BLM5026 Bilgisayar Oyunlarında Yapay Zeka

Ekin Doğucan Akkaya

25435004005

Proje Demosu Raporu

Demonun kısaca hedefleri: Çalışan bir ortam ve en az bir adet uygulanmış AI bileşeni göstermek.

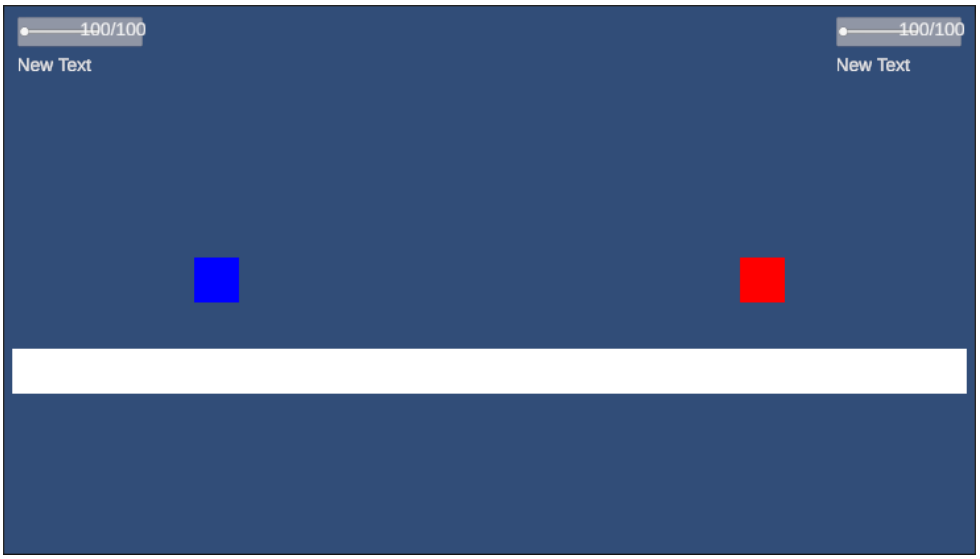
Ödevin bulunduğu github linki: https://github.com/EkinAkkaya0/BLM5026_Project

Genel Bakış

Bu projenin amacı üç farklı yapay zeka tekniğini kullanarak bir Unity oyunu tasarlamaktır. Benim oyunum İki boyutlu(2D) bir düzlemde var olan Mortal Kombat/Tekken/Street Fighter benzeri bir dövüş oyunudur. Şu ana kadar projemde sabit bir zemin, bir adet player ve bir adet enemy olmak üzere iki karakter ve bu karakterlerin can barlarını, kaç canları kaldığını ve en son hamlelerini gösteren basit bir UI düzeni bulunmaktadır. Proje önerisinde belirttiğim yapay zeka tekniklerinden ilki olan A* Algoritmasını Pathfinding görevini yerine getirmesi için eklemiş bulunmaktayım. Bu sayede enemy karakteri düzlemdeki noktaları(node) inceleyerek belirtilen minimum ve maksimum mesafeler arasında player karakterine yaklaşmakta veya uzaklaşmaktadır. Şimdilik yaptığı saldırılar ve bloklamalar tamamen basit olasılık hesabıyla yapılmaktadır.

Oyun Ortamı ve Karakterler

Oyun yatay bir zeminde geçen basit bir arena düzenine sahiptir. Zemin tek bir hatta düz bir şekilde uzanmaktadır ve iki ucunda karakterlerin zeminden dışarı çıkmasını engelleyen görünmez duvarlar bulunmaktadır. Oyunda iki karakter yer almaktadır. Biri oyuncunun kontrol ettiği Player(mavi) karakteri, diğeri ise yapay zekanın kontrol edeceği(Şimdilik sadece A* algoritması uygulanmıştır) Enemy(kırmızı) karakteridir. Her iki karakter de geçici olarak kare sprite'larla temsil edilmektedir.



Player Karakteri

Yürüme mekanizması

Player karakterinin oyun içindeki tüm hareketleri oyuncunun klavye girdileriyle gerçekleştirilmektedir. Hareket sistemi yürüyüş, zıplama ve blok/saldırı olmak üzere üç ana bölümden oluşmaktadır.

Hareket girişleri Unity'nin `Input.GetAxisRaw("Horizontal")` komutuyla alınır.

- **A** veya **<-**(sol ok) tuşu ile sola hareket edilir.
- **D** veya **->**(sağ ok) tuşu ile sağa hareket edilir.

Bu değerler PlayerController script'inde inputX değişkeni üzerinden Rigidbody2D bileşenine aktarılır. Böylece karakterin yatay doğrultuda kaymadan ilerlemesi sağlanır.

Zıplama mekanizması

Player karakteri sadece yere temas ederken zıplayabilir. Bunun amacı defalarca zıplama tuşuna basarak havada kalmayı veya ekrandan çıkacak seviyede yükselmeyi engellemektir. Bunun kontrolünü alt tarafa yerleştirilmiş küçük bir groundCheck nesnesi sağlamaktadır. Bu noktanın çevresi bir çember şeklinde kontrol edilerek karakterin zeminde olup olmadığı belirlenir. Yani özetle **Space** tuşuna basıldığında Player karakteri yerdeyse RigidBody Componentine yukarı yönlü bir kuvvet uygulanır, yani karakter zıplar.

