| Grafika komputerowa i GUI | |
|---|---|
| **Kierunek**<br>*Informatyczne Systemy Automatyki* | **Temat projektu**<br>*Wirtualne okno - Dno Oceanu* |
| **Imię, nazwisko, numer albumu**<br>*Mateusz Marko 273168* | **Data**<br>*30.10.2024r.* |
| **Grupa**<br>*Grupa 3 środa TP 17:05 – 18:45* | |

# Dokumentacja Projektu Wirtualne okno - Dno Oceanu



Politechnika Wrocławska

# Spis treści

# 1. Wstęp

## 1.1 Temat projektu

Celem projektu było stworzenie interaktywnego podwodnego środowiska w Unity, które symuluje widok z okna zanurzonego pod wodą. Początkowo planowałem umieścić kamerę wewnątrz łodzi podwodnej, jednak ostatecznie zrezygnowałem z tego pomysłu, uznając, że sama podwodna sceneria jest bardziej atrakcyjna. Zdecydowałem się na temat podwodnego okna, ponieważ wydawał mi się znacznie ciekawszy niż standardowe okno, w którym niewiele się dzieje. Chciałem przedstawić fascynujący świat podwodny, w którym użytkownik może obserwować ruchome ryby, bujne dno morskie oraz efekty wizualne charakterystyczne dla głębin oceanu.

# 2. Narzędzia i technologie

## 2.1 Unity 3D

Do realizacji projektu wybrałem silnik Unity 3D, który jest powszechnie używany w branży gier i aplikacji interaktywnych. Unity oferuje bogaty zestaw narzędzi do tworzenia środowisk 3D, obsługę fizyki, animacji oraz szeroką bazę assetów, co znacznie przyspiesza proces developmentu.

## 2.2 Język C# oraz Python

Skrypty w Unity pisałem w języku C# za pomocą Visual Studio 2022. C# jest językiem wysokopoziomowym, który doskonale integruje się z Unity i pozwala na efektywne tworzenie logiki gry oraz interakcji. Całe kontrolowanie pracy kamery nie działałoby jednak gdybym nie wysyłał informacji na bieżąco z Pycharma, w którym napisałem osobny skrypt w języku python do przechwytywania ruchu twarzy na kamerze.

## 2.3 Assety i Biblioteki

- MediaPipe - Wykorzystałem bibliotekę googla do śledzenia ruchu twarzy. W czasie rzeczywistym skrypt analizuje obraz z kamery a następnie wyliczane są przesunięcia za pomocą identyfikacji i śledzenia kluczowych punktów twarzy . Pozwoliło to na interaktywne sterowanie kamerą w środowisku Unity.

- Assety - Skorzystałem z różnych assetów dostępnych w Unity Asset Store oraz z darmowych źródeł online, takich jak Polytope Studio, AQUAS-Lite Water Plane którego użyłem do stworzenia realistycznej tafli wody z efektami falowania i odbicia światła czy Jiggly Bubble do stworzenia efektu podwodnych bąbli powietrza. Jednak nie wszystko było dostępne za darmo online więc niektóre tekstury stworzyłem samodzielnie, aby lepiej dopasować je do potrzeb projektu.

# 3. Implementacja

## 3.1 Tworzenie podwodnego środowiska

- **Dno morskie**

Dno morskie wykonałem od podstaw, korzystając z narzędzi w Terrain w Unity. Samodzielnie ukształtowałem teren, wygładzałem go i dostosowywałem jego kształt, aby oddać naturalne ukształtowanie dna morskiego. Nałożyłem różnorodne tekstury piasku, skał i roślinności. Modyfikowałem parametry takie jak długość trawy morskiej czy rozmiary kamieni, aby pasowały do scenerii i tworzyły spójny wizualnie świat.

