animator;

        [Header("Sounds")]
```

```
AudioSource audioSource;
[SerializeField] AudioSource jump;
[SerializeField] AudioSource walk;
[SerializeField] AudioSource hurt;
[SerializeField] AudioClip walkSound;
[SerializeField] AudioClip hurtSound;

[Header("Powerup Settings")]
[SerializeField] float dashCooldown = 0f;
[HideInInspector] public int tempHealth;
[HideInInspector] public bool upgradeAppl;
private PowerupData heldPowerup = null;
private bool powerupActive = false;
public Transform player, destination;
public GameObject playerg;
public PowerupData powerupData;
private PowerupData currentActivePowerup = null;

[Header("Camera Controller Settings")]
[SerializeField] float controllerSensitivity = 300f;
[SerializeField] float controllerSmoothTime = 0.2f; //lower = snappier, higher = smoother
private Vector2 controllerLookVelocity;
private Vector2 currentControllerLook;

//If making a different way to attack
/*[Header("Attack Settings")]
[SerializeField] float attackCooldown = 0.5f;
[SerializeField] float attackDistance = 1f;
[SerializeField] int attackDamage = 10;*/

//Animator
static readonly int Speed = Animator.StringToHash(name: "Speed");

float currentSpeed;
float velocity;
float jumpVelocity;


Transform mainCam;

const float ZeroF = 0f;

Vector3 movement;
```

```
    List<JumpTimer> timers;
    CountdownTimer jumpTimer;
    CountdownTimer jumpCooldownTimer;
    CountdownTimer attackTimer;
    CountdownTimer powerupTimer;
    CountdownTimer dashCooldownTimer;
    CountdownTimer throwCooldownTimer;

    StateMachine stateMachine;

    private void Awake()
    {
        mainCam = Camera.main.transform;
        freelookVCam.Follow = transform;
        freelookVCam.LookAt = transform;

        //Invoke event when observed transform is teleported, adjusting freeLookVCam's
position accordingly
        freelookVCam.OnTargetObjectWarped(transform, positionDelta: transform.position -
freelookVCam.transform.position - Vector3.forward);

        rb.freezeRotation = true;

        //Setting up timers
        jumpTimer = new CountdownTimer(jumpDuration);
        jumpCooldownTimer = new CountdownTimer(jumpCooldown);
        timers = new List<JumpTimer>(capacity: 2) { jumpTimer, jumpCooldownTimer };
        dashCooldownTimer = new CountdownTimer(dashCooldown);
        timers.Add(dashCooldownTimer);
        throwCooldownTimer = new CountdownTimer(0f);
        timers.Add(throwCooldownTimer);

        //jumpTimer.OnTimerStart =+ () => jumpVelocity = jumpForce;
        //jumpTimer.OnTimerStop += () => jumpCooldownTimer.Start();

        //State Machine
        stateMachine = new StateMachine();

        //Declare States
        //var locomotionState = new LocomotionState(player: this, animator);
        //var jumpState = new JumpState(player: this, animator);

        //Define transitions
```

```
        //At(from: locomotionState, to: jumpState, condition: new FuncPredicate(() =>
jumpTimer.isRunning));
        //At(from: jumpState, to: locomotionState, condition: new FuncPredicate(() =>
groundChecker.isGrounded && !jumpTimer.isRunning));

        //Set initial state
        //stateMachine.SetState(locomotionState);

        //Powerup
        powerupTimer = new CountdownTimer(0f);
        timers.Add(powerupTimer);

        powerupTimer.OnTimerStop += () =>
        {
            if (currentActivePowerup != null)
            {
                if (currentActivePowerup is CardPowerup cardPowerup)
                {
                    cardPowerup.HideGhostPreview();
                }
                currentActivePowerup.RemoveEffects(this);
                Debug.Log("Powerup ended");
                currentActivePowerup = null;
            }

            heldPowerup = null;
            powerupActive = false;
        };
    }

    //void At(IState from, IState to, IPredicate condition) => stateMachine.AddTransition(from,
to, condition);
    //void Any(IState to, IPredicate condition) => stateMachine.AddAnyTransition(to, condition);

    private void Start()
    {
        input.EnablePlayerActions();
        animator = GetComponentInChildren<Animator>();
    }

    void OnEnable()
    {
        input.Jump += OnJump;
        input.ActivatePowerup += OnPowerup;
```