Savunma mekanizması

Player karakteri **Shift** tuşuna basılı tuttuğunda blok(savunma) moduna geçer. Bu sırada aşağıdaki süreçler gerçekleşir;

- Karakterin rengi değişir(cyan olur).
- Hasar alma oranı düşer(Oran miktarı değiştirilebilir).
- isBlocking değişkeni aktif hale gelir.

Bu bilgi daha sonra saldırı alındığında `PlayerCombat.ReceiveDamage()` fonksiyonunda kontrol edilir ve hasar değeri azaltılarak uygulanır.

Saldırı mekanizması

Player karakteri, light ve heavy olmak üzere iki farklı saldırı yapabilmektedir. Bu saldırılar PlayerCombat scripti ile gerçekleştirilir. Saldırı sırasında karakterin önünde bulunan AttackPoint isimli küçük bir nokta referans olarak alınır. Bu noktanın etrafında belirli çapta bir çember oluşturulur ve Enemy karakteri bu çember içerisine girdiğinde hasar almış olur.

- **J** tuşu ile hafif saldırı yapılır. Bu saldırının bekleme süresi daha kısadır ama daha az hasar verir.
- **K** tuşu ile ağır saldırı yapılır. Bu saldırının bekleme süresi daha uzundur ama daha yüksek hasar verir.

Takip mekanizması

Player karakteri her zaman Enemy karakterine bakar. Player karakterinin yönü, düşmanın hangi tarafta olduğuna göre otomatik olarak çevrilmektedir. Bunu sağlayan şey localScale.x değerinin pozitif veya negatif yapılmasıdır. Bu sayede saldırı noktası(AttackPoint) her zaman doğru tarafa bakar.

Player Karakterinin Kod Yapısı

Player karakterini iki ana script yönetmektedir. Bunlardan biri PlayerController.cs diğeri ise PlayerCombat.cs dosyasıdır. Yukarıda bahsedilen tüm mekanizmalar bu iki script tarafından gerçekleştirilmektedir.

PlayerController.cs

Player karakterinin hareket kontrollerini bu script sağlar. Yaptığı ve kontrol ettiği mekanizmalar şu şekildedir;

- Yürüme ve Zıplama girişlerini okur.
- GroundCheck kontrolünü yapar.
- Karakterin nereye bakacağını belirler.
- Block durumunu Kontrol eder.

PlayerCombat.cs

Player karakterinin dövüş hareketlerini bu script sağlar. Yaptığı ve kontrol ettiği mekanizmalar şu şekildedir;

- Light ve Heavy saldırıların zamanlamasını yönetir.
- Hitbox kontrolü yapar.
- Enemy karakterine hasar gönderir.
- Player blok durumunu hasar alırken kontrol eder.
- UI'daki son aksiyon yazısını günceller.

Player Karakterinin Kodları

PlayerController.cs

```
using UnityEngine;

public class PlayerController : MonoBehaviour
{
    [Header("Movement")]
    public float moveSpeed = 7f;
    public float jumpForce = 12f;

    [Header("Ground Check")]
    public Transform groundCheck;
    public float groundCheckRadius = 0.1f;
    public LayerMask groundLayer;

    private Rigidbody2D rb;
    private float inputX;
    private bool isGrounded;

    private Transform enemy;
    private Vector3 initialScale;


    private void Awake()
    {
        rb = GetComponent<Rigidbody2D>();
        initialScale = transform.localScale;
    }

     private void Start()
    {
        // Sahnedeki Enemy'yi tag'den bul
        GameObject enemyObj = GameObject.FindGameObjectWithTag("Enemy");
        if (enemyObj != null)
        {
            enemy = enemyObj.transform;
        }
    }

    private void Update()
    {

        inputX = Input.GetAxisRaw("Horizontal");


        // Yerde miyiz?
        if (groundCheck != null)
        {
            isGrounded = Physics2D.OverlapCircle(
                groundCheck.position,
                groundCheckRadius,
                groundLayer
            );
        }

        // Zıplama
        if (Input.GetButtonDown("Jump") && isGrounded)
        {
            rb.linearVelocity = new Vector2(rb.linearVelocity.x, jumpForce);
        }

          if (enemy != null)
        {
            float dx = enemy.position.x - transform.position.x;

            // Çok yakınken deli gibi flip atmasın diye küçük bir eşik
            if (Mathf.Abs(dx) > 0.05f)
            {
                float dirToEnemy = Mathf.Sign(dx);

                if (dirToEnemy > 0)
                {
                    // Enemy sağda → sağa bak
                    transform.localScale = new Vector3(
                        Mathf.Abs(initialScale.x),
                        initialScale.y,
                        initialScale.z
```

```
                    );
                }
                else
                {
                    // Enemy solda → sola bak
                    transform.localScale = new Vector3(
                        -Mathf.Abs(initialScale.x),
                        initialScale.y,
                        initialScale.z
                    );
                }
            }
        }
    }

    private void FixedUpdate()
    {
        // Yürüyüş
        rb.linearVelocity = new Vector2(inputX * moveSpeed, rb.linearVelocity.y);
    }

    private void OnDrawGizmosSelected()
    {
        // GroundCheck gizmo (editor'da küçük bir daire görebilirsin)
        if (groundCheck != null)
        {
            Gizmos.DrawWireSphere(groundCheck.position, groundCheckRadius);
        }
    }
}
```

