Instrukcja laboratoryjna nr 4

Logika aplikacji, fizyka i obiekty interaktywne

1. Cel i kontekst ćwiczenia

Poprzednie laboratoria pozwoliły nam budować sceny i reagować na wejście użytkownika. Brakowało jednak kluczowego składnika każdej gry i symulacji: fizyki. Robert Nystrom w książce „Game Programming Patterns” zauważa, że logika gry to przede wszystkim zarządzanie stanem: każda interakcja zmienia stan świata, a silnik fizyki jest jednym z mechanizmów, który ten stan aktualizuje w sposób fizycznie wiarygodny. [1]

W tym laboratorium połączymy wszystkie dotychczasowe umiejętności: do naszej sceny dodamy pełną symulację fizyczną, zdarzenia kolizji, wyzwalacze (triggery) oraz prosty system zarządzania stanem aplikacji. Efektem końcowym będzie mini-aplikacja interaktywna z czytelnie zorganizowaną architekturą kodu.

Po zakończeniu tego laboratorium student

1. rozumie różnicę między Rigidbody a Colliderem i wie, kiedy stosować każdy z nich,
2. potrafi obsłużyć zdarzenia kolizji (OnCollisionEnter) i wyzwalacze (OnTriggerEnter),
3. rozumie, dlaczego obliczenia fizyczne należy wykonywać w FixedUpdate(), a nie w Update(),
4. potrafi zaimplementować prosty automat skończony (FSM) do zarządzania stanem aplikacji,
5. potrafi tworzyć i niszczyć obiekty w czasie działania aplikacji (Instantiate / Destroy),
6. potrafi użyć podstawowych technik debugowania (Debug.Log, Gizmos, breakpoint w IDE).

2. Wprowadzenie teoretyczne

2.1. System fizyki PhysX w Unity

Unity korzysta z silnika fizycznego NVIDIA PhysX, który symuluje zachowanie ciał sztywnych (ang. rigid bodies) w przestrzeni trójwymiarowej. Symulacja działa niezależnie od renderowania: jest wykonywana w stałych krokach czasowych (ang. fixed timestep), domyślnie co 0,02 s (50 Hz), co zapewnia deterministyczność: ta sama sekwencja działań zawsze da ten sam wynik fizyczny niezależnie od liczby FPS.

Architektura komponentów fizyki w Unity opiera się na ścisłym podziale odpowiedzialności:

Komponent	Odpowiedzialność	Wymagany do...
Collider	Definiuje kształt geometrii kolizji (invisible shape)	Wykrywania kontaktu fizycznego lub triggera
Rigidbody	Nadaje obiektowi masę i poddaje go symulacji fizycznej	Reagowania na grawitację, siły, impulsy
Physics Material	Określa tarcie i sprystość (bounciness) powierzchni	Realistycznych odbić i poślizgów

Kluczowa zasada: Collider ≠ Rigidbody

Obiekt może mieć Collider bez Rigidbody: będzie wtedy statyczną przeszkodą (np. podłoga, ściana). Obiekt z Rigidbody bez Collidera będzie symulowany fizycznie, ale nie będzie wykrywał kolizji. Pełna symulacja wymaga obu komponentów jednocześnie.

2.2. Rigidbody: właściwości i tryby pracy

Komponent Rigidbody eksponuje szereg parametrów wpływających na zachowanie fizyczne obiektu:

Parametr	Opis	Typowe wartości
Mass	Masa obiektu w kg: wpływa na siłę potrzebną do przyspieszenia i na wynik kolizji	0.1 (piłka) – 1000 (samochód)
Drag	Opór powietrza: zwalnia ruch liniowy w czasie	0 (kosmos) – 5 (woda)
Angular Drag	Opór kątowy: zwalnia obrót w czasie	0.05 (domyślne)
Use Gravity	Czy grawitacja sceny działa na obiekt	true (zwykle)
Is Kinematic	Wyłącza dynamikę: obiekt porusza się tylko przez Transform lub skrypt	false (obiekty swobodne)
Collision Detection	Algorytm wykrywania kolizji: Continuous zapobiega przenikaniu szybkich obiektów	Discrete (domyślne) / Continuous
Constraints	Blokada wybranych osi ruchu lub obrotu	Freeze Rotation XZ (postaci 2D)

