중간고사 REPORT



|  |  |
| --- | --- |
| 과 목 : | 디지털트윈 |
| 제출일자 : | 2024.04.29 |
| 담당교수 : | 박상욱 |
| 학 과 : | 정보통신공학전공 |
| 학 번 : | 201920479 |
| 이 름 : | 김행복 |

1. 디지털 트윈의 개념에 대해 설명하시오. 앞으로 에너지 분야(신재생 에너지 및 수소)에서 디지털 트윈 기술의 방향성은 어떻게 나아가야 할지 자신이 선호하는 연구 분야와 접목하여 제시하시오. (500자 내외)

|  |
| --- |
| 디지털 트윈은 현실 세계의 객체나 시스템을 가상 환경에서 정밀하게 복제하는 가상 모델 기술이다. 이 기술은 실제 데이터를 기반으로 하는 가상 모델을 생성하여, 실시간 분석과 시뮬레이션을 통해 효율성을 극대화하고 문제를 사전에 예측할 수 있게 한다. 특히, 수소 에너지 분야에서 디지털 트윈은 설비의 최적화, 유지보수 예측, 에너지 효율 증진 등에 활용될 수 있다.  앞으로 수소 에너지 분야에서 디지털 트윈 기술의 발전 방향은 빅데이터 분석과의 접목을 통해 더욱 구체화될 필요가 있다. 수집되는 방대한 양의 데이터를 효과적으로 분석하여, 수소 생산 및 저장, 운송 과정에서 발생할 수 있는 다양한 시나리오를 예측하고 대응할 수 있다. 이를 위해 다양한 고급 분석 기법을 활용하여, 디지털 트윈 모델의 정확도를 높이고, 실시간으로 대응 할 수 있는 동적 시스템으로 발전시키는 것이 중요하다. 디지털 트윈과 빅데이터 분석의 결합은 수소 에너지 시스템의 안정성을 강화하고, 에너지 효율을 높이는데 필수적인 요소가 될 것이다. |

2. 다음 클래스 및 함수에 대한 기능을 설명하시오.

1) transform.Translate(Vector3.forward);

|  |
| --- |
| Transform 클래스는 유니티에서 게임 오브젝트의 위치, 회전, 크기를 조정하는 클래스다.  transform.Translate() 함수는 유니티에서 게임 오브젝트를 특정 방향과 거리로 이동시키는 데 사용된다. 이 함수는 오브젝트의 현재 위치에서 지정된 벡터만큼 상대적으로 이동시키며, 이동은 오브젝트의 로컬 좌표계 또는 월드 좌표계를 기준으로 할 수 있다.  Vector3.forward는 Unity의 3차원 벡터 표현에서 (0, 0, 1)로 정의된다. 이것은 오브젝트가 z축 방향으로 앞쪽을 바라볼 때를 나타내는 방향 벡터다. |

2) transform.Rotate(Vector3.up);

|  |
| --- |
| Transform 클래스는 유니티에서 게임 오브젝트의 위치, 회전, 크기를 조정하는 클래스다.  Transform.Rotate() 함수는 Unity에서 게임 오브젝트를 특정 축 주변으로 회전시키는 데 사용된다. 이 함수는 각 축에 대한 회전 각도를 입력받아 오브젝트를 회전시키며, 회전은 오브젝트의 로컬 좌표계 또는 월드 좌표계를 기준으로 할 수 있다.  Vector3.up은 Unity에서 (0, 1, 0)으로 정의된 3차원 벡터입니다. 이는 오브젝트의 y축 방향으로 '위'를 가리키는 방향 벡터로 사용된다. |

3) transform.rotation = Quaternion.Lerp(transform.rotation, Quaternion.Euler(0, 0, 180), 0.05f);

|  |
| --- |
| Transform.rotation 속성은 Unity에서 게임 오브젝트의 회전을 쿼터니언(Quaternion) 형태로 관리한다. 쿼터니언은 3D 공간에서의 회전을 표현하기 위해 널리 사용되며, 짐벌락(Gimbal lock) 같은 문제를 피할 수 있게 해주는 효율적인 방법을 제공한다.  Quaternion.Lerp 함수는 Unity에서 두 쿼터니언 간의 선형 보간(Linear Interpolation)을 수행하여 부드러운 회전을 구현하는 데 사용됩니다. 이 함수는 시작 쿼터니언과 목표 쿼터니언 사이를 특정 비율로 보간하여, 두 쿼터니언 상태 간의 부드러운 전환을 가능하게 한다.  매개변수 (transform.rotation)는 보간의 시작점으로 사용되는 오브젝트의 현재 회전 상태다.  매개변수 (Quaternion.Euler(0, 0, 180))는 목표 회전 상태로, 오일러 각을 사용하여 (0, 0, 180)으로 정의된 쿼터니언이다. 이는 오브젝트가 z축을 중심으로 180도 회전한 상태를 나타낸다.  매개변수 (0.05f)는 보간 인자 t는 0과 1 사이의 값을 가지며, 이 경우 0.05로 설정되어 있다. 이 값은 시작 회전에서 목표 회전으로의 전환을 5% 진행하겠다는 것을 의미한다. |

4) GameObject.SetActive(true);

|  |
| --- |
| SetActive() 메소드는 Unity에서 게임 오브젝트의 활성화 상태를 설정하는 데 사용된다. 이 메소드를 사용하여 게임 오브젝트를 활성화하거나 비활성화할 수 있다.  true 매개변수는 게임 오브젝트를 활성화를 결정한다. 활성화되면 오브젝트가 업데이트를 받고, 렌더링이 되며, 모든 연결된 컴포넌트가 활동적으로 작동하기 시작한다. |

5) PlayerPrefs.SetInt("A", 1);

|  |
| --- |
| PlayerPrefs.SetInt() 메소드는 Unity에서 간단한 데이터를 로컬에 저장하기 위해 사용된다. 이 함수를 통해 개발자는 정수 타입의 데이터를 키-값 형태로 저장할 수 있으며, 이 데이터는 게임을 종료하고 다시 시작해도 유지된다. 여기서 “A”는 데이터를 식별하는 데 사용되는 문자열이며, 1은 저장하려는 정수 값이다. |

6) PlayerPrefs.GetInt("A");

|  |
| --- |
| PlayerPrefs 클래스는 Unity에서 게임의 설정이나 플레이어의 진행 상태 등의 간단한 정보를 로컬에 키-값 형식으로 저장하고 검색하는 기능을 제공한다. 이 클래스는 데이터를 문자열, 정수 또는 실수 형태로 저장하며, 게임이 종료되어도 저장된 데이터는 유지된다.  지정된 key에 해당하는 정수 값을 반환한다. 여기서는 5번 문제에서 저장한 1 값이 반환될 수 있다. |

3. 유니티 프로그램의 "충돌체"와 "파티클 시스템"의 개념을 설명하시오.

1) 충돌체

|  |
| --- |
| 충돌체 (Collider)  Unity에서 충돌체는 게임 오브젝트가 물리적 상호 작용을 할 수 있게 하는 컴포넌트다. 충돌체는 오브젝트의 물리적 경계를 정의하고, 이 경계가 다른 오브젝트의 충돌체와 겹칠 때 물리 엔진이 이를 감지하고 반응하도록 한다. 충돌체는 물리적 충돌 감지 외에도 트리거 이벤트를 발생시키는 데 사용될 수 있다.  주요 유형의 충돌체:  박스 충돌체 (Box Collider): 사각형 프리즘 형태의 객체에 사용.  구 충돌체 (Sphere Collider): 구형 객체에 적합.  캡슐 충돌체 (Capsule Collider): 인간의 몸이나 기둥 같은 형태에 적합.  메쉬 충돌체 (Mesh Collider): 복잡한 형상을 가진 객체에 사용, 오브젝트의 정확한 메쉬를 충돌 경계로 사용.  지형 충돌체 (Terrain Collider): 지형 오브젝트에 부착하여 지형의 물리적 경계를 정의. |

2) 파티클 시스템

|  |
| --- |
| 파티클 시스템 (Particle System)  파티클 시스템은 Unity에서 수많은 작은 입자들을 생성, 제어, 그리고 파괴하는 기능을 제공하는 컴포넌트다. 이 시스템을 통해 개발자들은 연기, 불, 먼지, 마법 효과 등 다양한 시각적 효과를 생성할 수 있다. 파티클 시스템은 각 파티클의 생명주기, 속도, 색상, 크기 등을 매우 세밀하게 조정할 수 있어 복잡하고 다이나믹한 효과를 효율적으로 구현할 수 있다.  파티클 시스템의 주요 구성 요소:  발생기 (Emitter): 파티클을 생성하는 역할.  파티클 (Particles): 개별적으로 독립된 작은 입자들.  시뮬레이터 (Simulator): 파티클의 동작과 상호 작용을 처리.  렌더러 (Renderer): 파티클을 화면에 시각적으로 표현.. |