PlayerCombat.cs

```
using UnityEngine;

public class PlayerCombat : MonoBehaviour
{
    [Header("References")]
    public Transform attackPoint;
    public LayerMask enemyLayers;

    [Header("Attack Settings")]
    public float attackRange = 0.6f;
    public int lightAttackDamage = 10;
    public int heavyAttackDamage = 20;
    public float lightAttackCooldown = 0.2f;
    public float heavyAttackCooldown = 0.6f;

    [Header("Block Settings")]
    public bool isBlocking = false;
    public float blockDamageMultiplier = 0.3f;

    private float nextLightAttackTime = 0f;
    private float nextHeavyAttackTime = 0f;

    private SpriteRenderer sr;
    private FighterHealth health;

    private void Awake()
    {
        sr = GetComponentInChildren<SpriteRenderer>();
        health = GetComponent<FighterHealth>();
    }

    private void Update()
    {
        // BLOCK
        isBlocking = Input.GetKey(KeyCode.LeftShift) || Input.GetKey(KeyCode.RightShift);

        if (sr != null)
        {
            sr.color = isBlocking ? Color.cyan : Color.blue;
        }

        // LIGHT ATTACK: J
```

```
        if (Input.GetKeyDown(KeyCode.J) && Time.time >= nextLightAttackTime)
        {
            nextLightAttackTime = Time.time + lightAttackCooldown;
            PerformAttack(lightAttackDamage, 0.1f, "Light Attack");
        }

        // HEAVY ATTACK: K
        if (Input.GetKeyDown(KeyCode.K) && Time.time >= nextHeavyAttackTime)
        {
            nextHeavyAttackTime = Time.time + heavyAttackCooldown;
            PerformAttack(heavyAttackDamage, 0.2f, "Heavy Attack");
        }
    }

    private void PerformAttack(int damage, float flashTime, string attackName)
{
    if (sr != null)
    {
        StopAllCoroutines();
        StartCoroutine(AttackFlash(flashTime));
    }

    if (attackPoint == null) return;

    Collider2D[] hitEnemies = Physics2D.OverlapCircleAll(
        attackPoint.position,
        attackRange,
        enemyLayers
    );

    int hitCount = 0;

    foreach (Collider2D enemy in hitEnemies)
    {
        // ÖNCE EnemyCombat'e sor (block var mı vs.)
        EnemyCombat enemyCombat = enemy.GetComponent<EnemyCombat>();
        FighterHealth enemyHealth = enemy.GetComponent<FighterHealth>();

        if (enemyCombat != null)
        {
            enemyCombat.ReceiveDamage(damage);
            hitCount++;
        }
        else if (enemyHealth != null)
        {
            // Yedek: block sistemi yoksa direkt can düşsün
            enemyHealth.TakeDamage(damage);
            hitCount++;
        }
    }

    if (hitCount > 0)
    {
        FighterUI.PlayerUI?.SetLastAction($"{attackName}: {damage} dmg (hit x{hitCount})");
    }
    else
    {
        FighterUI.PlayerUI?.SetLastAction($"{attackName}: missed");
    }
}



    private System.Collections.IEnumerator AttackFlash(float time)
    {
        Color original = sr.color;
        sr.color = Color.yellow;
        yield return new WaitForSeconds(time);
        sr.color = original;
    }

    public void ReceiveDamage(int amount)
    {
        if (health == null) return;

        int finalDamage = amount;

        if (isBlocking)
        {
            finalDamage = Mathf.CeilToInt(amount * blockDamageMultiplier);
        }
```

```
        health.TakeDamage(finalDamage);

        if (isBlocking)
        {
            FighterUI.PlayerUI?.SetLastAction($"Blocked: received {finalDamage} dmg (reduced)");
        }
        else
        {
            FighterUI.PlayerUI?.SetLastAction($"Got hit: {finalDamage} dmg");
        }
    }

    private void OnDrawGizmosSelected()
    {
        if (attackPoint == null) return;
        Gizmos.DrawWireSphere(attackPoint.position, attackRange);
    }
}
```

Enemy Karakteri

Hareket mekanizması

Enemy karakteri EnemyController.cs scripti sayesinde oyuncunun konumunu sürekli takip eder ve oyuncuyla arasındaki mesafeyi kontrol eder. Bunu yaparken manuel bir input yerine sahne üzerinde yerleştirilmiş PathNode'lar ile A* algoritmasını kullanır. Bu mekanizma ile enemy, oyuncuya yaklaşması veya uzaklaşması gerektiğini belirler.

EnemyController scripti;

- Oyuncuya olan yatay uzaklığı ölçer.
- Bu mesafe çok yakınsa geri çekilmeye çalışır, çok uzunsa oyuncuya doğru ilerler, ideal aralıktaysa olduğu yerde kalır.
- Gerekli durumlarda yeni bir path bulur ve node'lar arasında ilerleyerek pozisyon alır.

Zıplama mekanizması

Enemy karakteri de Player karakterindeki aynı zıplama mekanizmasına sahiptir. Yani sadece yere temas ederken zıplayabilmektedir. Bu amaçla Player karakterindeki GroundCheck gibi EnemyGroundCheck isimli bir nesne kullanılmaktadır. EnemyController, groundCheck noktasının zemine temas edip etmediğini kontrol ederek enemy karakteri havada değilken zıplama kararı verebilmesine olanak sağlar. Bu zıplama hareketi bir cooldown mantığına bağlıdır bu sayede sürekli art arda zıplamaların önüne geçilir.

