P O L I T E C H N I K A Ś W I Ę T O K R Z Y S K A
Wydział Elektrotechniki, Automatyki i Informatyki

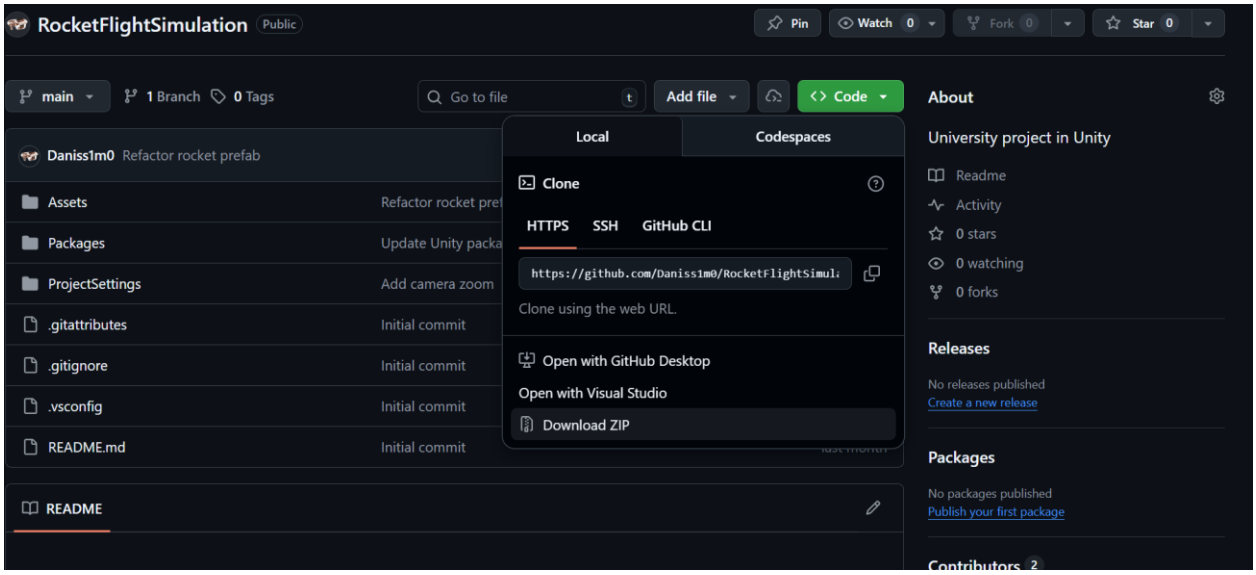
Projekt: Fizyka w animacji i grafice komputerowej		
Temat projektu: **Symulacja lotu rakiety**		
I termin		
Informatyka - IV rok	Wykonali: **Dunytskyi Daniil** **Kozieł Karol** **Rudnytskyi Dmytro**	Grupa: **4ID13A**
Rok akademicki - 2025/2026		Semestr 7

Opis zastosowanych technologii:

- **Użyty język programowania**: C#.
- **Silnik/Biblioteka:** Unity 3D.
- **IDE**: Microsoft Visual Studio 2022 Community Edition.
- **OS**: Windows 11.

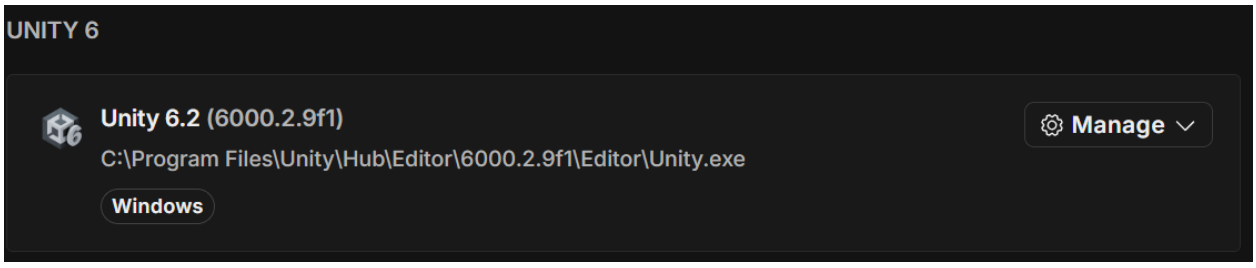
Instrukcja obsługi:

Projekt należy najpierw pobrać z repozytorium zespołu na GitHubie. Należy skopiować adres repozytorium i pobrać projekt za pomocą opcji *Code → Download ZIP* lub wykonać w terminalu polecenie *git clone <adres>*.