4. 수업시간에 진행한 내용을 기반으로 유니티의 "충돌체"와 "파티클 시스템"을 응용하여 간단한 게임을 구현하시오.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 게임 이미지    플레이어 시점  스크린샷, 멀티미디어 소프트웨어, 그래픽 소프트웨어, 비디오 게임 소프트웨어이(가) 표시된 사진  자동 생성된 설명    <플레이어 파트>  - PlayerInputHandler.cpp   |  | | --- | | using Unity.FPS.Game;  using UnityEngine;  namespace Unity.FPS.Gameplay  {  public class PlayerInputHandler : MonoBehaviour  {  [Tooltip("Sensitivity multiplier for moving the camera around")]  public float LookSensitivity = 1f;  [Tooltip("Additional sensitivity multiplier for WebGL")]  public float WebglLookSensitivityMultiplier = 0.25f;  [Tooltip("Limit to consider an input when using a trigger on a controller")]  public float TriggerAxisThreshold = 0.4f;  [Tooltip("Used to flip the vertical input axis")]  public bool InvertYAxis = false;  [Tooltip("Used to flip the horizontal input axis")]  public bool InvertXAxis = false;  GameFlowManager m\_GameFlowManager;  PlayerCharacterController m\_PlayerCharacterController;  bool m\_FireInputWasHeld;  void Start()  {  m\_PlayerCharacterController = GetComponent<PlayerCharacterController>();  DebugUtility.HandleErrorIfNullGetComponent<PlayerCharacterController, PlayerInputHandler>(  m\_PlayerCharacterController, this, gameObject);  m\_GameFlowManager = FindObjectOfType<GameFlowManager>();  DebugUtility.HandleErrorIfNullFindObject<GameFlowManager, PlayerInputHandler>(m\_GameFlowManager, this);  Cursor.lockState = CursorLockMode.Locked;  Cursor.visible = false;  }  void LateUpdate()  {  m\_FireInputWasHeld = GetFireInputHeld();  }  public bool CanProcessInput()  {  return Cursor.lockState == CursorLockMode.Locked && !m\_GameFlowManager.GameIsEnding;  }  public Vector3 GetMoveInput()  {  if (CanProcessInput())  {  Vector3 move = new Vector3(Input.GetAxisRaw(GameConstants.k\_AxisNameHorizontal), 0f,  Input.GetAxisRaw(GameConstants.k\_AxisNameVertical));  // constrain move input to a maximum magnitude of 1, otherwise diagonal movement might exceed the max move speed defined  move = Vector3.ClampMagnitude(move, 1);  return move;  }  return Vector3.zero;  }  public float GetLookInputsHorizontal()  {  return GetMouseOrStickLookAxis(GameConstants.k\_MouseAxisNameHorizontal,  GameConstants.k\_AxisNameJoystickLookHorizontal);  }  public float GetLookInputsVertical()  {  return GetMouseOrStickLookAxis(GameConstants.k\_MouseAxisNameVertical,  GameConstants.k\_AxisNameJoystickLookVertical);  }  public bool GetJumpInputDown()  {  if (CanProcessInput())  {  return Input.GetButtonDown(GameConstants.k\_ButtonNameJump);  }  return false;  }  public bool GetJumpInputHeld()  {  if (CanProcessInput())  {  return Input.GetButton(GameConstants.k\_ButtonNameJump);  }  return false;  }  public bool GetFireInputDown()  {  return GetFireInputHeld() && !m\_FireInputWasHeld;  }  public bool GetFireInputReleased()  {  return !GetFireInputHeld() && m\_FireInputWasHeld;  }  public bool GetFireInputHeld()  {  if (CanProcessInput())  {  bool isGamepad = Input.GetAxis(GameConstants.k\_ButtonNameGamepadFire) != 0f;  if (isGamepad)  {  return Input.GetAxis(GameConstants.k\_ButtonNameGamepadFire) >= TriggerAxisThreshold;  }  else  {  return Input.GetButton(GameConstants.k\_ButtonNameFire);  }  }  return false;  }  public bool GetAimInputHeld()  {  if (CanProcessInput())  {  bool isGamepad = Input.GetAxis(GameConstants.k\_ButtonNameGamepadAim) != 0f;  bool i = isGamepad  ? (Input.GetAxis(GameConstants.k\_ButtonNameGamepadAim) > 0f)  : Input.GetButton(GameConstants.k\_ButtonNameAim);  return i;  }  return false;  }  public bool GetSprintInputHeld()  {  if (CanProcessInput())  {  return Input.GetButton(GameConstants.k\_ButtonNameSprint);  }  return false;  }  public bool GetCrouchInputDown()  {  if (CanProcessInput())  {  return Input.GetButtonDown(GameConstants.k\_ButtonNameCrouch);  }  return false;  }  public bool GetCrouchInputReleased()  {  if (CanProcessInput())  {  return Input.GetButtonUp(GameConstants.k\_ButtonNameCrouch);  }  return false;  }  public bool GetReloadButtonDown()  {  if (CanProcessInput())  {  return Input.GetButtonDown(GameConstants.k\_ButtonReload);  }  return false;  }  public int GetSwitchWeaponInput()  {  if (CanProcessInput())  {  bool isGamepad = Input.GetAxis(GameConstants.k\_ButtonNameGamepadSwitchWeapon) != 0f;  string axisName = isGamepad  ? GameConstants.k\_ButtonNameGamepadSwitchWeapon  : GameConstants.k\_ButtonNameSwitchWeapon;  if (Input.GetAxis(axisName) > 0f)  return -1;  else if (Input.GetAxis(axisName) < 0f)  return 1;  else if (Input.GetAxis(GameConstants.k\_ButtonNameNextWeapon) > 0f)  return -1;  else if (Input.GetAxis(GameConstants.k\_ButtonNameNextWeapon) < 0f)  return 1;  }  return 0;  }  public int GetSelectWeaponInput()  {  if (CanProcessInput())  {  if (Input.GetKeyDown(KeyCode.Alpha1))  return 1;  else if (Input.GetKeyDown(KeyCode.Alpha2))  return 2;  else if (Input.GetKeyDown(KeyCode.Alpha3))  return 3;  else if (Input.GetKeyDown(KeyCode.Alpha4))  return 4;  else if (Input.GetKeyDown(KeyCode.Alpha5))  return 5;  else if (Input.GetKeyDown(KeyCode.Alpha6))  return 6;  else if (Input.GetKeyDown(KeyCode.Alpha7))  return 7;  else if (Input.GetKeyDown(KeyCode.Alpha8))  return 8;  else if (Input.GetKeyDown(KeyCode.Alpha9))  return 9;  else  return 0;  }  return 0;  }  float GetMouseOrStickLookAxis(string mouseInputName, string stickInputName)  {  if (CanProcessInput())  {  // Check if this look input is coming from the mouse  bool isGamepad = Input.GetAxis(stickInputName) != 0f;  float i = isGamepad ? Input.GetAxis(stickInputName) : Input.GetAxisRaw(mouseInputName);  // handle inverting vertical input  if (InvertYAxis)  i \*= -1f;  // apply sensitivity multiplier  i \*= LookSensitivity;  if (isGamepad)  {  // since mouse input is already deltaTime-dependant, only scale input with frame time if it's coming from sticks  i \*= Time.deltaTime;  }  else  {  // reduce mouse input amount to be equivalent to stick movement  i \*= 0.01f;  #if UNITY\_WEBGL  // Mouse tends to be even more sensitive in WebGL due to mouse acceleration, so reduce it even more  i \*= WebglLookSensitivityMultiplier;  #endif  }  return i;  }  return 0f;  }  }  } |   - Player Character Controller   |  | | --- | | using Unity.FPS.Game;  using UnityEngine;  using UnityEngine.Events;  namespace Unity.FPS.Gameplay  {  [RequireComponent(typeof(CharacterController), typeof(PlayerInputHandler), typeof(AudioSource))]  public class PlayerCharacterController : MonoBehaviour  {  [Header("References")] [Tooltip("Reference to the main camera used for the player")]  public Camera PlayerCamera;  [Tooltip("Audio source for footsteps, jump, etc...")]  public AudioSource AudioSource;  [Header("General")] [Tooltip("Force applied downward when in the air")]  public float GravityDownForce = 20f;  [Tooltip("Physic layers checked to consider the player grounded")]  public LayerMask GroundCheckLayers = -1;  [Tooltip("distance from the bottom of the character controller capsule to test for grounded")]  public float GroundCheckDistance = 0.05f;  [Header("Movement")] [Tooltip("Max movement speed when grounded (when not sprinting)")]  public float MaxSpeedOnGround = 10f;  [Tooltip(  "Sharpness for the movement when grounded, a low value will make the player accelerate and decelerate slowly, a high value will do the opposite")]  public float MovementSharpnessOnGround = 15;  [Tooltip("Max movement speed when crouching")] [Range(0, 1)]  public float MaxSpeedCrouchedRatio = 0.5f;  [Tooltip("Max movement speed when not grounded")]  public float MaxSpeedInAir = 10f;  [Tooltip("Acceleration speed when in the air")]  public float AccelerationSpeedInAir = 25f;  [Tooltip("Multiplicator for the sprint speed (based on grounded speed)")]  public float SprintSpeedModifier = 2f;  [Tooltip("Height at which the player dies instantly when falling off the map")]  public float KillHeight = -50f;  [Header("Rotation")] [Tooltip("Rotation speed for moving the camera")]  public float RotationSpeed = 200f;  [Range(0.1f, 1f)] [Tooltip("Rotation speed multiplier when aiming")]  public float AimingRotationMultiplier = 0.4f;  [Header("Jump")] [Tooltip("Force applied upward when jumping")]  public float JumpForce = 9f;  [Header("Stance")] [Tooltip("Ratio (0-1) of the character height where the camera will be at")]  public float CameraHeightRatio = 0.9f;  [Tooltip("Height of character when standing")]  public float CapsuleHeightStanding = 1.8f;  [Tooltip("Height of character when crouching")]  public float CapsuleHeightCrouching = 0.9f;  [Tooltip("Speed of crouching transitions")]  public float CrouchingSharpness = 10f;  [Header("Audio")] [Tooltip("Amount of footstep sounds played when moving one meter")]  public float FootstepSfxFrequency = 1f;  [Tooltip("Amount of footstep sounds played when moving one meter while sprinting")]  public float FootstepSfxFrequencyWhileSprinting = 1f;  [Tooltip("Sound played for footsteps")]  public AudioClip FootstepSfx;  [Tooltip("Sound played when jumping")] public AudioClip JumpSfx;  [Tooltip("Sound played when landing")] public AudioClip LandSfx;  [Tooltip("Sound played when taking damage froma fall")]  public AudioClip FallDamageSfx;  [Header("Fall Damage")]  [Tooltip("Whether the player will recieve damage when hitting the ground at high speed")]  public bool RecievesFallDamage;  [Tooltip("Minimun fall speed for recieving fall damage")]  public float MinSpeedForFallDamage = 10f;  [Tooltip("Fall speed for recieving th emaximum