Instrukcja laboratoryjna nr 1

Wprowadzenie do środowiska Unity i organizacji projektu

1. Cel i kontekst ćwiczenia

Pierwsze laboratorium jest technicznym i mentalnym startem kursu. W kolejnych tygodniach będziemy tworzyć mechaniki, interakcje, systemy kamer, prostą fizykę, interfejs użytkownika i elementy rozgrywki. Zanim jednak pojawi się „gra" w klasycznym znaczeniu, trzeba zbudować warsztat pracy: zrozumieć środowisko, sposób organizacji sceny, strukturę plików i reguły komunikacji między edytorem Unity a kodem C#.

Unity warto traktować jak laboratorium czasu rzeczywistego. Każda zmiana może zostać natychmiast uruchomiona, obserwowana i poprawiona. Taka praca różni się od klasycznego programu konsolowego: tutaj nie tylko obliczamy wynik, ale sterujemy obiektami żyjącymi w przestrzeni, reagującymi na czas, wejście użytkownika, światło, kamerę i fizykę.

Unity nie jest tylko programem do „układania klocków". To środowisko, w którym projektant sceny, programista, grafik techniczny i tester spotykają się w jednym narzędziu. Nawet prosta kostka obracająca się na ekranie jest efektem współpracy obiektu, komponentu Transform, renderera, materiału, kamery, światła i skryptu.

Po zakończeniu tego laboratorium student:

- rozumie architekturę silnika czasu rzeczywistego i model komponentowy Unity,
- sprawnie porusza się po interfejsie edytora Unity,
- potrafi samodzielnie utworzyć projekt i zorganizować jego strukturę katalogów,
- potrafi budować scenę 3D z obiektami, kamerą i oświetleniem,
- potrafi napisać i przyłączyć skrypt C# do obiektu gry,
- rozumie cykl życia skryptu i znaczenie metod Start() / Update().

2. Wprowadzenie teoretyczne

Ta część porządkuje pojęcia, które będą wracały przez cały kurs. Nie chodzi o zapamiętanie definicji, lecz o zbudowanie modelu mentalnego: co dzieje się w każdej klatce, czym jest `obiekt sceny`, dlaczego `komponenty` są ważniejsze niż dziedziczenie i jak C# jest integrowany z silnikiem.

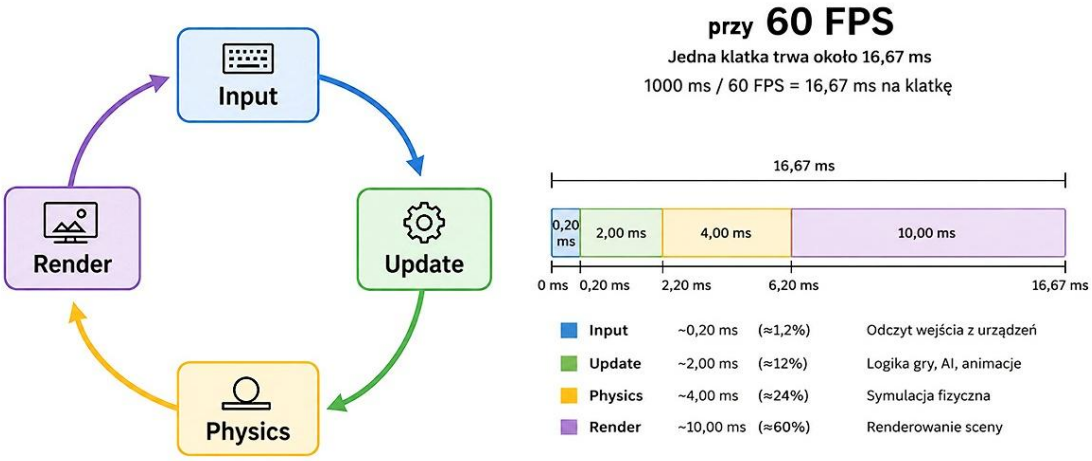
2.1. Architektura silników czasu rzeczywistego

Silniki czasu rzeczywistego, takie jak Unity czy Unreal Engine, działają w oparciu o tzw. pętlę gry, gdzie *(w każdej klatce obrazu)* silnik wykonuje sekwencję operacji:

Silnik czasu rzeczywistego to system, który nie czeka na pojedyncze wejście i nie kończy działania po zwróceniu wyniku. Zamiast tego pracuje w nieustannej pętli *(ang. game loop).* W każdej klatce *(ang. frame)* sprawdza wejście użytkownika, aktualizuje stan świata, wykonuje skrypty, oblicza fizykę, animacje i dźwięk, a następnie renderuje obraz. Ten rytm jest pulsem aplikacji interaktywnej.

W praktyce oznacza to, że gra nie „wie", że kostka ma obracać się przez sekundę. Gra wie, że minęło trochę czasu od poprzedniej klatki, a skrypt ma wykorzystać tę informację do wykonania małego kroku. Po setkach takich małych kroków użytkownik widzi płynny ruch.

- **Odczyt wejścia**: klawiatura, mysz, kontroler, ekran dotykowy lub dane z innego urządzenia
- **Aktualizacja stanu**: skrypty, animacje, sztuczna inteligencja, logika, prosta symulacja świata
- **Fizyka**: kolizje, grawitacja, siły, prędkości i reakcje obiektów
- **Renderowanie**: przygotowanie obrazu z perspektywy kamery i wyświetlenie go w Game View



Rys. 1. Schemat pętli gry

Warto zapamiętać

FPS nie jest tylko liczbą w rogu ekranu. To informacja, ile razy na sekundę aplikacja wykonuje pełny cykl pracy. Spadek FPS oznacza, że użytkownik dostaje rzadsze informacje zwrotne, a sterowanie zaczyna wydawać się mniej responsywne.