Repozytorium na githubie

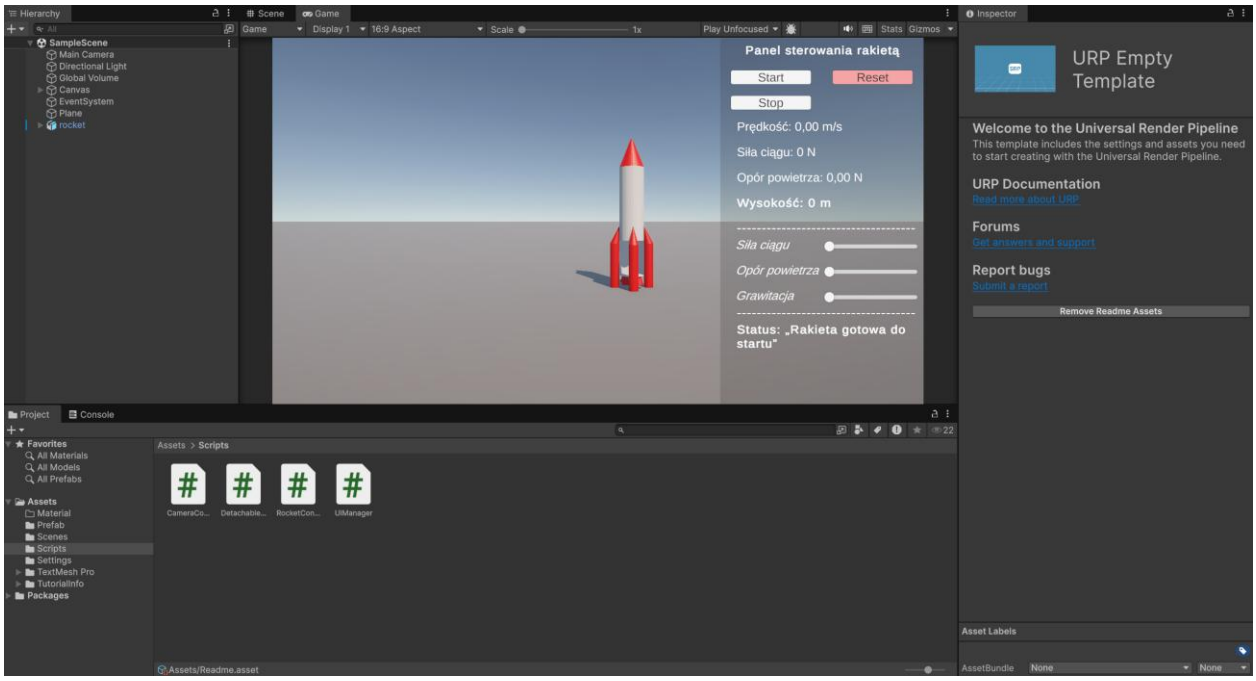
Po sklonowaniu projektu katalog należy otworzyć w Unity Hub i uruchomić projekt w wersji **Unity 6000.2.9f1**. Przed pierwszym uruchomieniem rekomenduje się sprawdzić panel Console w celu wykrycia ewentualnych błędów kompilacji oraz upewnić się, że wszystkie wymagane pakiety i zależności zostały poprawnie zainstalowane.



Silnik

Po otwarciu projektu należy załadować scenę główną (SampleScene.unity). Na scenie znajduje się rakieta umieszczona na platformie startowej oraz panel interfejsu użytkownika umieszczony po prawej stronie ekranu. Panel służy do uruchamiania i sterowania symulacją — udostępnia przyciski *„Start”*, *„Stop”* i *„Reset”* oraz suwaki pozwalające regulować siłę ciągu, współczynnik oporu aerodynamicznego oraz wartość grawitacji. Suwaki działają w czasie rzeczywistym: zmiana

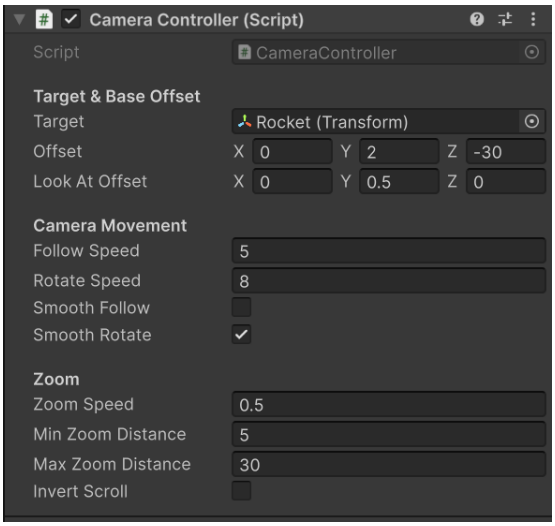
parametrów wpływa natychmiast na przebieg symulacji, co umożliwia szybkie przeprowadzanie eksperymentów bez konieczności ponownego uruchamiania sceny.



Widok sceny

Po naciśnięciu przycisku „*Start"* rakieta zaczyna się poruszać zgodnie z ustawionymi parametrami fizycznymi. W trakcie lotu panel po prawej stronie wyświetla telemetrię i status: wysokość nad ziemią, prędkość pionową, aktualną siłę ciągu oraz siłę oporu aerodynamicznego. Wartości są aktualizowane co klatkę, co ułatwia obserwację wpływu zmiany parametrów na dynamikę lotu. Przycisk *„Stop"* zatrzymuje symulację, natomiast *„Reset"* przywraca rakietę do pozycji początkowej i ponownie przyczepia części odłączone podczas testów, co pozwala na powtarzanie scenariuszy badawczych.