amount of fall damage")]  public float MaxSpeedForFallDamage = 30f;  [Tooltip("Damage recieved when falling at the mimimum speed")]  public float FallDamageAtMinSpeed = 10f;  [Tooltip("Damage recieved when falling at the maximum speed")]  public float FallDamageAtMaxSpeed = 50f;  public UnityAction<bool> OnStanceChanged;  public Vector3 CharacterVelocity { get; set; }  public bool IsGrounded { get; private set; }  public bool HasJumpedThisFrame { get; private set; }  public bool IsDead { get; private set; }  public bool IsCrouching { get; private set; }  public float RotationMultiplier  {  get  {  if (m\_WeaponsManager.IsAiming)  {  return AimingRotationMultiplier;  }  return 1f;  }  }  Health m\_Health;  PlayerInputHandler m\_InputHandler;  CharacterController m\_Controller;  PlayerWeaponsManager m\_WeaponsManager;  Actor m\_Actor;  Vector3 m\_GroundNormal;  Vector3 m\_CharacterVelocity;  Vector3 m\_LatestImpactSpeed;  float m\_LastTimeJumped = 0f;  float m\_CameraVerticalAngle = 0f;  float m\_FootstepDistanceCounter;  float m\_TargetCharacterHeight;  const float k\_JumpGroundingPreventionTime = 0.2f;  const float k\_GroundCheckDistanceInAir = 0.07f;  void Awake()  {  ActorsManager actorsManager = FindObjectOfType<ActorsManager>();  if (actorsManager != null)  actorsManager.SetPlayer(gameObject);  }  void Start()  {  // fetch components on the same gameObject  m\_Controller = GetComponent<CharacterController>();  DebugUtility.HandleErrorIfNullGetComponent<CharacterController, PlayerCharacterController>(m\_Controller,  this, gameObject);  m\_InputHandler = GetComponent<PlayerInputHandler>();  DebugUtility.HandleErrorIfNullGetComponent<PlayerInputHandler, PlayerCharacterController>(m\_InputHandler,  this, gameObject);  m\_WeaponsManager = GetComponent<PlayerWeaponsManager>();  DebugUtility.HandleErrorIfNullGetComponent<PlayerWeaponsManager, PlayerCharacterController>(  m\_WeaponsManager, this, gameObject);  m\_Health = GetComponent<Health>();  DebugUtility.HandleErrorIfNullGetComponent<Health, PlayerCharacterController>(m\_Health, this, gameObject);  m\_Actor = GetComponent<Actor>();  DebugUtility.HandleErrorIfNullGetComponent<Actor, PlayerCharacterController>(m\_Actor, this, gameObject);  m\_Controller.enableOverlapRecovery = true;  m\_Health.OnDie += OnDie;  // force the crouch state to false when starting  SetCrouchingState(false, true);  UpdateCharacterHeight(true);  }  void Update()  {  // check for Y kill  if (!IsDead && transform.position.y < KillHeight)  {  m\_Health.Kill();  }  HasJumpedThisFrame = false;  bool wasGrounded = IsGrounded;  GroundCheck();  // landing  if (IsGrounded && !wasGrounded)  {  // Fall damage  float fallSpeed = -Mathf.Min(CharacterVelocity.y, m\_LatestImpactSpeed.y);  float fallSpeedRatio = (fallSpeed - MinSpeedForFallDamage) /  (MaxSpeedForFallDamage - MinSpeedForFallDamage);  if (RecievesFallDamage && fallSpeedRatio > 0f)  {  float dmgFromFall = Mathf.Lerp(FallDamageAtMinSpeed, FallDamageAtMaxSpeed, fallSpeedRatio);  m\_Health.TakeDamage(dmgFromFall, null);  // fall damage SFX  AudioSource.PlayOneShot(FallDamageSfx);  }  else  {  // land SFX  AudioSource.PlayOneShot(LandSfx);  }  }  // crouching  if (m\_InputHandler.GetCrouchInputDown())  {  SetCrouchingState(!IsCrouching, false);  }  UpdateCharacterHeight(false);  HandleCharacterMovement();  }  void OnDie()  {  IsDead = true;  // Tell the weapons manager to switch to a non-existing weapon in order to lower the weapon  m\_WeaponsManager.SwitchToWeaponIndex(-1, true);  EventManager.Broadcast(Events.PlayerDeathEvent);  }  void GroundCheck()  {  // Make sure that the ground check distance while already in air is very small, to prevent suddenly snapping to ground  float chosenGroundCheckDistance =  IsGrounded ? (m\_Controller.skinWidth + GroundCheckDistance) : k\_GroundCheckDistanceInAir;  // reset values before the ground check  IsGrounded = false;  m\_GroundNormal = Vector3.up;  // only try to detect ground if it's been a short amount of time since last jump; otherwise we may snap to the ground instantly after we try jumping  if (Time.time >= m\_LastTimeJumped + k\_JumpGroundingPreventionTime)  {  // if we're grounded, collect info about the ground normal with a downward capsule cast representing our character capsule  if (Physics.CapsuleCast(GetCapsuleBottomHemisphere(), GetCapsuleTopHemisphere(m\_Controller.height),  m\_Controller.radius, Vector3.down, out RaycastHit hit, chosenGroundCheckDistance, GroundCheckLayers,  QueryTriggerInteraction.Ignore))  {  // storing the upward direction for the surface found  m\_GroundNormal = hit.normal;  // Only consider this a valid ground hit if the ground normal goes in the same direction as the character up  // and if the slope angle is lower than the character controller's limit  if (Vector3.Dot(hit.normal, transform.up) > 0f &&  IsNormalUnderSlopeLimit(m\_GroundNormal))  {  IsGrounded = true;  // handle snapping to the ground  if (hit.distance > m\_Controller.skinWidth)  {  m\_Controller.Move(Vector3.down \* hit.distance);  }  }  }  }  }  void HandleCharacterMovement()  {  // horizontal character rotation  {  // rotate the transform with the input speed around its local Y axis  transform.Rotate(  new Vector3(0f, (m\_InputHandler.GetLookInputsHorizontal() \* RotationSpeed \* RotationMultiplier),  0f), Space.Self);  }  // vertical camera rotation  {  // add vertical inputs to the camera's vertical angle  m\_CameraVerticalAngle += m\_InputHandler.GetLookInputsVertical() \* RotationSpeed \* RotationMultiplier;  // limit the camera's vertical angle to min/max  m\_CameraVerticalAngle = Mathf.Clamp(m\_CameraVerticalAngle, -89f, 89f);  // apply the vertical angle as a local rotation to the camera transform along its right axis (makes it pivot up and down)  PlayerCamera.transform.localEulerAngles = new Vector3(m\_CameraVerticalAngle, 0, 0);  }  // character movement handling  bool isSprinting = m\_InputHandler.GetSprintInputHeld();  {  if (isSprinting)  {  isSprinting = SetCrouchingState(false, false);  }  float speedModifier = isSprinting ? SprintSpeedModifier : 1f;  // converts move input to a worldspace vector based on our character's transform orientation  Vector3 worldspaceMoveInput = transform.TransformVector(m\_InputHandler.GetMoveInput());  // handle grounded movement  if (IsGrounded)  {  // calculate the desired velocity from inputs, max speed, and current slope  Vector3 targetVelocity = worldspaceMoveInput \* MaxSpeedOnGround \* speedModifier;  // reduce speed if crouching by crouch speed ratio  if (IsCrouching)  targetVelocity \*= MaxSpeedCrouchedRatio;  targetVelocity = GetDirectionReorientedOnSlope(targetVelocity.normalized, m\_GroundNormal) \*  targetVelocity.magnitude;  // smoothly interpolate between our current velocity and the target velocity based on acceleration speed  CharacterVelocity = Vector3.Lerp(CharacterVelocity, targetVelocity,  MovementSharpnessOnGround \* Time.deltaTime);  // jumping  if (IsGrounded && m\_InputHandler.GetJumpInputDown())  {  // force the crouch state to false  if (SetCrouchingState(false, false))  {  // start by canceling out the vertical component of our velocity  CharacterVelocity = new Vector3(CharacterVelocity.x, 0f, CharacterVelocity.z);  // then, add the jumpSpeed value upwards  CharacterVelocity += Vector3.up \* JumpForce;  // play sound  AudioSource.PlayOneShot(JumpSfx);  // remember last time we jumped because we need to prevent snapping to ground for a short time  m\_LastTimeJumped = Time.time;  HasJumpedThisFrame = true;  // Force grounding to false  IsGrounded = false;  m\_GroundNormal = Vector3.up;  }  }  // footsteps sound  float chosenFootstepSfxFrequency =  (isSprinting ? FootstepSfxFrequencyWhileSprinting : FootstepSfxFrequency);  if (m\_FootstepDistanceCounter >= 1f / chosenFootstepSfxFrequency)  {  m\_FootstepDistanceCounter = 0f;  AudioSource.PlayOneShot(FootstepSfx);  }  // keep track of distance traveled for footsteps sound  m\_FootstepDistanceCounter += CharacterVelocity.magnitude \* Time.deltaTime;  }  // handle air movement  else  {  // add air acceleration  CharacterVelocity += worldspaceMoveInput \* AccelerationSpeedInAir \* Time.deltaTime;  // limit air speed to a maximum, but only horizontally  float verticalVelocity = CharacterVelocity.y;  Vector3 horizontalVelocity = Vector3.ProjectOnPlane(CharacterVelocity, Vector3.up);  horizontalVelocity = Vector3.ClampMagnitude(horizontalVelocity, MaxSpeedInAir \* speedModifier);  CharacterVelocity = horizontalVelocity + (Vector3.up \* verticalVelocity);  // apply the gravity to the velocity  CharacterVelocity += Vector3.down \* GravityDownForce \* Time.deltaTime;  }  }  // apply the final calculated velocity value as a character movement  Vector3 capsuleBottomBeforeMove = GetCapsuleBottomHemisphere();  Vector3 capsuleTopBeforeMove = GetCapsuleTopHemisphere(m\_Controller.height);  m\_Controller.Move(CharacterVelocity \* Time.deltaTime);  // detect obstructions to adjust velocity accordingly  m\_LatestImpactSpeed = Vector3.zero;  if (Physics.CapsuleCast(capsuleBottomBeforeMove, capsuleTopBeforeMove, m\_Controller.radius,  CharacterVelocity.normalized, out RaycastHit hit, CharacterVelocity.magnitude \* Time.deltaTime, -1,  QueryTriggerInteraction.Ignore))  {  // We remember the last impact speed because the fall damage logic might need it  m\_LatestImpactSpeed = CharacterVelocity;  CharacterVelocity = Vector3.ProjectOnPlane(CharacterVelocity, hit.normal);  }  }  // Returns true if the slope angle represented by the given normal is under the slope angle limit of the character controller  bool IsNormalUnderSlopeLimit(Vector3 normal)  {  return Vector3.Angle(transform.up, normal) <= m\_Controller.slopeLimit;  }  // Gets the center point of the bottom hemisphere of the character controller capsule  Vector3 GetCapsuleBottomHemisphere()  {  return transform.position + (transform.up \* m\_Controller.radius);  }  // Gets the center point of the top hemisphere of the character controller capsule  Vector3 GetCapsuleTopHemisphere(float atHeight)  {  return transform.position + (transform.up \* (atHeight - m\_Controller.radius));  }  // Gets a reoriented direction that is tangent to a given slope  public Vector3 GetDirectionReorientedOnSlope(Vector3 direction, Vector3 slopeNormal)  {  Vector3 directionRight = Vector3.Cross(direction, transform.up);  return Vector3.Cross(slopeNormal, directionRight).normalized;  }  void UpdateCharacterHeight(bool force)  {  // Update height instantly  if (force)  {  m\_Controller.height = m\_TargetCharacterHeight;  m\_Controller.center = Vector3.up \* m\_Controller.height \* 0.5f;  PlayerCamera.transform.localPosition = Vector3.up \* m\_TargetCharacterHeight \* CameraHeightRatio;  m\_Actor.AimPoint.transform.localPosition = m\_Controller.center;  }  // Update smooth height  else if (m\_Controller.height != m\_TargetCharacterHeight)  {  // resize the capsule and adjust camera position  m\_Controller.height = Mathf.Lerp(m\_Controller.height, m\_TargetCharacterHeight,  CrouchingSharpness \* Time.deltaTime);  m\_Controller.center = Vector3.up \* m\_Controller.height \* 0.5f;  PlayerCamera.transform.localPosition = Vector3.Lerp(PlayerCamera.transform.localPosition,  Vector3.up \* m\_TargetCharacterHeight \* CameraHeightRatio, CrouchingSharpness \* Time.deltaTime);  m\_Actor.AimPoint.transform.localPosition = m\_Controller.center;  }  }  // returns false if there was an obstruction  bool SetCrouchingState(bool crouched, bool ignoreObstructions)  {  // set appropriate heights  if (crouched)  {  m\_TargetCharacterHeight = CapsuleHeightCrouching;  }  else  {  // Detect obstructions  if (!ignoreObstructions)  {  Collider[] standingOverlaps = Physics.OverlapCapsule(  GetCapsuleBottomHemisphere(),  GetCapsuleTopHemisphere(CapsuleHeightStanding),  m\_Controller.radius,  -1,  QueryTriggerInteraction.Ignore);  foreach (Collider c in standingOverlaps)  {  if (c != m\_Controller)  {  return false;  }  }  }  m\_TargetCharacterHeight = CapsuleHeightStanding;  }  if (OnStanceChanged != null)  {  OnStanceChanged.Invoke(crouched);  }  IsCrouching = crouched;  return true;  }  }  } |   - Health   |  | | --- | | using UnityEngine;  using UnityEngine.Events;  namespace Unity.FPS.Game  {  public class Health : MonoBehaviour  {  [Tooltip("Maximum amount of health")] public float MaxHealth = 10f;  [Tooltip("Health ratio at which the critical health vignette starts appearing")]  public float CriticalHealthRatio = 0.3f;  public UnityAction<float, GameObject> OnDamaged;  public UnityAction<float> OnHealed;  public UnityAction OnDie;  public float CurrentHealth { get; set; }  public bool Invincible { get; set; }  public bool CanPickup() => CurrentHealth < MaxHealth;  public float GetRatio() => CurrentHealth / MaxHealth;  public bool IsCritical() => GetRatio() <= CriticalHealthRatio;  bool m\_IsDead;  void Start()  {  CurrentHealth = MaxHealth;  }  public void Heal(float healAmount)  {  float healthBefore = CurrentHealth;  CurrentHealth += healAmount;  CurrentHealth = Mathf.Clamp(CurrentHealth, 0f, MaxHealth);  // call OnHeal action  float trueHealAmount = CurrentHealth - healthBefore;  if (trueHealAmount > 0f)  {  OnHealed?.Invoke(trueHealAmount);  }  }  public void TakeDamage(float damage, GameObject damageSource)  {  if (Invincible)  return;  float healthBefore = CurrentHealth;  CurrentHealth -= damage;  CurrentHealth = Mathf.Clamp(CurrentHealth, 0f, MaxHealth);  // call OnDamage action  float trueDamageAmount = healthBefore - CurrentHealth;  if (trueDamageAmount > 0f)  {  OnDamaged?.Invoke(trueDamageAmount, damageSource);  }  HandleDeath();  }  public void Kill()  {  CurrentHealth = 0f;  // call OnDamage action  OnDamaged?.Invoke(MaxHealth, null);  HandleDeath();  }  void HandleDeath()  {  if (m\_IsDead)  return;  // call OnDie action  if (CurrentHealth <= 0f)  {  m\_IsDead = true;  OnDie?.Invoke();  }  }  }  } |   - Player Weapons Manager   |  | | --- | | using System.Collections.Generic;  using Unity.FPS.Game;  using UnityEngine;  using UnityEngine.Events;  namespace Unity.FPS.Gameplay  {  [RequireComponent(typeof(PlayerInputHandler))]  public class PlayerWeaponsManager : MonoBehaviour  {  public enum WeaponSwitchState  {  Up,  Down,  PutDownPrevious,  PutUpNew,  }  [Tooltip("List of weapon the player will start with")]  public List<WeaponController> StartingWeapons = new List<WeaponController>();  [Header("References")] [Tooltip("Secondary camera used to avoid seeing weapon go throw geometries")]  public Camera WeaponCamera;  [Tooltip("Parent transform where all weapon will be added in the hierarchy")]  public Transform WeaponParentSocket;  [Tooltip("Position for weapons when active but not actively aiming")]  public Transform DefaultWeaponPosition;  [Tooltip("Position for weapons when aiming")]  public Transform AimingWeaponPosition;  [Tooltip("Position for innactive weapons")]  public Transform DownWeaponPosition;  [Header("Weapon Bob")]  [Tooltip("Frequency at which the weapon will move around in the screen when the player is in movement")]  public float BobFrequency = 10f;  [Tooltip("How fast the weapon bob is applied, the bigger value the fastest")]  public float BobSharpness = 10f;  [Tooltip("Distance the weapon bobs when not aiming")]  public float DefaultBobAmount = 0.05f;  [Tooltip("Distance the weapon bobs when aiming")]  public float AimingBobAmount = 0.02f;  [Header("Weapon Recoil")]  [Tooltip("This will affect how fast the recoil moves the weapon, the bigger the value, the fastest")]  public float RecoilSharpness = 50f;  [Tooltip("Maximum distance the recoil can affect the weapon")]  public float MaxRecoilDistance = 0.5f;  [Tooltip("How fast the weapon goes back to it's original position after the recoil is finished")]  public float RecoilRestitutionSharpness = 10f;  [Header("Misc")] [Tooltip("Speed at which the aiming animatoin is played")]  public float AimingAnimationSpeed = 10f;  [Tooltip("Field of view when not aiming")]  public float DefaultFov = 60f;  [Tooltip("Portion of the regular FOV to apply to the weapon camera")]  public float WeaponFovMultiplier = 1f;  [Tooltip("Delay before switching weapon a second time, to avoid recieving multiple inputs from mouse wheel")]  public float WeaponSwitchDelay = 1f;  [Tooltip("Layer to set FPS weapon gameObjects to")]  public LayerMask FpsWeaponLayer;  public bool IsAiming { get; private set; }  public bool IsPointingAtEnemy { get; private set; }  public int ActiveWeaponIndex { get; private set; }  public UnityAction<WeaponController> OnSwitchedToWeapon;  public UnityAction<WeaponController, int> OnAddedWeapon;  public UnityAction<WeaponController, int> OnRemovedWeapon;  WeaponController[] m\_WeaponSlots = new WeaponController[9]; // 9 available weapon slots  PlayerInputHandler m\_InputHandler;  PlayerCharacterController m\_PlayerCharacterController;  float m\_WeaponBobFactor;  Vector3 m\_LastCharacterPosition;  Vector3 m\_WeaponMainLocalPosition;  Vector3 m\_WeaponBobLocalPosition;  Vector3 m\_WeaponRecoilLocalPosition;  Vector3 m\_AccumulatedRecoil;  float m\_TimeStartedWeaponSwitch;  WeaponSwitchState m\_WeaponSwitchState;  int m\_WeaponSwitchNewWeaponIndex;  void Start()  {  ActiveWeaponIndex = -1;  m\_WeaponSwitchState = WeaponSwitchState.Down;  m\_InputHandler = GetComponent<PlayerInputHandler>();  DebugUtility.HandleErrorIfNullGetComponent<PlayerInputHandler, PlayerWeaponsManager>(m\_InputHandler, this,  gameObject);  m\_PlayerCharacterController = GetComponent<PlayerCharacterController>();  DebugUtility.HandleErrorIfNullGetComponent<PlayerCharacterController, PlayerWeaponsManager>(  m\_PlayerCharacterController, this, gameObject);  SetFov(DefaultFov);  OnSwitchedToWeapon += OnWeaponSwitched;  // Add starting weapons  foreach (var weapon in StartingWeapons)  {  AddWeapon(weapon);  }  SwitchWeapon(true);  }  void Update()  {  // shoot handling  WeaponController activeWeapon = GetActiveWeapon();  if (activeWeapon != null && activeWeapon.IsReloading)  return;  if (activeWeapon != null && m\_WeaponSwitchState == WeaponSwitchState.Up)  {  if (!activeWeapon.AutomaticReload && m\_InputHandler.GetReloadButtonDown() && activeWeapon.CurrentAmmoRatio < 1.0f)  {  IsAiming = false;  activeWeapon.StartReloadAnimation();  return;  }  // handle aiming down sights  IsAiming = m\_InputHandler.GetAimInputHeld();  // handle shooting  bool hasFired = activeWeapon.HandleShootInputs(  m\_InputHandler.GetFireInputDown(),  m\_InputHandler.GetFireInputHeld(),  m\_InputHandler.GetFireInputReleased());  // Handle accumulating recoil  if (hasFired)  {  m\_AccumulatedRecoil += Vector3.back \* activeWeapon.RecoilForce;  m\_AccumulatedRecoil = Vector3.ClampMagnitude(m\_AccumulatedRecoil, MaxRecoilDistance);  }  }  // weapon switch handling  if (!IsAiming &&  (activeWeapon == null || !activeWeapon.IsCharging) &&  (m\_WeaponSwitchState == WeaponSwitchState.Up || m\_WeaponSwitchState == WeaponSwitchState.Down))  {  int switchWeaponInput = m\_InputHandler.GetSwitchWeaponInput();  if (switchWeaponInput != 0)  {  bool switchUp = switchWeaponInput > 0;  SwitchWeapon(switchUp);  }  else  {  switchWeaponInput = m\_InputHandler.GetSelectWeaponInput();  if (switchWeaponInput != 0)  {  if (GetWeaponAtSlotIndex(switchWeaponInput - 1) != null)  SwitchToWeaponIndex(switchWeaponInput - 1);  }  }  }  // Pointing at enemy handling  IsPointingAtEnemy = false;  if (activeWeapon)  {  if (Physics.Raycast(WeaponCamera.transform.position, WeaponCamera.transform.forward, out RaycastHit hit,  1000, -1, QueryTriggerInteraction.Ignore))  {  if (hit.collider.GetComponentInParent<Health>() != null)  {  IsPointingAtEnemy = true;  }  }  }  }  // Update various animated features in LateUpdate because it needs to override the animated arm position  void LateUpdate()  {  UpdateWeaponAiming();  UpdateWeaponBob();  UpdateWeaponRecoil();  UpdateWeaponSwitching();  // Set final weapon socket position based on all the combined animation influences  WeaponParentSocket.localPosition =  m\_WeaponMainLocalPosition + m\_WeaponBobLocalPosition + m\_WeaponRecoilLocalPosition;  }  // Sets the FOV of the main camera and the weapon camera simultaneously  public void SetFov(float fov)  {  m\_PlayerCharacterController.PlayerCamera.fieldOfView = fov;  WeaponCamera.fieldOfView = fov \* WeaponFovMultiplier;  }  // Iterate on all weapon slots to find the next valid weapon to switch to  public void SwitchWeapon(bool ascendingOrder)  {  int newWeaponIndex = -1;  int closestSlotDistance = m\_WeaponSlots.Length;  for (int i = 0; i < m\_WeaponSlots.Length; i++)  {  // If the weapon at this slot is valid, calculate its "distance" from the active slot index (either in ascending or descending order)  // and select it if it's the closest distance yet  if (i != ActiveWeaponIndex && GetWeaponAtSlotIndex(i) != null)  {  int distanceToActiveIndex = GetDistanceBetweenWeaponSlots(ActiveWeaponIndex, i, ascendingOrder);  if (distanceToActiveIndex < closestSlotDistance)  {  closestSlotDistance = distanceToActiveIndex;  newWeaponIndex = i;  }  }  }  // Handle switching to the new weapon index  SwitchToWeaponIndex(newWeaponIndex);  }  // Switches to the given weapon index in weapon slots if the new index is a valid weapon that is different from our current one  public void SwitchToWeaponIndex(int newWeaponIndex, bool force = false)  {  if (force || (newWeaponIndex != ActiveWeaponIndex && newWeaponIndex >= 0))  {  // Store data related to weapon switching animation  m\_WeaponSwitchNewWeaponIndex = newWeaponIndex;  m\_TimeStartedWeaponSwitch = Time.time;  // Handle case of switching to a valid weapon for the first time (simply put it up without putting anything down first)  if (GetActiveWeapon() == null)  {  m\_WeaponMainLocalPosition = DownWeaponPosition.localPosition;  m\_WeaponSwitchState = WeaponSwitchState.PutUpNew;  ActiveWeaponIndex = m\_WeaponSwitchNewWeaponIndex;  WeaponController newWeapon = GetWeaponAtSlotIndex(m\_WeaponSwitchNewWeaponIndex);  if (OnSwitchedToWeapon != null)  {  OnSwitchedToWeapon.Invoke(newWeapon);  }  }  // otherwise, remember we are putting down our current weapon for switching to the next one  else  {  m\_WeaponSwitchState = WeaponSwitchState.PutDownPrevious;  }  }  }  public WeaponController HasWeapon(WeaponController weaponPrefab)  {  // Checks if we already have a weapon coming from the specified prefab  for (var index = 0; index < m\_WeaponSlots.Length; index++)  {  var w = m\_WeaponSlots[index];  if (w != null && w.SourcePrefab == weaponPrefab.gameObject)  {  return w;  }  }  return null;  }  // Updates weapon position and camera FoV for the aiming transition  void UpdateWeaponAiming()  {  if (m\_WeaponSwitchState == WeaponSwitchState.Up)  {  WeaponController activeWeapon = GetActiveWeapon();  if (IsAiming && activeWeapon)  {  m\_WeaponMainLocalPosition = Vector3.Lerp(m\_WeaponMainLocalPosition,  AimingWeaponPosition.localPosition + activeWeapon.AimOffset,  AimingAnimationSpeed \* Time.deltaTime);  SetFov(Mathf.Lerp(m\_PlayerCharacterController.PlayerCamera.fieldOfView,  activeWeapon.AimZoomRatio \* DefaultFov, AimingAnimationSpeed \* Time.deltaTime));  }  else  {  m\_WeaponMainLocalPosition = Vector3.Lerp(m\_WeaponMainLocalPosition,  DefaultWeaponPosition.localPosition, AimingAnimationSpeed \* Time.deltaTime);  SetFov(Mathf.Lerp(m\_PlayerCharacterController.PlayerCamera.fieldOfView, DefaultFov,  AimingAnimationSpeed \* Time.deltaTime));  }  }  }  // Updates the weapon bob animation based on character speed  void UpdateWeaponBob()  {  if (Time.deltaTime > 0f)  {  Vector3 playerCharacterVelocity =  (m\_PlayerCharacterController.transform.position - m\_LastCharacterPosition) / Time.deltaTime;  // calculate a smoothed weapon bob amount based on how close to our max grounded movement velocity we are  float characterMovementFactor = 0f;  if (m\_PlayerCharacterController.IsGrounded)  {  characterMovementFactor =  Mathf.Clamp01(playerCharacterVelocity.magnitude /  (m\_PlayerCharacterController.MaxSpeedOnGround \*  m\_PlayerCharacterController.SprintSpeedModifier));  }  m\_WeaponBobFactor =  Mathf.Lerp(m\_WeaponBobFactor, characterMovementFactor, BobSharpness \* Time.deltaTime);  // Calculate vertical and horizontal weapon bob values based on a sine function  float bobAmount = IsAiming ? AimingBobAmount : DefaultBobAmount;  float frequency = BobFrequency;  float hBobValue = Mathf.Sin(Time.time \* frequency) \* bobAmount \* m\_WeaponBobFactor;  float vBobValue = ((Mathf.Sin(Time.time \* frequency \* 2f) \* 0.5f) + 0.5f) \* bobAmount \*  m\_WeaponBobFactor;  // Apply weapon bob  m\_WeaponBobLocalPosition.x = hBobValue;  m\_WeaponBobLocalPosition.y = Mathf.Abs(vBobValue);  m\_LastCharacterPosition = m\_PlayerCharacterController.transform.position;  }  }  // Updates the weapon recoil animation  void UpdateWeaponRecoil()  {  // if the accumulated recoil is further away from the current position, make the current position move towards the recoil target  if (m\_WeaponRecoilLocalPosition.z >= m\_AccumulatedRecoil.z \* 0.99f)  {  m\_WeaponRecoilLocalPosition = Vector3.Lerp(m\_WeaponRecoilLocalPosition, m\_AccumulatedRecoil,  RecoilSharpness \* Time.deltaTime);  }  // otherwise, move recoil position to make it recover towards its resting pose  else  {  m\_WeaponRecoilLocalPosition = Vector3.Lerp(m\_WeaponRecoilLocalPosition, Vector3.zero,  RecoilRestitutionSharpness \* Time.deltaTime);  m\_AccumulatedRecoil = m\_WeaponRecoilLocalPosition;  }  }  // Updates the animated transition of switching weapons  void UpdateWeaponSwitching()  {  // Calculate the time ratio (0 to 1) since weapon switch was triggered  float switchingTimeFactor = 0f;  if (WeaponSwitchDelay == 0f)  {  switchingTimeFactor = 1f;  }  else  {  switchingTimeFactor = Mathf.Clamp01((Time.time - m\_TimeStartedWeaponSwitch) / WeaponSwitchDelay);  }  // Handle transiting to new switch state  if (switchingTimeFactor >= 1f)  {  if (m\_WeaponSwitchState == WeaponSwitchState.PutDownPrevious)  {  // Deactivate old weapon  WeaponController oldWeapon = GetWeaponAtSlotIndex(ActiveWeaponIndex);  if (oldWeapon != null)  {  oldWeapon.ShowWeapon(false);  }  ActiveWeaponIndex = m\_WeaponSwitchNewWeaponIndex;  switchingTimeFactor = 0f;  // Activate new weapon  WeaponController newWeapon = GetWeaponAtSlotIndex(ActiveWeaponIndex);  if (OnSwitchedToWeapon != null)  {  OnSwitchedToWeapon.Invoke(newWeapon);  }  if (newWeapon)  {  m\_TimeStartedWeaponSwitch = Time.time;  m\_WeaponSwitchState = WeaponSwitchState.PutUpNew;  }  else  {  // if new weapon is null, don't follow through with putting weapon back up  m\_WeaponSwitchState = WeaponSwitchState.Down;  }  }  else if (m\_WeaponSwitchState == WeaponSwitchState.PutUpNew)  {  m\_WeaponSwitchState = WeaponSwitchState.Up;  }  }  // Handle moving the weapon socket position for the animated weapon switching  if (m\_WeaponSwitchState == WeaponSwitchState.PutDownPrevious)  {  m\_WeaponMainLocalPosition = Vector3.Lerp(DefaultWeaponPosition.localPosition,  DownWeaponPosition.localPosition, switchingTimeFactor);  }  else if (m\_WeaponSwitchState == WeaponSwitchState.PutUpNew)  {  m\_WeaponMainLocalPosition = Vector3.Lerp(DownWeaponPosition.localPosition,  DefaultWeaponPosition.localPosition, switchingTimeFactor);  }  }  // Adds a weapon to our inventory  public bool AddWeapon(WeaponController weaponPrefab)  {  // if we already hold this weapon type (a weapon coming from the same source prefab), don't add the weapon  if (HasWeapon(weaponPrefab) != null)  {  return false;  }  // search our weapon slots for the first free one, assign the weapon to it, and return true if we found one. Return false otherwise  for (int i = 0; i < m\_WeaponSlots.Length; i++)  {  // only add the weapon if the slot is free  if (m\_WeaponSlots[i] == null)  {  // spawn the weapon prefab as child of the weapon socket  WeaponController weaponInstance = Instantiate(weaponPrefab, WeaponParentSocket);  weaponInstance.transform.localPosition = Vector3.zero;  weaponInstance.transform.localRotation = Quaternion.identity;  // Set owner to this gameObject so the weapon can alter projectile/damage logic accordingly  weaponInstance.Owner = gameObject;  weaponInstance.SourcePrefab = weaponPrefab.gameObject;  weaponInstance.ShowWeapon(false);  // Assign the first person layer to the weapon  int layerIndex =  Mathf.RoundToInt(Mathf.Log(FpsWeaponLayer.value,  2)); // This