

## 1. Temat projektu

Prosta gra 3D – symulator statku kosmicznego:

- sterowanie „statkiem kosmicznym” z obrotami wokół wszystkich osi,
- kamera umieszczona za pojazdem,
- wykorzystanie kwaternionów do rotacji statku/sceny,
- otoczenie statku i fabuła gry opracowane we własnym zakresie

## 2. Opis zastosowanych bibliotek i algorytmów

W projekcie został wykorzystany:

- Silnik Unity jest fundamentem całego projektu. Jego wszechstronne narzędzia i możliwości pozwalają na efektywne tworzenie interaktywnych środowisk 3D. Unity oferuje wsparcie dla wielu platform, co pozwala na łatwe przenoszenie projektu między różnymi urządzeniami, takimi jak komputery osobiste, konsole, urządzenia mobilne czy też platformy VR. C# jest najbardziej powszechnie używanym językiem programowania w Unity. Jest to silnie typowany język, który oferuje wydajność, bezpieczeństwo typów i zaawansowane funkcje obiektowe.
- W celu obsługi wejścia od gracza zdecydowano się na skorzystanie z nowego systemu wejścia Unity, jest to rozszerzenie InputAction Asset. Umożliwia on w nowoczesny sposób zarządzanie wejściem, tworzenie elastycznej obsługi dla wielu urządzeń jednocześnie w tym przypadku myszy, klawiatury i pada.
- Do implementacji dynamicznej kamery śledzącej statek gracza użyto modułu Cinemachine w Unity. Cinemachine pozwala na łatwe definiowanie różnych ustawień kamery.
- Własny algorytm obsługujący rotacje za pomocą kwaternionów. Algorytm ten umożliwia precyzyjne obliczenia związane z obrotami statku kosmicznego. Pozbywa się problemu tzw. „gimbal lock”, na który możemy się natknąć korzystając z standardowych kątów eulera. Główne kroki tego algorytmu obejmują:
  - Inicjalizacja kwaternionu – Tworzymy obiekt klasy CustomQuaternion, który domyślnie reprezentuje jednostkowy kwaternion (brak obrotu). Możemy także utworzyć kwaternion z określonymi wartościami x, y, z, w lub z osią i kątem w

radianach.

- Tworzenie kwaternionu z osią i kątem – metoda `FromAxisAngle`, przyjmuje wektor osi i kąt w stopniach. Metoda ta oblicza kąt obrotu a następnie sinus kąta do utworzenia kwaternionu z osi i  $\sin$ ,  $\cos$  połowy kąta.
- Mnożenie kwaternionów – metoda `Multiply` pozwala na pomnożenie dwóch kwaternionów i utworzenie kwaternionu wynikowego. Pozwala to na skumulowanie obrotów wokół różnych osi w jednym kwaternionie.
- Normalizacja kwaternionu – metoda `Normalize` umożliwia normalizację kwaternionu w celu utrzymania jednostkowej długości kwaternionu, który jest wymagany, aby poprawnie reprezentować obroty.
- Konwersja do macierzy rotacji – Metoda `ToMatrix4x4`, konwertuje kwaternion na macierz 4x4 w celu umożliwienia wykonywania operacji takich jak mnożenie wektora rotacji
- Pomnożenie kwaternionu przez zadany wektor – metoda `RotateVector` mnoży wektor przez kwaternion co pozwala na uzyskanie obrotu wektora.
- Algorytm generowania wrogów/zasobów umożliwia generowanie w zasadzie dowolnych obiektów. Realizowany jest przez klasę `Spawner`. Zmienna tablicowa przechowuje obiekty, a także zakres współrzędnych, reprezentujących kwadrat, wewnątrz którego zostaną utworzone obiekty. Metoda działania jest następująca:
  - Pętla wykonuje się tyle razy ile wynosi ilość obiektów, które chcemy by się wygenerowały
  - Wewnątrz pętli losowana jest pozycja xyz z wspomnianego zakresu dla danej instancji obiektu
  - Losowany jest obiekt, który zostanie wygenerowany
  - Następuje inicjalizacja obiektu przy pomocy ustalonych wcześniej zmiennych
  - Cykl się powtarza aż do spełnienia warunku pętli
- Algorytm reprezentacji zerowej grawitacji w kosmosie dla obiektów pozwala na wprawienie w ruch obiektów znajdujących się na scenie. Każdy obiekt poza statkiem gracza i przeciwników obsługiwany jest przez klasę `GravityController`. Klasa ta zawiera zmienne, które określają min oraz max momentu i siły, która ma działać na obiekt. Następnie podczas ładowania projektu wyłączamy w komponencie fizycznym obiektów grawitację, a następnie za pomocą metod `AddForce` i `AddTorque` zostaje

nadana siła oraz moment skierowany do przodu. Wartość tej siły i momentu jest wylosowana z zakresu zdefiniowanego przez zmienne.

- Algorytm do obsługi wrogów jest ważnym aspektem tego projektu, ponieważ definiuje zachowanie przeciwników w danej sytuacji. Podejście zastosowane w tym projekcie jest następujące. W metodzie, która wykonuje się co klatkę, tworzone są sfery o zdefiniowanym promieniu, reprezentujące zasięg widzenia oraz ataku przeciwnika. Te sfery są przytwierdzone do obiektu przeciwnika. Wykrywają one kolizję ze wskazaną warstwą, w tym przypadku jest to warstwa dla statku gracza oraz jego bazy. Wynikiem detekcji kolizji jest ustawienie odpowiednich zmiennych na true lub false, w zależności od tego, czy obiekt w zasięgu widzenia, czy ataku. Następnie zmienne te są sprawdzane przy pomocy if'ów. Dzięki temu podejściu mamy możliwość zdefiniowania określonego zachowania w przypadku gdy np. statek gracza jest w zasięgu ataku a baza gracza w zasięgu widzenia. Następnie właśnie za pomocą odpowiednich metod określane jest, co ma się wydarzyć, jak ma się zachować obiekt przeciwnika w takiej sytuacji.

### 3. Opis implementacji wraz z fragmentami kodu źródłowego

Przedstawię tylko najważniejsze fragmenty dotyczące programu, które są kluczowe z punktu widzenia tematu i założeń projektowych.

Pierwszą i jedną z najważniejszych klas jest CustomQuaternion. Klasa ta zawiera własną implementację dla kwaternionów, które są wykorzystywane podczas poruszania się statkiem gracza. Zmienne x,y,z,w definiują składowe kwaternionu. Definiowana jest także stała do wykorzystywania do obliczeń kąta obrotu. Klasa ta posiada kilka konstruktorów.

- CustomQuaternion(): Konstruktor domyślny, tworzy kwaternion jednostkowy (brak obrotu).
- CustomQuaternion(float x, float y, float z, float w): Konstruktor z określonymi składowymi kwaternionu.
- CustomQuaternion(Vector3 axis, float angle): Konstruktor z osią i kątem obrotu w

radianach.