2.3. Rodzaje Colliderów

Unity oferuje kilka typów colliderów zoptymalizowanych do różnych kształtów. Koszt obliczeniowy rośnie wraz ze złożonością kształtu:

Typ Collidera	Kształt	Koszt obliczeniowy	Zastosowanie
Box Collider	Prostopadłościan	Najniższy	Skrzynie, ściany, podłogi, prototypowanie
Sphere Collider	Sfera	Najniższy	Piłki, granaty, postaci uproszczone
Capsule Collider	Kapsuła (walec + półsfery)	Niski	Postaci gracza, NPC (standard humanoid)
Mesh Collider (convex)	Wypukła siatka polygonowa	Średnio wysoki	Złożone obiekty dynamiczne
Mesh Collider (concave)	Dowolną siatka polygonowa	Wysoki (tylko static!)	Teren, budynki jako tło statyczne
Terrain Collider	Wygenerowany z heightmap	Wbudowany w Terrain	Tereny Unity Terrain

Mesh Collider concave, a dynamika

Wklęsły (concave) Mesh Collider może być stosowany wyłącznie na obiektach statycznych (bez Rigidbody lub z Is Kinematic = true). Próba użycia wklęsłego Mesh Collidera na dynamicznym Rigidbody spowoduje błąd środowiska uruchomieniowego (runtime error) i wyłączenie kolizji.

2.4. Zdarzenia fizyczne: Collision i Trigger

Unity wywołuje metody cyklu życia skryptu automatycznie w momencie wystąpienia kontaktu między colliderami. Istnieją dwa niezależne mechanizmy:

Mechanizm	Warunek aktywacji	Przekazywany parametr	Typowe zastosowanie
OnCollisionEnter	Dwa Collidery wchodzą w kontakt (normalny)	Collision col	Reakcja na uderzenie, dźwięk, obrażenia
OnCollisionStay	Collidery pozostają w kontakcie	Collision col	Tarcie, stały kontakt
OnCollisionExit	Collidery kończą kontakt	Collision col	Powrót do stanu spoczynku
OnTriggerEnter	Obiekt wchodzi w strefę trigger	Collider other	Strefa aktywacji, pick-up, checkpoint
OnTriggerStay	Obiekt przebywa w strefie trigger	Collider other	Strefa działania, damage-per-second
OnTriggerExit	Obiekt opuszcza strefę trigger	Collider other	Dezaktywacja strefy

Trigger to Collider z zaznaczoną opcją `Is Trigger`. Taki Collider nie blokuje ruchu fizycznego: obiekty przelatują przez niego swobodnie, ale silnik wykrywa moment wejścia, przebywania i wyjścia. Triggery są niewidoczną strefą aktywacji. Przykładowe zastosowania to: drzwi otwierające się po wejściu gracza w strefę, pola zbierania punktów, miny zbliżeniowe lub strefy wyzwalaące cutscenki.

2.5. Wzorzec projektowy: Automat skończony (FSM)

Automat skończony (ang. Finite State Machine, FSM) to jeden z najczęściej stosowanych wzorzeców w programowaniu gier. Nystrom w „Game Programming Patterns” poświęca mu osobny rozdział, podkreślając, że FSM elegancko rozwiązuje problem zarządzania zachowaniem obiektów posiadających wiele wzajemnie wykluczających się stanów. [1]

FSM składa się z trzech elementów:

- Stanów (ang. states): zestawu wzajemnie wykluczających się trybów zachowania (np. Idle, Running, Jumping, Dead),
- Przejść (ang. transitions): warunków, które powodują zmianę stanu (np. naciśnięcie spacji wywołuje przejście z Running do Jumping),
- Akcji (ang. actions): zachowań wykonywanych w danym stanie lub przy przejściu (np. odtworzenie animacji skoku).

W C# najprostszy FSM implementujemy przez typ wyliczeniowy (enum) i instrukcję switch. To podejście jest czytelne, łatwe w rozszerzeniu i wystarczające dla przytłaczającej większości projektów akademickich i indie.