Saldırı mekanizması

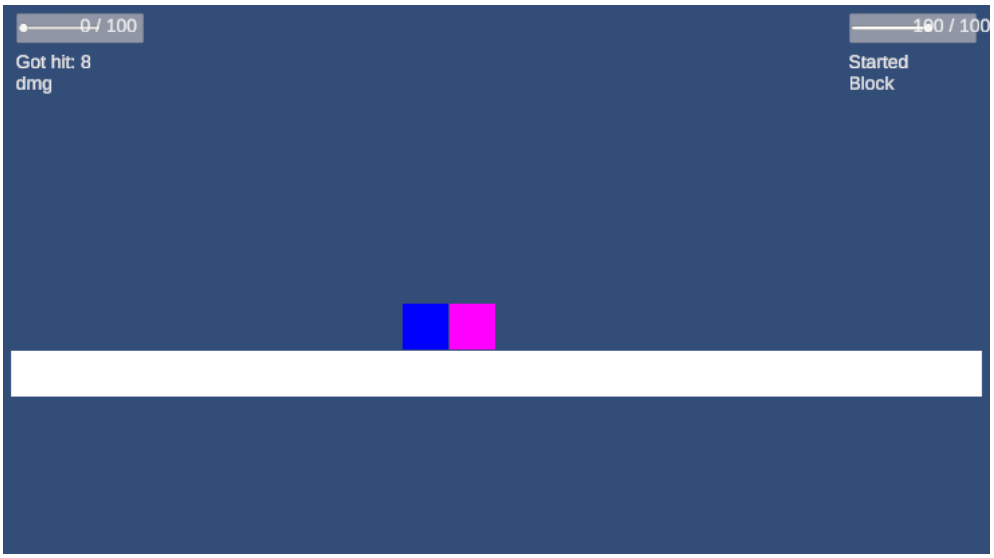
Enemy karakteri, oyuncu saldırı menziline girdiğinde ve blok yapmıyorsa saldırı gerçekleştirmeyi dener. Bu saldırılar Player'da olduğu gibi light attack ve heavy attack olmak üzere iki türdür. EnemyCombat script'i saldırıların uygulanmasından sorumludur. Saldırı sırasında;

- Enemy karakterinin önünde bulunan AttackPoint referans alınır.
- AttackPoint etrafında bir çember oluşturulur.
- Bu çemberin içinde Player bulunuyorsa hasar uygulanır.
- Light saldırı daha hızlıdır ama az hasar verir, heavy saldırı daha yavaştır ama daha çok hasar verir.
- UI üzerinde "Enemy: Light Attack (hit/miss)" gibi bir bilgilendirme yazısı görünür.

Savunma mekanizması

Enemy karakteri, oyuncu yakın mesafeye girdiğinde savunma moduna geçebilir. Bu süreç tamamen EnemyController tarafından kontrol edilmektedir. Blok moduna girildiğinde;

- Enemy karakterinin rengi magenta olur.
- Gelen hasarın büyük bir kısmı azaltılır.
- Blok sadece belirli bir süre aktif kalır ve sonrasında cooldowna girer. Bu sayede sürekli blok yapılmamış olur.



Takip mekanizması

Enemy karakteri de her zaman Player karakterine bakacak şekilde yönünü otomatik ayarlar. Bu iş Player hareketindeki mantığın aynısıdır. Enemy karakterinin yönü, Player karakterinin Enemy karakterinin hangi tarafında olduğuna göre localScale.x değerinin işaretinin değiştirilmesiyle belirlenir. Bu sayede AttackPoint her zaman doğru yönde kalır.

Enemy Karakterinin Kod Yapısı

EnemyController.cs

Bu script enemy karakterinin yapay zekâ tarafını, yani hangi eylemi ne zaman yapması gerektiğini belirlemektedir. Görevleri şunlardır;

- Player karakterine olan mesafeyi sürekli ölçer.
- Belirlenen ideal dövüş mesafesinde kalmaya çalışır.
- Gerekirse A* pathfinding ile doğru noktaya gider.
- Blok kararlarını verir.
- Taktiksel jump kararlarını verir.
- Light/Heavy saldırıların ne zaman deneneceğine karar verir.
- Enemy karakterinin yönünü Player karakterine göre otomatik çevirir.
- Hareketi Rigidbody2D üzerinden gerçekleştirir.

EnemyCombat.cs

Bu script ise EnemyController scriptinin verdiği kararları fiziksel olarak uygulamaktadır. Görevleri şunlardır;

- Light ve heavy saldırıları gerçekleştirir.
- AttackPoint etrafındaki çember üzerinden Player karakterine hasar verir.
- Blok durumunda gelen hasarı azaltır.
- UI'daki son aksiyon yazılarını günceller.

Enemy Karakterinin Kodları

EnemyController.cs

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

public class EnemyController : MonoBehaviour
{
    [Header("Movement")]
    public float moveSpeed = 6f;

    [Header("Desired Distance To Player")]
    public float minDistance = 1.0f;      // Çok yaklaşma sınırı
    public float maxDistance = 3.0f;      // Çok uzaklaşma sınırı
    public float desiredDistance = 1.0f;  // Tercih edilen ideal mesafe

    [Header("Ground Check")]
    public Transform groundCheck;
    public float groundCheckRadius = 0.1f;
    public LayerMask groundLayer;
    private bool isGrounded;


    [Header("Pathfinding")]
    public PathNodeManager pathManager;
    public float nodeReachThreshold = 0.25f;
    public float repathInterval = 0.3f;

    [Header("Combat")]
    public EnemyCombat combat;