- **Efekty wizualne**

Wykorzystałem asset **AQUAS-Lite Water Plane** do stworzenia realistycznej powierzchni wody. Zmodyfikowałem ruch fal oraz shader odbicia promieni świetlnych, aby uzyskać pożądany efekt. Następnie w ustawieniach kamery ustawiłem tryb oświetlenia na **Linear**. Dodałem efekt mgły o kolorze odpowiadającym dnu morza, z początkiem na 0 i końcem na 70 jednostkach, co tworzy wrażenie głębi i gęstej wody. W dalszej kolejności ukształtowałem dalekie formacja skalne jako tło**,** oddaliłem skybox z chmurami i dostosowałem go tak, aby pod wpływem tafli wody chmury się rozmywały, tworząc efekt realistycznego nieba widzianego spod wody. Całość starałem się utrzymać w stylu **low poly**, który nadaje projektowi lekko bajkowego, growego charakteru, zamiast dążyć do fotorealizmu.

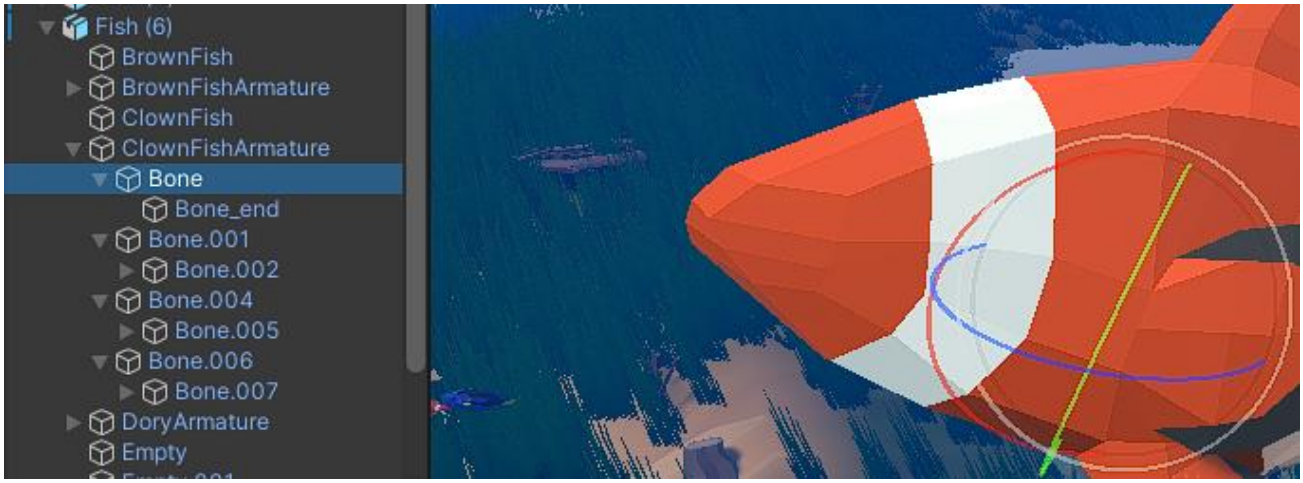
- **Interakcje środowiskowe**

Dodałem box collidery do obiektów w scenie, takich jak skały i rośliny, aby ryby nie przenikały przez nie, lecz odbijały się lub omijały je zapobiegając kolizjom. Trawa morska została zaanimowana tak, aby poruszała się pod wpływem fikcyjnych prądów morskich, co dodaje dynamiki i realizmu scenerii. Na koniec Wykorzystałem asset Jiggly Bubble do stworzenia efektu podwodnych bąbli powietrza unoszących się ku powierzchni.

*Rysunek 1 finalny rezultat dna oceanu*

## 3.2 Ruch ryb

Skorzystałem z kilku gotowych modeli ryb, tworząc z nich grupy (gromady) i rozmieszczając je losowo w scenie w różnej wielkości, aby poruszały się w różnych kierunkach zapełniając scene. Każdy model został dostosowany pod względem wielkości i orientacji, aby pasował do stylu i skali projektu. Do animacji ryb użyłem Animator Controller w Unity do zarządzania animacjami, gdzie osobno można było poruszać już wbudowanymi „kośćmi" modelu ryby. Sam ruch ryb z kolei zaimplemenotwałem za pomocą skryptu **FishMovement.cs**.