function converts a layermask to a layer index  foreach (Transform t in weaponInstance.gameObject.GetComponentsInChildren<Transform>(true))  {  t.gameObject.layer = layerIndex;  }  m\_WeaponSlots[i] = weaponInstance;  if (OnAddedWeapon != null)  {  OnAddedWeapon.Invoke(weaponInstance, i);  }  return true;  }  }  // Handle auto-switching to weapon if no weapons currently  if (GetActiveWeapon() == null)  {  SwitchWeapon(true);  }  return false;  }  public bool RemoveWeapon(WeaponController weaponInstance)  {  // Look through our slots for that weapon  for (int i = 0; i < m\_WeaponSlots.Length; i++)  {  // when weapon found, remove it  if (m\_WeaponSlots[i] == weaponInstance)  {  m\_WeaponSlots[i] = null;  if (OnRemovedWeapon != null)  {  OnRemovedWeapon.Invoke(weaponInstance, i);  }  Destroy(weaponInstance.gameObject);  // Handle case of removing active weapon (switch to next weapon)  if (i == ActiveWeaponIndex)  {  SwitchWeapon(true);  }  return true;  }  }  return false;  }  public WeaponController GetActiveWeapon()  {  return GetWeaponAtSlotIndex(ActiveWeaponIndex);  }  public WeaponController GetWeaponAtSlotIndex(int index)  {  // find the active weapon in our weapon slots based on our active weapon index  if (index >= 0 &&  index < m\_WeaponSlots.Length)  {  return m\_WeaponSlots[index];  }  // if we didn't find a valid active weapon in our weapon slots, return null  return null;  }  // Calculates the "distance" between two weapon slot indexes  // For example: if we had 5 weapon slots, the distance between slots #2 and #4 would be 2 in ascending order, and 3 in descending order  int GetDistanceBetweenWeaponSlots(int fromSlotIndex, int toSlotIndex, bool ascendingOrder)  {  int distanceBetweenSlots = 0;  if (ascendingOrder)  {  distanceBetweenSlots = toSlotIndex - fromSlotIndex;  }  else  {  distanceBetweenSlots = -1 \* (toSlotIndex - fromSlotIndex);  }  if (distanceBetweenSlots < 0)  {  distanceBetweenSlots = m\_WeaponSlots.Length + distanceBetweenSlots;  }  return distanceBetweenSlots;  }  void OnWeaponSwitched(WeaponController newWeapon)  {  if (newWeapon != null)  {  newWeapon.ShowWeapon(true);  }  }  }  } |   - JetPack   |  | | --- | | using Unity.FPS.Game;  using UnityEngine;  using UnityEngine.Events;  namespace Unity.FPS.Gameplay  {  [RequireComponent(typeof(AudioSource))]  public class Jetpack : MonoBehaviour  {  [Header("References")] [Tooltip("Audio source for jetpack sfx")]  public AudioSource AudioSource;  [Tooltip("Particles for jetpack vfx")] public ParticleSystem[] JetpackVfx;  [Header("Parameters")] [Tooltip("Whether the jetpack is unlocked at the begining or not")]  public bool IsJetpackUnlockedAtStart = false;  [Tooltip("The strength with which the jetpack pushes the player up")]  public float JetpackAcceleration = 7f;  [Range(0f, 1f)]  [Tooltip(  "This will affect how much using the jetpack will cancel the gravity value, to start going up faster. 0 is not at all, 1 is instant")]  public float JetpackDownwardVelocityCancelingFactor = 1f;  [Header("Durations")] [Tooltip("Time it takes to consume all the jetpack fuel")]  public float ConsumeDuration = 1.5f;  [Tooltip("Time it takes to completely refill the jetpack while on the ground")]  public float RefillDurationGrounded = 2f;  [Tooltip("Time it takes to completely refill the jetpack while in the air")]  public float RefillDurationInTheAir = 5f;  [Tooltip("Delay after last use before starting to refill")]  public float RefillDelay = 1f;  [Header("Audio")] [Tooltip("Sound played when using the jetpack")]  public AudioClip JetpackSfx;  bool m\_CanUseJetpack;  PlayerCharacterController m\_PlayerCharacterController;  PlayerInputHandler m\_InputHandler;  float m\_LastTimeOfUse;  // stored ratio for jetpack resource (1 is full, 0 is empty)  public float CurrentFillRatio { get; private set; }  public bool IsJetpackUnlocked { get; private set; }  public bool IsPlayergrounded() => m\_PlayerCharacterController.IsGrounded;  public UnityAction<bool> OnUnlockJetpack;  void Start()  {  IsJetpackUnlocked = IsJetpackUnlockedAtStart;  m\_PlayerCharacterController = GetComponent<PlayerCharacterController>();  DebugUtility.HandleErrorIfNullGetComponent<PlayerCharacterController, Jetpack>(m\_PlayerCharacterController,  this, gameObject);  m\_InputHandler = GetComponent<PlayerInputHandler>();  DebugUtility.HandleErrorIfNullGetComponent<PlayerInputHandler, Jetpack>(m\_InputHandler, this, gameObject);  CurrentFillRatio = 1f;  AudioSource.clip = JetpackSfx;  AudioSource.loop = true;  }  void Update()  {  // jetpack can only be used if not grounded and jump has been pressed again once in-air  if (IsPlayergrounded())  {  m\_CanUseJetpack = false;  }  else if (!m\_PlayerCharacterController.HasJumpedThisFrame && m\_InputHandler.GetJumpInputDown())  {  m\_CanUseJetpack = true;  }  // jetpack usage  bool jetpackIsInUse = m\_CanUseJetpack && IsJetpackUnlocked && CurrentFillRatio > 0f &&  m\_InputHandler.GetJumpInputHeld();  if (jetpackIsInUse)  {  // store the last time of use for refill delay  m\_LastTimeOfUse = Time.time;  float totalAcceleration = JetpackAcceleration;  // cancel out gravity  totalAcceleration += m\_PlayerCharacterController.GravityDownForce;  if (m\_PlayerCharacterController.CharacterVelocity.y < 0f)  {  // handle making the jetpack compensate for character's downward velocity with bonus acceleration  totalAcceleration += ((-m\_PlayerCharacterController.CharacterVelocity.y / Time.deltaTime) \*  JetpackDownwardVelocityCancelingFactor);  }  // apply the acceleration to character's velocity  m\_PlayerCharacterController.CharacterVelocity += Vector3.up \* totalAcceleration \* Time.deltaTime;  // consume fuel  CurrentFillRatio = CurrentFillRatio - (Time.deltaTime / ConsumeDuration);  for (int i = 0; i < JetpackVfx.Length; i++)  {  var emissionModulesVfx = JetpackVfx[i].emission;  emissionModulesVfx.enabled = true;  }  if (!AudioSource.isPlaying)  AudioSource.Play();  }  else  {  // refill the meter over time  if (IsJetpackUnlocked && Time.time - m\_LastTimeOfUse >= RefillDelay)  {  float refillRate = 1 / (m\_PlayerCharacterController.IsGrounded  ? RefillDurationGrounded  : RefillDurationInTheAir);  CurrentFillRatio = CurrentFillRatio + Time.deltaTime \* refillRate;  }  for (int i = 0; i < JetpackVfx.Length; i++)  {  var emissionModulesVfx = JetpackVfx[i].emission;  emissionModulesVfx.enabled = false;  }  // keeps the ratio between 0 and 1  CurrentFillRatio = Mathf.Clamp01(CurrentFillRatio);  if (AudioSource.isPlaying)  AudioSource.Stop();  }  }  public bool TryUnlock()  {  if (IsJetpackUnlocked)  return false;  OnUnlockJetpack.Invoke(true);  IsJetpackUnlocked = true;  m\_LastTimeOfUse = Time.time;  return true;  }  }  } |   - Actor   |  | | --- | | using UnityEngine;  namespace Unity.FPS.Game  {  // This class contains general information describing an actor (player or enemies).  // It is mostly used for AI detection logic and determining if an actor is friend or foe  public class Actor : MonoBehaviour  {  [Tooltip("Represents the affiliation (or team) of the actor. Actors of the same affiliation are friendly to each other")]  public int Affiliation;  [Tooltip("Represents point where other actors will aim when they attack this actor")]  public Transform AimPoint;  ActorsManager m\_ActorsManager;  void Start()  {  m\_ActorsManager = GameObject.FindObjectOfType<ActorsManager>();  DebugUtility.HandleErrorIfNullFindObject<ActorsManager, Actor>(m\_ActorsManager, this);  // Register as an actor  if (!m\_ActorsManager.Actors.Contains(this))  {  m\_ActorsManager.Actors.Add(this);  }  }  void OnDestroy()  {  // Unregister as an actor  if (m\_ActorsManager)  {  m\_ActorsManager.Actors.Remove(this);  }  }  }  } |   - Damageable   |  | | --- | | using UnityEngine;  namespace Unity.FPS.Game  {  public class Damageable : MonoBehaviour  {  [Tooltip("Multiplier to apply to the received damage")]  public float DamageMultiplier = 1f;  [Range(0, 1)] [Tooltip("Multiplier to apply to self damage")]  public float SensibilityToSelfdamage = 0.5f;  public Health Health { get; private set; }  void Awake()  {  // find the health component either at the same level, or higher in the hierarchy  Health = GetComponent<Health>();  if (!Health)  {  Health = GetComponentInParent<Health>();  }  }  public void InflictDamage(float damage, bool isExplosionDamage, GameObject damageSource)  {  if (Health)  {  var totalDamage = damage;  // skip the crit multiplier if it's from an explosion  if (!isExplosionDamage)  {  totalDamage \*= DamageMultiplier;  }  // potentially reduce damages if inflicted by self  if (Health.gameObject == damageSource)  {  totalDamage \*= SensibilityToSelfdamage;  }  // apply the damages  Health.TakeDamage(totalDamage, damageSource);  }  }  }  } |   <HUD 파트>  - Game Flow Manager   |  | | --- | | using UnityEngine;  using UnityEngine.SceneManagement;  namespace Unity.FPS.Game  {  public class GameFlowManager : MonoBehaviour  {  [Header("Parameters")] [Tooltip("Duration of the fade-to-black at the end of the game")]  public float EndSceneLoadDelay = 3f;  [Tooltip("The canvas group of the fade-to-black screen")]  public CanvasGroup EndGameFadeCanvasGroup;  [Header("Win")] [Tooltip("This string has to be the name of the scene you want to load when winning")]  public string WinSceneName = "WinScene";  [Tooltip("Duration of delay before the fade-to-black, if winning")]  public float DelayBeforeFadeToBlack = 4f;  [Tooltip("Win game