    [Header("Block & Jump")]
    public float blockCloseDistance = 1.5f;
    public float blockDuration = 0.6f;
    public float blockCooldown = 10f;
    public float jumpForce = 10f;
    public float jumpCooldown = 10f;

    private Transform player;
    private Rigidbody2D rb;
    private float moveDir;              // -1: sola, 0: dur, 1: sağa
    private Vector3 initialScale;

    private List<PathNode> currentPath = new List<PathNode>();
    private int currentPathIndex = 0;
    private float nextRepathTime = 0f;

    // Block & jump zamanlayıcıları
    private float blockEndTime = 0f;
    private float nextBlockAllowedTime = 0f;
    private float nextJumpTime = 0f;

    private void Awake()
    {
        rb = GetComponent<Rigidbody2D>();
        initialScale = transform.localScale;
    }

    private void Start()
    {
        GameObject playerObj = GameObject.FindGameObjectWithTag("Player");
        if (playerObj != null)
```

```
    {
        player = playerObj.transform;
    }
    else
    {
        Debug.LogError("EnemyController: 'Player' tag'li obje bulunamadı!");
    }
}

private void Update()
{
    if (player == null) return;

            // Ground check
    if (groundCheck != null)
    {
        isGrounded = Physics2D.OverlapCircle(
            groundCheck.position,
            groundCheckRadius,
            groundLayer
        );
    }


    float distanceX = Mathf.Abs(player.position.x - transform.position.x);
    float directionToPlayer = Mathf.Sign(player.position.x - transform.position.x);

    // Yüze dön: her zaman player'a bak
    if (directionToPlayer > 0)
    {
        transform.localScale = new Vector3(
            Mathf.Abs(initialScale.x),
            initialScale.y,
            initialScale.z
        );
    }
    else if (directionToPlayer < 0)
    {
        transform.localScale = new Vector3(
            -Mathf.Abs(initialScale.x),
            initialScale.y,
            initialScale.z
        );
    }

    bool closeToPlayer = distanceX < blockCloseDistance;

    // ---------- BLOCK LOGIC ----------
    if (combat != null)
    {
        // Block süresi bittiyse block'u kapat
        if (combat.isBlocking && Time.time >= blockEndTime)
        {
            combat.SetBlocking(false);
        }

        // Yeterince yakınsa ara sıra block'a girsin
        if (!combat.isBlocking && closeToPlayer && Time.time >= nextBlockAllowedTime)
        {
            // Basit: %25 ihtimalle block denesin
            if (Random.value < 0.25f)
            {
                combat.SetBlocking(true);
                blockEndTime = Time.time + blockDuration;
                nextBlockAllowedTime = Time.time + blockCooldown;
                FighterUI.EnemyUI?.SetLastAction("Started Block");
            }
        }
    }

            // ---- JUMP KARARI (TAKTİKSEL, SEYREK) ----
    if (combat != null && !combat.isBlocking && isGrounded && Time.time >= nextJumpTime)
    {
        float veryCloseDist = 1.0f;   // çok yakın: kaçış için
        float midDist       = 2.5f;   // bunun altında: atılma için

        bool inAttackRange = distanceX <= (combat.attackRange + 0.2f);
        float rand = Random.value;

        if (inAttackRange)
```

```
        {
            // Saldırı mesafesindeysek: normalde saldır, BAZEN geri zıpla
            if (distanceX < veryCloseDist && rand < 0.10f) // %10 back-jump
            {
                float dirAway = -Mathf.Sign(player.position.x - transform.position.x);
                rb.linearVelocity = new Vector2(dirAway * 5f, jumpForce);

                nextJumpTime = Time.time + jumpCooldown;
                FighterUI.EnemyUI?.SetLastAction("Back Jump (escape)");

                currentPath.Clear();
                currentPathIndex = 0;
            }
        }
        else if (distanceX < midDist)
        {
            // Saldırı mesafesinin biraz dışındayız: BAZEN ileri atılma jump
            if (rand < 0.20f) // %20 forward-jump
            {
                float dirToPlayer = Mathf.Sign(player.position.x - transform.position.x);
                rb.linearVelocity = new Vector2(dirToPlayer * 5f, jumpForce);

                nextJumpTime = Time.time + jumpCooldown;
                FighterUI.EnemyUI?.SetLastAction("Forward Jump (attack)");

                currentPath.Clear();
                currentPathIndex = 0;
            }
        }
    }


    // ---------- PATHFINDING / MESAFE AYARI ----------
    float hysteresis = 0.2f; // ileri-geri titremeyi azaltmak için tampon

    if (Time.time >= nextRepathTime)
    {
        if (distanceX > maxDistance + hysteresis || distanceX < minDistance - hysteresis)
        {
            RequestNewPath();
        }
        else
        {
            // İdeal bandın içindeyiz → path'i temizle, yerinde dur
            currentPath.Clear();
            currentPathIndex = 0;
            moveDir = 0f;
        }

        nextRepathTime = Time.time + repathInterval;
    }

    UpdateMoveDirectionAlongPath();

            // ---------- COMBAT ----------
    if (combat != null && !combat.isBlocking && isGrounded)
    {
        // Saldırı mesafesindeysek saldırmayı dene
        float attackRangeWithMargin = combat.attackRange + 0.2f;

        if (distanceX <= attackRangeWithMargin)
        {
            if (Random.value < 0.7f)
                combat.TryLightAttack();
            else
                combat.TryHeavyAttack();
        }
    }

}

private void FixedUpdate()
{
    // Sadece x ekseninde hareket
    rb.linearVelocity = new Vector2(moveDir * moveSpeed, rb.linearVelocity.y);
}

private void RequestNewPath()
{
    if (pathManager == null || player == null) return;
```