*Rysunek 2 Model ryby z jej szkieletem do animacji*

- **Kluczowe elementy kodu obejmują:**

➢ **Losowe wybieranie punktu docelowego -** Ryby wybierają losowy punkt w przestrzeni i płyną w jego kierunku, co sprawia, że ich ruch wydaje się naturalny i nieprzewidywalny. Ruch ryb jest ograniczony do zdefiniowanego obszaru poprzez zmienne ***minBounds*** i ***maxBounds***, dzięki czemu nie opuszczają one sceny.

```
Odwołania: 2
void SetNewTargetPosition()
{
    float x = Random.Range(minBounds.x, maxBounds.x);
    float y = Random.Range(minBounds.y, maxBounds.y);
    float z = Random.Range(minBounds.z, maxBounds.z);

    targetPosition = new Vector3(x, y, z);
}
```

*Rysunek 3 fragment kodu z losowym dobieraniem punktu docelowego*

- **Obliczanie kierunku ruchu ryby i korekta jej rotacji -** Ze względu na różne orientacje modeli ryb w jednej gromadzie, konieczne było zastosowanie korekty rotacji w kodzie, aby ryby były zwrócone przodem do kierunku ruchu i nie pływały chociażby brzuchem do góry.

```
Vector3 directionXZ = new Vector3(targetPosition.x - transform.position.x, 0, targetPosition.z - transform.position.z).normalized;

if (directionXZ != Vector3.zero)
{

    Quaternion targetRotation = Quaternion.LookRotation(directionXZ) * rotationOffset;
    transform.rotation = Quaternion.Slerp(transform.rotation, targetRotation, turnSpeed * Time.deltaTime);
}
```

*Rysunek 4 fragment kodu odpowiadający za kierunek i rotacje ryby*

- **Napotkane problemy**

Ryby początkowo pływały do góry nogami lub w niepożądanych kierunkach i ciężko było kontrolować ruch całej ławicy na raz. Po wielu próbach i błędach dostosowałem **rotationOffset** oraz obrót modeli w edytorze, aby uzyskać poprawną orientację. Ponadto ryby czasami poruszały się w sposób nienaturalny, np. pionowo lub do tyłu co dziwnie wyglądało. Udało mi się to jednak naprawić poprzez modyfikację kodu i ograniczenie ruchu w osi Y dzięki temu uzyskałem bardziej realistyczne zachowanie.

## 3.3 Śledzenie twarzy i sterowanie kamerą

Do śledzenia ruchu twarzy wykorzystałem bibliotekę **MediaPipe** w Pythonie. Jest to platforma stworzona przez Google, umożliwiająca przetwarzanie danych wizyjnych w czasie rzeczywistym. Skrypt analizuje obraz z kamery internetowej i identyfikuje kluczowe punkty twarzy. Na tej podstawie obliczane są przesunięcia w osiach X, Y i Z.

**W kontekście śledzenia twarzy działa następująco:**

1) **Przechwytywanie obrazu -** Najpierw skrypt wykorzystuje bibliotekę OpenCV do przechwytywania obrazu z kamery. Następnie mediapipe korzysta z zaawansowanych modeli uczenia maszynowego do wykrywania twarzy na obrazie przy pomocy face mesh.

```
#inicjalizacja kamery
cap = cv2.VideoCapture(0)
if not cap.isOpened():
    print("kamera nie dziala")
    exit()

#inicjalizacja mediapipe face mesh
mp_face_mesh = mp.solutions.face_mesh
face_mesh = mp_face_mesh.FaceMesh(static_image_mode=False, max_num_faces=1,
                                  min_detection_confidence=0.5, min_tracking_confidence=0.5)
```