Zaimplementowano także następujące metody:

- `FromAxisAngle(Vector3 axis, float angle)`: Tworzy kwaternion na podstawie osi i kąta obrotu.
- `Multiply(CustomQuaternion a, CustomQuaternion b)`: Wykonuje mnożenie kwaternionów.
- `ToMatrix4x4()`: Konwertuje kwaternion na macierz 4x4, co jest przydatne do transformacji obiektów w przestrzeni.
- `RotateVector(Vector3 vector)`: Obraca wektor przy użyciu kwaternionu.
- `Normalize()`: Normalizuje kwaternion, aby zachować jednostkową długość.
- `Forward()`, `Up()`, `Right()`, `Back()`: Zwracają kierunki w przestrzeni według aktualnego obrotu.

Operacje zastosowane na kwaternionach wykorzystują ogólne wzory, które były przedstawiane na wykładzie. Poniżej znajduje się zrzut ekranu przedstawiający opisaną klasę wraz z komentarzami do kodu.

```

using UnityEngine;

public class CustomQuaternion
{
    //Zmienna
    public float x;
    public float y;
    public float z;
    public float w; //skalar

    const float AngleMultiplier = 0.001f; // stała do obliczenia kąta obrotu

    // Konstruktor domyślny (kwaternion jednostkowy (brak obrotu))
    public CustomQuaternion()
    {
        x = 0f;
        y = 0f;
        z = 0f;
        w = 1f;
    }

    // Konstruktor z określonymi zmiennymi
    public CustomQuaternion(float x, float y, float z, float w)
    {
        this.x = x;
        this.y = y;
        this.z = z;
        this.w = w;
    }

    // Konstruktor z osią i kątem w radianach
    public CustomQuaternion(Vector3 axis, float angle)
    {
        angle *= 0.5f;
        float sinHalf = Mathf.Sin(angle);
        x = sinHalf * axis.x;
        y = sinHalf * axis.y;
        z = sinHalf * axis.z;
        w = Mathf.Cos(angle);
    }

    // Metoda statyczna tworząca kwaternion na podstawie osi i kąta obrotu
    public static CustomQuaternion FromAxisAngle(Vector3 axis, float angle)
    {
        float halfAngle = angle * AngleMultiplier; // Obliczenie kąta obrotu
        float sinHalf = Mathf.Sin(halfAngle); // Obliczenie sinusa kąta
        return new CustomQuaternion(axis.x * sinHalf, axis.y * sinHalf, axis.z * sinHalf,
            Mathf.Cos(halfAngle)); // Utworzenie kwaternionu z osi i sinusa, kosinusa połowy kąta
    }

    // Metoda statyczna wykonująca mnożenie kwaternionów.
    public static CustomQuaternion Multiply(CustomQuaternion a, CustomQuaternion b)
    {
        // Obliczenia dla x, y, z, w
        float x = a.w * b.x + a.x * b.w + a.y * b.z - a.z * b.y;
        float y = a.w * b.y + a.y * b.w + a.z * b.x - a.x * b.z;
        float z = a.w * b.z + a.z * b.w + a.x * b.y - a.y * b.x;
        float w = a.w * b.w - a.x * b.x - a.y * b.y - a.z * b.z;

        return new CustomQuaternion(x, y, z, w); // Utworzenie nowego kwaternionu z wyników
    }

    // Metoda konwertująca kwaternion na macierz 4x4
    public Matrix4x4 ToMatrix4x4()
    {
        // Obliczenia dla różnych elementów macierzy rotacji
        float xx = x * x;
        float xy = x * y;
        float xz = x * z;
        float xw = x * w;

        float yy = y * y;
        float yz = y * z;
        float yw = y * w;

        float zz = z * z;
        float zw = z * w;

        // Utworzenie nowej macierzy 4x4
        Matrix4x4 matrix = new Matrix4x4();

        // Przypisanie wartości do poszczególnych elementów macierzy rotacji
        matrix[0, 0] = 1 - 2 * (yy + zz);
        matrix[0, 1] = 2 * (xy - zw);
        matrix[0, 2] = 2 * (xz + yw);

        matrix[1, 0] = 2 * (xy + zw);
        matrix[1, 1] = 1 - 2 * (xx + zz);
        matrix[1, 2] = 2 * (yz - xw);

        matrix[2, 0] = 2 * (xz - yw);
        matrix[2, 1] = 2 * (yz + xw);
        matrix[2, 2] = 1 - 2 * (xx + yy);

        matrix[3, 0] = 1f;

        return matrix;
    }

    // Metoda obracająca wektor przy użyciu kwaternionu
    public Vector3 RotateVector(Vector3 vector)
    {
        Matrix4x4 rotationMatrix = ToMatrix4x4(); // Konwersja kwaternionu na macierz rotacji
        return rotationMatrix * vector; // Pomnożenie macierzy rotacji przez wektor
    }

    public void Normalize()
    {
        float magnitude = Mathf.Sqrt(x * x + y * y + z * z + w * w);
        x /= magnitude;
        y /= magnitude;
        z /= magnitude;
        w /= magnitude;
    }

    // Metody zwracające kierunku w przestrzeni według aktualnego obrotu
    public Vector3 Forward()
    {
        return new Vector3(2 * (x * z + w * y), // składowa x
            2 * (y * z - w * x), // składowa y
            1 - 2 * (x * x + y * y)); // składowa z
    }

    public Vector3 Up()
    {
        return new Vector3(2 * (x * y - w * z), 1 - 2 * (x * x + z * z), 2 * (y * z + w * x));
    }

    public Vector3 Right()
    {
        return new Vector3(1 - 2 * (y * y + z * z), 2 * (x * y + w * z), 2 * (x * z - w * y));
    }

    public Vector3 Back()
    {
        return -Forward();
    }
}

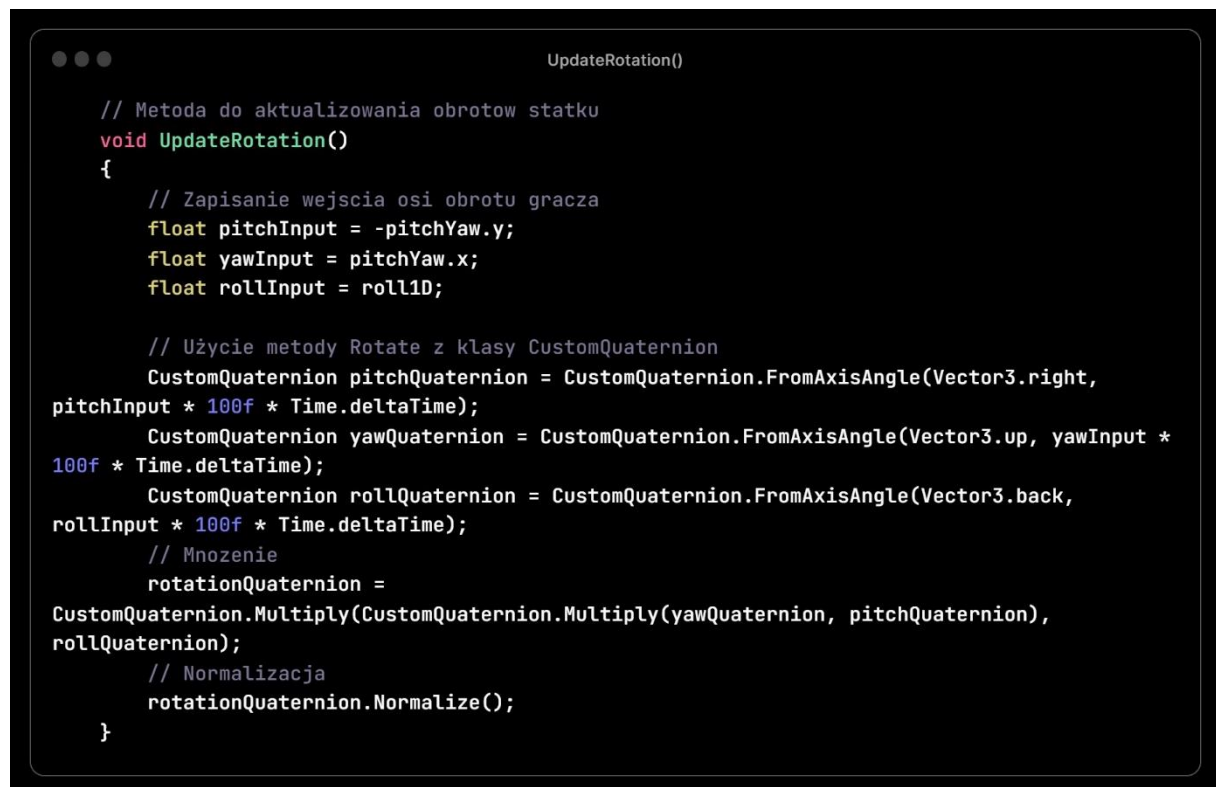
```