Widok kamery można regulować rolką myszy — przybliżanie i oddalanie działa płynnie, a kamera automatycznie śledzi rakietę z ustawionego kąta widoku. Jeżeli zachodzi potrzeba zmiany kąta widoku, można ją dostosować bezpośrednio w edytorze przez modyfikację transformu obiektu kamery; w standardowym użyciu nie jest to wymagane. Parametry kamery — takie jak prędkość podążania, miękkość obrotu i zakres zoomu — można modyfikować w inspektorze komponentu kamery.



Przykład korzystania ze skryptu w edytorze

Symulacja uwzględnia trzy podstawowe siły: siłę ciągu generowaną przez silnik, działanie grawitacji oraz opór powietrza obliczany według uproszczonego modelu aerodynamicznego. Jeżeli siła oporu działająca na konkretne elementy konstrukcji przekroczy wartość progu wytrzymałości (breakForce), elementy te zostają odłączone od rakiety i stają się niezależnymi obiektami fizycznymi, poddającymi się sile grawitacji i innym siłom zewnętrznym.

Dobór parametrów i ich wpływu na zachowanie rakiety:

Symulacja opiera się na prostych, klasycznych zależnościach mechaniki i aerodynamiki. Najważniejsze wzory użyte w implementacji to

- opór aerodynamiczny

$$F_D = \frac{1}{2}\rho C_D A v^2$$

  gdzie ρ — gęstość powietrza, C_D — współczynnik oporu, A — powierzchnia przekroju, v v — prędkość względem powietrza.
- siła wzdłuż osi rakiety (układ pionowy, kierunek dodatni do góry)

$$F_{net} = T + F_{drag} - mg$$

  T — siła ciągu skierowana ku górze, $W = mg$ — siła ciężkości; F_D działa przeciwnie do kierunku ruchu).
- przyspieszenie

$$a = F_{net}/m$$

- równania ruchu (dyskretne przyrosty czasu Δt)

$$v_{t+\Delta t} = v_t + a\,\Delta t$$
$$y_{t+\Delta t} = y_t + v_{t+\Delta t}\,\Delta t$$

Z powyższych równań wynika jasny wpływ parametrów:

- Siła ciągu T: bezpośrednio zwiększa F_{net}. Aby rakieta uniosła się z miejsca, konieczne jest $T > mg$. Nadmiar ciągu daje dodatnie przyspieszenie.
- Masa m: większa masa obniża przyspieszenie przy tej samej F_{net} (a więc wolniejszy wzrost prędkości i mniejsza wysokość osiągnięta w danym czasie).
- Grawitacja g: zwiększenie g powiększa siłę ciężkości mg i zmniejsza (lub odwraca) F_{net}.
- Współczynnik C_D, pole A, gęstość ρ: zwiększają opór F_D, ich wpływ rośnie gwałtownie wraz ze wzrostem prędkości. Przy dużych prędkościach opór może zdominować siłę ciągu i spowodować odczepienie części (gdy lokalny $F_D > breakForce$).
- Prędkość v: kluczowy czynnik dla oporu - opór rośnie kwadratowo z prędkością, stąd przy doborze T trzeba uwzględnić, że przy wyższych prędkościach potrzebny jest znacznie większy ciąg, by utrzymać lub zwiększyć przyspieszenie.

Proponowany przebieg testu:

A. Test spadania i oporu powietrza — cel: zbadać wpływ oporu na hamowanie

1. Ustawić siłę ciągu na około 30 N.
2. Ustawić grawitację na maksymalną (większa g — agresywna siła ciężkości).
3. Uruchomić symulację i czekać, aż rakieta osiągnie prędkość 100 m/s — (Uwaga: jeżeli przy T = 30 N i przy dużym g rakieta nie osiąga prędkości, zwiększyć T tymczasowo; celem jest uzyskanie stanu z dużą prędkością).
4. Ustawić opór powietrza na maksymalny.
5. Ustawić siłę ciągu na 0.

Oczekiwany efekt: po ustawieniu oporu i odcięciu ciągu rakieta zostanie gwałtownie spowolniona przez F_D; przy wystarczająco dużym oporze spadek prędkości będzie bardzo szybki.

B. Test stanu równowagi — cel: znaleźć warunki równowagi

1. Ustawić siłę ciągu na około 100 N.
2. Ustawić grawitację i opór powietrza na maksymalne wartości.
3. Uruchomić symulację i czekać, aż rakieta osiągnie stan równowagi (prędkość dąży do stałej, $a \approx 0$).
4. Aby zobaczyć wpływ grawitacji na poziom równowagi, chwilowo ustawić grawitację na 0 i obserwować zmianę stanu równowagi.

Oczekiwany efekt: w stanie równowagi $F_{net} \approx 0$, czyli $T \approx mg + F_D$. Przy dużych C_D, A prędkość równowagi będzie mniejsza; przy zmniejszeniu g (do 0) równowaga przesunie się do takiego v, gdzie $F_D \approx T$ (zmiana g pokazuje wkład ciężaru).

C. Test odpadania części — cel: zweryfikować mechanizm odłączania

1. Ustawić grawitację na 0 (żeby odpadanie było sterowane głównie przez opór, nie grawitację).
2. Ustawić opór powietrza na maksymalną wartość.
3. Ustawić siłę ciągu na około 165 N.
4. Uruchomić symulację i czekać, aż odpadną wszystkie części oprócz głównego trzonu rakiety.