*Rysunek 5 fragment kodu inicjalizacji kamery oraz mediapipe face mesh*

2) **Landmarki twarzy -** Po wykryciu twarzy identyfikowane są punkty charakterystyczne (landmarki), takie jak oczy, nos, usta. W moim projekcie korzystałem z punktu na czubku nosa jako referencji. Dla lepszych rezultatów można ustawić środek głowy przez co ruszanie jedynie twarzą nie spowoduje ruchu kamery jednak w moim przypadku średnio się to sprawdziło. Następnie współrzędne punktów są normalizowane względem rozmiaru obrazu, aby uzyskać wartości niezależne od rozdzielczości kamery. Dodatkowo dodałem funkcjonalność gdzie zasłonięcie landmarku (u nas nosa) powoduje reset pozycji kamery.

```python
#przetwarzanie obrazu i wyszukiwanie punktów charakterystycznych twarzy
results = face_mesh.process(rgb_frame)

if results.multi_face_landmarks:
    face_landmarks = results.multi_face_landmarks[0]

    #pobierz pozycje kluczowych punktów (np nos)
    nose_tip = face_landmarks.landmark[1]  #czubek nosa

    #współrzędne normalizowane na piksele
    x = int(nose_tip.x * frame_width)
    y = int(nose_tip.y * frame_height)
    z = nose_tip.z  # Wartość Z jest już znormalizowana

    #normalizacja pozycji twarzy
    x_norm = (x - frame_width / 2) / (frame_width / 2)
    y_norm = (y - frame_height / 2) / (frame_height / 2)
    x_norm = np.clip(x_norm, -1, a_max: 1)
    y_norm = np.clip(y_norm, -1, a_max: 1)
    #wartość Z z MediaPipe jest ujemna przed płaszczyzną ekranu, więc odwracamy znak
    z_norm = -z
```

*Rysunek 6 fragment kodu inicjalizujący wykrywanie landmarków oraz normalizacja współrzędnych*

3) **Obliczanie przesunięć -** Porównując aktualne pozycje z pozycjami początkowymi, ustawionymi wcześniej, obliczane są przesunięcia w osiach X, Y i Z. Aby ruch kamery był nieustannie płynny, wartości przesunięć są uśredniane za pomocą kolejek by nie było żadnych nagłych przeskoków wartości. Na koniec obliczone przesunięcia są wysyłane do Unity za pomocą protokołu UDP.

```python
    #uśrednianie wartości
    x_queue.append(x_norm)
    y_queue.append(y_norm)
    z_queue.append(z_norm)
    x_avg = np.mean(x_queue)
    y_avg = np.mean(y_queue)
    z_avg = np.mean(z_queue)

    #ustawienie pozycji początkowej
    if initial_z is None:
        initial_x = x_avg
        initial_y = y_avg
        initial_z = z_avg

    #przesunięcia względem pozycji początkowej
    delta_x = x_avg - initial_x
    delta_y = y_avg - initial_y
    delta_z = z_avg - initial_z
    delta_z_scaled = delta_z * 85
    #wysyłanie danych do Unity
    message = f"{delta_x},{delta_y},{delta_z_scaled}"
    sock.sendto(message.encode(), (unity_ip, unity_port))
```

*Rysunek 7 fragment kodu wyliczający przesunięcia oraz uśredniający wartości*

## Integracja z Unity:

W Unity stworzyłem skrypt **EyeTrackingReceiver.cs**, który odbiera dane z Pythonowego skryptu:

1) **Odbiór danych UDP -** Skrypt nasłuchuje na określonym porcie i odbiera wiadomości zawierające przesunięcia w osiach X, Y i Z. Odebrane dane są przekształcane na wartości numeryczne i udostępniane innym komponentom.

```
void ReceiveData()
{
    client = new UdpClient(port);
    while (running)
    {
        try
        {
            IPEndPoint anyIP = new IPEndPoint(IPAddress.Any, 0);
            byte[] data = client.Receive(ref anyIP);
            string text = Encoding.UTF8.GetString(data);

            string[] coords = text.Split(',');
            float deltaX = float.Parse(coords[0], CultureInfo.InvariantCulture);
            float deltaY = float.Parse(coords[1], CultureInfo.InvariantCulture);
            float deltaZ = float.Parse(coords[2], CultureInfo.InvariantCulture);

            Vector3 newFaceDeltas = new Vector3(deltaX, deltaY, deltaZ);

            lock (lockObject)
            {
                faceDeltas = newFaceDeltas;
            }
        }
        catch (System.Exception ex)
        {
            Debug.LogError("Error in ReceiveData: " + ex);
        }
    }
}
```

*Rysunek 8 fragment kodu odpowiadający za odbieranie danych z pythona*

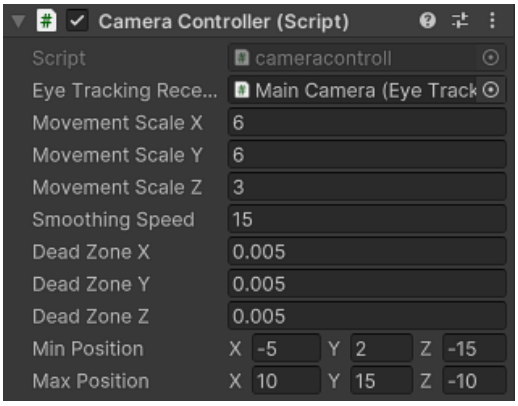
2) **Sterowanie kamerą** - Skrypt **CameraController.cs** wykorzystuje dane z **EyeTrackingReceiver.cs** do sterowania pozycją kamery w Unity. Kamera reaguje oraz skaluje w zależności od potrzeb ruchy głowy użytkownika, przesuwając się w odpowiednim kierunku. Wprowadziłem także strefy martwe, aby drobne ruchy głowy nie powodowały niepożądanych przesunięć kamery. Ponadto dzięki wygładzaniu ruchu **smoothedPosition** kamera porusza się jeszcze płynniej, co zwiększa komfort użytkowania.

```
void Update()
{
    if (eyeTrackingReceiver != null)
    {
        Vector3 faceDeltas = eyeTrackingReceiver.GetFaceDeltas();

        if (faceDeltas == Vector3.zero)
        {
            //jesli brak danych powoli powracaj do początkowej
            smoothedPosition = Vector3.Lerp(smoothedPosition, initialPosition, Time.deltaTime * smoothingSpeed);
            transform.position = smoothedPosition;
            return;
        }

        //zastosuj strefy martwe
        float deltaX = Mathf.Abs(faceDeltas.x) < deadZoneX ? 0 : faceDeltas.x;
        float deltaY = Mathf.Abs(faceDeltas.y) < deadZoneY ? 0 : faceDeltas.y;
        float deltaZ = Mathf.Abs(faceDeltas.z) < deadZoneZ ? 0 : faceDeltas.z;

        //skalowanie ruchu
        deltaX *= movementScaleX;
        deltaY *= movementScaleY;
        deltaZ *= movementScaleZ;

        //oblicz pozycję docelową kamery
        Vector3 targetPosition = initialPosition + new Vector3(-deltaX, -deltaY, deltaZ); //deltaZ +

        //ogranicz pozycję
        targetPosition.x = Mathf.Clamp(targetPosition.x, minPosition.x, maxPosition.x);
        targetPosition.y = Mathf.Clamp(targetPosition.y, minPosition.y, maxPosition.y);
        targetPosition.z = Mathf.Clamp(targetPosition.z, minPosition.z, maxPosition.z);

        //wygładzanie ruchu kamery
        smoothedPosition = Vector3.Lerp(smoothedPosition, targetPosition, Time.deltaTime * smoothingSpeed);

        //aktualizuj pozycję kamery
        transform.position = smoothedPosition;
    }
}
```