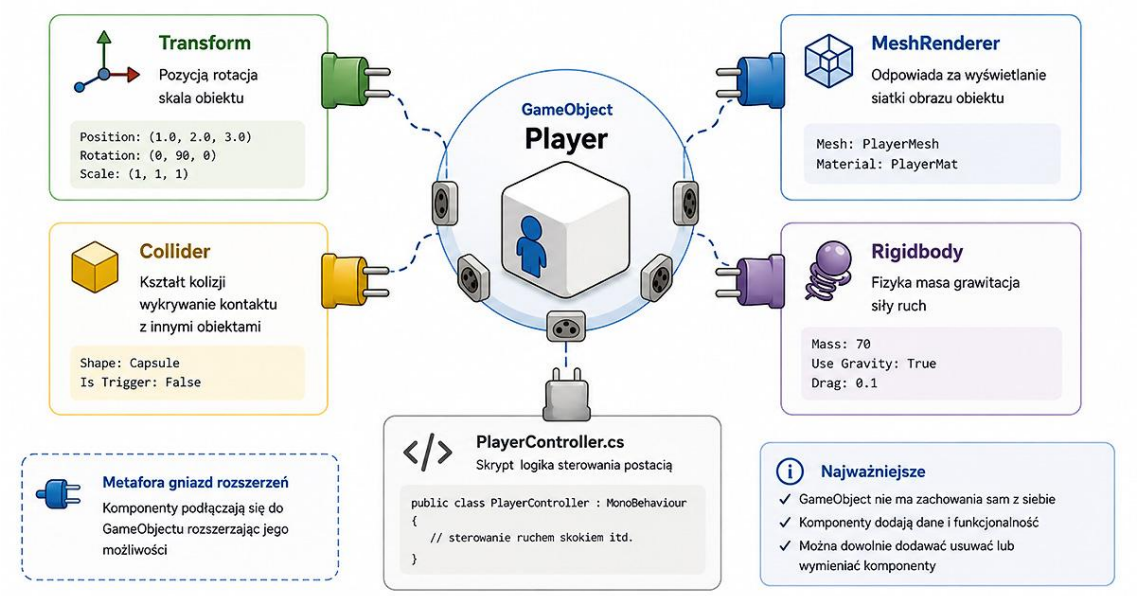
2.2. Model komponentowy (entity-component system)

W klasycznym programowaniu obiektowym często projektujemy hierarchie klas: Pojazd → Samochod → SamochodSportowy. W grach takie hierarchie szybko stają się sztywne. Co zrobić z obiektem, który jest jednocześnie źródłem światła, przeciwnikiem, elementem interaktywnym i emiterem dźwięku? Unity proponuje inne podejście: model komponentowy. Każdy obiekt w scenie, zwany GameObject, jest domyślnie jedynie kontenerem *(nie posiada żadnego zachowania)*. Właściwe cechy i działanie nadają mu dołączane komponenty. Takie podejście zapewnia elastyczność i umożliwia wielokrotne wykorzystanie gotowych elementów.

Najprostszy sposób myślenia jest następujący: GameObject odpowiada na pytanie „co istnieje w scenie?”, komponent odpowiada na pytanie „co ten obiekt potrafi albo jakie ma właściwości?”. Kostka staje się widoczna dzięki MeshRenderer, zajmuje miejsce dzięki Transform, może podlegać fizyce dzięki Rigidbody, a zachowanie otrzymuje dzięki skryptowi C#.

Komponent	Opis i zastosowanie
Transform	Definiuje pozycję, obrót i skalę obiektu: jedyny komponent obowiązkowy
MeshRenderer	Renderuje siatkę 3D na ekranie z uwzględnieniem materiałów
Rigidbody	Włącza symulację fizyczną (grawitacja, kolizje dynamiczne)
Collider	Definiuje kształt detekcji kolizji bez widocznej geometrii
Camera	Określa punkt widzenia renderowanej sceny
Light	Emituje światło wpływające na wygląd materiałów
AudioSource	Odtwarza pliki dźwiękowe powiązane z obiektem

Skrypt C#	Własna logika: dowolne zachowanie zdefiniowane przez programistę



Rys. 2. GameObject jako kontener komponentów

2.3. Scena, hierarchia i prefabrykaty

Scena w Unity jest plikiem opisującym konkretny świat lub fragment aplikacji. Może to być menu główne, poziom gry, ekran testowy albo osobna scena demonstracyjna do eksperymentów. Dobra praktyka polega na tym, aby scena nie była przypadkowym zbiorem obiektów, ale czytelną strukturą, którą można zrozumieć po rzucie okiem na panel Hierarchy.

Hierarchia obiektów ***(assets)*** pozwala budować zależności rodzic-dziecko. Jeżeli obiekt dziecka znajduje się wewnątrz obiektu rodzica, jego pozycja, obrót i skala są interpretowane względem rodzica. To bardzo wygodne przy tworzeniu złożonych obiektów: samochód może mieć koła jako dzieci, postać może mieć broń jako dziecko dłoni, a interfejs może mieć przyciski wewnątrz panelu.

Prefabrykat ***(prefab)*** jest natomiast zapisanym wzorcem obiektu. Warto myśleć o nim jak o przepisie: jeżeli przepis zmienimy, wszystkie dania przygotowane według tego przepisu mogą zostać zaktualizowane. Dzięki prefabrykatom nie powielamy ręcznie konfiguracji obiektów. W małych projektach oszczędza to czas, w dużych projektach zapobiega chaosowi.

Przykład z praktyki

Jeżeli w grze występuje 50 monet do zebrania, nie tworzymy 50 niezależnych obiektów konfigurowanych ręcznie. Tworzymy prefab Coin z modelem, colliderem, animacją i skryptem zbierania, a następnie rozmieszczamy instancje w scenie.

