Data: 24.06.2025 r.

Konrad Kuleta 272293
Mikołaj Lipiński 273024

Projektowanie i programowanie gier

Praca Własna

Przeciwnicy w twoim pokoju

Wykonawcy	Konrad Kuleta Mikołaj Lipiński
Nr indeksu	272293 273024
Wydział	Wydział Informatyki i Telekomunikacji
Termin zajęć	Wtorek (parzyste), 13:15 – 16:15
Numer grupy	Grupa nr 4
Data oddania projektu	-
Ocena końcowa	

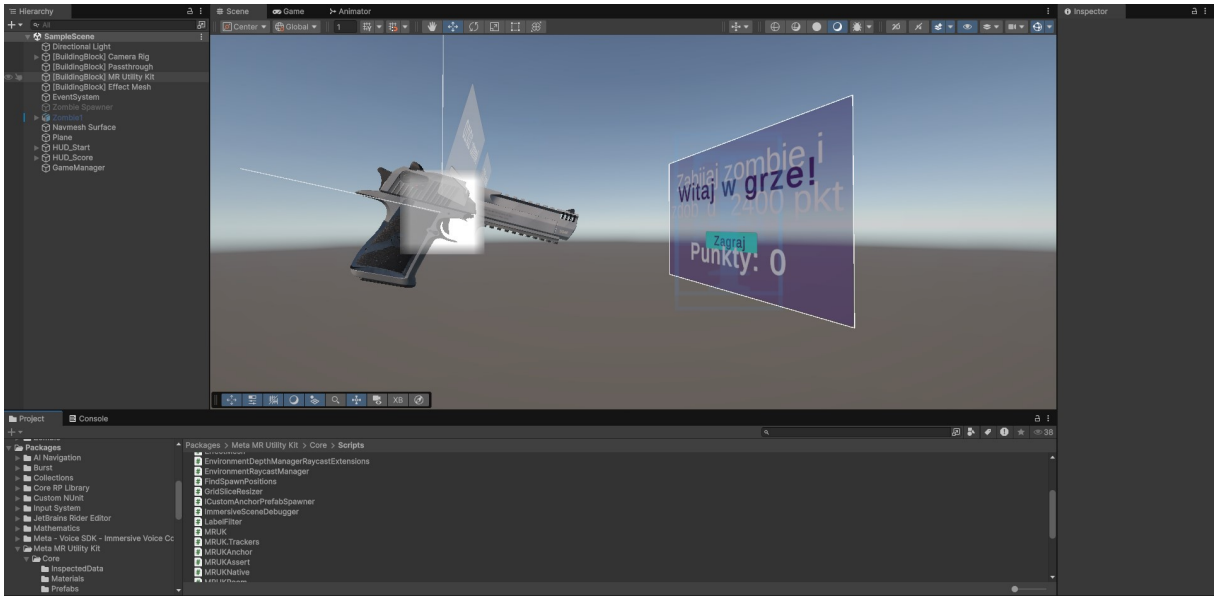
Prowadzący:
dr inż. arch. Tomasz Zamojski

Spis treści

Spis rysunków

1 Wprowadzenie

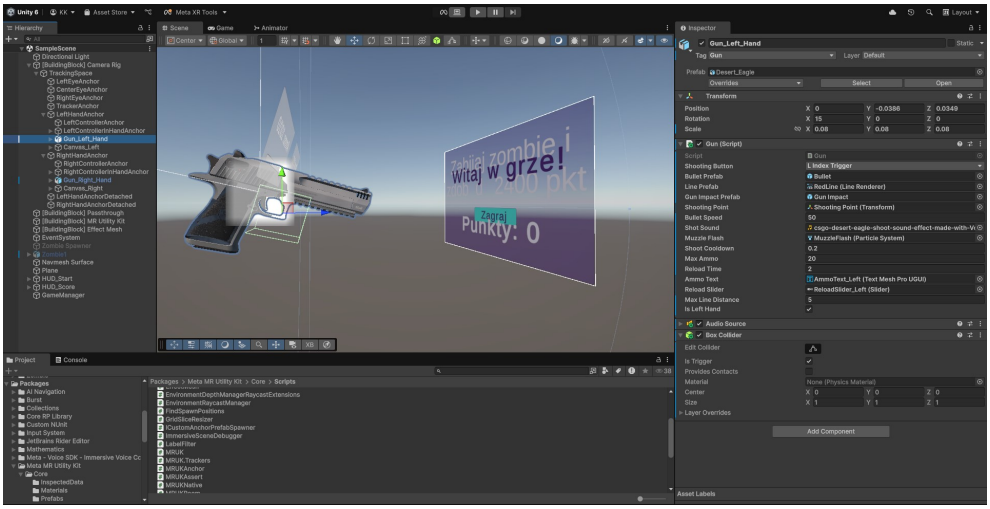
Gra „Przeciwnicy w twoim pokoju” to projekt stworzony w technologii rozszerzonej rzeczywistości (XR), przeznaczony na platformę Meta Quest. Celem gry jest immersyjna rozgrywka w rzeczywistej przestrzeni gracza, w której pojawiają się przeciwnicy – zombie – generowani w czasie rzeczywistym dzięki funkcji passthrough. Gracz, korzystając z wirtualnej broni, musi zdobyć określoną liczbę punktów w jak najkrótszym czasie, eliminując nadchodzące zombie poprzez trafienia w głowę (jednym strzałem) lub dwukrotne trafienie w ciało. Za każdego pokonanego przeciwnika gracz otrzymuje 150 punktów. Rozgrywkę rozpoczyna i kończy interfejs HUD, a podczas gry punkty są aktualizowane na bieżąco. Gra została zaprojektowana z myślą o dostosowaniu do różnych pomieszczeń, pozostawiając graczowi wybór przestrzeni, w której chce się zmierzyć z wyzwaniem.