*Rysunek 9 fragment kodu odpowiadający za sterowanie ruchem kamery już w unity oraz jego korektą i płynnością*

- ## Napotkane problemy

Implementacja dodatkowych efektów podwodnych jak trzęsienie kamery czy rozmazanie wodą wymagało użycia **Universal Render Pipeline (URP)**. Niestety, napotkałem problemy z kompatybilnością z moim projektem, które powodowały, że shadery nie wczytywały się poprawnie, a wszystkie tekstury stawały się różowe. Starałem się to naprawić jednak finalnie po dostosowaniu tekstur do wymagań program przestał działać, bezpowrotnie niszcząc tekstury wszystkich obiektów sceny przez co musiałem zaczynać od nowa. Po wielu próbach i błędach zdecydowałem się zrezygnować z URP i pozostać przy standardowych ustawieniach, dostosowując efekty do możliwości projektu.




*Rysunek 10 kontroler w unity umożliwiający ustawienie skali ruchu max granic oraz ustawienia wygładzania*

# 4. Testowanie i wyniki

Wstępnie sprawdzałem, jak kamera reaguje na ruchy głowy, dostosowując parametry skalowania i wygładzania a następnie przetestowałem zachowanie ryb w różnych warunkach, upewniając się, że poruszają się płynnie i w obrębie zdefiniowanego obszaru. Udało się stworzyć immersyjne podwodne środowisko z dynamicznymi elementami, takimi jak poruszająca się trawa morska i bąble powietrza. Jak widać na poniższym filmie wszystko działa zarówno ruch w przestrzeni X jak i Y jak i Z a także łączone. W zależności od preferencji można dostosować skalę ruchu w każdym z kierunków a także granice poruszania się kamery czy wygładzenie. Gdy chcemy zresetować położenie kamery wystarczy zakryć nasz kluczowy punkt (u nas nos) i pozycja zostanie zresetowana. Minimalne opóźnienie poruszenia sceny względem kamery wynika z niestabilnego przesyłania informacji jako że nie miałem kamerki internetowej i użyłem kamerki w telefonie która wysyła obraz siecią WiFi.

https://www.youtube.com/watch?v=sAhsR06ssLA

# 5. Wnioski

- Pracując nad tym projektem, połączyłem różne narzędzia i języki programowania, takie jak Unity, C#, Python i MediaPipe. Dzięki temu udało mi się stworzyć interaktywne i immersyjne środowisko podwodne na wzór okna pod wodą.
- Nauczyłem się tworzyć i modyfikować elementy w Unity od ukształtowania terenu, przez modele i tekstury, aż po oświetlenie i cienie. Pozwoliło mi to w pełni dostosować scenerię do mojej własnej wizji artystycznej i nabrać wprawy w samym programie.
- Zmagałem się z wyzwaniami takimi jak problemy z kompatybilnością shaderów i assetów oraz implementacją naturalnego ruchu ryb. Te doświadczenia trochę pomogły mi rozumieć i jak radzić sobie z takimi problemami bardziej technicznymi i złożonymi jak renderowanie czy raytracing.
- Skutecznie połączyłem Unity z pythonem za pomocą mediapipe, co pogłębiło moją wiedzę w zakresie wizji maszynowej i przetwarzania obrazu w czasie rzeczywistym.
- Uzyskane rezultaty stanowią dla mnie solidną bazę do dalszych prac nad projektem. Może kiedyś wprowadze dodatkowe elementy i efekty, by zwiększyć interaktywność oraz trochę udoskonale aspekty wizualne, może by przerobić ten projekt także na jakąś grę.

# 6. Bibliografia i źródła


https://docs.unity.com/

https://github.com/google-ai-edge/mediapipe

https://assetstore.unity.com/

https://www.youtube.com/watch?v=pOo48Vdtwwk&t=233s.

https://www.youtube.com/watch?v=XtQMytORBmM&t=1029s

https://polyhaven.com/textures