message")]  public string WinGameMessage;  [Tooltip("Duration of delay before the win message")]  public float DelayBeforeWinMessage = 2f;  [Tooltip("Sound played on win")] public AudioClip VictorySound;  [Header("Lose")] [Tooltip("This string has to be the name of the scene you want to load when losing")]  public string LoseSceneName = "LoseScene";  public bool GameIsEnding { get; private set; }  float m\_TimeLoadEndGameScene;  string m\_SceneToLoad;  void Awake()  {  EventManager.AddListener<AllObjectivesCompletedEvent>(OnAllObjectivesCompleted);  EventManager.AddListener<PlayerDeathEvent>(OnPlayerDeath);  }  void Start()  {  AudioUtility.SetMasterVolume(1);  }  void Update()  {  if (GameIsEnding)  {  float timeRatio = 1 - (m\_TimeLoadEndGameScene - Time.time) / EndSceneLoadDelay;  EndGameFadeCanvasGroup.alpha = timeRatio;  AudioUtility.SetMasterVolume(1 - timeRatio);  // See if it's time to load the end scene (after the delay)  if (Time.time >= m\_TimeLoadEndGameScene)  {  SceneManager.LoadScene(m\_SceneToLoad);  GameIsEnding = false;  }  }  }  void OnAllObjectivesCompleted(AllObjectivesCompletedEvent evt) => EndGame(true);  void OnPlayerDeath(PlayerDeathEvent evt) => EndGame(false);  void EndGame(bool win)  {  // unlocks the cursor before leaving the scene, to be able to click buttons  Cursor.lockState = CursorLockMode.None;  Cursor.visible = true;  // Remember that we need to load the appropriate end scene after a delay  GameIsEnding = true;  EndGameFadeCanvasGroup.gameObject.SetActive(true);  if (win)  {  m\_SceneToLoad = WinSceneName;  m\_TimeLoadEndGameScene = Time.time + EndSceneLoadDelay + DelayBeforeFadeToBlack;  // play a sound on win  var audioSource = gameObject.AddComponent<AudioSource>();  audioSource.clip = VictorySound;  audioSource.playOnAwake = false;  audioSource.outputAudioMixerGroup = AudioUtility.GetAudioGroup(AudioUtility.AudioGroups.HUDVictory);  audioSource.PlayScheduled(AudioSettings.dspTime + DelayBeforeWinMessage);  // create a game message  //var message = Instantiate(WinGameMessagePrefab).GetComponent<DisplayMessage>();  //if (message)  //{  // message.delayBeforeShowing = delayBeforeWinMessage;  // message.GetComponent<Transform>().SetAsLastSibling();  //}  DisplayMessageEvent displayMessage = Events.DisplayMessageEvent;  displayMessage.Message = WinGameMessage;  displayMessage.DelayBeforeDisplay = DelayBeforeWinMessage;  EventManager.Broadcast(displayMessage);  }  else  {  m\_SceneToLoad = LoseSceneName;  m\_TimeLoadEndGameScene = Time.time + EndSceneLoadDelay;  }  }  void OnDestroy()  {  EventManager.RemoveListener<AllObjectivesCompletedEvent>(OnAllObjectivesCompleted);  EventManager.RemoveListener<PlayerDeathEvent>(OnPlayerDeath);  }  }  } |   - Enemy Manager   |  | | --- | | using System.Collections.Generic;  using Unity.FPS.Game;  using UnityEngine;  namespace Unity.FPS.AI  {  public class EnemyManager : MonoBehaviour  {  public List<EnemyController> Enemies { get; private set; }  public int NumberOfEnemiesTotal { get; private set; }  public int NumberOfEnemiesRemaining => Enemies.Count;  void Awake()  {  Enemies = new List<EnemyController>();  }  public void RegisterEnemy(EnemyController enemy)  {  Enemies.Add(enemy);  NumberOfEnemiesTotal++;  }  public void UnregisterEnemy(EnemyController enemyKilled)  {  int enemiesRemainingNotification = NumberOfEnemiesRemaining - 1;  EnemyKillEvent evt = Events.EnemyKillEvent;  evt.Enemy = enemyKilled.gameObject;  evt.RemainingEnemyCount = enemiesRemainingNotification;  EventManager.Broadcast(evt);  // removes the enemy from the list, so that we can keep track of how many are left on the map  Enemies.Remove(enemyKilled);  }  }  } |   - Actor Manager   |  | | --- | | using System.Collections.Generic;  using UnityEngine;  namespace Unity.FPS.Game  {  public class ActorsManager : MonoBehaviour  {  public List<Actor> Actors { get; private set; }  public GameObject Player { get; private set; }  public void SetPlayer(GameObject player) => Player = player;  void Awake()  {  Actors = new List<Actor>();  }  }  } |   - Object Manager   |  | | --- | | using System.Collections.Generic;  using UnityEngine;  namespace Unity.FPS.Game  {  public class ObjectiveManager : MonoBehaviour  {  List<Objective> m\_Objectives = new List<Objective>();  bool m\_ObjectivesCompleted = false;  void Awake()  {  Objective.OnObjectiveCreated += RegisterObjective;  }  void RegisterObjective(Objective objective) => m\_Objectives.Add(objective);  void Update()  {  if (m\_Objectives.Count == 0 || m\_ObjectivesCompleted)  return;  for (int i = 0; i < m\_Objectives.Count; i++)  {  // pass every objectives to check if they have been completed  if (m\_Objectives[i].IsBlocking())  {  // break the loop as soon as we find one uncompleted objective  return;  }  }  m\_ObjectivesCompleted = true;  EventManager.Broadcast(Events.AllObjectivesCompletedEvent);  }  void OnDestroy()  {  Objective.OnObjectiveCreated -= RegisterObjective;  }  }  } |   <적파트>  - Enemy Controller   |  | | --- | | using System.Collections.Generic;  using Unity.FPS.Game;  using UnityEngine;  using UnityEngine.AI;  using UnityEngine.Events;  namespace Unity.FPS.AI  {  [RequireComponent(typeof(Health), typeof(Actor), typeof(NavMeshAgent))]  public class EnemyController : MonoBehaviour  {  [System.Serializable]  public struct RendererIndexData  {  public Renderer Renderer;  public int MaterialIndex;  public RendererIndexData(Renderer renderer, int index)  {  Renderer = renderer;  MaterialIndex = index;  }  }  [Header("Parameters")]  [Tooltip("The Y height at which the enemy will be automatically killed (if it falls off of the level)")]  public float SelfDestructYHeight = -20f;  [Tooltip("The distance at which the enemy considers that it has reached its current path destination point")]  public float PathReachingRadius = 2f;  [Tooltip("The speed at which the enemy rotates")]  public float OrientationSpeed = 10f;  [Tooltip("Delay after death where the GameObject is destroyed (to allow for animation)")]  public float DeathDuration = 0f;  [Header("Weapons Parameters")] [Tooltip("Allow weapon swapping for this enemy")]  public bool SwapToNextWeapon = false;  [Tooltip("Time delay between a weapon swap and the next attack")]  public float DelayAfterWeaponSwap = 0f;  [Header("Eye color")] [Tooltip("Material for the eye color")]  public Material EyeColorMaterial;  [Tooltip("The default color of the bot's eye")] [ColorUsageAttribute(true, true)]  public Color DefaultEyeColor;  [Tooltip("The attack color of the bot's eye")] [ColorUsageAttribute(true, true)]  public Color AttackEyeColor;  [Header("Flash on hit")] [Tooltip("The material used for the body of the hoverbot")]  public Material BodyMaterial;  [Tooltip("The gradient representing the color of the flash on hit")] [GradientUsageAttribute(true)]  public Gradient OnHitBodyGradient;  [Tooltip("The duration of the flash on hit")]  public float FlashOnHitDuration = 0.5f;  [Header("Sounds")] [Tooltip("Sound played when recieving damages")]  public AudioClip DamageTick;  [Header("VFX")] [Tooltip("The VFX prefab spawned when the enemy dies")]  public GameObject DeathVfx;  [Tooltip("The point at which the death VFX is spawned")]  public Transform DeathVfxSpawnPoint;  [Header("Loot")] [Tooltip("The object this enemy can drop when dying")]  public GameObject LootPrefab;  [Tooltip("The chance the object has to drop")] [Range(0, 1)]  public float DropRate = 1f;  [Header("Debug Display")] [Tooltip("Color of the sphere gizmo representing the path reaching range")]  public Color PathReachingRangeColor = Color.yellow;  [Tooltip("Color of the sphere gizmo representing the attack range")]  public Color AttackRangeColor = Color.red;  [Tooltip("Color of the sphere gizmo representing the detection range")]  public Color DetectionRangeColor = Color.blue;  public UnityAction onAttack;  public UnityAction onDetectedTarget;  public UnityAction onLostTarget;  public UnityAction onDamaged;  List<RendererIndexData> m\_BodyRenderers = new List<RendererIndexData>();  MaterialPropertyBlock m\_BodyFlashMaterialPropertyBlock;  float m\_LastTimeDamaged = float.NegativeInfinity;  RendererIndexData m\_EyeRendererData;  MaterialPropertyBlock m\_EyeColorMaterialPropertyBlock;  public PatrolPath PatrolPath { get; set; }  public GameObject KnownDetectedTarget => DetectionModule.KnownDetectedTarget;  public bool IsTargetInAttackRange => DetectionModule.IsTargetInAttackRange;  public bool IsSeeingTarget => DetectionModule.IsSeeingTarget;  public bool HadKnownTarget => DetectionModule.HadKnownTarget;  public NavMeshAgent NavMeshAgent { get; private set; }  public DetectionModule DetectionModule { get; private set; }  int m\_PathDestinationNodeIndex;  EnemyManager m\_EnemyManager;  ActorsManager m\_ActorsManager;  Health m\_Health;  Actor m\_Actor;  Collider[] m\_SelfColliders;  GameFlowManager m\_GameFlowManager;  bool m\_WasDamagedThisFrame;  float m\_LastTimeWeaponSwapped = Mathf.NegativeInfinity;  int m\_CurrentWeaponIndex;  WeaponController m\_CurrentWeapon;  WeaponController[] m\_Weapons;  NavigationModule m\_NavigationModule;  void Start()  {  m\_EnemyManager = FindObjectOfType<EnemyManager>();  DebugUtility.HandleErrorIfNullFindObject<EnemyManager, EnemyController>(m\_EnemyManager, this);  m\_ActorsManager = FindObjectOfType<ActorsManager>();  DebugUtility.HandleErrorIfNullFindObject<ActorsManager, EnemyController>(m\_ActorsManager, this);  m\_EnemyManager.RegisterEnemy(this);  