2.4. Cykl życia skryptu w Unity

Skrypt C# dziedziczący po MonoBehaviour jest podłączany do obiektu sceny jako komponent. Unity wywołuje wybrane metody automatycznie w określonych momentach życia obiektu. Programista nie tworzy własnej pętli głównej, tylko dopisuje zachowanie do istniejącej pętli silnika. Skrypt C# automatycznie uczestniczy w pętli gry. Najważniejsze metody cyklu życia, które będziemy wykorzystywać, to:

```
void Awake()         // wywoływana raz przy tworzeniu obiektu (przed Start)
void Start()         // wywoływana raz na początku pierwszej klatki
void Update()        // wywoływana co klatkę: główna logika aplikacji
void FixedUpdate()   // wywoływana ze stałą częstotliwością: obliczenia fizyczne
```

```
void OnDestroy()     // wywoływana przy niszczeniu obiektu
```

Kluczowa różnica między Update(), a FixedUpdate(): pierwsza metoda działa z częstotliwością zależną od FPS (niestabilną), druga: z definiowalną, stałą częstotliwością (domyślnie 50 Hz). Wszelkie obliczenia fizyczne należy wykonywać w FixedUpdate(), aby uniknąć niespójności.

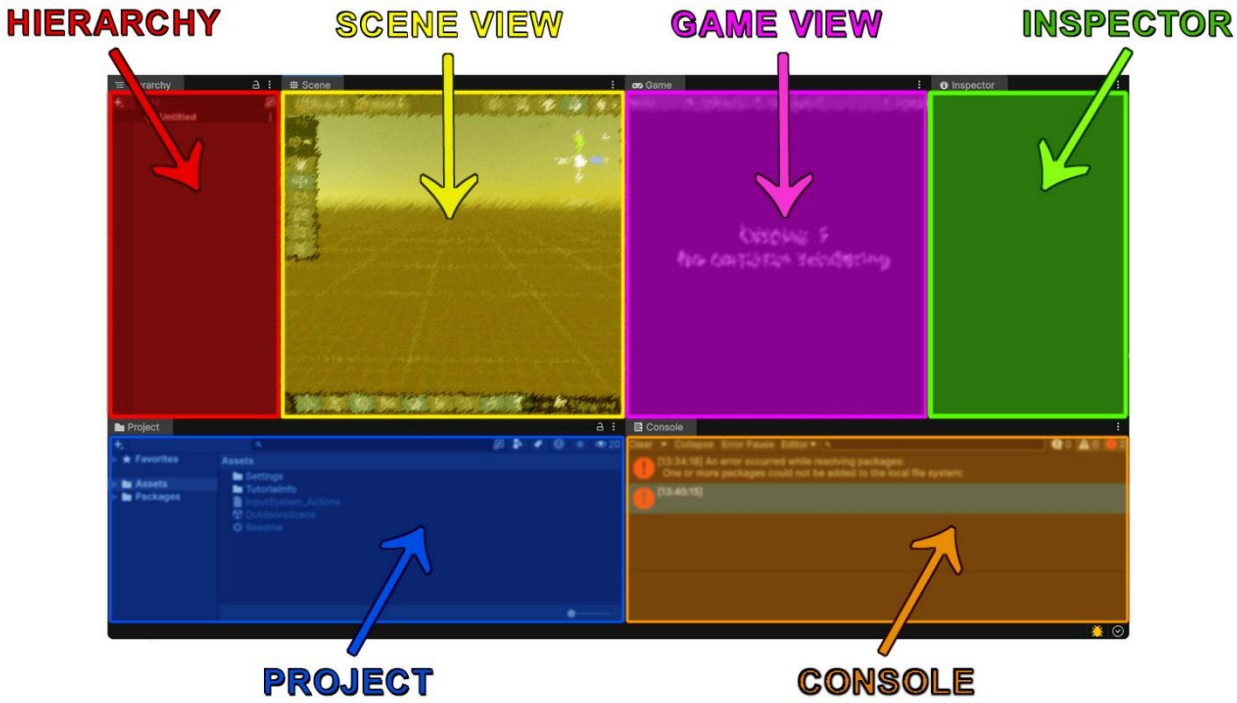
Najczęstszy błąd początkujących polega na traktowaniu Update() jak miejsca na wszystko. Tymczasem Update() jest dobre dla logiki zależnej od klatki i wejścia użytkownika, natomiast FixedUpdate() należy stosować przy pracy z fizyką. Różnica jest istotna, ponieważ liczba klatek na sekundę może się zmieniać, a symulacja fizyczna wymaga stabilnego kroku czasowego.

Time.deltaTime

Jeżeli chcemy obracać obiekt o 90 stopni na sekundę, nie możemy obracać go o 90 stopni w każdej klatce. Przy 30 FPS wykonałby 2700 stopni na sekundę, a przy 120 FPS aż 10800. Dlatego mnożymy prędkość przez Time.deltaTime, czyli czas od poprzedniej klatki.

2.5. Interfejs edytora Unity

Na początku edytor Unity może wyglądać jak zestaw niezależnych okien. W praktyce każdy panel odpowiada na inne pytanie projektowe, a ich znajomość jest niezbędna do sprawnej pracy. Scene View służy do budowania świata, Game View pokazuje to, co zobaczy użytkownik, Hierarchy mówi, jakie obiekty istnieją, Inspector pokazuje ich komponenty, Project przechowuje pliki, a Console informuje, czy kod i projekt działają poprawnie.