2.6. Tworzenie i niszczenie obiektów w czasie działania

Unity udostępnia dwie statyczne metody klasy `Object`, które pozwalają dynamicznie zarządzać obiektami sceny:

- Instantiate(prefab, position, rotation): tworzy nową instancję prefabrykatu w podanym miejscu.
- Destroy(gameObject, delay): usuwa obiekt ze sceny po opcjonalnym opóźnieniu w sekundach.

Częste tworzenie i niszczenie obiektów generuje obciążenie garbage collectora. W wydajnych aplikacjach (np. duża liczba pocisków) stosuje się wzorzec Object Pool, który recyclinguje nieaktywne obiekty zamiast je niszczyć. Na tych zajęciach stosujemy prostsze podejście Instantiate/Destroy, które jest wystarczające dla realizowanych ćwiczeń.

3. Przygotowanie do ćwiczenia

Scena bazowa	Tworzymy nową scenę ScenaLab04 w folderze _Scenes. Dodajemy podłogę (Plane), kamerę i światło.
Warstwy (Layers)	Zalecane: Edit → Project Settings → Tags and Layers: dodaj warstwę Interaktywny. Ułatwi filtowanie raycastów i kolizji.
Physics Settings	Edit → Project Settings → Physics: Fixed Timestep = 0.02 (domyślne). Gravity = (0, -9.81, 0).
Prefabrykaty	Przygotuj folder Assets/Prefabs. Będziemy tworzyć kilka prefabrykatów obiektów fizycznych.

4. Æwiczenia praktyczne

4.1. Symulacja fizyczna: Rigidbody i kolizje

Zaczniemy od zbudowania prostej sceny fizycznej i zaobserwowania działania silnika PhysX. To ćwiczenie demonstracyjne: jego celem jest wyrobienie intuicji dotyczącej zachowania obiektów fizycznych przed przejściem do kodowania logiki.

Krok 1: Przygotowanie sceny fizycznej

1. Dodajemy do sceny podłogę: Hierarchy → 3D Object → Plane. Position (0, 0, 0), Scale (3, 1, 3). Upewniamy się, że posiada Box Collider (domyślnie dodawany przez Unity).
2. Dodajemy ścianę: Hierarchy → 3D Object → Cube. Ustawiamy Position (0, 1, 5), Scale (6, 2, 0.2). Komponent Box Collider jest domyślnie obecny.
3. Tworzymy obiekt spadający: Hierarchy → 3D Object → Sphere. Position (0, 5, 0).
4. Do obiektu Sphere dodajemy komponent Rigidbody: Inspector → Add Component → Physics → Rigidbody.
5. Uruchamiamy scenę (Play) i obserwujemy: sfera spada pod wpływem grawitacji i odbija się od podłogi.
6. Eksperymentujemy ze wartością Mass i Drag w Rigidbody: obserwujemy wpływ na zachowanie.

Krok 2: Physics Material: sprężystość i tarcie

7. W folderze Assets/Materials tworzymy Physic Material: prawym przyciskiem → Create → Physic Material. Nadajemy nazwę MatFizycznyOdbić.
8. W Inspektorze ustawiamy:
   - Dynamic Friction: 0.4
   - Static Friction: 0.6
   - Bounciness: 0.7 (wartość 0 = brak odbicia, 1 = idealne odbicie)

- Bounce Combine: Maximum

9. Przypisujemy Physics Material do Collidera sfery: zaznaczamy Sphere → Inspector → Sphere Collider → Material.
10. Uruchamiamy scenę ponownie i porównujemy zachowanie z poprzednim uruchomieniem.

4.2. Obsługa zdarzeń kolizji i triggerów

Teraz napisze my skrypty, które reagują na kontakt fizyczny. Ten etap jest fundamentem budowania logiki aplikacji opartej na stanie świata gry.