```
    input.Look += OnLook;
}

void OnDisable()
{
    input.Jump -= OnJump;
    input.ActivatePowerup -= OnPowerup;
    input.Look -= OnLook;
}

void OnJump(bool performed)
{
    if (!performed) return;

    if (groundChecker.isGrounded)
    {
        walk.Stop();
        jump.Play();
        animator.SetBool("isJumping", true);
        animator.SetBool("isFalling", true);
        jumpTimer.Start();
        currentJumpCount = 1;
        rb.velocity = new Vector3(rb.velocity.x, jumpForce, rb.velocity.z);
        canDoubleJump = true;
    }

    else if (canDoubleJump && currentJumpCount < maxJumpCount)
    {
        doubleJumpRequested = true;
        currentJumpCount++;
        canDoubleJump = false;
    }
}


void OnPowerup()
{
    if (heldPowerup == null || !powerupActive) return;

    //HatBlink Behavior
    if (currentActivePowerup is HatBlink)
    {
        DashThrough();
    }
```

```
        else if (currentActivePowerup is CardPowerup cardPowerup)
        {
            if (!throwCooldownTimer.isRunning)
            {
                cardPowerup.ThrowPlatform(this);
                throwCooldownTimer.Reset(cardPowerup.throwCooldown);
                throwCooldownTimer.Start();
            }

            cardPowerup.HideGhostPreview();
        }
    }

    void OnLook(Vector2 lookDelta, bool isMouse)
    {
        if (isMouse)
        {
            freelookVCam.m_XAxis.Value += lookDelta.x;
            freelookVCam.m_YAxis.Value += lookDelta.y;
        }

        else
        {
            {
                currentControllerLook = Vector2.SmoothDamp(
                    currentControllerLook,
                    lookDelta,
                    ref controllerLookVelocity,
                    controllerSmoothTime
                    );

                freelookVCam.m_XAxis.Value += lookDelta.x * controllerSensitivity *
Time.deltaTime;
                freelookVCam.m_YAxis.Value -= lookDelta.y * controllerSensitivity *
Time.deltaTime;
            }
        }
    }

    private void Update()
    {
        //Debug.Log("Camera forward: " + Camera.main.transform.forward);

        movement = new Vector3(input.Direction.x, y: 0f, z: input.Direction.y);
```

```
    HandleAnimator();
    HandleTimers();

    if (heldPowerup is CardPowerup cardPowerup && powerupActive)
    {
        cardPowerup.UpdateGhost(this);
    }
}

private void FixedUpdate()
{
    HandleJump();
    HandleMovement();
}
void HandleAnimator()
{
    //animator.SetFloat(id: Speed, currentSpeed);
}

void HandleTimers()
{
    foreach (var timer in timers)
    {
        timer.Tick(Time.deltaTime);
    }
}

public void PickupPowerup(PowerupData newPowerup)
{
    if (powerupActive)
    {
        currentActivePowerup.RemoveEffects(this);
        powerupTimer.Stop();
        currentActivePowerup = null;
    }

    heldPowerup = newPowerup;
    HandlePowerup();
    Debug.Log($"Picked up powerup: {heldPowerup.GetType().Name}");
}

public void PickupKey(KeyData newKey)
{
```

```
            if (!SceneManager.instance.collectedKeyIDs.Contains(newKey.keyID))
            {
                SceneManager.instance.collectedKeyIDs.Add(newKey.keyID);
                Debug.Log($"Picked up key: {newKey.keyID}");
            }

            else
            {
                Debug.Log($"Key {newKey.keyID} is already collected");
            }
        }

        public void HandleJump()
        {
            if (doubleJumpRequested)
            {
                animator.SetBool("isDoubleJumping", true);
                jump.Play();
                walk.Stop();
                float doubleJumpHeight = jumpMaxHeight * doubleJumpMultiplier;
                float doubleJumpVelocity = Mathf.Sqrt(2 * doubleJumpHeight *
Mathf.Abs(Physics.gravity.y));
                //jumpVelocity = jumpForce * doubleJumpMultiplier;
                jumpVelocity = doubleJumpVelocity;
                doubleJumpRequested = false;
            }

            else if (!groundChecker.isGrounded)
            {
                walk.Stop();
                jumpVelocity += Physics.gravity.y * gravityMultiplier * Time.fixedDeltaTime;
            }