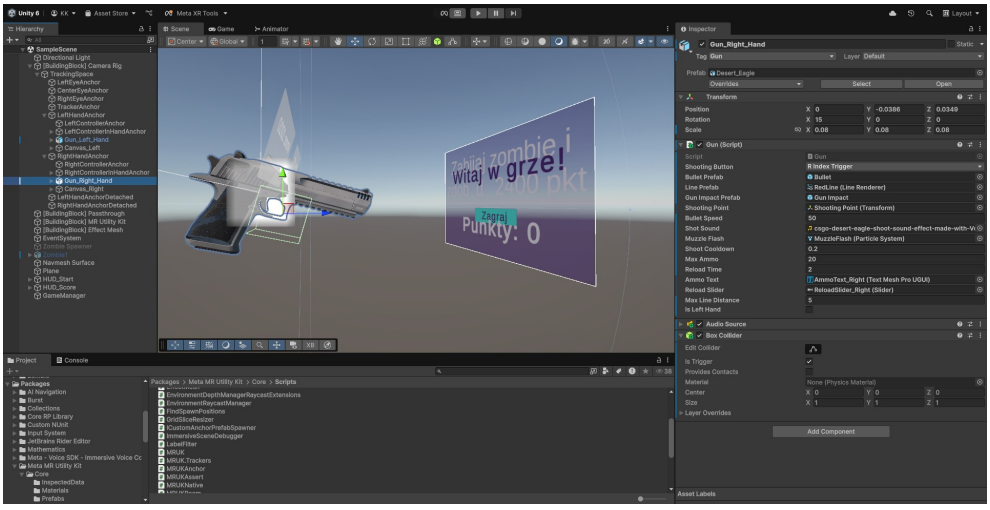
Rysunek 1: Widok głównej sceny projektu

2 System broni

W grze zastosowano dwa oddzielne obiekty broni przypisane do odpowiednich kontrolerów zestawu XR – lewego i prawego. Obie bronie umożliwiają eliminowanie przeciwników za pomocą wirtualnych strzałów, a logika działania jest identyczna w obu przypadkach.



Rysunek 2: Gun_Left_Hand



Rysunek 3: Gun_Right_Hand

2.1 Skrypt do pistoletu

2.1.1 Metoda Start() - Inicjalizacja

```
void Start()
{
    audioSource = GetComponent<AudioSource>();
    currentAmmo = maxAmmo;

    laserLineInstance = Instantiate(linePrefab);
    laserLineInstance.positionCount = 2;

    UpdateAmmoUI();
    if (reloadSlider != null)
        reloadSlider.gameObject.SetActive(false);
}
```

Listing 1: Metoda Start()

1. Przypisuje komponent *AudioSource*.
2. Tworzy wizualną linię lasera *LineRenderer* jako celownik.
3. Ustawia początkową liczbę nabojów i ukrywa pasek przeładowania.

2.1.2 Metoda Update() - Główna pętla działania broni

```
void Update()
{
    if (laserLineInstance != null)
    {
        laserLineInstance.SetPosition(0, shootingPoint.position);
        laserLineInstance.SetPosition(1, shootingPoint.position +
   shootingPoint.forward * maxLineDistance);
    }

    if (isReloading) return;

    OVRInput.Controller handController = isLeftHand ? OVRInput.Controller.
   LTouch : OVRInput.Controller.RTouch;
    OVRInput.Button reloadButton = isLeftHand ? OVRInput.Button.Three :
   OVRInput.Button.Two;
```

```
13
14      if (OVRInput.GetDown(shootingButton) && Time.time >= lastShotTime +
    shootCooldown)
15      {
16          if (currentAmmo > 0)
17          {
18              Shoot(handController);
19              currentAmmo--;
20              lastShotTime = Time.time;
21              UpdateAmmoUI();
22
23              if (currentAmmo == 0)
24              {
25                  StartCoroutine(Reload());
26              }
27          }
28      }
29
30      if (OVRInput.GetDown(reloadButton, handController) && !isReloading &&
    currentAmmo < maxAmmo)
31      {
32          StartCoroutine(Reload());
33      }
34  }
```

Listing 2: Metoda Update()

1. Rysuje linię celowania (laser).
2. Rozpoznaje rękę gracza i odpowiednie przyciski.
3. Obsługuje strzał: tylko jeśli dostępna amunicja i cooldown minął.
4. Wywołuje automatyczne lub ręczne przeładowanie.

2.1.3 Metoda UpdateAmmoUI() - aktualizacja UI amunicji

```
void UpdateAmmoUI()
{
    if (ammoText != null)
        ammoText.text = $"Ammo: {currentAmmo}/{maxAmmo}";
}
```

Listing 3: Metoda UpdateAmmoUI()

1. Aktualizuje licznik amunicji na ekranie gracza.

2.1.4 Metoda Reload() – proces przeładowania

```
IEnumerator Reload()
{
    isReloading = true;
    if (reloadSlider != null)
    {
        reloadSlider.gameObject.SetActive(true);
        reloadSlider.value = 0f;
    }

    float timer = 0f;
    while (timer < reloadTime)
    {
        timer += Time.deltaTime;
        if (reloadSlider != null)
            reloadSlider.value = timer / reloadTime;
        yield return null;
    }

    currentAmmo = maxAmmo;
    UpdateAmmoUI();

    if (reloadSlider != null)
        reloadSlider.gameObject.SetActive(false);
    isReloading = false;
}
```

Listing 4: Metoda Reload()

1. Uruchamia pasek ładowania (jeśli dostępny).
2. Symuluje czas ładowania przez *yield return*.
3. Przywraca pełną amunicję i aktualizuje interfejs.