Rys. 3. Ekran edytora Unity

Panel	Funkcja
Scene View	Interaktywny widok 3D sceny, gdzie edytujemy obiekty przez przeciąganie
Game View	Symulacja perspektywy kamery w grze/aplikacji -> widok końcowy
Hierarchy	Drzewo wszystkich GameObject-ów w aktywnej scenie

Inspector	Podgląd i edycja komponentów zaznaczonego obiektu
Project	Struktura plików projektu: skrypty, modele, tekstury, sceny
Console	Log komunikatów, ostrzeżeń i błędów w czasie działania aplikacji

3. Przygotowanie środowiska pracy

Dobrze skonfigurowane środowisko skraca czas rozwiązywania problemów. W gamedevie drobne błędy instalacyjne potrafią udawać błędy programistyczne: brak modułu build support, źle ustawiony edytor skryptów albo niezgodne wersje Unity mogą zablokować pracę nad właściwym zadaniem.

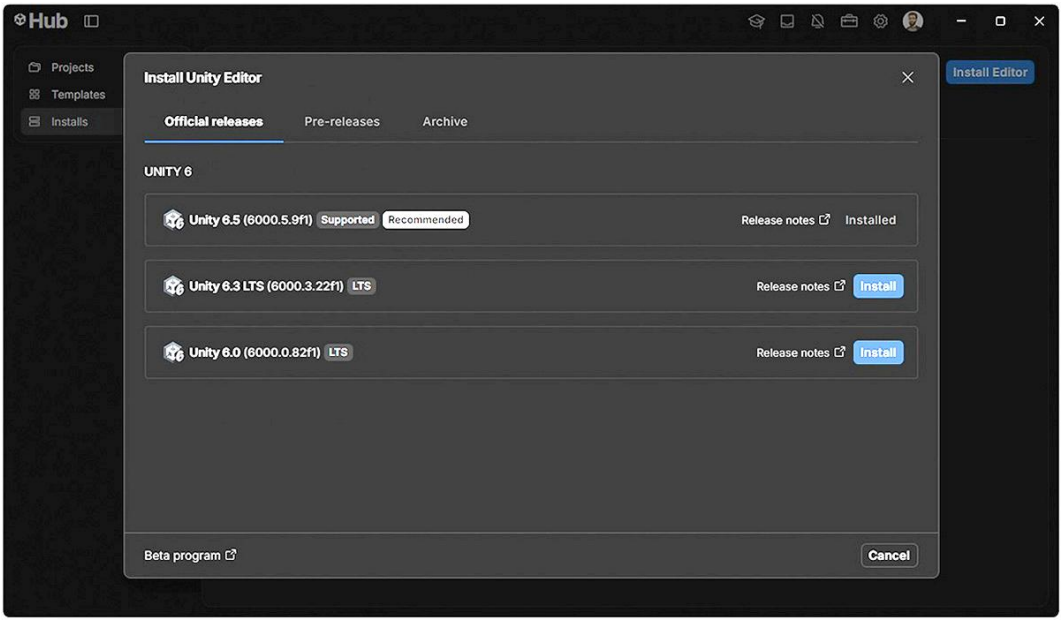
3.1. Wymagania systemowe

System	Windows 10/11 *(64-bit),* macOS 12+, lub Ubuntu 20.04 LTS
Unity	Unity 6 LTS *(zalecana)* lub Unity 2022 LTS: pobierz przez Unity Hub
IDE	Visual Studio 2022 Community lub JetBrains Rider (z wtyczką Unity)
RAM / Dysk	Min. 8 GB RAM *(zalecane 16 GB);* ok. 10 GB wolnego miejsca na dysku

3.2. Instalacja i uruchomienie Unity Hub

Unity Hub jest menedżerem projektów, wersji edytora i licencji. Pozwala mieć równolegle kilka wersji Unity, co jest standardem w pracy zawodowej: starszy projekt może wymagać starszego edytora, a nowy prototyp - wersji aktualnej. Dzięki Hub-owi nie trzeba ręcznie szukać plików wykonywalnych ani pamiętać, który projekt był tworzony w której wersji.

Po instalacji Unity Hub zainstaluj Unity Editor. W tym celu naciśnij Installs → Install Editor → wybierz wersję LTS (long term support)



Rys. 4. Instalacja wersji edytora Unity

Zaznacz moduły: Microsoft Visual Studio Community oraz Windows Build Support

CoreCLR
to nowoczesne, wieloplatformowe środowisko uruchomieniowe (runtime) dla platformy .NET, służące do wykonywania kodu napisanego w językach takich jak C# czy F#

4. Ćwiczenia praktyczne

Część praktyczna prowadzi od pustego projektu do działającej sceny. Pracuj metodycznie: po każdym większym kroku zapisuj projekt i sprawdzaj, czy efekt jest widoczny w edytorze. Dobra praca z Unity polega na krótkich iteracjach: zmiana, uruchomienie, obserwacja, poprawka.

4.1. Utworzenie projektu i organizacja struktury katalogów

Porządek w katalogu Assets jest jednym z najtańszych sposobów uniknięcia problemów. W projekcie jednoosobowym pozwala szybko odnaleźć pliki. W projekcie zespołowym decyduje o tym, czy inni członkowie zespołu są w stanie pracować bez ciągłego pytania, gdzie co się znajduje. Unity daje swobodę, ale ta swoboda wymaga dyscypliny.