Rysunek 1 Klasa CustomQuaternion

Kolejną klasą, którą przedstawię, jest `PlayerComponent`, lecz tu ograniczę się do poszczególnych fragmentów z racji jej obszerności.

Klasa ta przyjmuje wiele zmiennych, chociażby do ustawienia momentu obrotowego na poszczególnych osiach, określenia siły napędu, wartości boostu, tempa odnawiania, utraty czy też zmiennych używanych do wytracania prędkości, obsługi wejścia od gracza.

Jedną z ważniejszych metod jest `UpdateRotation()`, która wykonywana jest co klatkę. Jest to metoda służąca do aktualizowania obrotów statku. Na początku zapisywane są wejścia osi obrotu, pochodzących z wejścia od gracza. Następnie używana jest metoda `FromAxisAngle()`, z klasy `CustomQuaternion` dla każdej osi obrotu, później mnożenie otrzymanych kwaternionów z wykorzystaniem `Multiply` również z klasy `CustomQuaternion` a na końcu wynikowy kwaternion jest normalizowany.



```
UpdateRotation()

// Metoda do aktualizowania obrotow statku
void UpdateRotation()
{
    // Zapisanie wejścia osi obrotu gracza
    float pitchInput = -pitchYaw.y;
    float yawInput = pitchYaw.x;
    float rollInput = roll10;

    // Użycie metody Rotate z klasy CustomQuaternion
    CustomQuaternion pitchQuaternion = CustomQuaternion.FromAxisAngle(Vector3.right,
pitchInput * 100f * Time.deltaTime);
    CustomQuaternion yawQuaternion = CustomQuaternion.FromAxisAngle(Vector3.up, yawInput *
100f * Time.deltaTime);
    CustomQuaternion rollQuaternion = CustomQuaternion.FromAxisAngle(Vector3.back,
rollInput * 100f * Time.deltaTime);
    // Mnożenie
    rotationQuaternion =
CustomQuaternion.Multiply(CustomQuaternion.Multiply(yawQuaternion, pitchQuaternion),
rollQuaternion);
    // Normalizacja
    rotationQuaternion.Normalize();
}
```

*Rysunek 2 Metoda `UpdateRotation()` z klasy `PlayerComponent`*

Do obsługi samego ruchu została napisana metoda `HandleMove()`, która rozpoczyna się od wykorzystania poprzednio opisanej metody `UpdateRotation()`. Używa ona następnie wyliczonego kwaternionu do zastosowania momentu obrotowego dla każdej z osi za pomocą metody `AddRelativeTorque` znajdującej się w komponencie `Rigidbody`, który odpowiada za fizykę w Unity. Linijki te pozwalają uzyskać efekt rotacji statku. Następnie znajduje się obsługa

ruchu statku w trzech wymiarach, która zawsze rozpoczyna się od sprawdzenia, czy został wykonany jakiś ruch. Następnie stosowana jest siła napędu za pomocą `AddRelativeForce()` w określonym kierunku z wykorzystaniem `RotateVector()`. W momencie, gdy nie ma ruchu, to uzyskana prędkość, jest zmniejszana z wykorzystaniem zdefiniowanej zmiennej do wytracania prędkości.

```
HandleMove()

// Metoda do obsługi ruchu statku
void HandleMove()
{
    // Aktualizacja kwaternionu rotacji na podstawie ruchu gracza
    UpdateRotation();

    // Zastosowanie sil i momentow obrotowych przy uzyciu macierzy rotacji
    rb.AddRelativeTorque(rotationQuaternion.Back() * rollID * rollTorque *
Time.deltaTime);
    rb.AddRelativeTorque(rotationQuaternion.Right() * Mathf.Clamp(-pitchYaw.y, -1, 1f) *
pitchTorque * Time.deltaTime);
    rb.AddRelativeTorque(rotationQuaternion.Up() * Mathf.Clamp(pitchYaw.x, -1f, 1f) *
yawTorque * Time.deltaTime);

    // Obsługa ruchu statku w trzech wymiarach (przód, góra/dół, bok)
    if (thrustID > 0.1f || thrustID < -0.1)
    {
        // Określenie aktualnej siły napędu
        float currentThrust;
        if (boosting)
        {
            currentThrust = thrust * boostMultiplier; // Jeśli używamy boost to zwiększamy
sile napędu o mnożnik boosta
        }
        else
        {
            currentThrust = thrust;
        }
        // Zastosowanie siły napędu do przodu w kierunku określonym przez rotację statku
        rb.AddRelativeForce(rotationQuaternion.RotateVector(Vector3.forward) * thrustID *
currentThrust * Time.deltaTime);
        // Aktualizacja wartości dla późniejszego wytracania prędkości
        glide = thrust;
    }
    else
    {
        // Wytracanie prędkości, gdy gracz nie rusza się
        rb.AddRelativeForce(rotationQuaternion.RotateVector(Vector3.forward) * glide *
Time.deltaTime);
        glide -= thrustGlideReduction;
    }

    // Obsługa ruchu w górę/dół
    if (upDownID > 0.1f || upDownID < -0.1)
    {
        // Zastosowanie siły napędu w górę/dół w zależności od rotacji
        rb.AddRelativeForce(rotationQuaternion.RotateVector(Vector3.up) * upDownID *
upThrust * Time.fixedDeltaTime);
        // Aktualizacja wartości dla późniejszego wytracania prędkości
        verticalGlide = upDownID * upThrust;
    }
    else
    {
        // Wytracanie prędkości w górę/dół gdy gracz nie rusza się
        rb.AddRelativeForce(rotationQuaternion.RotateVector(Vector3.up) * verticalGlide *
Time.fixedDeltaTime);
        verticalGlide -= upDownGlideReduction;
    }

    // Obsługa ruchu w bok
    if (strafeID > 0.1f || strafeID < -0.1)
    {
        // Zastosowanie siły napędu w bok w zależności od rotacji
        rb.AddRelativeForce(rotationQuaternion.RotateVector(Vector3.right) * strafeID *
strafeThrust * Time.fixedDeltaTime);
        // Aktualizacja wartości dla późniejszego wytracania prędkości
        horizontalGlide = strafeID * strafeThrust;
    }
    else
    {
        // Wytracanie prędkości w bok gdy gracz nie rusza się
        rb.AddRelativeForce(rotationQuaternion.RotateVector(Vector3.right) *
horizontalGlide * Time.fixedDeltaTime);
        horizontalGlide -= leftRightGlideReduction;
    }
}
```