2.1.5 Metoda Shoot() – strzał i rozpoznanie trafienia

```
public void Shoot(OVRInput.Controller handController)
{
    if (shotSound != null && audioSource != null)
        audioSource.PlayOneShot(shotSound);

    if (muzzleFlash != null)
        muzzleFlash.Play();

    Ray ray = new Ray(shootingPoint.position + shootingPoint.forward *
   0.01f, shootingPoint.forward);
    bool hasHit = Physics.Raycast(ray, out RaycastHit hit, maxLineDistance
   );
    Vector3 endPoint;

    if (hasHit)
    {
        endPoint = hit.point;

        Zombie zombie = hit.transform.GetComponentInParent<Zombie>();

        if (zombie)
        {
            bool isHeadshot = hit.transform.name.Equals("Z_Head", System.
   StringComparison.OrdinalIgnoreCase);

            zombie.TakeDamage(1, isHeadshot);
        }
        else
        {
            Quaternion gunImpactRotation = Quaternion.LookRotation(-hit.
   normal);

```

```
            GameObject gunImpact = Instantiate(gunImpactPrefab, hit.point,
    gunImpactRotation);
            Destroy(gunImpact, 1);
        }
    }
    else
    {
        endPoint = shootingPoint.position + shootingPoint.forward *
   maxLineDistance;
    }

    if (laserLineInstance != null)
    {
        laserLineInstance.SetPosition(0, shootingPoint.position);
        laserLineInstance.SetPosition(1, endPoint);
    }

    GameObject bullet = Instantiate(bulletPrefab, shootingPoint.position,
   shootingPoint.rotation);
    Rigidbody rb = bullet.GetComponent<Rigidbody>();
    if (rb != null)
        rb.linearVelocity = shootingPoint.forward * bulletSpeed;

    Destroy(bullet, 6f);

    OVRInput.SetControllerVibration(1f, 0.7f, handController);
    StartCoroutine(StopHaptics(handController));
}
```

Listing 5: Metoda Shoot()

1. Odtwarza efekt dźwiękowy i cząsteczkowy strzału.
2. Tworzy promień *Raycast* do wykrycia trafienia.
3. Jeśli trafiono zombie, sprawdza czy to headshot (trafienie w Z_Head).
4. Tworzy wizualny impakt w miejscu trafienia.
5. Generuje pocisk i nadaje mu prędkość.
6. Włącza wibrację kontrolera.

2.1.6 Coroutine StopHaptics() – wyłączenie wibracji

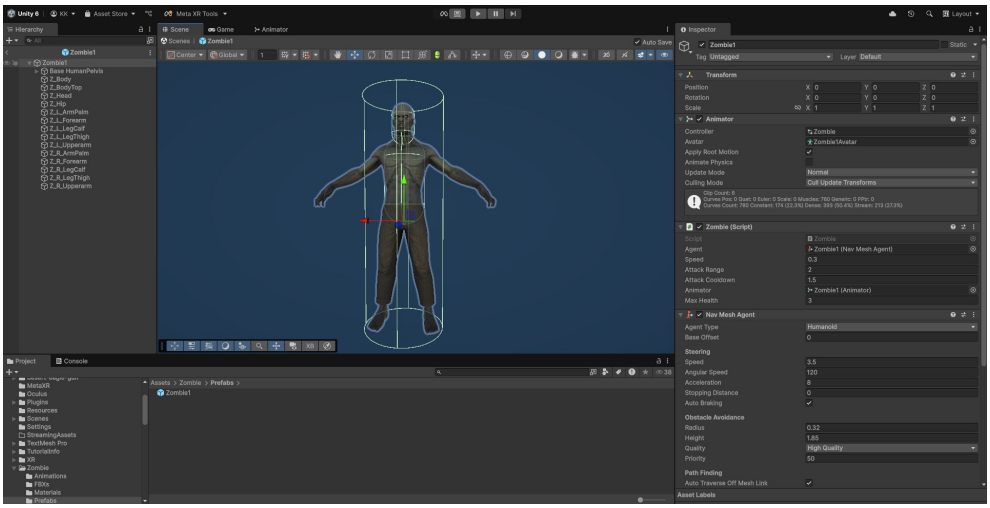
```
IEnumerator StopHaptics(OVRInput.Controller hand)
{
    yield return new WaitForSeconds(0.1f);
    OVRInput.SetControllerVibration(0f, 0f, hand);
}
```

Listing 6: Coroutine StopHaptics()

1. Wstrzymuje na 0.1 sekundy, po czym wyłącza wibracje kontrolera, aby nie trwały zbyt długo.

3 System przeciwników – Zombie

W grze „Przeciwnicy w twoim pokoju" głównym wyzwaniem dla gracza są pojawiające się w otoczeniu zombie. Przeciwnicy ci stanowią podstawowy element rozgrywki i są generowani dynamicznie w rzeczywistej przestrzeni użytkownika za pomocą technologii XR. Każdy zombie porusza się w kierunku gracza i może zostać wyeliminowany poprzez strzał w głowę (headshot) lub dwukrotne trafienie w ciało.
W tej sekcji opisano implementację prefabrykatu Zombie, systemu jego obrażeń, logiki poruszania się oraz integracji z punktacją.