Krok 1. Tworzenie projektu

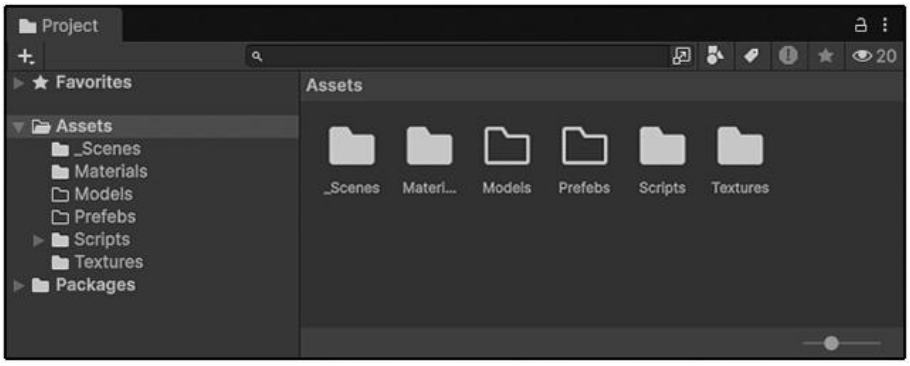
1. Otwieramy Unity Hub i klikamy New Project
2. Wybieramy szablon 3D (Core): nie URP ani HDRP, bo te pipeline'y omawiamy później
3. Nadajemy projektowi nazwę (np. Lab01_Unity_Intro) i wskazujemy katalog docelowy
4. Klikamy Create project i czekamy na inicjalizację (pierwsze uruchomienie trwa kilka minut)

Krok 2. Organizacja struktury katalogów

W panelu Project (folder Assets) tworzymy następującą hierarchię katalogów. Klikamy prawym przyciskiem → Create → Folder:

```
Assets/
├── _Scenes/          ← zapisane sceny projektu
├── Scripts/          ← skrypty C#
├── Prefabs/          ← prefabrykaty obiektów
├── Materials/        ← materiały i shadery
├── Models/           ← modele 3D (.fbx, .obj)
└── Textures/         ← pliki tekstur (.png, .jpg)
```

Podkreślenie na początku nazwy (_Scenes) to konwencja wymuszająca sortowanie tego folderu na szczycie listy. Stosujemy ją dla katalogów używanych najczęściej.



Rys. 5. Panel Project po utworzeniu folderów

Krok 3: Konfiguracja IDE

IDE jest miejscem, w którym piszesz kod. Unity potrafi wygenerować pliki projektu dla Visual Studio lub Ridera, dzięki czemu otrzymujesz autouzupełnianie, podświetlanie błędów, możliwość przechodzenia do definicji klas i wygodniejsze debugowanie. Bez tego skrypty C# szybko staną się nieczytelne i trudne do naprawiania.

1. Przechodzimy do Edit → Preferences → External Tools
2. W polu External Script Editor wybieramy zainstalowaną wersję Visual Studio lub Rider
3. Klikamy Regenerate project files, aby Unity wygenerował pliki .sln

4.2. Budowa pierwszej sceny 3D

Scena jest środowiskiem, w którym osadzone są wszystkie obiekty aplikacji. Każda nowa scena zawiera domyślnie kamerę główną (Main Camera) i źródło światła kierunkowego (Directional Light). Naszym zadaniem jest rozbudowanie tej sceny o dodatkowe elementy. W tym ćwiczeniu nie tworzymy jeszcze rozgrywki, ale budujemy minimalny świat testowy: podłoże, kilka brył, kamerę i światło. Takie sceny testowe są powszechne w produkcji, ponieważ pozwalają szybko sprawdzić mechanikę bez pełnej oprawy graficznej.

Krok 1. Zapis sceny

Unity zapisuje projekt i scenę jako powiązane, ale nie identyczne elementy. Scena zawiera rozmieszczenie obiektów i ich konfigurację. Zapisuj ją regularnie, ponieważ utrata zmian w scenie jest jednym z najczęstszych problemów początkujących.

1. Zapisujemy scenę skrótem Ctrl+S.
2. Nadajemy jej nazwę ScenaLab01 i wskazujemy folder _Scenes

Krok 2. Dodanie podłoża (plane)

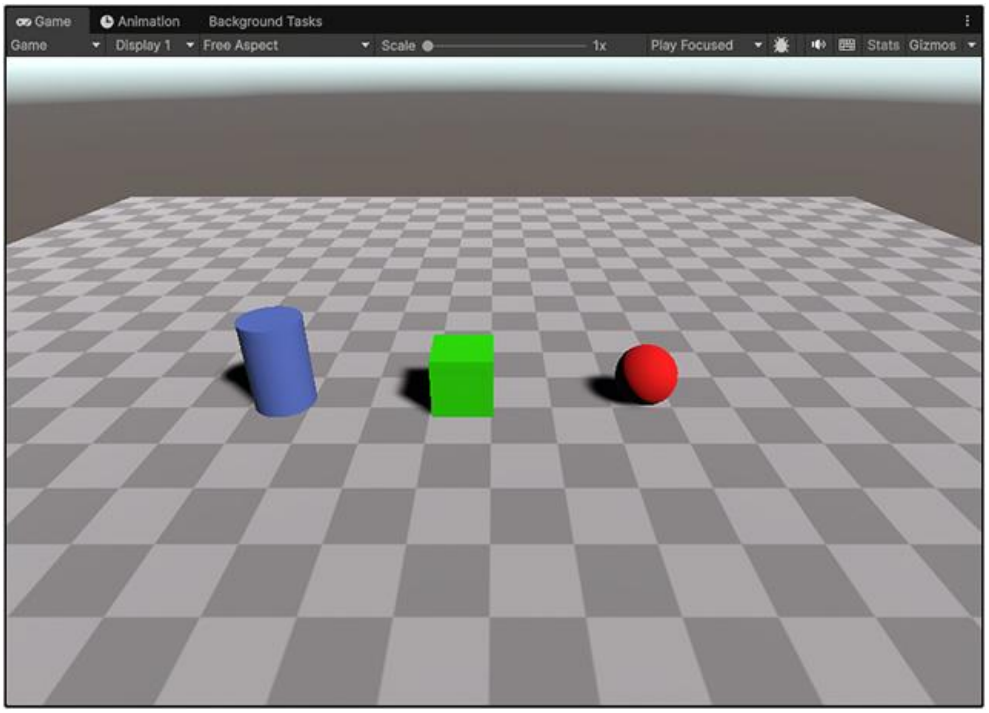
1. W panelu Hierarchy klikamy prawym przyciskiem → 3D Object → Plane
2. W panelu Inspector ustawiamy:
    - Position: (0, 0, 0)
    - Scale: (3, 1, 3): co odpowiada siatce 30×30 jednostek
3. Tworzymy materiał: klikamy prawym przyciskiem w Assets/Materials → Create → Material
4. Nadajemy materiałowi nazwę MatPodloze i przypisujemy mu kolor szary (Albedo)
5. Przeciągamy materiał na obiekt Plane w widoku sceny

> **Jednostki**
> Unity domyślnie traktuje 1 jednostkę jako 1 metr w wielu kontekstach fizycznych. Nie jest to obowiązkowe, ale trzymanie się tej skali ułatwia późniejszą pracę z kamerą, ruchem i fizyką.