Rysunek 3 Metoda `HandleMove()` z klasy `PlayerComponent`

Przedstawione wyżej metody są najważniejsze w klasie `PlayerComponent`. Poza nimi znajdują się również metody do obsługi wejścia od gracza, aktualizacji boostu, a także obsługi kolizji.

Statek gracza ma możliwość wystrzeliwania rakiet. Zdolność ta obsługiwana jest w klasie `ShootingScript`. Zawiera ona zmienne dotyczące pozycji dział, zasięgu strzału, czasu przeładowania, ilości amunicji, a także obiektu który ma zostać wystrzelony i z jaką prędkością. Poniższa metoda `HandleShooting()`, używana jest do wyznaczenia kierunku, w który ma zostać wystrzelony obiekt z działa. Metoda ta rozpoczyna się od sprawdzenia, czy gracz ma możliwość wystrzału. Gdy posiada on amunicję, użył przycisku do strzału i minął określony czas, od ostatniego wystrzału, następuje sprawdzenie z wykorzystaniem własnej implementacji, użycia `RayCastu`, o nazwie `TargetInfo` i metody `isTargetInRange`, która to zwraca czy promień wystrzeliwany z uśrednionej pozycji dział w kierunku określonym przez zwrot kamery i długości przez zasięg strzału trafi w cel, czy też nie. Jeśli trafi, to zostaje zapisana pozycja celu w który trafił, jeśli natomiast nie to używany jest domyślny punkt określony przez raycast. Następnie w pętli `foreach` dla każdej pozycji dział losowane jest przesunięcie, które określa rozrzut pocisku w promieniu 0.5. Następnie przekazywana jest pozycja i rotacja dział oraz punkt, w który ma zostać wystrzelona rakietka do metody `ShootMissile()`. Kolejne kroki to zmniejszenie ilości amunicji, aktualizacja gui i zmiennych odpowiadających za dostęp do strzału.



```

    HandleShooting()

    void HandleShooting()
    {
        if (shooting && missileAmmo > 0 && canShoot)
        {
            if (Gamepad.current != null)
            {
                StartCoroutine(HandleShootingVibration());
            }
            RaycastHit hitInfo;
            Vector3 targetPosition;

            if (TargetInfo.isTargetInRange(middlePoint.transform.position,
            cam.transform.forward, out hitInfo, shootRange))
            {
                targetPosition = hitInfo.point;
            }
            else
            {
                // Domyslny punkt gdy raycast nie trafi w zaden punkt
                targetPosition = middlePoint.transform.position + cam.transform.forward *
            shootRange;
            }
            foreach (Transform missileGun in gunsTransform)
            {
                //Vector3 localHitPosition =
            missileGun.transform.InverseTransformPoint(targetPosition);
                Vector3 randomOffset = Random.onUnitSphere * 1.0f; // Przesuniecie o losowa
            wartosc na sferze o promieniu 0.5
                Vector3 adjustedTargetPosition = targetPosition + randomOffset;
                // Wystrzel pocisk w kierunku punktu
                ShootMissile(missileGun.position, adjustedTargetPosition,
            missileGun.rotation);
                missileAmmo--;
                if (ammoText)
                {
                    ammoText.text = missileAmmo.ToString();
                }
            }
            shooting = false;
            canShoot = false;
            StartCoroutine(EnableShootingAfterDelay(reloadingTime));
        }
        else
        {
            shooting = false;
        }
    }
}

```

*Rysunek 4 Metoda HandleShooting() z klasy ShootingScript*

Metoda ShootMissile() odpowiada za wystrzelanie obiektu. Na początku obliczany jest kierunek, w który powinna zostać wystrzelona rakietą na podstawie pozycji działa oraz pozycji celu. Następnie dokonywana jest inicjalizacja obiektu w miejscu pozycji działa jego rotacji. Przekazywane do klasy rakiety jest, kto ją wystrzeliwiuje, w celu zablokowania możliwości trafienia siebie samego lub kogoś z własnej drużyny. Następnie za pomocą metody AddForce z komponentu fizycznego rakiety jest ona wystrzeliwana w kierunku wcześniej obliczonym z siłą określoną przez zmienną missileSpeed. Następnie uruchamiana jest zmienna, która ma za zadanie zniszczyć raketę po czasie osiągnięcia określonego zasięgu.

```

    ShootMissile()

    public void ShootMissile(Vector3 startPosition, Vector3 targetPosition, Quaternion
rotation)
    {
        // Oblicz kierunek
        Vector3 direction = targetPosition - startPosition;
        GameObject missile = Instantiate(missilePrefab, startPosition, rotation);
        // Przekazanie referencji do obiektu który inicjalizuje do rakiety
        Missile missileScript = missile.GetComponent<Missile>();
        if (missileScript != null)
        {
            missileScript.SetShooter(this.gameObject);
        }

        Rigidbody missileRb = missile.GetComponent<Rigidbody>();
        missileRb.AddForce(direction.normalized * missileSpeed, ForceMode.Impulse);

        StartCoroutine(DestroyAfterTime(missile, shootRange / missileSpeed));
    }

```

*Rysunek 5 Metoda ShootMissile() z klasy ShootingScript*

Mniej ważne metody dostępne w klasie to te służące do obsługi wejścia od gracza, obsługi wibracji dla pada, odblokowania możliwości strzelania po przeładowaniu czy też obsługa kolizji do zbierania amunicji.