Krok 1: Skrypt reakcji na kolizję

Tworzymy skrypt C# o nazwie OdpowiedzNaKolizje i przyłączamy go do obiektu Sphere:

```
using UnityEngine;

public class OdpowiedzNaKolizje : MonoBehaviour
{
    [SerializeField] private AudioClip dzwiekKolizji;
    [SerializeField] private float     progSilyDzwieku = 1f;

    private AudioSource audioSource;
    private Renderer    objRenderer;
    private Color       kolorStartowy;

    void Start()
    {
        audioSource  = GetComponent<AudioSource>();
        objRenderer  = GetComponent<Renderer>();
        kolorStartowy = objRenderer.material.color;
    }

    // Wywoływane przez silnik przy pierwszym kontakcie
    void OnCollisionEnter(Collision collision)
    {
        // collision.relativeVelocity: względna prędkość kontaktu
        float sila = collision.relativeVelocity.magnitude;

        Debug.Log($"Kolizja z: {collision.gameObject.name} | Siła: {sila:F2}");

        // Feedback wizualny proporcjonalny do siły uderzenia
        float t = Mathf.Clamp01(sila / 10f);
        objRenderer.material.color = Color.Lerp(kolorStartowy, Color.red, t);

        // Feedback dźwiękowy: tylko gdy siła przekracza próg
        if (sila > progSilyDzwieku && audioSource != null && dzwiekKolizji != null)
            audioSource.PlayOneShot(dzwiekKolizji, Mathf.Clamp01(sila / 10f));
    }

    void OnCollisionExit(Collision collision)
    {
        // Powrót do oryginalnego koloru po oderwaniu
        objRenderer.material.color = kolorStartowy;
    }
}
```

11. Do obiektu Sphere dodajemy również komponent Audio Source (Add Component → Audio → Audio Source). Odznaczamy Play On Awake.
12. Uruchamiamy scenę i obserwujemy zmianę koloru przy uderzeniu.

Krok 2: Strefa Trigger

13. Tworzymy obiekt strefy: Hierarchy → 3D Object → Cube. Zmieniamy nazwę na StrefaAktywacji.
14. Ustawiamy Position (3, 0.5, 3), Scale (2, 1, 2).
15. W komponencie Box Collider zaznaczamy opcję Is Trigger.
16. Usuwamy lub dezaktywujemy komponent Mesh Renderer (opcjonalnie: trigger może być niewidoczny).
17. Tworzymy skrypt StrefaAktywacji i przyłączamy go do obiektu:

```
using UnityEngine;
using TMPro;

public class StrefaAktywacji : MonoBehaviour
{
    [SerializeField] private TMP_Text etykietaStanu;
    [SerializeField] private Color    kolorAktywny   = new Color(0f, 1f, 0f, 0.3f);
    [SerializeField] private Color    kolorNieaktywny = new Color(1f, 0f, 0f, 0.3f);

    private Renderer strefaRenderer;

    void Start()
    {
        strefaRenderer = GetComponent<Renderer>();
        UstawKolor(kolorNieaktywny);
        if (etykietaStanu != null) etykietaStanu.text = "Strefa: pusta";
    }

    void OnTriggerEnter(Collider other)
    {
        Debug.Log($"{other.name} wszedł do strefy.");
        UstawKolor(kolorAktywny);
        if (etykietaStanu != null)
            etykietaStanu.text = $"Strefa: {other.name}";
    }

    void OnTriggerExit(Collider other)
    {
        Debug.Log($"{other.name} opuścił strefę.");
        UstawKolor(kolorNieaktywny);
        if (etykietaStanu != null) etykietaStanu.text = "Strefa: pusta";
    }

    private void UstawKolor(Color c)
    {
        if (strefaRenderer != null) strefaRenderer.material.color = c;
    }
}
```

18. Dodajemy do Canvas (z Lab 3 lub nowy) element TextMeshPro o nazwie TekstStrefy i przeciągamy go do pola Etykieta Stanu w Inspektorze.
19. Uruchamiamy scenę i przemieszczamy sferę w strefę za pomocą siły lub skryptu z Lab 3. Obserwujemy zmianę koloru i tekstu HUDu.

4.3. Logika aplikacji: automat skończony (FSM)

"Jeśli masz do czynienia ze zbiorem zachowań, które wzajemnie się wykluczają, FSM jest najprostszym narzędziem zapewniającym porządek". Zaimplementujemy FSM zarządzający stanem prostej aplikacji: może to być stan rozgrywki, stan obiektu lub stan całej symulacji.