Rysunek 4: Model Zombie

3.1 Skrypt do Zombie

3.1.1 Metoda Start() - Inicjalizacja zombie

```
void Start()
    {
        agent = GetComponent<NavMeshAgent>();
        animator = GetComponent<Animator>();
        target = Camera.main.transform;

        currentHealth = maxHealth;
        agent.speed = speed;
        agent.stoppingDistance = 0.5f;      // Zeby podchodzil mozliwie
   blisko
    }
```

Listing 7: Metoda Start()

1. Przypisuje komponenty i ustawia ich wartości.
2. *target* to pozycja kamery – zombie będzie do niej podążał.
3. Ustawia prędkość i minimalną odległość zatrzymania przed graczem.

3.1.2 Metoda Update() – Główna logika zachowania zombie

```
void Update()
    {
        if (isDead || !agent || !target || !agent.isOnNavMesh)
            return;

        float distance = Vector3.Distance(transform.position, target.
   position);

        // Zombie patrzy na gracza (tylko w poziomie)
        Vector3 lookDir = target.position - transform.position;
        lookDir.y = 0f;
        if (lookDir.sqrMagnitude > 0.01f)
        {
            Quaternion rotation = Quaternion.LookRotation(lookDir);
            transform.rotation = Quaternion.Slerp(transform.rotation,
   rotation, Time.deltaTime * 5f);
```

```
15          }
16
17          float remainingDistance = agent.remainingDistance;
18
19          // Przerwij atak jesli gracz sie oddali
20          if (isAttacking && distance > 0.5f)
21          {
22              isAttacking = false;
23              animator.SetBool("isAttacking", false);
24
25              agent.isStopped = false;
26              agent.SetDestination(target.position);
27          }
28
29          // Wejdz w atak tylko jesli agent dotarl naprawde blisko
30          else if (!isAttacking && !agent.pathPending && agent.hasPath &&
    remainingDistance > 0 && remainingDistance <= agent.stoppingDistance +
    0.05f)
31          {
32              isAttacking = true;
33              animator.SetBool("isAttacking", true);
34
35              agent.isStopped = true;
36              agent.ResetPath();
37          }
38
39          // Jesli nie atakuje, to ruszaj
40          else if (!isAttacking)
41          {
42              if (agent.isStopped)
43                  agent.isStopped = false;
44
45              agent.SetDestination(target.position);
46          }
47
48          animator.SetFloat("Speed", agent.velocity.magnitude);
49      }
```

Listing 8: Metoda Update()

1. Zombie obraca się w stronę gracza.
2. Sprawdza dystans i decyduje, czy ma atakować, zatrzymać się lub kontynuować ruch.
3. Aktualizuje animacje zależnie od prędkości *Speed* i stanu *isAttacking*.

3.1.3 Metoda Kill() – eliminacja zombie

```
public void Kill()
    {
        if (isDead) return; // Zabezpieczenie przed powtornym wywolaniem

        isDead = true;
        isFalling = false;

        // Powiadom spawner, ze zombie umarl
        ZombieSpawner spawner = FindFirstObjectByType<ZombieSpawner>();
        if (spawner != null)
            spawner.NotifyZombieDied();

        // DODAJ PUNKTY do GameManagera
        var gm = FindFirstObjectByType<GameStartUI>();
        if (gm != null)
            gm.AddPoints(150);

        if (agent != null)
            agent.enabled = false;

        animator.SetBool("isAttacking", false);
        animator.SetTrigger("gotShot");

        StartCoroutine(DieAfterDelay(5f));
    }
```

Listing 9: Metoda Kill()

1. Oznacza przeciwnika jako martwego.
2. Powiadamia *ZombieSpawner*, aby wiedział, że zombie został zabity.
3. Dodaje punkty do gry.
4. Wyłącza NavMeshAgenta, zatrzymuje animację ataku.
5. Rozpoczyna coroutine opóźnionego usunięcia.

3.1.4 Metoda DieAfterDelay() – opóźnienie przed zniknięciem

```
private IEnumerator DieAfterDelay(float delay)
    {
        yield return new WaitForSeconds(delay);
        Destroy(gameObject);
    }
```

Listing 10: Metoda DieAfterDelay()

1. Po śmierci zombie pozostaje przez 5 sekund, co pozwala na dokończenie animacji lub efektów.

3.1.5 Metoda TakeDamage() – odbieranie obrażeń przez zombie

```
public void TakeDamage(int amount, bool isHeadshot = false)
    {
        if (isDead || isFalling) return;

        animator.SetBool("hitToHead", isHeadshot);
        animator.SetTrigger("gotShot");

        if (isHeadshot)
        {
            Kill();
            return;
        }

        torsoHits++;
        currentHealth -= amount;

        if (torsoHits >= 2 || currentHealth <= 0)
        {
            Kill(); // Po 2 trafieniu - pelna smierc
        }
        else
        {
            // Odpal lekka reakcje - animacje upadania, ale wroc po chwili
    do Idle
            StartCoroutine(TemporaryFallingForward());
```

```
        }
    }
```

Listing 11: Metoda TakeDamage()

1. Jeżeli trafienie to headshot – natychmiastowe zabicie.
2. W przypadku trafień w tułów: dwa trafienia = śmierć.
3. Jeśli nie umiera, to aktywuje animację potknięcia (tymczasowe spowolnienie).