Sama rakietka, czyli ten wystrzelony obiekt, który traktowany jest jak pocisk, posiada klasę o nazwie Missile. Zawiera ona zmienne, które określają masę, w jaką pocisk może trafić, particle i dźwięk po uderzeniu, a także obrażenia, jakie zadaje. Najważniejszą metodą w tej klasie jest metoda obsługująca kolizje. Sprawdzane wewnątrz niej jest, kto został trafiony, jaką warstwę posiada trafiony obiekt i jeśli posiada on komponent odpowiadający za zdrowie, to zadawane mu są obrażenia, a jeśli ten obiekt jest bazą gracza, to zmniejszana jest wartość tarczy tej bazy. Inicjalizowane są także particle i dźwięk uderzenia, jeśli takie, są dodane i niszczone jest obiekt samej rakiety po uderzeniu.

```
TriggerEnter i ApplyDamage

void ApplyDamage(HealthComponent healthComponent)
{
    healthComponent.TakeDamage(missileDamage);
}

private void OnTriggerEnter(Collider collision)
{
    if (shooter.tag != null)
    {
        if (shooter.tag != collision.tag)
        {
            if (shootableMask == (shootableMask | (1 << collision.gameObject.layer)))
            {
                if (impactParticles)
                {
                    Destroy(Instantiate(impactParticles,
collision.ClosestPoint(transform.position), Quaternion.identity), 5f);
                }
                if (impactSound)
                {
                    Destroy(Instantiate(impactSound, transform.position,
Quaternion.identity), 5f);
                }
                if (collision.gameObject.GetComponentInParent<HealthComponent>())
                {
                    ApplyDamage(collision.gameObject.GetComponentInParent<HealthComponent>
());
                }
                else if (collision.gameObject.CompareTag("Base"))
                {
                    GameManager.Instance.addBarrierBase(-missileDamage);
                }
                if (gameObject != null)
                {
                    Destroy(gameObject);
                }
            }
        }
    }
}
```

Rysunek 6 ApplyDamage() i OnTriggerEnter() z klasy Missile

Kolejna klasa ważna w projekcie to EnemyComponent. Jak nazwa wskazuje, znajduje się tu implementacja przeciwników. Jako zmienne dostępne są warstwy do wykrywania gracza i jego bazy, zasięg widzenia wrogów, ich siła napędu i szybkość do rotacji, zmienne boolowskie, które używane są do określenia zachowania, a także zmienne znane z poprzedniej klasy ShootingScript takie jak zasięg strzału, czas ładowania, amunicja, pozycja dział. Ważniejszą metodą jest tu metoda wykonująca się co klatkę, czyli FixedUpdate(). Wewnątrz znajduje się logika związana z określeniem zachowania wroga. Na początku tworzona jest sfera o

promieniu określonym przez zasięg widzenia bądź też strzału, która wykrywa warstwę gracza i jego bazy. Sfera ta jest doczepiona do pozycji obiektu przeciwnika. Zwraca ona true do wyżej wspomnianych zmiennych boolowskich, jeśli gracz bądź baza znajduje się w zasięgu ataku lub widzenia. Następnie znajdują się if'y, wewnątrz nich sprawdzane zostaje stan tych zmiennych i na podstawie tego wywoływana odpowiednia metoda np. do ataku lub gonienia gracza albo ataku jego bazy. Jest również warunek, w którym przypisywana jest maksymalna wartość amunicji do aktualnej amunicji w momencie, gdy braknie jej przeciwnikowi.

```
FixedUpdate()

private void FixedUpdate()
{
    basePlayerInSightRange = Physics.CheckSphere(transform.position, sightRange,
    layerPlayerBase);
    basePlayerInAttackRange = Physics.CheckSphere(transform.position, shootRange,
    layerPlayerBase);

    playerInSightRange = Physics.CheckSphere(transform.position, sightRange, layerPlayer);
    playerInAttackRange = Physics.CheckSphere(transform.position, shootRange,
    layerPlayer);

    if(!playerInSightRange && !playerInAttackRange && !basePlayerInAttackRange &&
    !basePlayerInSightRange)
    {
        MoveToBasePlayer();
    }

    if (basePlayerInSightRange && basePlayerInAttackRange)
    {
        AttackBasePlayer();
    }
    if (playerInSightRange && !playerInAttackRange && !basePlayerInAttackRange)
    {
        ChasePlayer();
    }
    if(playerInSightRange && playerInAttackRange && !basePlayerInAttackRange)
    {
        AttackPlayer();
    }
    if(missileAmmo <= 0)
    {
        missileAmmo = maxMissileAmmo;
    }
}
```

Rysunek 7 FixedUpdate() EnemyComponent

Inne metody, które możemy tutaj znaleźć do obsługi ruchu, strzelania są uproszczoną wersją opisywanych wcześniej metod znajdujących się w PlayerComponent oraz ShootingScript.

Każdy z przeciwników oraz gracz korzysta z klasy `HealthComponent`, która zawiera całą logikę związaną z otrzymywaniem, zadawaniem obrażeń oraz leczeniem. Przechowuje ona zmienne określające maksymalną i aktualną ilość punktów zdrowia, a także obiekt dla partycji wykonywanych podczas zniszczenia oraz pasek zdrowia. Wszystkie metody w tej klasie są wirtualne, więc zostały zaimplementowane klasy dziedziczące po `HealthComponent`, które mogą dodać dodatkową logikę do tych metod. W metodzie `Start()` wykonującej się przy uruchomieniu jest przypisywana maksymalna wartość punktów zdrowia do aktualnej i aktualizowany pasek zdrowia. Metoda `AddHealth()` służy do zwiększania poziomu zdrowia, natomiast metoda `TakeDamage()` do jego zmniejszania poprzez zadawanie obrażeń. Znajduje się tu również warunek, który wykona się gdy życie spadnie do 0 i obiekt, przechowujący partycję nie jest pusty. Zostanie wtedy zainicjalizowany obiekt partycji, który będzie odpowiadał za efekt zniszczenia, a następnie usunięty ten obiekt.

```
HealthComponent

public class HealthComponent : MonoBehaviour
{
    public float maxHealth = 100f;
    public float currentHealth = 0;
    [SerializeField] GameObject destructionParticles;
    [SerializeField] Image healthBar;

    public virtual void Start()
    {
        currentHealth = maxHealth;
        if (healthBar)
        {
            UpdateHealthBar(maxHealth, currentHealth);
        }
    }

    public virtual void AddHealth(float health)
    {
        currentHealth += health;
        UpdateHealthBar(maxHealth, currentHealth);
    }

    public virtual void TakeDamage(float damage)
    {
        currentHealth -= damage;
        if (healthBar)
        {
            UpdateHealthBar(maxHealth, currentHealth);
        }
        if (currentHealth <= 0) {
            if (destructionParticles)
            {
                Destroy(Instantiate(destructionParticles, transform.position,
                Quaternion.identity), 5f);
            }
            Destroy(this.gameObject);
        }
    }

    protected virtual void OnDestroy()
    {
    }

    public void UpdateHealthBar(float maxHealth, float currentHealth)
    {
        healthBar.fillAmount = currentHealth/maxHealth;
    }
}
```