Krok 1: Definicja stanów

Tworzymy skrypt C# o nazwie ZarzadcaStanu. Będzie on obsługiwał cztery stany aplikacji:

```
using UnityEngine;
using TMPro;
public class ZarzadcaStanu : MonoBehaviour {
    public enum StanAplikacji { // Definicja możliwych stanów aplikacji
        Oczekiwanie, // aplikacja czeka na akcję użytkownika
        Aktywny, // symulacja jest w toku
        Pauza, // symulacja wstrzymana
        Koniec // sesja zakończona
    }
    // Bieżący stan: prywatny, zmiana tylko przez metodę ZmienStan()
    private StanAplikacji stanAktualny = StanAplikacji.Oczekiwanie;
    [SerializeField] private TMP_Text etykietaStanu;
    [SerializeField] private GameObject objektFizyczny; // np. Sphere
    void Start() => AktualizujUI();

    // Jedyna metoda zmieniająca stan: centralizuje logikę przejść
    public void ZmienStan(StanAplikacji nowyStan) {
        if (stanAktualny == nowyStan) return;  // brak zmiany

        Debug.Log($"Zmiana stanu: {stanAktualny} → {nowyStan}");
        stanAktualny = nowyStan;
        WykonajAkcjeWejscia(nowyStan);
        AktualizujUI();
    }
    // Akcje wykonywane przy wejściu w nowy stan
    private void WykonajAkcjeWejscia(StanAplikacji stan) {
        switch (stan) {
            case StanAplikacji.Aktywny:
                Time.timeScale = 1f;       // wznów fizykę
                AktywujObiekt(true);
                break;
            case StanAplikacji.Pauza:
                Time.timeScale = 0f;       // zatrzymaj fizykę
                break;
            case StanAplikacji.Koniec:
                Time.timeScale = 0f;
                AktywujObiekt(false);
                break;
            case StanAplikacji.Oczekiwanie:
                Time.timeScale = 1f;
                break;
        }
    }
    private void AktywujObiekt(bool aktywny) {
        if (objektFizyczny != null)
            objektFizyczny.SetActive(aktywny);
    }
    private void AktualizujUI() {
        if (etykietaStanu != null)
            etykietaStanu.text = $"Stan: {stanAktualny}";
    }
    // Metody publiczne wywoływane przez przyciski UI
    public void RozpocznijAktywny() => ZmienStan(StanAplikacji.Aktywny);
    public void PrzePauza() => ZmienStan(
        stanAktualny == StanAplikacji.Aktywny
            ? StanAplikacji.Pauza
            : StanAplikacji.Aktywny);
    public void ZakonczSesje() => ZmienStan(StanAplikacji.Koniec);
    public void Resetuj() => ZmienStan(StanAplikacji.Oczekiwanie);
}
```

20. Tworzymy pusty obiekt GameManager w hierarchii i przyłączamy skrypt ZarzadcaStanu.
21. W Canvas dodajemy trzy przyciski: Rozpocznij, Pauza, Zakończ. Każdy łączymy przez On Click() z odpowiednią metodą ZarzadcaStanu: RozpocznijAktywny, PrzePauza, ZakonczSesje.
22. Dodajemy tekst TekstStanu do Canvas i przeciagamy go do pola Etykieta Stanu w Inspektorze.
23. Uruchamiamy scenę i testujemy przejścia między stanami.

Time.timeScale = 0 a Update()

Ustawienie Time.timeScale = 0 zatrzymuje fizykę (FixedUpdate) i animacje, ale metoda Update() nadal jest wywoływana. Dzięki temu można obsługiwać wejście użytkownika (np. klawisz Escape = wyjście z pauzy) nawet w czasie zatrzymania symulacji.

4.4. Dynamiczne tworzenie obiektów: Instantiate i Destroy

Zbudujemy prosty mechanizm „spawnowania”: dynamicznego tworzenia obiektów w czasie działania aplikacji. To wzorzec powszechny w grach (pociski, wrogowie, efekty cząsteczkowe) i symulacjach (generowanie danych testowych, cząsteczki).