Oczekiwany efekt: przy bardzo dużym F_D na poszczególnych częściach wartości te mogą przekroczyć ich *breakForce* i spowodować ich odłączenie.

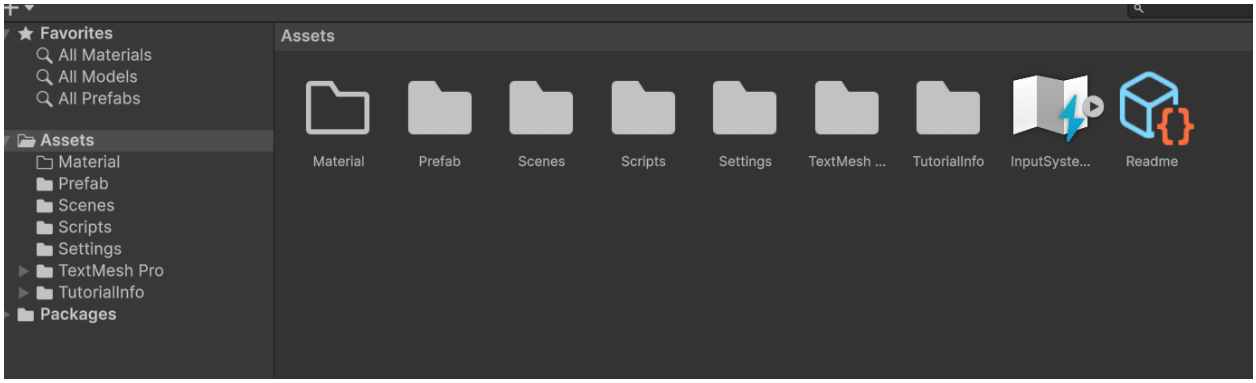
Opis implementacji z fragmentami kodu źródłowego:

Implementacja projektu została podzielona na cztery główne skrypty odpowiadające za logikę fizyczną rakiety, zarządzanie interfejsem użytkownika, obsługę kamery oraz działanie odłączanych elementów aerodynamicznych. Struktura projektu została utrzymana w sposób bardzo prosty i przejrzysty. W głównym katalogu Assets znajdują się jedynie podstawowe foldery niezbędne do działania projektu:

Ogólna struktura gry:

- Material – folder zawierający użyte w scenie materiały
- Prefab – katalog z prefabem rakiety.
- Scenes – folder przechowujący scenę SampleScene.unity.

- Scripts – najważniejszy katalog zawierający cztery skrypty odpowiedzialne za logikę działania aplikacji.



Assets

W ramach folderu Scripts rozmieszczono cztery najważniejsze pliki źródłowe implementujące całą logikę działania symulacji:

- CameraController.cs
- DetachablePart.cs
- RocketController.cs
- UIManager.cs

Poniżej przedstawiono szczegółowy opis ich działania wraz z fragmentami kodu źródłowego ilustrującymi najważniejsze elementy implementacji.

```
public class CameraController : MonoBehaviour
{
    [Header("Target & Base Offset")]
    public Transform target;
    //...

    [Header("Camera Movement")]
    public float followSpeed = 5f;
    //...

    [Header("Zoom")]
    public float zoomSpeed = 0.5f;
    //...

    void Start()
    {
        if (target == null)
        {
            var rc = FindAnyObjectByType<RocketController>();
            if (rc != null) target = rc.transform;
        }

        //...
        if (target != null)
        {
            Vector3 desiredPosition = target.position + offset *
currentZoomMultiplier;
            transform.position = desiredPosition;
            transform.rotation = Quaternion.LookRotation((target.position +
lookAtOffset) - transform.position);
        }
    }
```

```
    void Update()
    {
        HandleZoom();
    }

    void LateUpdate()
    {
        if (target == null) return;

        //...
        if (smoothRotate)
        {
            Quaternion desiredRotation = Quaternion.LookRotation(lookPoint -
transform.position);
            transform.rotation = Quaternion.Slerp(transform.rotation,
desiredRotation, rotateSpeed * Time.deltaTime);
        }
        else
        {
            transform.LookAt(lookPoint);
        }
    }

    private void HandleZoom()
    {
        float scroll = 0f;

        if (Mathf.Abs(scroll) > 0.0001f)
        {
            float direction = invertScroll ? 1f : -1f;
            currentZoomMultiplier += scroll * zoomSpeed * direction;
            //...
            currentZoomMultiplier = Mathf.Clamp(currentZoomMultiplier, minMul,
maxMul);
        }
    }
}
```