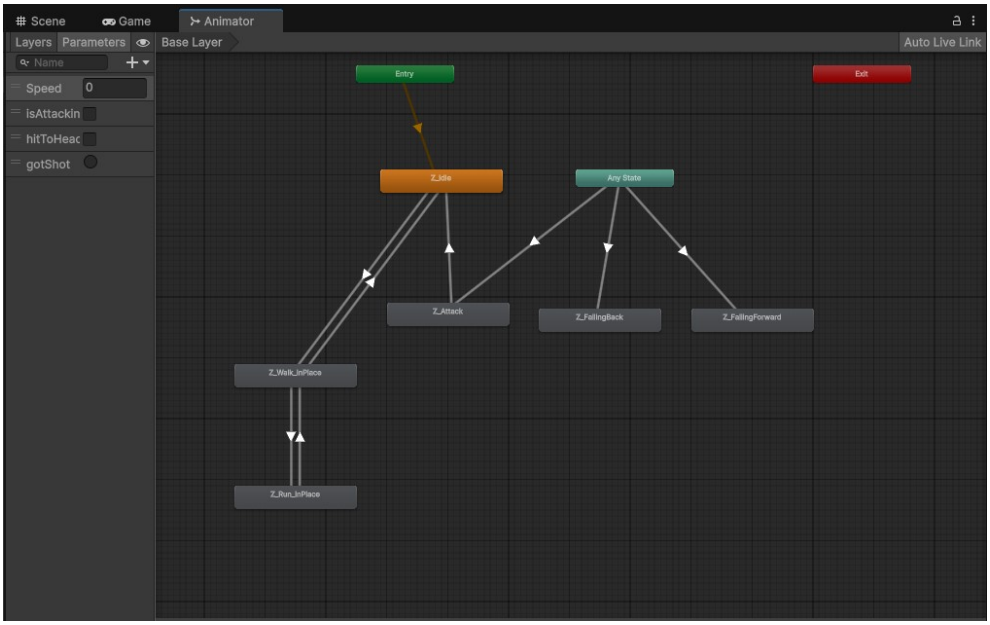
3.1.6 Metoda TemporaryFallingForward() – animacja przewrócenia

```
private IEnumerator TemporaryFallingForward()
    {
        isFalling = true;

        if (agent != null)
        {
            agent.isStopped = true;
            agent.ResetPath();

            // Spowolnij zombie
            float originalSpeed = agent.speed;
            agent.speed = 0.05f;

            animator.SetTrigger("gotShot");

            yield return new WaitForSeconds(0.5f); // Czas potkniecia

            // Przywroc oryginalna predkosc i wznow ruch
            if (!isDead)
            {
                agent.speed = originalSpeed;
                agent.isStopped = false;
                agent.SetDestination(target.position);
            }
        }

        isFalling = false;
    }
}
```

Listing 12: Metoda TemporaryFallingForward()

1. Po trafieniu zombie zatrzymuje się i odtwarza animację trafienia.
2. Na moment spowalnia jego ruch, a następnie powraca do normalnego tempa.
3. Działa tylko jeśli zombie jeszcze nie został zabity.

3.2 Animacje do Zombie

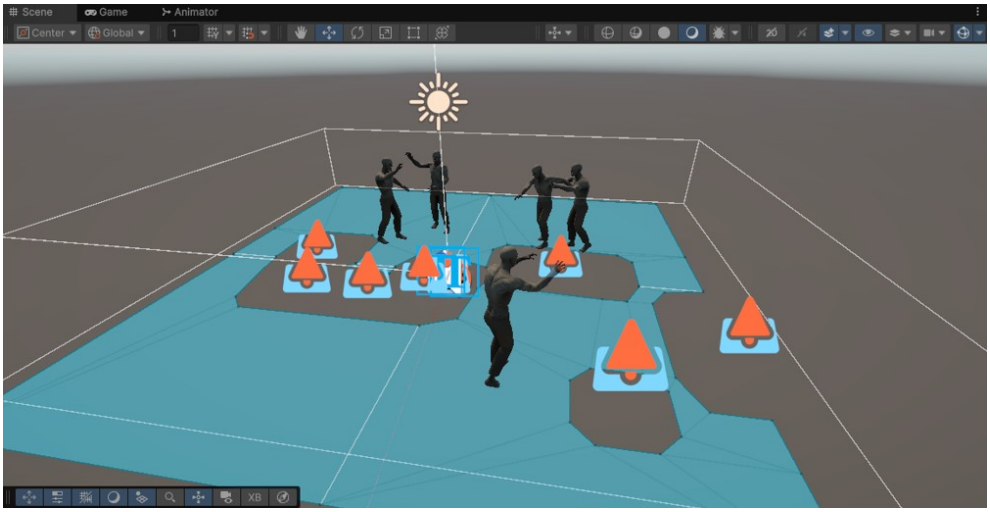


Rysunek 5: Animacje do Zombie

Na przedstawionym zrzucie ekranu widać Animator Controller dla obiektu Zombie, zawierający zdefiniowane stany animacji i przejścia między nimi. Animator ten odpowiada za sterowanie ruchem, atakiem oraz reakcjami zombie na obrażenia.

4 Spawner Zombie

ZombieSpawner to komponent odpowiedzialny za dynamiczne generowanie przeciwników na scenie w czasie rzeczywistym. Korzysta z systemu MR Utility Kit (MRUK) do analizowania otoczenia użytkownika i szukania odpowiednich powierzchni (np. ścian) do umieszczania zombie.