```
        PathNode start = pathManager.GetClosestNode(transform.position);

        // Player'a göre ideal hedef pozisyon
        float dirToPlayer = Mathf.Sign(player.position.x - transform.position.x);
        float prefDist = Mathf.Clamp(desiredDistance, minDistance, maxDistance);

        Vector3 desiredPos = player.position - new Vector3(dirToPlayer * prefDist, 0f, 0f);
        PathNode goal = pathManager.GetClosestNode(desiredPos);

        if (start == null || goal == null)
        {
            currentPath.Clear();
            currentPathIndex = 0;
            return;
        }

        currentPath = pathManager.FindPath(start, goal) ?? new List<PathNode>();
        currentPathIndex = 0;
    }

    private void UpdateMoveDirectionAlongPath()
    {
        // Block halindeyken yürümeyi kes
        if (combat != null && combat.isBlocking)
        {
            moveDir = 0f;
            return;
        }

        if (currentPath == null || currentPath.Count == 0 || currentPathIndex >= currentPath.Count)
        {
            moveDir = 0f;
            return;
        }

        Vector3 targetPos = currentPath[currentPathIndex].transform.position;
        float diffX = targetPos.x - transform.position.x;

        // Hedef node'a yeterince yaklaştıysak bir sonrakine geç
        if (Mathf.Abs(diffX) <= nodeReachThreshold)
        {
            currentPathIndex++;

            if (currentPathIndex >= currentPath.Count)
            {
                moveDir = 0f;
                return;
            }

            targetPos = currentPath[currentPathIndex].transform.position;
            diffX = targetPos.x - transform.position.x;
        }

        moveDir = Mathf.Sign(diffX);

        // Çok küçük farkta ileri-geri zıplamasın
        if (Mathf.Abs(diffX) < 0.01f)
        {
            moveDir = 0f;
        }
    }
}
```

EnemyCombat.cs

```
using UnityEngine;

public class EnemyCombat : MonoBehaviour
{
    [Header("References")]
    public Transform attackPoint;
    public LayerMask playerLayers;

    [Header("Attack Settings")]
    public float attackRange = 1.3f;
    public int lightAttackDamage = 8;
    public int heavyAttackDamage = 16;
    public float lightAttackCooldown = 0.5f;
    public float heavyAttackCooldown = 1.0f;

    [Header("Block Settings")]
    public bool isBlocking = false;
    public float blockDamageMultiplier = 0.3f; // block varken alınan hasar oranı

    private float nextLightAttackTime = 0f;
    private float nextHeavyAttackTime = 0f;

    private SpriteRenderer sr;
    private Transform player;
    private FighterHealth health;

    private void Awake()
    {
        sr = GetComponentInChildren<SpriteRenderer>();
        health = GetComponent<FighterHealth>();
    }

    private void Start()
    {
        GameObject playerObj = GameObject.FindGameObjectWithTag("Player");
        if (playerObj != null)
        {
            player = playerObj.transform;
        }
        else
        {
            Debug.LogError("EnemyCombat: Player not found!");
        }

        UpdateBlockVisual(); // başlangıç rengi
    }

    // ---- BLOCK ----

    public void SetBlocking(bool value)
    {
        isBlocking = value;
        UpdateBlockVisual();
    }

    private void UpdateBlockVisual()
    {
        if (sr == null) return;

        // Block'tayken cyan, değilken kırmızı
        sr.color = isBlocking ? Color.magenta : Color.red;
    }

    public void ReceiveDamage(int amount)
    {
        if (health == null) return;

        int final = isBlocking
            ? Mathf.CeilToInt(amount * blockDamageMultiplier)
            : amount;

        health.TakeDamage(final);

        if (isBlocking)
            FighterUI.EnemyUI?.SetLastAction($"Blocked: received {final} dmg (reduced)");
        else
            FighterUI.EnemyUI?.SetLastAction($"Got hit: {final} dmg");
    }
```

```
// ---- ATTACK ----

public void TryLightAttack()
{
    if (Time.time < nextLightAttackTime) return;
    if (player == null) return;

    float distanceX = Mathf.Abs(player.position.x - transform.position.x);
    if (distanceX > attackRange + 0.2f) return;

    nextLightAttackTime = Time.time + lightAttackCooldown;
    PerformAttack(lightAttackDamage, 0.1f, "Light Attack");
}

public void TryHeavyAttack()
{
    if (Time.time < nextHeavyAttackTime) return;
    if (player == null) return;

    float distanceX = Mathf.Abs(player.position.x - transform.position.x);
    if (distanceX > attackRange + 0.2f) return;

    nextHeavyAttackTime = Time.time + heavyAttackCooldown;
    PerformAttack(heavyAttackDamage, 0.2f, "Heavy Attack");
}

private void PerformAttack(int damage, float flashTime, string attackName)
{
    if (sr != null)
    {
        StopAllCoroutines();
        StartCoroutine(AttackFlash(flashTime));
    }

    if (attackPoint == null)
    {
        Debug.LogWarning("EnemyCombat: AttackPoint not assigned!");
        return;
    }

    Collider2D[] hitPlayers = Physics2D.OverlapCircleAll(
        attackPoint.position,
        attackRange,
        playerLayers
    );

    int hitCount = 0;

    foreach (Collider2D col in hitPlayers)
    {
        PlayerCombat pc = col.GetComponent<PlayerCombat>();
        if (pc != null)
        {
            pc.ReceiveDamage(damage);
            hitCount++;
        }
    }

    if (hitCount > 0)
    {
        FighterUI.EnemyUI?.SetLastAction($"{attackName}: {damage} dmg (hit x{hitCount})");
    }
    else
    {
        FighterUI.EnemyUI?.SetLastAction($"{attackName}: missed");
    }