Rysunek 8 Klasa `HealthComponent`

Projekt zawiera również klasę Spawner, która przechowuje zmienne i metody do generowania obiektów. Zmienne, które zostały zadeklarowane to tablica obiektów, wewnątrz której pojawiać się będą obiekty, które chcemy generować. Zmienne służące do określania wielkości przestrzeni, w której mają być inicjalizowane obiekty. A także kolor do reprezentacji wizualnej wspomnianej przestrzeni i int pozwalający na dodanie dodatkowej ilości obiektów, które będą generowane. Metoda Spawn() jako parametr przyjmuje ilość obiektów, które mają zostać utworzone. Następnie przy pomocy pętli for wykonywanej tyle ile zostało przekazane w parametrze metody, plus ewentualnie określone przez zmienną przechowującą wspomnianą wcześniej dodatkową ilość. Wewnątrz pętli losowana jest pozycja xyz punktu w przestrzeni, który w tym przypadku jest kwadratem. Następnie losowany jest obiekt znajdujący się w tablicy i na końcu jest on inicjalizowany w wylosowanym punkcie w tej przestrzeni. Znajduje się tu także metoda OnDrawGizmos(), która wizualnie przedstawia tę przestrzeń i pozwala ją ustawić na scenie.



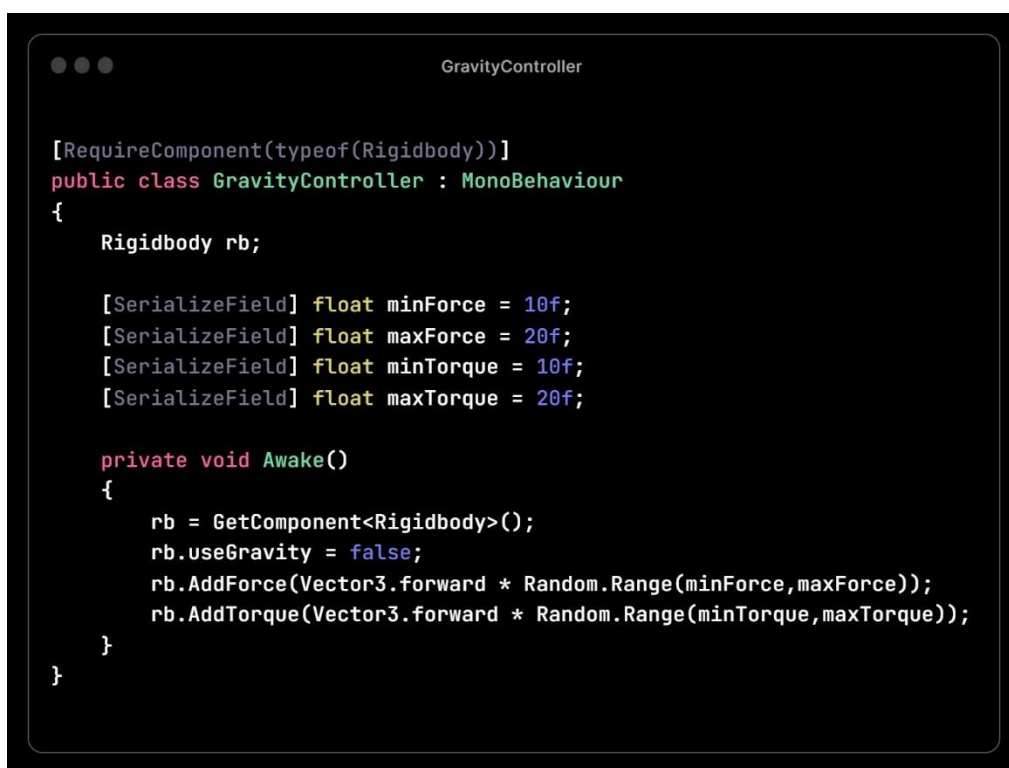
```
public class Spawner : MonoBehaviour
{
    [SerializeField] GameObject[] prefab;
    [SerializeField] float minRandomSpawn = -500f;
    [SerializeField] float maxRandomSpawn = 500f;
    [SerializeField] Color color;
    [SerializeField] int extraSpawn = 0;

    public void Spawn(int amount)
    {
        for (int i = 0; i < amount + extraSpawn; i++)
        {
            float randomX = Random.Range(minRandomSpawn, maxRandomSpawn) +
transform.position.x;
            float randomY = Random.Range(minRandomSpawn, maxRandomSpawn) +
transform.position.y;
            float randomZ = Random.Range(minRandomSpawn, maxRandomSpawn) +
transform.position.z;
            Vector3 randomSpawnPoint = new Vector3(randomX, randomY, randomZ);
            int randomPrefab = Random.Range(0, prefab.Length);
            Instantiate(prefab[randomPrefab], randomSpawnPoint, Random.rotation,
this.transform);
        }
    }

    private void OnDrawGizmos()
    {
        Gizmos.color = color;
        Gizmos.DrawWireCube(transform.position, new Vector3(maxRandomSpawn * 2, maxRandomSpawn
* 2, maxRandomSpawn * 2));
    }
}
```

Rysunek 9 Klasa Spawner

W celu uzyskania symulacji zerowej grawitacji w kosmosie została utworzona klasa GravityController, którą posiada każdy obiekt na scenie, prócz gracza i przeciwników, ponieważ posiadają oni własną wariację tej klasy. Klasa ta zawiera zmienne typu float do określenia wartości minimalnych i maksymalnych dla siły oraz momentu. Metoda Awake(), która uruchamiana jest w momencie, kiedy zostanie załadowana poszczególna instancja obiektu, który go posiada w swoim ciele, pobiera komponent fizyczny (rigidbody) obiektu do którego jest dołączona, wyłącza grawitację w tym komponencie, a następnie stosowana jest siła oraz moment za pomocą metod AddForce() oraz AddTorque() o wartościach wylosowanych z określonego powyższymi zmiennymi zakresu.