Rysunek 6: Zombie Spawner

4.1 Skrypt do Zombie Spawner

4.1.1 Metoda Update() – cykliczne sprawdzanie i generowanie zombie

```
void Update()
    {
        // Sprawdz czy MRUK jest gotowy
        if (MRUK.Instance == null || !MRUK.Instance.IsInitialized)
            return;

        // Odliczanie czasu
        timer += Time.deltaTime;
        if (timer > spawnTimer)
        {
            timer = 0f;
```

```
            SpawnZombie();
        }
    }
```

Listing 13: Metoda Update()

1. Sprawdza, czy instancja MRUK została poprawnie zainicjowana.
2. Odlicza czas przy użyciu *Time.deltaTime*.
3. Jeśli czas przekroczy ustalony *spawnTimer*, wywołuje metodę *SpawnZombie*() i resetuje licznik.

4.1.2 Metoda SpawnZombie() – generowanie nowego zombie w przestrzeni XR

```
public void SpawnZombie()
    {
        if (currentZombieCount >= maxZombies)
            return;

        MRUKRoom room = MRUK.Instance.GetCurrentRoom();
        int currentTry = 0;

        while (currentTry < spawnTry)
        {
            bool found = room.GenerateRandomPositionOnSurface(
                MRUK.SurfaceType.VERTICAL,
                minEdgeDistance,
                new LabelFilter(spawnLabels),
                out Vector3 pos,
                out Vector3 norm
            );

            if (found)
            {
                Vector3 spawnPos = pos + norm * normalOffset;
                spawnPos.y = 0; // Dopasuj do podlogi

                Instantiate(prefabToSpawn, spawnPos, Quaternion.identity);
                currentZombieCount++;
                return;
            }
```

```
28
29              currentTry++;
30          }
31      }
```

Listing 14: Metoda SpawnZombie()

1. **Kontrola limitu:** jeśli osiągnięto maksymalną liczbę przeciwników $maxZombies$, metoda kończy działanie.
2. **Pobieranie przestrzeni:** odczytuje aktualne pomieszczenie MRUK $GetCurrentRoom()$.
3. **Losowanie pozycji:**

a) próbuje maksymalnie $spawnTry$ razy znaleźć losową pozycję na pionowej powierzchni $SurfaceType.VERTICAL$,

b) filtruje powierzchnie na podstawie etykiet $LabelFilter$.

4. **Generowanie:**

a) jeśli znaleziono odpowiednią pozycję, przeciwnik jest spawnowany z przesunięciem $normalOffset$ od ściany i y = 0, by był przyklejony do podłogi,

b) zwiększa licznik $currentZombieCount$.

4.1.3 Metoda NotifyZombieDied() – informacja o śmierci przeciwnika

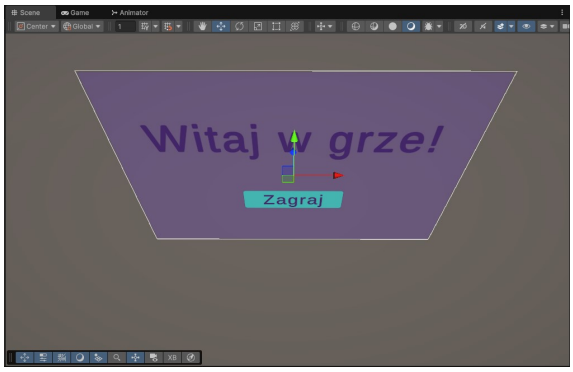
```
1       public void NotifyZombieDied()
2       {
3           currentZombieCount = Mathf.Max(0, currentZombieCount - 1);
4       }
```

Listing 15: Metoda NotifyZombieDied()

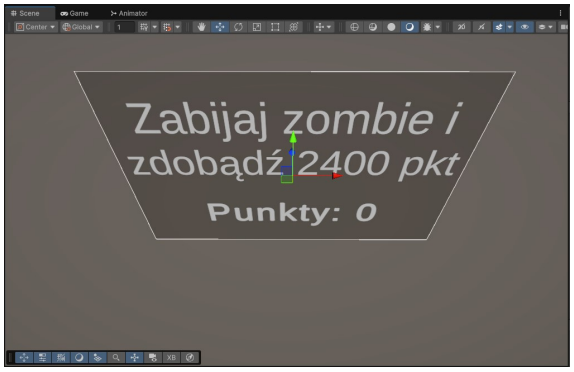
1. Metoda wywoływana przez prefab zombie w momencie śmierci (np. w $Kill()$).
2. Zmniejsza licznik aktywnych zombie, dzięki czemu spawner może ponownie wygenerować kolejnego przeciwnika.
3. Zabezpieczenie przed zejściem poniżej zera przez $Mathf.Max$.

5 System interfejsów HUD

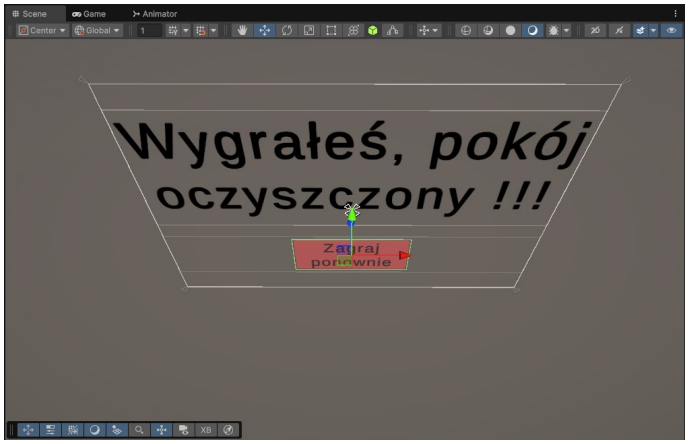
System HUD (Heads-Up Display) odpowiada za obsługę przebiegu gry – od jej uruchomienia, przez naliczanie punktów, aż po zakończenie. Składa się z trzech głównych paneli: HUD_Start (ekran powitalny z przyciskiem startu), HUD_Score (panel z aktualnym wynikiem oraz komunikatem o wygranej) oraz RestartButton, który umożliwia ponowne rozpoczęcie rozgrywki. Interfejs na bieżąco aktualizuje liczbę punktów zdobywanych za eliminację przeciwników, zapewniając graczowi przejrzysty i płynny przebieg gry.