CameraController.cs

Skrypt zarządza pozycją i orientacją kamery względem rakiety: na starcie ustawia kamerę w pozycji docelowej, a następnie płynnie podąża za obiektem i obraca się w jego stronę. Obsługuje też przybliżanie/oddalanie za pomocą rolki myszy, z ograniczeniem minimalnej i maksymalnej odległości. Dzięki parametrom można łatwo dostosować płynność śledzenia i czułość zoomu.

```
public class DetachablePart : MonoBehaviour
{
    public float breakForce = 50f;
    //...
    Transform originalParent;
    Vector3 originalLocalPos;
    Quaternion originalLocalRot;
    Rigidbody rb;
    Collider col;

    void Awake()
    {
        originalParent = transform.parent;
        originalLocalPos = transform.localPosition;
        originalLocalRot = transform.localRotation;
        col = GetComponent<Collider>();
        rb = GetComponent<Rigidbody>();

        if (rb != null)
        {
```

```
            rb.isKinematic = true;
            rb.useGravity = false;
        }
        if (col != null)
        {
            col.isTrigger = true;
        }
    }

    public void Detach(Vector3 inheritVelocity, Vector3 flightDir, float
dragForce)
    {
        if (isDetached) return;

        transform.parent = null;

        rb = GetComponent<Rigidbody>();
        if (rb == null) rb = gameObject.AddComponent<Rigidbody>();

        rb.mass = Mathf.Max(0.001f, partMass);
        rb.isKinematic = false;
        rb.useGravity = true;

        rb.linearVelocity = inheritVelocity;

        float impulse = Mathf.Clamp(dragForce * 0.02f, 0f, dragForce * 0.2f);
        rb.AddForce(-flightDir.normalized * impulse, ForceMode.Impulse);
        rb.AddTorque(Random.insideUnitSphere * impulse, ForceMode.Impulse);

        isDetached = true;

        if (col != null)
        {
            col.isTrigger = false;
        }
    }

    public void Reattach(Transform parent)
    {
        if (rb != null)
        {
            Destroy(rb);
            rb = null;
        }

        transform.parent = parent == null ? originalParent : parent;
        transform.localPosition = originalLocalPos;
        transform.localRotation = originalLocalRot;
        isDetached = false;

        col = GetComponent<Collider>();
        if (col != null)
        {
            col.isTrigger = true;
        }
    }
}
```

DetachablePart.cs

Ten skrypt odpowiada za elementy rakiety, które są przyczepione w stanie spoczynkowym, a po spełnieniu warunku przeciążenia przechodzą na „fizyczny” tryb Rigidbody i oddzielają się od korpusu. Metoda *Detach()* nadaje części odziedziczoną prędkość, dodaje impuls i moment obrotowy oraz włącza grawitację, natomiast *Reattach()* przywraca oryginalne położenie i usuwa

Rigidbody. W praktyce skrypt izoluje logikę odłączania i zapobiega kolizjom części podczas gdy są przyczepione.

```
public class RocketController : MonoBehaviour
{
    public float dragCoefficient = 0.1f;
    public float thrust = 100f;
    public float gravity = 9.81f;

    //...;

    DetachablePart[] parts;

    void Start()
    {
        transform.position = new Vector3(transform.position.x, 0.0f,
transform.position.z);
        if (uiManager == null)
        {
            uiManager = FindFirstObjectByType<UIManager>();
            if (uiManager != null && uiManager.rocket == null) uiManager.rocket =
this;
        }
        parts = GetComponentsInChildren<DetachablePart>(true);
    }

    void Update()
    {
        if (!isRunning)
        {
            CheckReattachedPartsOnGround();
            return;
        }

        float vAbs = Mathf.Abs(velocity);
        Vector3 flightDir = velocity >= 0f ? transform.up : -transform.up;

        float sumPartDrag = 0f;
        if (parts != null && parts.Length > 0)
        {
            foreach (var p in parts)
            {
                if (p == null || p.isDetached) continue;
                Vector3 start = p.transform.position + flightDir * 0.01f;
                bool blocked = Physics.Linecast(start, start + flightDir *
partProbeDistance);
                if (blocked) continue;
                float partDrag = 0.5f * airDensity * p.Cd * p.area * vAbs * vAbs;
                sumPartDrag += partDrag;
            }
        }

        dragForce = -Mathf.Sign(velocity) * sumPartDrag;

        float netForce = thrust + dragForce - mass * gravity;
        float acceleration = netForce / mass;

        velocity += acceleration * Time.deltaTime;
        transform.position += new Vector3(0f, velocity * Time.deltaTime, 0f);

        if (transform.position.y < 0f)
        {
            transform.position = new Vector3(transform.position.x, 0f,
transform.position.z);
            velocity = 0f;
```

```
            isRunning = false;
        }

        CheckPartsForDetachment(flightDir, vAbs);
    }

    void CheckPartsForDetachment(Vector3 flightDir, float vAbs)
    {
        if (parts == null || parts.Length == 0) return;
        Vector3 inheritVelocity = new Vector3(0f, velocity, 0f);
        foreach (var p in parts)
        {
            if (p == null || p.isDetached) continue;
            Vector3 start = p.transform.position + flightDir * 0.01f;
            bool blocked = Physics.Linecast(start, start + flightDir *
partProbeDistance);
            if (blocked) continue;
            float partDrag = 0.5f * airDensity * p.Cd * p.area * vAbs * vAbs;
            if (partDrag > p.breakForce)
            {
                p.Detach(inheritVelocity, flightDir, partDrag);
            }
        }
    }

    void CheckReattachedPartsOnGround()
    {
        if (parts == null || parts.Length == 0) return;
        foreach (var p in parts)
        {
            if (p == null) continue;
            if (p.isDetached) continue;
        }
    }

    public void StartSimulation()
    {
        if (!isRunning)
        {
            isRunning = true;
            if (uiManager != null) uiManager.UpdateUI();
        }
    }

    public void StopSimulation()
    {
        isRunning = false;
        if (uiManager != null) uiManager.UpdateUI();
    }

    public void ResetSimulation()
    {
        isRunning = false;
        velocity = 0f;
        transform.position = new Vector3(transform.position.x, 2.5f,
transform.position.z);
        if (parts != null)
        {
            foreach (var p in parts)
            {
                if (p != null) p.Reattach(this.transform);
            }
            parts = GetComponentsInChildren<DetachablePart>(true);
        }
        if (uiManager != null) uiManager.UpdateUI();
```

```
    }

    public string GetStatus()
    {
        //...
    }
}
```