Krok 3. Dodanie obiektów 3D

Dodajemy co najmniej trzy różne prymitywy (Hierarchy → prawym przyciskiem → 3D Object):

- Cube: sześcian, ustawiamy w pozycji (0, 0.5, 0)
- Sphere: sfera, ustawiamy w pozycji (3, 0.5, 0)
- Cylinder: walec, ustawiamy w pozycji (-3, 0.5, 0)



Rys. 6. Scena z trzema prymitywami

Obiekt	Pozycja	Materiał	Cel dydaktyczny
Cube	(0, 0.5, 0)	Kolor A	Obiekt, do którego dołączymy skrypt obrotu
Sphere	(3, 0.5, 0)	Kolor B	Porównanie innej geometrii i materiału
Cylinder	(-3, 0.5, 0)	Kolor C	Trzeci punkt odniesienia w scenie

Prymitywy są proste, ale bardzo przydatne: zanim powstaną finalne modele, programiści często używają ich jako zastępników, aby testować mechaniki. Każdemu obiektowi tworzymy odrębny materiał w innym kolorze, aby w scenie widać było wyraźną różnicę pomiędzy elementami. Materiały przechowujemy w Assets/Materials.

Krok 4. Konfiguracja kamery i oświetlenia

1. Zaznaczamy Main Camera i ustawiamy:
    - Position: (0, 85 −8)
    - Rotation: (30, 0, 0)
2. Sprawdzamy w Game View, czy scena jest w pełni widoczna
3. Zaznaczamy Directional Light i ustawiamy:
    - Rotation: (45, −45, 0): oświetlenie półtonowe, które dobrze eksponuje bryły 3D
    - Intensity: 1.0 (wartość domyślna)
4. Opcjonalnie: w Window → Rendering → Lighting można skonfigurować Ambient Color

Test kamery

Po ustawieniu kamery zawsze oceniaj efekt w Game View, nie tylko w Scene View. Scene View jest narzędziem edycji, ale użytkownik zobaczy wyłącznie obraz z kamery.

4.3. Pierwszy skrypt C# i ruch obiektu

Teraz scena przestaje być statyczną makietą. Skrypt C# pozwoli obiektowi zmieniać stan w czasie. Skrypty w Unity to klasy dziedziczące po MonoBehaviour, które Unity automatycznie włącza w pętlę gry. Stworzymy skrypt odpowiedzialny za obrót obiektu Cube wokół osi Y. Zaczynamy od obrotu kostki, ponieważ ten przykład pokazuje trzy fundamenty: komponent jako nośnik logiki, Update() jako metoda wykonywana co klatkę oraz Time.deltaTime jako mechanizm stabilizacji ruchu.

Krok 1. Utworzenie skryptu

1. W Assets/Scripts klikamy prawym przyciskiem → Create → C# Script
2. Nadajemy skryptowi nazwę ObrotObiektu (nazwy klas bez spacji i polskich znaków!)
3. Klikamy dwa razy na plik, aby otworzyć go w IDE
4. Upewnij się, że nazwa pliku i nazwa klasy są identyczne: ObrotObiektu

Krok 2. Implementacja skryptu

```
using UnityEngine;

public class ObrotObiektu : MonoBehaviour
{
    // Prędkość obrotu w stopniach na sekundę
    // [SerializeField] sprawia, że pole widoczne jest w Inspectorze
    [SerializeField] private float predkoscObrotu = 90f;

    void Update()
    {
        // Time.deltaTime to czas od poprzedniej klatki
        // Dzięki temu ruch ma podobna prędkość przy rożnym FPS
        transform.Rotate(0f, predkoscObrotu * Time.deltaTime, 0f);
    }
}
```

Atrybut [SerializeField] jest praktyką często stosowaną w Unity. Udostępnia on udostępnia prywatne pole w panelu Inspector, nie naruszając zasady enkapsulacji *(hermetyzacji zmiennej)*. Pozwala więc programiście zmienić jego wartość bez udostępniania pola jako publicznego elementu API klasy. Dzięki temu kod pozostaje bardziej kontrolowany, a jednocześnie wygodny do testowania. Jest to preferowana praktyka zamiast deklarowania pól jako public.

Inspector z polem predkoscObrotu, na którym widać komponent ObrotObiektu i edytowalne pole Predkosc Obrotu.

Krok 3. Przyłączenie skryptu do obiektu

1. Zapisujemy skrypt (*Ctrl+S* w IDE)
2. W edytorze Unity zaznaczamy obiekt Cube w hierarchii
3. Przeciągamy plik skryptu z panelu Project na obiekt Cube w hierarchii LUB klikamy Add Component w Inspektorze i wyszukujemy ObrotObiektu

4. Klikamy przycisk Play (▶) i obserwujemy obrót kostki w Game View
5. Prędkość można zmieniać na żywo w panelu Inspector w trakcie odtwarzania

> **Uwaga na tryb Play**
>
> Zmiany wprowadzone podczas działania trybu Play zwykle nie są zapisywane po zatrzymaniu symulacji. To celowe: możesz eksperymentować bez ryzyka trwałego zepsucia konfiguracji. Jeżeli znajdziesz dobrą wartość parametru, zapisz ją sobie i ustaw ponownie po wyjściu z Play Mode.