A screenshot of a code editor window titled "GravityController". The code is written in C# and implements a MonoBehaviour script. It includes a [RequireComponent(typeof(Rigidbody))] attribute, a public class declaration, and a private Awake() method. The Awake() method finds the Rigidbody component, disables gravity, and applies a random force and torque to the object.

```
GravityController

[RequireComponent(typeof(Rigidbody))]
public class GravityController : MonoBehaviour
{
    Rigidbody rb;

    [SerializeField] float minForce = 10f;
    [SerializeField] float maxForce = 20f;
    [SerializeField] float minTorque = 10f;
    [SerializeField] float maxTorque = 20f;

    private void Awake()
    {
        rb = GetComponent<Rigidbody>();
        rb.useGravity = false;
        rb.AddForce(Vector3.forward * Random.Range(minForce,maxForce));
        rb.AddTorque(Vector3.forward * Random.Range(minTorque,maxTorque));
    }
}
```

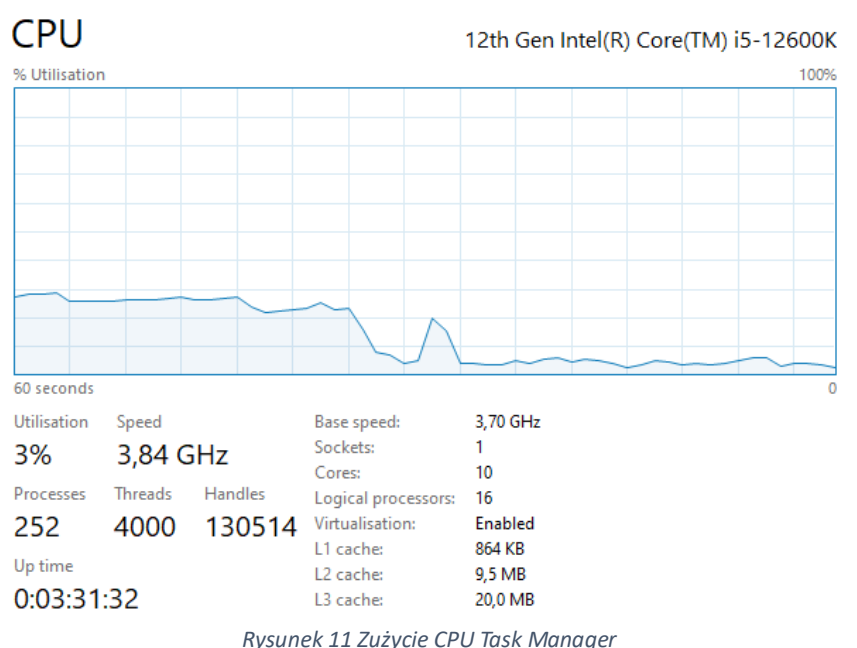
Rysunek 10 GravityController

Utworzona została także, chociażby klasa GameManager, która kontroluje rozgrywkę poprzez metody do tworzenia nowych fal wrogów, generowania zasobów, sprawdzania ilości żywych przeciwników, zabójstw oraz obsługująca pauze, barierę bazy gracza oraz określająca koniec gry. W projekcie zostały również utworzone pomniejsze klasy takie jak TowardsPlayer do śledzenia statku gracza przy pomocy kamery lub ParticleDestroyer do zagwarantowania zniszczenia obiektu po jego animacji.

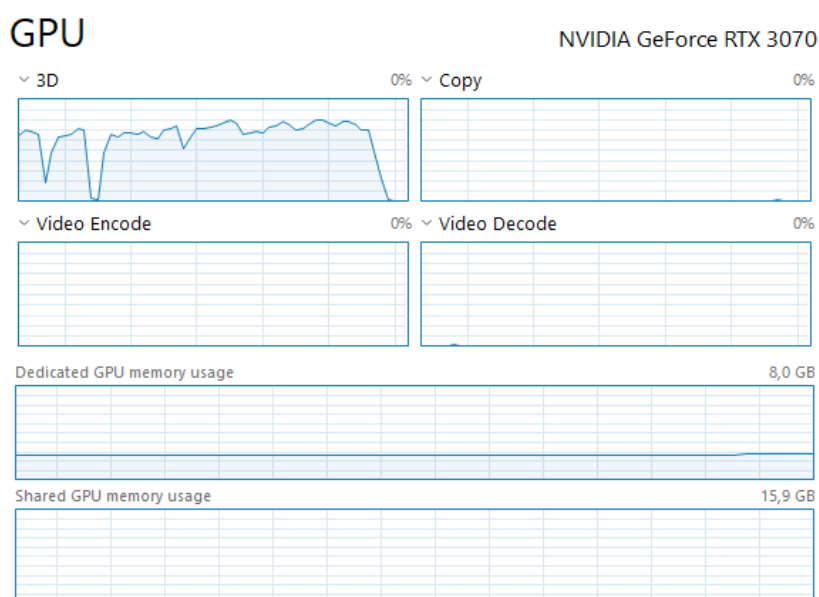
W grze obecna jest także muzyka, dźwięki i różne efekty zniszczeń oraz trafień, które są udostępniane za darmo w asset store. Za ich odtwarzanie odpowiadają komponenty Unity.

## 4. Wyniki testów wydajnościowych

Testy przeprowadzone na programie potwierdziły jego poprawne i stabilne działanie. Na poniższych zdjęciach możemy zobaczyć m.in. zużycie CPU i GPU, korzystając ze wbudowanego w system Windows menedżera zadań. Widać, że poziom obciążenia CPU utrzymuje się poniżej 30%, natomiast GPU osiąga maksymalnie 80%. Warto jednak podkreślić, że te wyniki mogą być obarczone pewnym błędem z uwagi na możliwy wpływ innych aktywnych procesów w trakcie przeprowadzania testu.