RocketController.cs

Główny skrypt symulacji oblicza siły działające na rakietę: ciąg, skumulowany opór od członów oraz ciężar, następnie integruje przyspieszenie i prędkość, aktualizując pozycję. Dla każdej nieodłączonej części obliczany jest lokalny opór i w przypadku przekroczenia *breakForce* wywoływane jest odłączenie przez *DetachablePart*. Skrypt także obsługuje start/stop/reset symulacji oraz współpracuje z UI, udostępniając wartości telemetryczne.

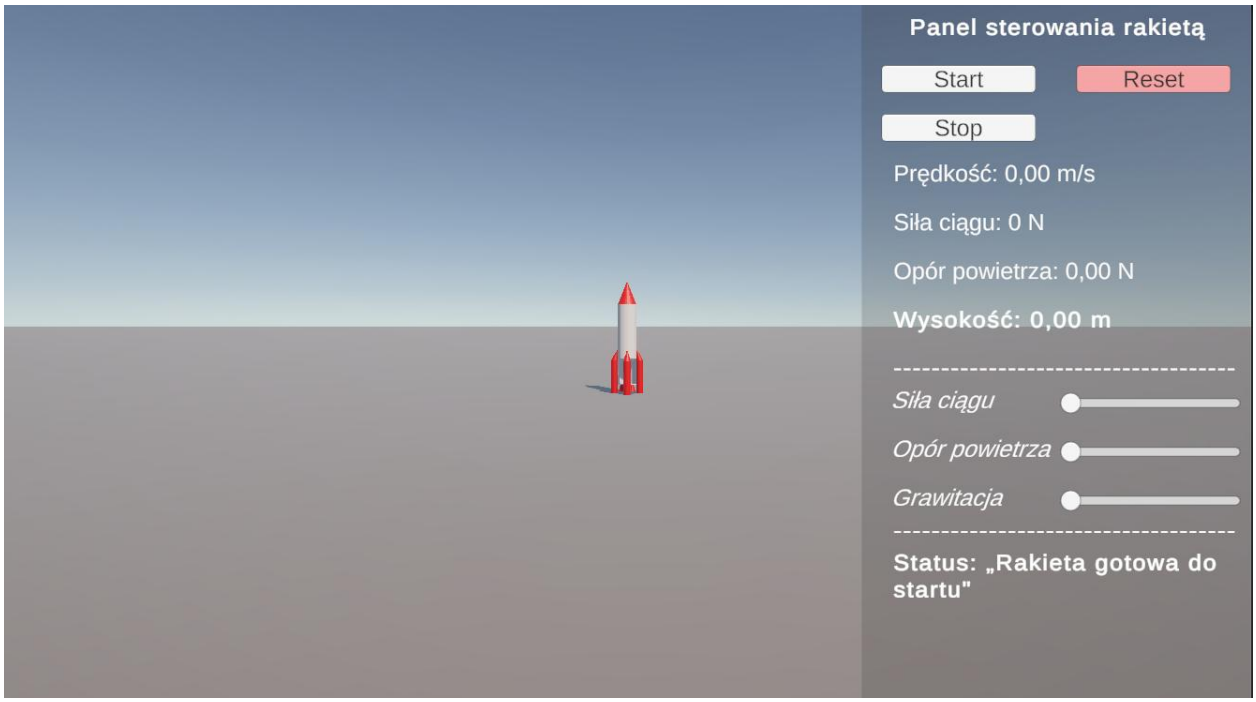
```
using //...

public class UIManager : MonoBehaviour
{
    public Slider airResistanceSlider;
    //...
    public Button startButton;
    //...

    public TextMeshProUGUI statusText;
    //...

    public RocketController rocket;
    public float thrustMultiplier;
    private float densitySliderMultiplier = 0.0001f;

    void Start()
    {
        if (airResistanceSlider != null) airResistanceSlider.minValue = 0f;
airResistanceSlider.maxValue = 2f;
        //...
        UpdateUI();
    }

    void Update()
    {
        UpdateUI();
    }

    public void UpdateUI()
    {
        if (rocket == null)
        {
            speedText.text = "-";
            thrustText.text = "-";
            dragText.text = "-";
            heightText.text = "-";
            statusText.text = "Brak rakiety";
            return;
        }

        //...
    }
}
```