4.4. Tworzenie prefabrykatu i zarządzanie instancjami

Prefabrykaty (prefabs) są jednym z najważniejszych mechanizmów Unity, ponieważ pozwalają zdefiniować pojedynczy obiekt, ze wszystkimi komponentami i skryptami w jednym miejscu i wielokrotnie go instancjonować w scenie. W produkcji gier prefabrykatem może być pocisk, wróg, drzwi, moneta, przycisk interfejsu albo cała sekcja poziomu. W tym laboratorium prefabrykat utworzymy z obracającej się kostki.

Krok 1. Tworzenie prefabrykatu

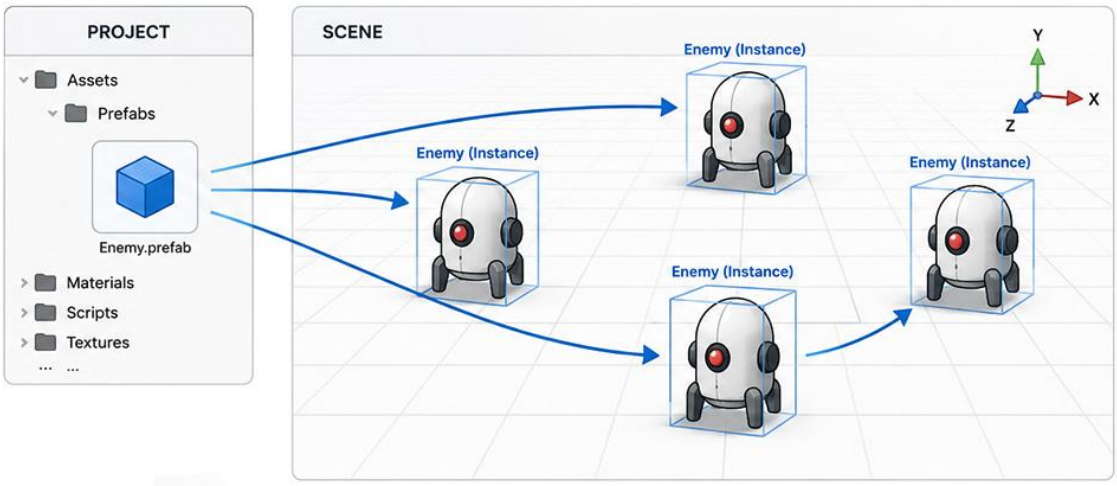
1. Zaznaczamy obiekt Cube (z przypisanym skryptem ObrotObiektu) w hierarchii
2. Przeciągamy go do katalogu Assets/Prefabs w panelu Project
3. Unity automatycznie zamienia obiekt w prefabrykat: ikonę obiektu w hierarchii zmieni kolor na niebieski

Krok 2. Instancjonowanie prefabrykatu

1. Przeciągamy prefabrykat z panelu Project do widoku sceny kilka razy
2. Każdą instancję umieszczamy w innym miejscu (zmieniamy Position w Inspector)
3. Uruchamiamy scenę (Play) i potwierdzamy, że wszystkie instancje obracają się niezależnie

Krok 3. Edycja prefabrykatu (prefab mode)

1. Klikamy podwójnie na prefabrykat w hierarchii lub panelu Project, aby wejść w Prefab Mode
2. Zmieniamy kolor materiału lub dodajemy komponent: zmiana dotknie wszystkich instancji
3. Zamykamy Prefab Mode przyciskiem cofania (strzałka w lewo obok nazwy sceny)



Rys. 7. Prefab i instancje

5. Dokumentacja projektu

Dokumentacja jest nieodłącznym elementem procesu wytwarzania oprogramowania. Jej celem nie jest formalne wymogi zaliczeniowe, lecz utrwalenie wiedzy o podjętych decyzjach projektowych. Zmusza do nazwania decyzji, opisania struktury i wskazania, dlaczego coś zostało zrobione w określony sposób. W pracy zawodowej dobra dokumentacja skraca „onboarding" nowych osób i ułatwia powrót do projektu po przerwie. Informacje te okażą się cenne przy powrocie do projektu po dłuższej przerwie lub przy przekazaniu projektu innemu programiście.

W ramach tego laboratorium dokumentacja powinna być krótka, konkretna i oparta na faktach z wykonanego projektu. Nie chodzi o referat o Unity, lecz o zapis: co zostało utworzone, jak jest zorganizowane i jak działa.

5.1. Zakres dokumentacji

W ramach laboratorium należy przygotować krótki dokument tekstowy (plik .md /.pdf) zawierający:

- nazwę projektu, datę i autora,
- opis struktury katalogów projektu ze zrzutem ekranu panelu `Project`,
- opis hierarchii obiektów w scenie (lista obiektów i ich powiązań `parent-child`),
- opis zastosowanych komponentów z uzasadnieniem wyboru,
- omówienie napisanego `skryptu C#`: co robi i dlaczego tak został zaprojektowany,
- zrzut ekranu działającej sceny w trybie `Game View`
- Krótka refleksja: co było najłatwiejsze, co sprawiło trudność i co warto sprawdzić przed następnymi zajęciami.