Rysunek 11 Zużycie CPU Task Manager

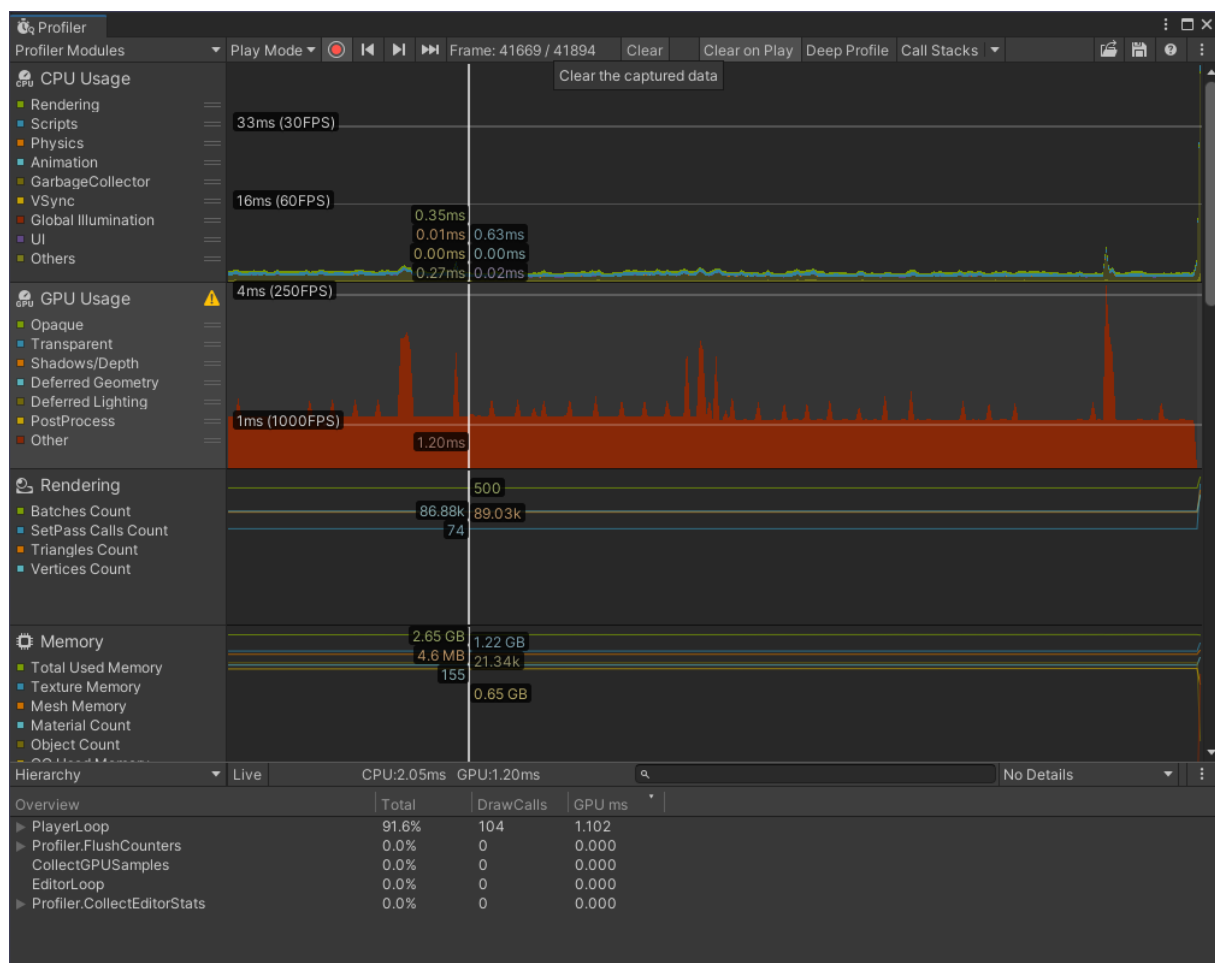


Rysunek 12 Zużycie GPU Task Manager

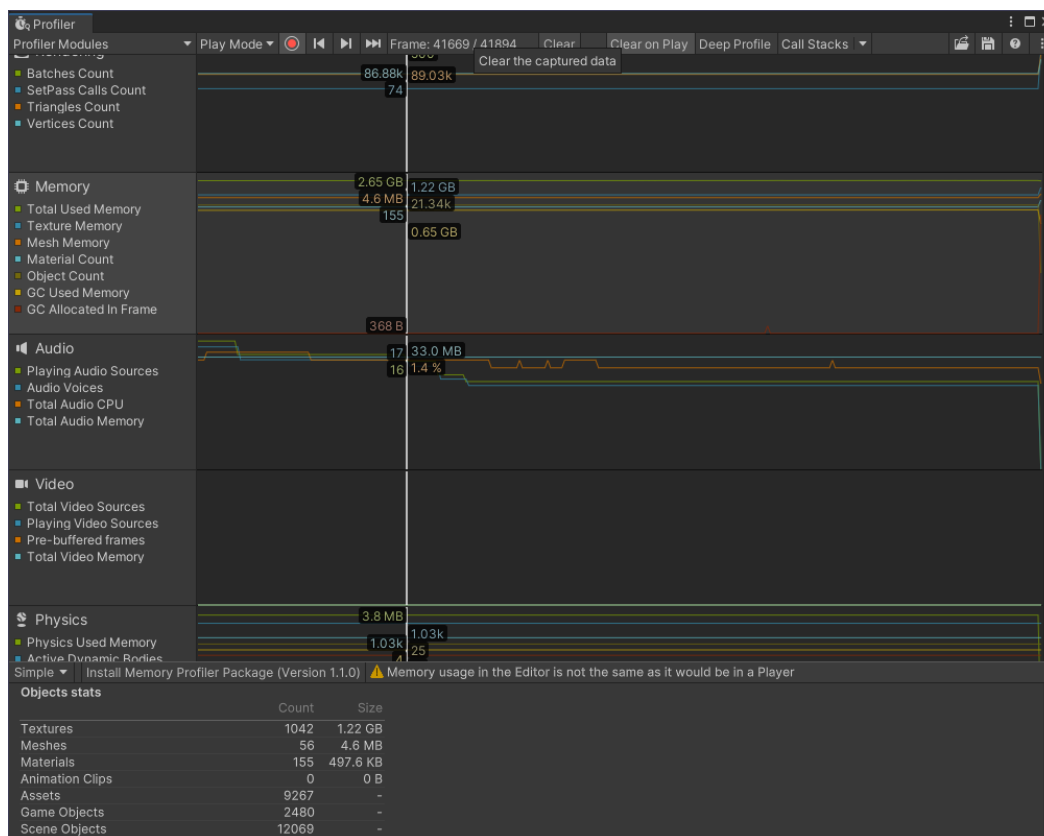


Unity oferuje zaawansowane narzędzie do monitorowania wydajności aplikacji, znane jako Unity Profiler. Profiler zbiera i prezentuje szczegółowe dane dotyczące wydajności w obszarach takich jak procesor, pamięć, moduł renderowania i dźwięk. Na poniższych zrzutach ekranu przedstawione zostały uzyskane informacje podczas używania naszej gry. Przedstawiają one dokładne zużycie zasobów wraz z czasem trwania każdej klatki, szczegółowo podzielone na poszczególne moduły i komponenty.

Analizując te wykresy, możemy stwierdzić, że utworzona aplikacja zużywa mało zasobów komputera i jest wydajna. Otrzymujemy wynik około 500 fps dla CPU, przyjmując, że czas trwania 1 klatki dla CPU trwał 2.05ms oraz około 800 fps dla GPU przyjmując 1.20ms. Jeśli chodzi o zużycie pamięci RAM, całkowite zużycie aplikacji wynosi 2.65 GB, z czego 1.22 GB to tekstury. Na kolejnym zrzucie widzimy również dokładnie jaka była ich ilość. Ciekawą statystyką jest również przedstawiona liczba trójkątów i wierzchołków, a także zużycie zasobów przez audio lub fizykę.



Rysunek 13 Unity Profiler 1



Rysunek 14 Unity Profiler 2

Ostatni zrzut ekranu przedstawia wynik uzyskany za pomocą wbudowanych statystyk live podczas uruchomienia aplikacji w edytorze Unity. Można zauważyć również ilość wygenerowanych FPS, czas trwania klatek, ilość wierzchołków, trójkątów, poziom głośności itd.



Rysunek 15 Unity Live Statistics Window

Wszystkie te wyniki potwierdzają wysoką jakość działania programu, jego optymalizację i efektywność.