    Debug.Log($"Enemy attacked. Damage: {damage} | Hit count: {hitCount}");
}

private System.Collections.IEnumerator AttackFlash(float time)
{
    if (sr == null)
        yield break;

    Color original = sr.color;
    sr.color = Color.magenta;
    yield return new WaitForSeconds(time);
    // Block durumuna göre rengi geri çek
```

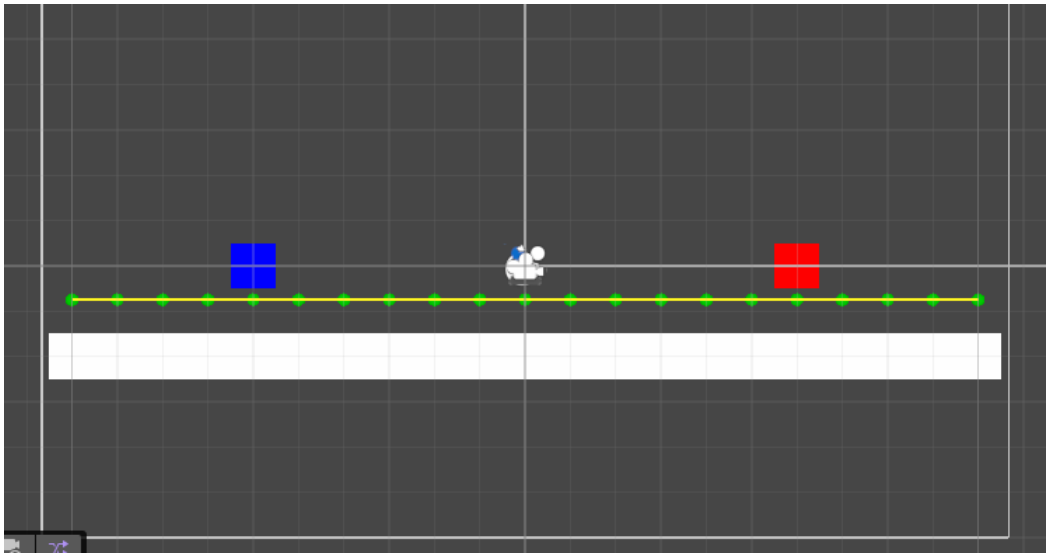
```
        UpdateBlockVisual();
    }

    private void OnDrawGizmosSelected()
    {
        if (attackPoint == null) return;
        Gizmos.DrawWireSphere(attackPoint.position, attackRange);
    }
}
```

Node sistemi ve Pathfinding Yapısı

Oyun arenası tek bir düz zemin üzerinde geçmektedir ve karakterler yatay eksende hareket etmektedir. Enemy karakteri, oyuncuya yaklaşma ve uzaklaşma kararlarını verirken sahne üzerinde yerleştirilmiş node'ları(noktaları) referans olarak kullanır. Bu node sistemi, yapay zekanın arenayı anlamasını ve kendini player karakterine göre uygun bir pozisyonda konumlandırmasını sağlar.

Node'lar zemin üzerinde düzenli aralıklarla yerleştirilmiştir. Her node, zemindeki tek boyutlu hareket hattı üzerinde bir referans noktası gibi çalışır. Enemy karakteri, bu node'ları kullanarak sahnede nereye gitmesi gerektiğini hesaplar. Bu sistem sayesinde;

- Enemy karakteri sahnenin neresinde olduğunu bilir.
- Hedef pozisyona gitmek için en uygun node rotasını seçer.
- Player karakterine yaklaşırken veya uzaklaşırken daha akıcı hareket eder.



PathNodeManager yapısı

Node'ların tamamı PathNodeManager adındaki script tarafından yönetilmektedir. Bu script;

- Tüm node'ları listeler.
- Enemy karakterinin bulunduğu pozisyona en yakın node'u bulur.
- Enemy karakterinin gitmek istediği hedef pozisyona en yakın node'u belirler.
- Bu iki node arasında A* algoritmasıyla en uygun yolu hesaplar.

Bu yapı sayesinde enemy karakteri, sahnede sadece x koordinatlarına bağlı kalmak yerine kendisi için hesaplanmış bir güzergâhı takip eder.

A* (A-Star) Pathfinding mekanizması ve mesafe yönetimi

Enemy karakteri, doğrudan player'a doğru yürü gibi basit bir komutla hareket etmez. Bunun yerine node'lar üzerinde çalışan bir A* algoritması kullanır. A*, iki nokta arasındaki en uygun rotayı bulan klasik bir arama yöntemidir. Bu mekanizma şu şekilde çalışmaktadır;

1. Enemy karakterinin bulunduğu konuma en yakın node belirlenir.
2. Enemy karakterinin gitmesi gereken hedef mesafe hesaplanır.
3. Hedef mesafeye karşılık gelen pozisyona en yakın node hedef olarak alınır.
4. A* algoritması başlangıç node'u ile hedef node arasında en kısa rotayı hesaplar.
5. Enemy bu rotadaki node'ları sırayla takip ederek hareket eder.

Enemy karakterinin amacı her zaman Player karakteri ile arasında ideal bir dövüş mesafesinde durmaktır. Örneğin enemy karakteri çok yaklaştığında geri çekilmek ister, çok uzaklaştığında ise oyuncuya yaklaşır. Bu mantık pathfinding ile birleştiğinde enemy karakteri sadece saldırmak için değil, aynı zamanda dövüş pozisyonunu korumak için sürekli akıllı hareketler yapmış olur. Bunun için üç parametre kullanılır;

- **minDistance:** Player karakterine bu kadar yaklaşıldığında geri çekilmesi gerekir.
- **desiredDistance:** Enemy karakterinin en verimli şekilde dövüşebildiği mesafeyi belirtir.
- **maxDistance:** Enemy karakteri bu sınırdan daha fazla uzaklaşırsa yaklaşması gerekir.

Node Sistemi ve Pathfinding Yapısı Kodları

PathNode.cs

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

public class PathNode : MonoBehaviour
{
    [Tooltip("Bu nodun bağlı olduğu komşu nodelar")]
    public List<PathNode> neighbors = new List<PathNode>();

    private void OnDrawGizmos()
    {
        // Node noktası
        Gizmos.color = Color.green;
        Gizmos.DrawSphere(transform.position, 0.15f);

        // Komşulara çizgi
        Gizmos.color = Color.yellow;
        foreach (var n in neighbors)
        {
            if (n != null)
            {
                Gizmos.DrawLine(transform.position, n.transform.position);
            }
        }
    }
}
```