m\_Health = GetComponent<Health>();  DebugUtility.HandleErrorIfNullGetComponent<Health, EnemyController>(m\_Health, this, gameObject);  m\_Actor = GetComponent<Actor>();  DebugUtility.HandleErrorIfNullGetComponent<Actor, EnemyController>(m\_Actor, this, gameObject);  NavMeshAgent = GetComponent<NavMeshAgent>();  m\_SelfColliders = GetComponentsInChildren<Collider>();  m\_GameFlowManager = FindObjectOfType<GameFlowManager>();  DebugUtility.HandleErrorIfNullFindObject<GameFlowManager, EnemyController>(m\_GameFlowManager, this);  // Subscribe to damage & death actions  m\_Health.OnDie += OnDie;  m\_Health.OnDamaged += OnDamaged;  // Find and initialize all weapons  FindAndInitializeAllWeapons();  var weapon = GetCurrentWeapon();  weapon.ShowWeapon(true);  var detectionModules = GetComponentsInChildren<DetectionModule>();  DebugUtility.HandleErrorIfNoComponentFound<DetectionModule, EnemyController>(detectionModules.Length, this,  gameObject);  DebugUtility.HandleWarningIfDuplicateObjects<DetectionModule, EnemyController>(detectionModules.Length,  this, gameObject);  // Initialize detection module  DetectionModule = detectionModules[0];  DetectionModule.onDetectedTarget += OnDetectedTarget;  DetectionModule.onLostTarget += OnLostTarget;  onAttack += DetectionModule.OnAttack;  var navigationModules = GetComponentsInChildren<NavigationModule>();  DebugUtility.HandleWarningIfDuplicateObjects<DetectionModule, EnemyController>(detectionModules.Length,  this, gameObject);  // Override navmesh agent data  if (navigationModules.Length > 0)  {  m\_NavigationModule = navigationModules[0];  NavMeshAgent.speed = m\_NavigationModule.MoveSpeed;  NavMeshAgent.angularSpeed = m\_NavigationModule.AngularSpeed;  NavMeshAgent.acceleration = m\_NavigationModule.Acceleration;  }  foreach (var renderer in GetComponentsInChildren<Renderer>(true))  {  for (int i = 0; i < renderer.sharedMaterials.Length; i++)  {  if (renderer.sharedMaterials[i] == EyeColorMaterial)  {  m\_EyeRendererData = new RendererIndexData(renderer, i);  }  if (renderer.sharedMaterials[i] == BodyMaterial)  {  m\_BodyRenderers.Add(new RendererIndexData(renderer, i));  }  }  }  m\_BodyFlashMaterialPropertyBlock = new MaterialPropertyBlock();  // Check if we have an eye renderer for this enemy  if (m\_EyeRendererData.Renderer != null)  {  m\_EyeColorMaterialPropertyBlock = new MaterialPropertyBlock();  m\_EyeColorMaterialPropertyBlock.SetColor("\_EmissionColor", DefaultEyeColor);  m\_EyeRendererData.Renderer.SetPropertyBlock(m\_EyeColorMaterialPropertyBlock,  m\_EyeRendererData.MaterialIndex);  }  }  void Update()  {  EnsureIsWithinLevelBounds();  DetectionModule.HandleTargetDetection(m\_Actor, m\_SelfColliders);  Color currentColor = OnHitBodyGradient.Evaluate((Time.time - m\_LastTimeDamaged) / FlashOnHitDuration);  m\_BodyFlashMaterialPropertyBlock.SetColor("\_EmissionColor", currentColor);  foreach (var data in m\_BodyRenderers)  {  data.Renderer.SetPropertyBlock(m\_BodyFlashMaterialPropertyBlock, data.MaterialIndex);  }  m\_WasDamagedThisFrame = false;  }  void EnsureIsWithinLevelBounds()  {  // at every frame, this tests for conditions to kill the enemy  if (transform.position.y < SelfDestructYHeight)  {  Destroy(gameObject);  return;  }  }  void OnLostTarget()  {  onLostTarget.Invoke();  // Set the eye attack color and property block if the eye renderer is set  if (m\_EyeRendererData.Renderer != null)  {  m\_EyeColorMaterialPropertyBlock.SetColor("\_EmissionColor", DefaultEyeColor);  m\_EyeRendererData.Renderer.SetPropertyBlock(m\_EyeColorMaterialPropertyBlock,  m\_EyeRendererData.MaterialIndex);  }  }  void OnDetectedTarget()  {  onDetectedTarget.Invoke();  // Set the eye default color and property block if the eye renderer is set  if (m\_EyeRendererData.Renderer != null)  {  m\_EyeColorMaterialPropertyBlock.SetColor("\_EmissionColor", AttackEyeColor);  m\_EyeRendererData.Renderer.SetPropertyBlock(m\_EyeColorMaterialPropertyBlock,  m\_EyeRendererData.MaterialIndex);  }  }  public void OrientTowards(Vector3 lookPosition)  {  Vector3 lookDirection = Vector3.ProjectOnPlane(lookPosition - transform.position, Vector3.up).normalized;  if (lookDirection.sqrMagnitude != 0f)  {  Quaternion targetRotation = Quaternion.LookRotation(lookDirection);  transform.rotation =  Quaternion.Slerp(transform.rotation, targetRotation, Time.deltaTime \* OrientationSpeed);  }  }  bool IsPathValid()  {  return PatrolPath && PatrolPath.PathNodes.Count > 0;  }  public void ResetPathDestination()  {  m\_PathDestinationNodeIndex = 0;  }  public void SetPathDestinationToClosestNode()  {  if (IsPathValid())  {  int closestPathNodeIndex = 0;  for (int i = 0; i < PatrolPath.PathNodes.Count; i++)  {  float distanceToPathNode = PatrolPath.GetDistanceToNode(transform.position, i);  if (distanceToPathNode < PatrolPath.GetDistanceToNode(transform.position, closestPathNodeIndex))  {  closestPathNodeIndex = i;  }  }  m\_PathDestinationNodeIndex = closestPathNodeIndex;  }  else  {  m\_PathDestinationNodeIndex = 0;  }  }  public Vector3 GetDestinationOnPath()  {  if (IsPathValid())  {  return PatrolPath.GetPositionOfPathNode(m\_PathDestinationNodeIndex);  }  else  {  return transform.position;  }  }  public void SetNavDestination(Vector3 destination)  {  if (NavMeshAgent)  {  NavMeshAgent.SetDestination(destination);  }  }  public void UpdatePathDestination(bool inverseOrder = false)  {  if (IsPathValid())  {  // Check if reached the path destination  if ((transform.position - GetDestinationOnPath()).magnitude <= PathReachingRadius)  {  // increment path destination index  m\_PathDestinationNodeIndex =  inverseOrder ? (m\_PathDestinationNodeIndex - 1) : (m\_PathDestinationNodeIndex + 1);  if (m\_PathDestinationNodeIndex < 0)  {  m\_PathDestinationNodeIndex += PatrolPath.PathNodes.Count;  }  if (m\_PathDestinationNodeIndex >= PatrolPath.PathNodes.Count)  {  m\_PathDestinationNodeIndex -= PatrolPath.PathNodes.Count;  }  }  }  }  void OnDamaged(float damage, GameObject damageSource)  {  // test if the damage source is the player  if (damageSource && !damageSource.GetComponent<EnemyController>())  {  // pursue the player  DetectionModule.OnDamaged(damageSource);    onDamaged?.Invoke();  m\_LastTimeDamaged = Time.time;    // play the damage tick sound  if (DamageTick && !m\_WasDamagedThisFrame)  AudioUtility.CreateSFX(DamageTick, transform.position, AudioUtility.AudioGroups.DamageTick, 0f);    m\_WasDamagedThisFrame = true;  }  }  void OnDie()  {  // spawn a particle system when dying  var vfx = Instantiate(DeathVfx, DeathVfxSpawnPoint.position, Quaternion.identity);  Destroy(vfx, 5f);  // tells the game flow manager to handle the enemy destuction  m\_EnemyManager.UnregisterEnemy(this);  // loot an object  if (TryDropItem())  {  Instantiate(LootPrefab, transform.position, Quaternion.identity);  }  // this will call the OnDestroy function  Destroy(gameObject, DeathDuration);  }  void OnDrawGizmosSelected()  {  // Path reaching range  Gizmos.color = PathReachingRangeColor;  Gizmos.DrawWireSphere(transform.position, PathReachingRadius);  if (DetectionModule != null)  {  // Detection range  Gizmos.color = DetectionRangeColor;  Gizmos.DrawWireSphere(transform.position, DetectionModule.DetectionRange);  // Attack range  Gizmos.color = AttackRangeColor;  Gizmos.DrawWireSphere(transform.position, DetectionModule.AttackRange);  }  }  public void OrientWeaponsTowards(Vector3 lookPosition)  {  for (int i = 0; i < m\_Weapons.Length; i++)  {  // orient weapon towards player  Vector3 weaponForward = (lookPosition - m\_Weapons[i].WeaponRoot.transform.position).normalized;  m\_Weapons[i].transform.forward = weaponForward;  }  }  public bool TryAtack(Vector3 enemyPosition)  {  if (m\_GameFlowManager.GameIsEnding)  return false;  OrientWeaponsTowards(enemyPosition);  if ((m\_LastTimeWeaponSwapped + DelayAfterWeaponSwap) >= Time.time)  return false;  // Shoot the weapon  bool didFire = GetCurrentWeapon().HandleShootInputs(false, true, false);  if (didFire && onAttack != null)  {  onAttack.Invoke();  if (SwapToNextWeapon && m\_Weapons.Length > 1)  {  int nextWeaponIndex = (m\_CurrentWeaponIndex + 1) % m\_Weapons.Length;  SetCurrentWeapon(nextWeaponIndex);  }  }  return didFire;  }  public bool TryDropItem()  {  if (DropRate == 0 || LootPrefab == null)  return false;  else if (DropRate == 1)  return true;  else  return (Random.value <= DropRate);  }  void FindAndInitializeAllWeapons()  {  // Check if we already found and initialized the weapons  if (m\_Weapons == null)  {  m\_Weapons = GetComponentsInChildren<WeaponController>();  DebugUtility.HandleErrorIfNoComponentFound<WeaponController, EnemyController>(m\_Weapons.Length, this,  gameObject);  for (int i = 0; i < m\_Weapons.Length; i++)  {  m\_Weapons[i].Owner = gameObject;  }  }  }  public WeaponController GetCurrentWeapon()  {  FindAndInitializeAllWeapons();  // Check if no weapon is currently selected  if (m\_CurrentWeapon == null)  {  // Set the first weapon of the weapons list as the current weapon  SetCurrentWeapon(0);  }  DebugUtility.HandleErrorIfNullGetComponent<WeaponController, EnemyController>(m\_CurrentWeapon, this,  gameObject);  return m\_CurrentWeapon;  }  void SetCurrentWeapon(int index)  {  m\_CurrentWeaponIndex = index;  m\_CurrentWeapon = m\_Weapons[m\_CurrentWeaponIndex];  if (SwapToNextWeapon)  {  m\_LastTimeWeaponSwapped = Time.time;  }  else  {  m\_LastTimeWeaponSwapped = Mathf.NegativeInfinity;  }  }  }  } | |