Krok 1: Przygotowanie prefabrykatu

24. Tworzymy nową sferę: Hierarchy → 3D Object → Sphere. Dodajemy do niej komponent Rigidbody.
25. Tworzymy materiał w kolorze jasnoniebieskim i przypisujemy go do sfery.
26. Przeciagamy sferę do folderu Assets/Prefabs. Usuwamy oryginał ze sceny.

Krok 2: Skrypt SpawnerObiektow

Tworzymy skrypt SpawnerObiektow i przyłączamy go do obiektu GameManager:

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

public class SpawnerObiektow : MonoBehaviour
{
    [SerializeField] private GameObject prefabObiektu;
    [SerializeField] private Transform  punktSpawnu; // puste Transform w scenie
    [SerializeField] private float zasieguSpawnu = 3f;
    [SerializeField] private int maksObiektow  = 10;

    // Lista śledzonych instancji: umożliwia usunięcie ich wszystkich naraz
    private List<GameObject> aktywneLiscie = new List<GameObject>();

    // Wywoływana przez przycisk UI lub zdarzenie
    public void SpawnujObiekt()
    {
        if (aktywneLiscie.Count >= maksObiektow)
        {
            Debug.LogWarning("Osiągnięto limit obiektów.");
            return;
        }

        // Losowa pozycja wokół punktu spawnu
        Vector3 offset = new Vector3(
            Random.Range(-zasieguSpawnu, zasieguSpawnu),
            3f,   // wysokość początkowa
```

```
            Random.Range(-zasieguSpawnu, zasieguSpawnu)
        );
        Vector3 pozycja = (punktSpawnu != null ? punktSpawnu.position : Vector3.zero) +
offset;

        // Tworzenie instancji prefabrykatu
        GameObject nowyObiekt = Instantiate(prefabObiektu, pozycja, Quaternion.identity);
        nowyObiekt.name = $"Obiekt_{aktywneLiscie.Count + 1}";
        aktywneLiscie.Add(nowyObiekt);

        Debug.Log($"Stworzono: {nowyObiekt.name} w {pozycja}");
    }

    // Usuwa najstarszy obiekt z listy
    public void UsunNajstarszy()
    {
        if (aktywneLiscie.Count == 0) return;
        GameObject doUsuniecia = aktywneLiscie[0];
        aktywneLiscie.RemoveAt(0);
        Destroy(doUsuniecia);
    }

    // Usuwa wszystkie śledzone obiekty
    public void UsunWszystkie()
    {
        foreach (var obj in aktywneLiscie)
            if (obj != null) Destroy(obj);
        aktywneLiscie.Clear();
    }
}
```

27. Tworzymy pusty obiekt PunktSpawnu w hierarchii (Position np. 0, 0, 0) i przeciagamy go do pola Punkt Spawnu w Inspektorze.
28. Przeciagamy prefabrykat sfery do pola Prefab Obiektu.
29. Dodajemy do Canvas przyciski: Spawn, Usuń najstarszy, Usuń wszystkie. Podłączamy je przez On Click() do odpowiednich metod SpawnerObiektow.
30. Testujemy: klikamy Spawn kilka razy i obserwujemy spadające sfery. Klikamy Usuń i obserwujemy usunięcie obiektów.

4.5. Debugowanie aplikacji

Hocking w „Unity in Action” zwraca uwagę, że debugowanie w Unity wymaga równoległego korzystania z kilku narzędzi: konsoli komunikatów, Gizmos (rysowanie pomoc niczych linii w Scene View) oraz pełnoprawnego debugera IDE. Każde z nich ujawnia inny rodzaj problemów/

Krok 1: Gizmos: wizualizacja danych w Scene View

Dodajemy do skryptu SpawnerObiektow metodę rysującą zasięg spawnu w Scene View:

```
// Dodajemy do klasy SpawnerObiektow:
void OnDrawGizmosSelected()
{
    // Gizmos są widoczne tylko po zaznaczeniu obiektu w edytorze
    Gizmos.color = new Color(1f, 0.3f, 0.3f, 0.4f);
    Vector3 centrum = punktSpawnu != null ? punktSpawnu.position :
transform.position;