Rysunek 7: HUD - Start



Rysunek 8: HUD - Score



Rysunek 9: HUD - Restart

5.1 Wspólny skrypt do HUD_Start oraz HUD_Score

5.1.1 Metoda LateUpdate() – pozycjonowanie i rotacja UI

```
void LateUpdate()
    {
        if (!head) return;

        Vector3 targetPos = head.position + head.forward * offset.z +
   Vector3.up * offset.y;
        transform.position = targetPos;

        Quaternion lookRotation = Quaternion.LookRotation(transform.
   position - head.position);
        transform.rotation = lookRotation;
    }
```

Listing 16: Metoda LateUpdate()

1. Ustawia UI 2 metry przed głową gracza i lekko w górę.
2. Obraca UI, by zawsze był skierowany w stronę gracza.
3. Działa w *LateUpdate*(), zapewniając płynność i czytelność.

5.2 Skrypt do HUD_Start

```
using System.Diagnostics;
using UnityEngine;

public class StartGameTarget : MonoBehaviour
{
    public GameStartUI gameManager;

    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Gun"))
        {
            gameManager.BeginGameplay();
        }
    }
}
```

Listing 17: Skrypt do HUD_Start

1. Po wejściu pistoletem (obiektem z tagiem "*Gun*") w obszar triggera, uruchamia grę przez *BeginGameplay*() z *GameStartUI*.
2. Działa jako interaktywny start – gra zaczyna się po zbliżeniu broni do wyznaczonego celu, bez potrzeby przycisku.

5.3 Skrypt do HUD_Score

```
using System.Diagnostics;
using UnityEngine;

public class RestartGameTarget : MonoBehaviour
{
    public GameStartUI gameManager;

    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Gun"))
        {
            gameManager.StartGame();
        }
    }
}
```

Listing 18: Skrypt do HUD_Start

1. Po wejściu pistoletem w trigger, uruchamia ponownie grę przez *StartGame*() z *GameStartUI*.
2. Działa jako interaktywny restart po zakończeniu gry – element *HUD_Score*.

5.4 Skrypt do GameStartUI - Zarządzanie interfejsem

Chociaż GameStartUI to osobny obiekt w scenie, pełni on kluczową rolę w systemie interfejsów HUD. Odpowiada za dynamiczne przełączanie widoczności różnych paneli w zależności od stanu gry – takich jak ekran startowy (HUD_Start), punktacja (HUD_Score) czy ekran końcowy z wynikiem i przyciskiem restartu. Dodatkowo zarządza logiką rozpoczęcia, zakończenia i resetu rozgrywki, łącząc UI z systemem przeciwników.

5.4.1 Metoda Start()

```
void Start()
{
    ShowStartUI();
}
```

Listing 19: Metoda Start()

1. Ustawia początkowy widok interfejsu – wywołuje metodę odpowiedzialną za pokazanie ekranu startowego.

5.4.2 Metoda AddPoints(int amount)

```
public void AddPoints(int amount)
{
    if (gameEnded) return;

    score += amount;
    UpdateScoreUI();

    if (score >= 2400)
    {
        EndGame();
    }
}
```

Listing 20: Metoda AddPoints(int amount)

1. Dodaje punkty (np. po zabiciu zombie).
2. Sprawdza, czy gra jeszcze trwa.
3. Jeśli gracz osiągnął 2400 punktów – kończy rozgrywkę.

5.4.3 Metoda UpdateScoreUI()

```
void UpdateScoreUI()
{
    if (scoreText != null)
        scoreText.text = $"Punkty: {score}";
}
```

Listing 21: Metoda UpdateScoreUI()

1. Aktualizuje wyświetlaną liczbę punktów w interfejsie HUD.

5.4.4 Metoda EndGame()

```
void EndGame()
{
    gameEnded = true;

    if (zombieSpawner != null)
        zombieSpawner.SetActive(false);

    if (hudScore != null)
        hudScore.SetActive(true);

    if (scoreText != null)
        scoreText.gameObject.SetActive(false);

    if (infoScore != null)
        infoScore.SetActive(false);

    if (winText != null)
        winText.SetActive(true);

    if (restartButton != null)
        restartButton.SetActive(true);

    Zombie[] allZombies = FindObjectsOfType<Zombie>();
    foreach (Zombie z in allZombies)
        Destroy(z.gameObject);
}
```

Listing 22: Metoda EndGame()

1. Zatrzymuje dalsze naliczanie punktów i wyłącza spawner zombie.
2. Ukrywa zbędne elementy UI (np. wynik), pokazuje ekran zwycięstwa i przycisk restartu.
3. Usuwa wszystkie zombie ze sceny.