            //If not jumping and grounded, keep jump velocity at 0
            if (!jumpTimer.isRunning && groundChecker.isGrounded)
            {

                animator.SetBool("isFalling", false);
                animator.SetBool("isJumping", false);
                animator.SetBool("isDoubleJumping", false);
                jumpVelocity = ZeroF;
                jumpTimer.Stop();
                currentJumpCount = 0;
                return;
```

```
        }

        //If jumping or falling calculate velocity
        if (jumpTimer.isRunning)
        {
            //Progress point for inital burst of velocity
            float launchPoint = 0.9f;
            if (jumpTimer.Progress > launchPoint)
            {
                //Calculate the velocity required to reach the jump height using physics equations v
= sqrt(2gh) (height (h), gravity (g), velocity (v)
                jumpVelocity = Mathf.Sqrt(f: 2 * (jumpMaxHeight * doubleJumpMultiplier) *
Mathf.Abs(Physics.gravity.y));
            }

            else
            {
                //Gradually apply less velocity as the jump progresses
                jumpVelocity += (1 - jumpTimer.Progress) * jumpForce * Time.fixedDeltaTime;
            }
        }

        else
        {
            jumpVelocity += Physics.gravity.y * gravityMultiplier * Time.fixedDeltaTime;
        }

        //Apply Velocity
        rb.velocity = new Vector3(rb.velocity.x, jumpVelocity, rb.velocity.z);

    }

    public void HandleMovement()
    {
        //Rotate movement direction to match camera rotation
        var adjustedDirection = Quaternion.AngleAxis(mainCam.eulerAngles.y, Vector3.up) *
movement;
        if (adjustedDirection.magnitude > ZeroF)
        {
            HandleRotation(adjustedDirection);
            HandleHorizontalMovement(adjustedDirection);
            SmoothSpeed(adjustedDirection.magnitude);
        }
        else
```

```
        {
            SmoothSpeed(ZeroF);

            //Reset horizontal velocity
            rb.velocity = new Vector3(x: ZeroF, rb.velocity.y, z: ZeroF);
            animator.SetBool("isFalling", false);
            animator.SetBool("isWalking", false);

            walk.Stop();
        }
    }

    void HandleHorizontalMovement(Vector3 adjustedDirection)
    {
        //Move the player

        animator.SetBool("isFalling", false);
        animator.SetBool("isWalking", true);
        Vector3 velocity = adjustedDirection * moveSpeed * Time.fixedDeltaTime;
        rb.velocity = new Vector3(velocity.x, rb.velocity.y, velocity.z);
        if (!walk.isPlaying && groundChecker.isGrounded)
        {
            walk.Play();
        }

    }

    void HandleRotation(Vector3 adjustedDirection)
    {
        //Adjust rotation to match movement direction
        var targetRotation = Quaternion.LookRotation(adjustedDirection);
        transform.rotation = Quaternion.RotateTowards(from: transform.rotation, to:
targetRotation, maxDegreesDelta: rotationSpeed * Time.deltaTime);
        transform.LookAt(worldPosition: transform.position + adjustedDirection);
    }

    void SmoothSpeed(float value)
    {
        currentSpeed = Mathf.SmoothDamp(current: currentSpeed, target: value, ref velocity,
smoothTime);

    }
    public void SetInvulnerable(bool value)
    {
```

```
            //Disable damage handling
        }

        public void SetPassThroughEnemies(bool value)
        {
            gameObject.layer = value ? LayerMask.NameToLayer("HatBlink") :
LayerMask.NameToLayer("Player");
        }

        public void DashThrough()
        {
            if (!powerupActive || dashCooldownTimer.isRunning) return;

            if (CompareTag("Player"))
            {
                playerg.SetActive(false);
                player.position = destination.position;
                playerg.SetActive(true);

                Debug.Log("Dash");

                dashCooldownTimer.Reset(dashCooldown);
                dashCooldownTimer.Start();
            }
        }
        void HandlePowerup()
        {
            if (!powerupActive && heldPowerup != null)
            {
                heldPowerup.ApplyEffects(this);
                powerupTimer.Reset(heldPowerup.duration);
                powerupTimer.Start();
                powerupActive = true;
                currentActivePowerup = heldPowerup;

                Debug.Log($"Activated powerup: {heldPowerup.name}");
            }
        }
    }
}
```