    // Rysujemy sześcian reprezentujący obszar spawnu
    Gizmos.DrawWireCube(centrum,
        new Vector3(zasieguSpawnu * 2, 1f, zasieguSpawnu * 2));
}
```

31. Po zapisaniu skryptu zaznaczamy GameManager w hierarchii: w Scene View pojawi się czerwony kontur obszaru spawnu. To sposób wizualizacji danych debugowych bez uruchamiania gry.

Krok 2: Debugowanie przez IDE

32. W Visual Studio lub Rider klikamy na numerek wiersza w skrypcie, w którym chcemy zatrzymać wykonanie: pojawi się czerwona kropka (breakpoint).
33. W edytorze Unity przechodzimy do Edit → Attach Debugger (VS: Debug → Attach to Unity).
34. Uruchamiamy scenę w Unity. Gdy wykonanie dotrze do breakpointu, IDE przejmuje kontrolę.
35. Możemy wtedy:
    - sprawdzić wartości zmiennych lokalnych i pól klasy,
    - wykonać kod krok po kroku (Step Over, Step Into),
    - wpisać wyrażenia w oknie Watch, aby obliczyć dowolne wartości.

Krok 3: Identyfikacja typowych błędów fizycznych

Podczas testowania sceny należy sprawdzić i udokumentować wystąpienie (lub brak) następujących problemów:

Objaw	Prawdopodobna przyczyna	Rozwiązanie
Obiekt przelatuje przez podłogę	Collision Detection = Discrete przy dużej prędkości	Zmień na Continuous lub Continuous Dynamic
Obiekt „trzepie się” na podłodze	Bounciness > 0 i brak tłumienia, lub Rigidbody Sleep Threshold	Zwiększ Drag, zmniejsz Bounciness
Trigger nie działa	Brak Rigidbody na jednym z obiektów lub brak Collidera	Dodaj Rigidbody (może Is Kinematic) lub Collider
FSM nie zmienia stanu	Brak podłączenia przycisku lub zła metoda w On Click()	Sprawdź Inspector, uruchom Debug.Log przy każdym przycisku
Instantiate tworzy w złym miejscu	Brak przypisania Punkt Spawnu w Inspektorze	Przeciągnij Transform do pola lub użyj wartości domyślnej

5. Integracja: mini-aplikacja interaktywna

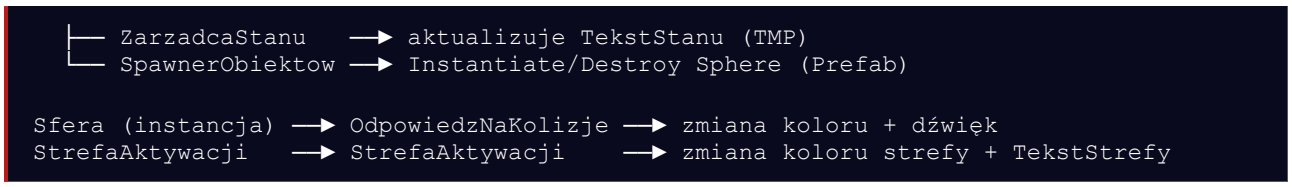
Wszystkie zbudowane elementy: fizykę, triggery, FSM i spawner: łączymy teraz w spójne doświadczenie. Na potrzeby ćwiczenia implementujemy następujący scenariusz:

- Stan Oczekiwanie: scena statyczna, wszystkie obiekty nieaktywne.
- Kliknięcie przycisku Rozpocznij → przejście do stanu Aktywny: spawner zaczyna możliwość tworzenia obiektów, trigger jest aktywny.
- Obiekty fizyczne spadają i wchodzą w kontakt z koliderami: zdarzenia kolizji zmieniają HUD.
- Kliknięcie przycisku Pauza → Time.timeScale = 0, symulacja zatrzymana, UI nadal działa.
- Kliknięcie Zakończ → wszystkie obiekty usunięte, wyświetlony komunikat końcowy.

Poniżej przedstawiony jest schemat połączeń między skryptami:

```
Canvas (UI)
  ├── Przycisk Rozpocznij  ──►  ZarzadcaStanu.RozpocznijAktywny()
  ├── Przycisk Pauza       ──►  ZarzadcaStanu.PrzePauza()
  ├── Przycisk Zakończ     ──►  ZarzadcaStanu.ZakonczSesje()
  ├── Przycisk Spawn       ──►  SpawnerObiektow.SpawnujObiekt()
  └── Przycisk Usuń wszy.  ──►  SpawnerObiektow.UsunWszystkie()

GameManager
```



Zakres implementacji

Nie jest wymagane pełne zaimplementowanie wszystkich powiązań diagramu: wystarczy co najmniej: działający FSM z 3 stanami, jeden trigger, jedna kolizja z reakcją wizualną oraz Instantiate/Destroy choć jednego obiektu. Integracja z HUD (TekstStanu) jest obowiązkowa.

6. Lista kontrolna wykonanych zadań

Przed zakończeniem laboratorium sprawdź, czy wszystkie poniższe elementy zostały zrealizowane:

☐	Scena zawiera statyczne Collidery (podłoga, ściana) i co najmniej jeden obiekt z Rigidbody
☐	Physics Material skonfigurowany z Bounciness > 0 i przypisany do co najmniej jednego Collidera
☐	Skrypt OdpowiedzNaKolizje działa: kolizja wywołuje zmianę koloru obiektu (OnCollisionEnter)
☐	Trigger (StrefaAktywacji) reaguje na wejście i wyjście obiektu (OnTriggerEnter / Exit)
☐	FSM (ZarzadcaStanu) obsługuje co najmniej 3 stany połączone z przyciskami UI
☐	Time.timeScale jest poprawnie przełączane między stanami Aktywny i Pauza
☐	SpawnerObiektow tworzy instancje prefabrykatu i usuwa je (Instantiate / Destroy)
☐	Gizmos rysują zasięg spawnu w Scene View (OnDrawGizmosSelected)
☐	Debugger IDE podłączony do Unity i użyty przynajmniej raz do inspekcji zmiennej
☐	Dokumentacja zawiera tabelę zidentyfikowanych błędów fizycznych i ich rozwiązania

7. Literatura i odwołania

7.1. Pozycje obowiązkowe

[1] Nystrom, R. (2014). Game Programming Patterns. Genever Benning. ISBN 978-0990582908. Rozdziały: „Flyweight” (organizacja danych), „State” (automaty skończone / FSM), „Object Pool” (wydajne tworzenie obiektów).

[2] Hocking, J. (2018). Unity in Action: Multiplatform Game Development in C#. Manning Publications. ISBN 978-1617294969. Rozdziały 4 (fizyka i kolizje), 7 (projektowanie gry).

[3] Unity Technologies. (2024). Unity Manual: Physics overview. Dostępne online: https://docs.unity3d.com/Manual/PhysicsSection.html

7.2. Dokumentacja techniczna

[4] Unity Technologies. (2024). Rigidbody: Unity Scripting API. https://docs.unity3d.com/ScriptReference/Rigidbody.html

[5] Unity Technologies. (2024). MonoBehaviour.OnCollisionEnter. https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnCollisionEnter.html

[6] Unity Technologies. (2024). MonoBehaviour.OnTriggerEnter. https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnTriggerEnter.html

[7] Unity Technologies. (2024). Object.Instantiate. https://docs.unity3d.com/ScriptReference/Object.Instantiate.html

[8] Unity Technologies. (2024). Time.timeScale https://docs.unity3d.com/ScriptReference/Time-timeScale.html

[9] NVIDIA PhysX SDK Documentation. (2023). https://developer.nvidia.com/physx-sdk

7.3. Materiały uzupełniające

Zasób	Temat	URL / źródło
Unity Learn: Physics	Oficjalny kurs fizyki w Unity ze ćwiczeniami	learn.unity.com
Game Programming Patterns: online	Bezpłatna wersja książki Nyströma	gameprogrammingpatterns.com
Microsoft C# Docs: enum	Dok. typu wyliczeniowego i switch	docs.microsoft.com/csharp
Catlike Coding: Object Management	Tutorial dot. Instantiate i Object Pool w Unity	catlikecoding.com