5.4.5 Metoda StartGame()

```
public void StartGame()
{
    ShowStartUI();
}
```

Listing 23: Metoda StartGame()

1. Resetuje stan gry do wartości początkowych (używana przy restarcie).

5.4.6 Metoda ShowStartUI()

```
void ShowStartUI()
{
    score = 0;
    gameEnded = false;

    if (zombieSpawner != null)
        zombieSpawner.SetActive(false);

    if (hudStart != null)
        hudStart.SetActive(true);

    if (hudScore != null)
        hudScore.SetActive(false);

    if (scoreText != null)
    {
        scoreText.text = "Punkty: 0";
        scoreText.gameObject.SetActive(true);
    }

```

```
    if (infoScore != null)
        infoScore.SetActive(true);

    if (winText != null)
        winText.SetActive(false);

    if (restartButton != null)
        restartButton.SetActive(false);
}
```

Listing 24: Metoda ShowStartUI()

1. Ustawia ekran początkowy: wyświetla HUD startowy, resetuje punkty, chowa inne panele.

5.4.7 Metoda BeginGameplay()

```
public void BeginGameplay()
{
    if (hudStart != null)
        hudStart.SetActive(false);

    if (hudScore != null)
        hudScore.SetActive(true);

    if (zombieSpawner != null)
        zombieSpawner.SetActive(true);

    if (scoreText != null)
    {
        scoreText.text = "Punkty: 0";
        scoreText.gameObject.SetActive(true);
    }

    if (infoScore != null)
        infoScore.SetActive(true);

    if (winText != null)
        winText.SetActive(false);

    if (restartButton != null)
```

```
25          restartButton.SetActive(false);
26
27      score = 0;
28      gameEnded = false;
29  }
```

Listing 25: Metoda BeginGameplay()

1. Rozpoczyna rozgrywkę: uruchamia spawner zombie, pokazuje HUD z wynikiem i chowa ekran startowy oraz przycisk restartu.

6 Pozostałe komponenty sceny

1. Directional Light Zapewnia podstawowe oświetlenie całej sceny. Symuluje światło słoneczne, rzuca cienie i wpływa na wygląd modeli 3D.

2. [BuildingBlock] Passthrough Umożliwia podgląd rzeczywistego świata przez gogle VR. Działa jako "przezroczyste tło" w aplikacji XR.

3. [BuildingBlock] MR Utility Kit Zestaw narzędzi Meta (MRUK) do interakcji ze światem rzeczywistym – np. rozpoznawanie ścian, płaszczyzn i powierzchni do rozmieszczania obiektów.

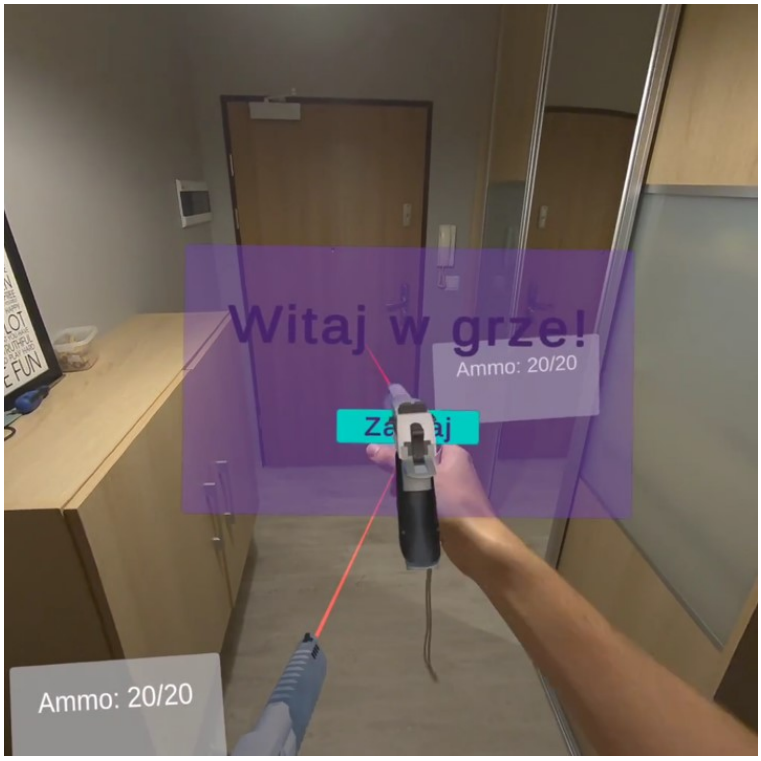
4. [BuildingBlock] Effect Mesh Dekoracyjny lub pomocniczy obiekt do wizualizacji (np. siatki, efekty nakładane na podłoże czy ściany). Może też pokazywać granice MR.

5. EventSystem System odpowiedzialny za obsługę zdarzeń UI w Unity – np. wykrywanie kliknięć, interakcji z przyciskami czy triggerami.

6. Navmesh Surface Komponent potrzebny do wyznaczenia obszaru, po którym mogą poruszać się zombie (AI). Definiuje "chodliwą"przestrzeń.

7. Plane Płaska powierzchnia w scenie – może pełnić rolę podłogi, tła lub testowej płaszczyzny do generowania zombie w przestrzeni.

7 Prezentacja wideo działania gry



Kliknij obrazek, aby odtworzyć wideo online

8 Podsumowanie

Podsumowując, jesteśmy naprawdę dumni z tego, że udało nam się samodzielnie stworzyć pełnoprawną grę w rozszerzonej rzeczywistości przy użyciu Unity. Spędziliśmy nad projektem wiele godzin, ucząc się, eksperymentując i rozwiązując różne problemy – i zdecydowanie było warto. Efekt końcowy prezentuje się bardzo dobrze, a sama gra działa płynnie i daje sporo satysfakcji. Wierzymy, że po kilku drobnych poprawkach, takich jak dopracowanie animacji czy optymalizacja działania zombie przy dużym obciążeniu, moglibyśmy śmiało udostępnić ją szerszemu gronu odbiorców.