PathNodeManager.cs

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

public class PathNodeManager : MonoBehaviour
{
    public List<PathNode> nodes = new List<PathNode>();

    private void Awake()
    {
        RefreshNodes();
    }

    private void OnValidate()
    {
        RefreshNodes();
    }

    private void RefreshNodes()
    {
        nodes.Clear();

        // Bu objenin çocuklarındaki tüm PathNode componentlerini topla
        GetComponentsInChildren<PathNode>(includeInactive: true, result: nodes);

        // X pozisyonuna göre sırala (soldan sağa)
        nodes.Sort((a, b) =>
            a.transform.position.x.CompareTo(b.transform.position.x)
        );

        // Lineer komşuluk: her node önceki ve sonraki ile bağlı
        for (int i = 0; i < nodes.Count; i++)
        {
            nodes[i].neighbors.Clear();

            if (i > 0)
                nodes[i].neighbors.Add(nodes[i - 1]);

            if (i < nodes.Count - 1)
                nodes[i].neighbors.Add(nodes[i + 1]);
        }
    }

    // Verilen dünya pozisyonuna en yakın nodu bul
    public PathNode GetClosestNode(Vector3 worldPos)
    {
        PathNode closest = null;
        float bestDist = float.MaxValue;

        foreach (var n in nodes)
        {
            float d = Vector2.SqrMagnitude((Vector2)(n.transform.position - worldPos));
            if (d < bestDist)
            {
                bestDist = d;
                closest = n;
            }
        }

        return closest;
    }

    // --- A* PATHFINDING ---

    public List<PathNode> FindPath(PathNode start, PathNode goal)
    {
        if (start == null || goal == null)
            return null;

        var openSet = new List<PathNode>();
        var closedSet = new HashSet<PathNode>();

        var cameFrom = new Dictionary<PathNode, PathNode>();
        var gScore = new Dictionary<PathNode, float>();
        var fScore = new Dictionary<PathNode, float>();

        // Tüm nodelar için başlangıç skorları
        foreach (var node in nodes)
        {
```

```
            gScore[node] = Mathf.Infinity;
            fScore[node] = Mathf.Infinity;
        }

        gScore[start] = 0f;
        fScore[start] = Heuristic(start, goal);
        openSet.Add(start);
        while (openSet.Count > 0)
        {
            // openSet içinden en düşük fScore'a sahip olanı bul
            PathNode current = openSet[0];
            float bestF = fScore[current];

            for (int i = 1; i < openSet.Count; i++)
            {
                PathNode n = openSet[i];
                float f = fScore[n];
                if (f < bestF)
                {
                    bestF = f;
                    current = n;
                }
            }

            // Hedefe ulaştık
            if (current == goal)
            {
                return ReconstructPath(cameFrom, current);
            }
t
            openSet.Remove(current);
            closedSet.Add(current);

            foreach (var neighbor in current.neighbors)
            {
                if (neighbor == null || closedSet.Contains(neighbor))
                    continue;

                float tentativeG = gScore[current] +
                                   Vector2.Distance(current.transform.position,
                                                    neighbor.transform.position);

                if (!openSet.Contains(neighbor))
                {
                    openSet.Add(neighbor);
                }
                else if (tentativeG >= gScore[neighbor])
                {
                    continue; // Daha iyi bir yol değil
                }

                cameFrom[neighbor] = current;
                gScore[neighbor] = tentativeG;
                fScore[neighbor] = tentativeG + Heuristic(neighbor, goal);
            }
        }

        // Yol bulunamadı
        return null;
    }

    private float Heuristic(PathNode a, PathNode b)
    {
        // Düz çizgi mesafe (bizim lineer arenada zaten gayet yeterli)
        return Vector2.Distance(a.transform.position, b.transform.position);
    }

    private List<PathNode> ReconstructPath(Dictionary<PathNode, PathNode> cameFrom, PathNode current)
    {
        var totalPath = new List<PathNode> { current };

        while (cameFrom.ContainsKey(current))
        {
            current = cameFrom[current];
            totalPath.Insert(0, current);
        }

        return totalPath;
    }
}
```