UIManager.cs

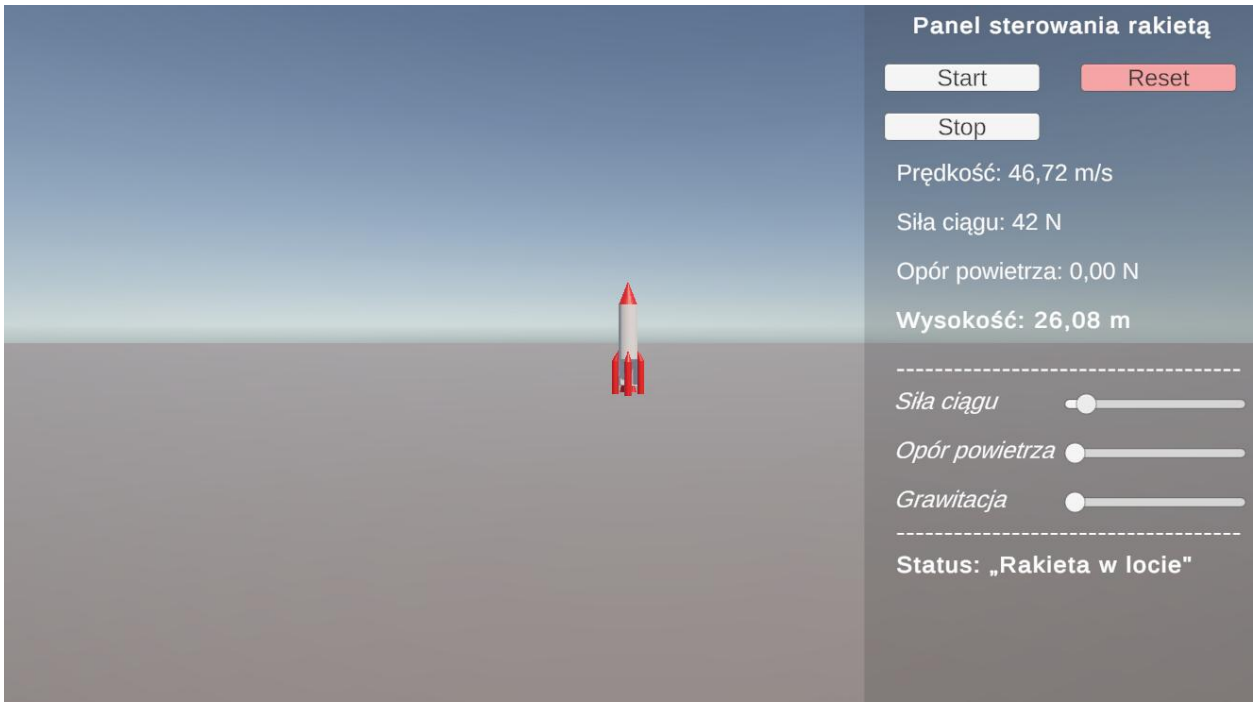
Skrypt łączy elementy interfejsu (suwaki, przyciski, pola tekstowe) z parametrami symulacji i reaguje na zmiany przez natychmiastowe przypisanie wartości do *RocketController* i odświeżenie

wyświetlanych danych. W *Start* podpinane są listenery do suwaków i przycisków, a w *Update* odczytywane są i prezentowane telemetryczne wartości (wysokość, prędkość, siła ciągu, opór). Skrypt organizuje interakcję użytkownika z symulacją i zapewnia wygodny panel kontrolny do prowadzenia testów.

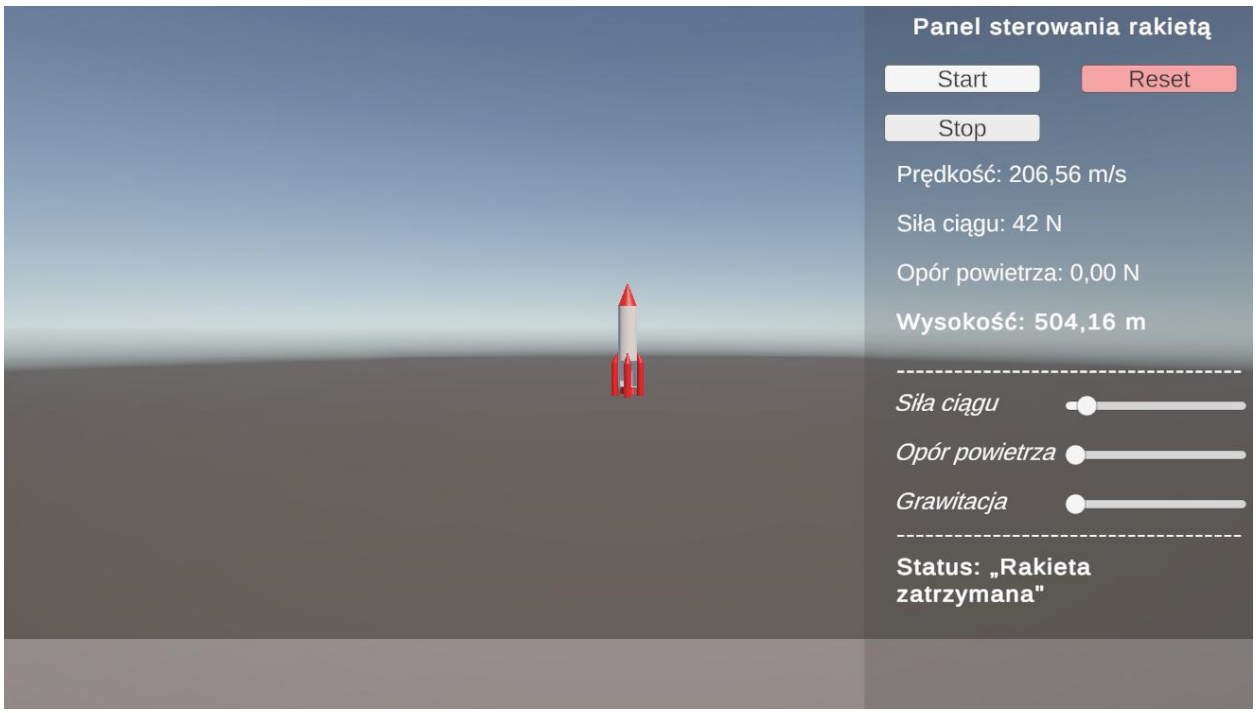
Przykłady działania ze zrzutami ekranu:



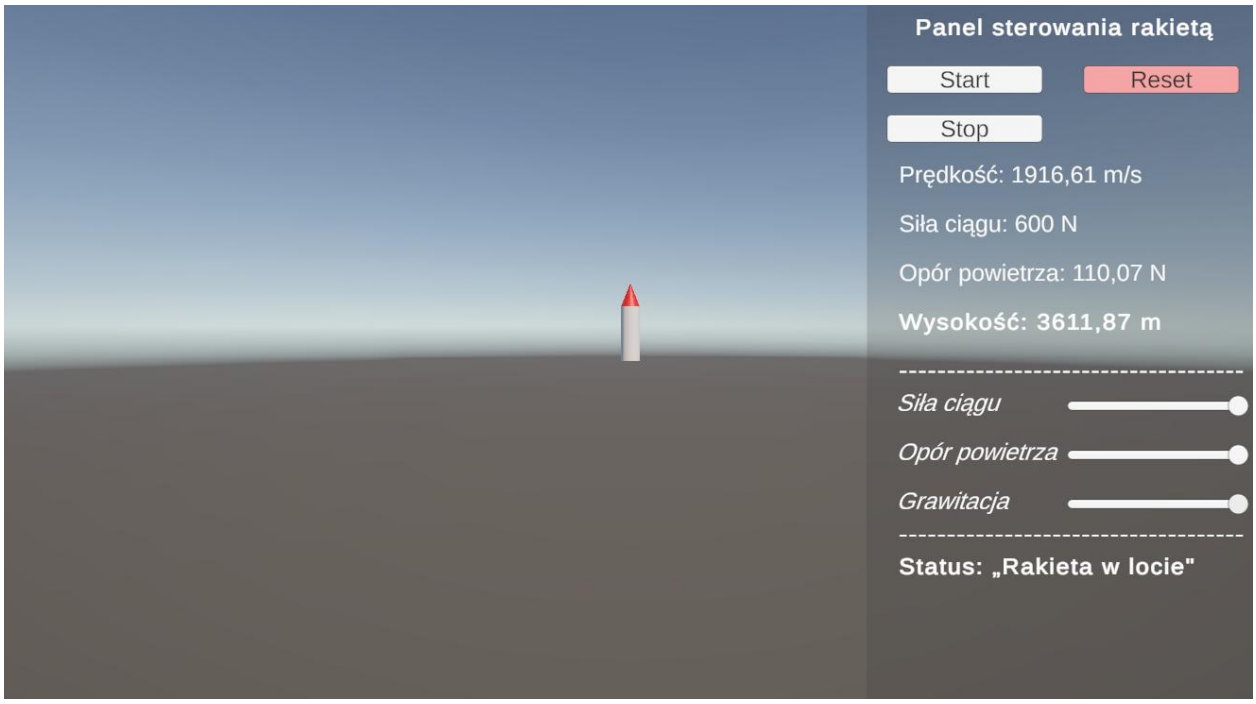
Status: „Rakieta gotowa do startu”



Status: „Rakieta w locie”



Status: „Rakieta zatrzymana”



Rakieta (odpadanie części)

Podsumowanie:

W ramach projektu zrealizowano wszystkie założone wymagania: przygotowano scenę i prefab rakiety, zaimplementowano podstawową fizykę (grawitacja, siła ciągu, uproszczony opór aerodynamiczny), działający mechanizm odłączania części, interfejs użytkownika oraz kamerę z płynnym śledzeniem i zoomem. Przeprowadzono wstępne testy scenariuszy (start, zachowanie przy różnych parametrach, odpadanie części) i wyniki pierwszych eksperymentów. Kod jest podzielony modułowo (RocketController, DetachablePart, CameraController, UIManager), co ułatwia dalszą rozbudowę i testowanie. Jako kierunki rozwoju proponuje się m.in.: wdrożenie dokładniejszego modelu aerodynamiki (przekrojowa projekcja i zmienne Cd), użycie

Joint/ConfigurableJoint dla realistycznego stagingu, dodanie efektów wizualnych i dźwiękowych, logowania wyników testów (CSV) oraz mechanizmu przywracania stanu/restore z prefabów. Te rozszerzenia pozwolą na bardziej wiarygodne symulacje i bogatszą prezentację wyników.