mini-szablon dokumentacji

````
# Laboratorium 1 – Wprowadzenie do Unity
**Autor:** Imię i nazwisko
**Data:** RRRR-MM-DD
**Nazwa:** Lab01-Unity-Intro

## 1. Cel
Celem ćwiczenia było ...

## 2. Struktura projektu
Projekt został uporządkowany w następującej strukturze katalogów: ...
Krótko opisz przeznaczenie katalogów oraz ewentualne odstępstwa od powyższej struktury.

## 3. Scena
**Nazwa sceny:** ScenaLab01

Najważniejsze obiekty: ...
Opisz zastosowane materiały, położenie obiektów oraz ewentualne zależności parent–child.

## 4. Skrypt
**Nazwa skryptu:** ...
**Obiekt:** ...
Zadaniem skryptu jest ...

Najważniejszy fragment:
```csharp
...
```
`Update()` jest wykonywana w każdej klatce. Zastosowanie `Time.deltaTime` powoduje, że ...
Parametr `predkoscObrotu` został udostępniony w Inspectorze przy użyciu atrybutu ...

## 5. Prefabrykaty
**Nazwa prefabrykatu:** `Cube`
Prefabrykat utworzono na podstawie obiektu ...

**Liczba instancji w scenie:** ...
Zweryfikowano, że: ...

## 6. Problemy i rozwiązania
| Problem | Przyczyna | Rozwiązanie |
|---|---|---|
````

```
| np. skrypt nie pojawiał się w Add Component | błąd kompilacji | poprawiono błąd wskazany
w Console |
| ... | ... | ... |

Jeżeli podczas realizacji ćwiczenia nie wystąpiły problemy, wpisz:
> Nie odnotowano istotnych problemów technicznych.

## 7. Zrzuty ekranu
### Rys. 1. Struktura projektu
Wstaw zrzut panelu Project

### Rys. 2. Hierarchia sceny
Wstaw zrzut panelu Hierarchy

### Rys. 3. Skrypt jako komponent
Wstaw zrzut Inspectora obiektu Cube z komponentem ObrotObiektu

### Rys. 4. Działająca scena
Wstaw zrzut Game View wykonany w trybie Play

## Krótka refleksja
**Najłatwiejszy element:**
.....
**Największa trudność:**
.....
**Co warto sprawdzić lub powtórzyć przed kolejnym laboratorium:**
.....
```

5.2. Dobre praktyki

Poniższe zasady warto stosować od pierwszych zajęć. Są proste, ale wynikają z realnych problemów produkcyjnych: nieczytelnych nazw, nieuporządkowanych zasobów, kodu bez intencji i scen, których nikt nie potrafi odtworzyć.

Zasady, które warto stosować od pierwszych zajęć

1. Nadawaj opisowe, angielskie nazwy bez spacji *(np. RotatingCube zamiast kostka1)*
2. Każdy folder trzyma jeden rodzaj zasobów: nie mieszaj skryptów z modelami
3. Komentuj w kodzie „DLACZEGO” a nie „CO”: sam kod opisuje co robi
4. Zapisuj scenę przed każdą istotniejszą zmianą (Ctrl+S)
5. Testuj często, po każdej istotnej zmianie klikaj `Play`
6. Czytaj `Console`. Czerwony błąd w `Console` jest częścią procesu, nie powodem do paniki. Komunikat błędu zwykle wskazuje plik, linię i przyczynę problemu
7. Nie poprawiaj wszystkiego naraz. Zmieniaj jeden element, testuj i dopiero idź dalej

Praktyka programisty gier

Najlepsi twórcy gier nie unikają błędów - szybko je izolują. Jeżeli po zmianie skryptu, materiału i kamery coś przestaje działać, nie wiadomo, która zmiana była przyczyną. Małe iteracje dają większą kontrolę.

6. Lista kontrolna wykonanych zadań

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

☐	Struktura katalogów projektu zorganizowana wg przedstawionego schematu
☐	Scena zawiera podłoże (`Plane`) z własnym materiałem

☐	W scenie znajdują się co najmniej 3 różne prymitywy 3D z różnymi materiałami
☐	Kamera poprawnie obejmuje całą scenę (widoczne w Game View)
☐	Oświetlenie kierunkowe skonfigurowane (obrót 45/−45/0)
☐	Skrypt C# ObrotObiektu napisany i przyłączony do obiektu Cube
☐	Prefabrykat Cube utworzony i umieszczony w scenie kilkakrotnie
☐	Skrypt działa poprawnie: obiekt obraca się w trybie Play
☐	Dokumentacja projektu przygotowana w formacie .md lub .pdf

7. Materiały uzupełniające

Materiały poniżej warto traktować jako mapę dalszej nauki. Unity jest duże, ale nie trzeba poznawać wszystkiego naraz. Po tym laboratorium szczególnie przydatne będą zasoby dotyczące podstaw edytora, komponentów, skryptów MonoBehaviour i przepływu pracy z prefabrykatami.

Zasób	Co warto przejrzeć	Dlaczego warto
Unity Learn	Unity Essentials, Junior Programmer	Oficjalne, praktyczne moduły prowadzące od podstaw do pierwszych mechanik
Unity Manual	GameObjects, Components, Scenes, Prefabs, Scripting	Dokumentacja techniczna edytora i najważniejszych pojęć
Unity Scripting API	MonoBehaviour, Transform, Time, GameObject	Referencja klas i metod używanych w kodzie C#
Unity in Action - J.Hocking	Rozdziały 1-3	Dobre wprowadzenie do sposobu myślenia o Unity jako środowisku produkcyjnym
Game Programming Patterns	Prototype, Component, Game Loop	Wzorce projektowe opisane w kontekście gier
Microsoft C# Documentation	Klasy, pola, modyfikatory dostępu, typy wartości	Uzupełnienie wiedzy o języku C#, bez którego skrypty Unity będą trudne do rozwijania

8. Propozycje rozszerzeń dla chętnych

Jeżeli zakończysz ćwiczenie przed czasem, wykonaj jedno z rozszerzeń. Nie są wymagane do zaliczenia podstawowego laboratorium, ale pomagają szybciej przejść od statycznej sceny do myślenia kategoriami mechanik gry.

1. Dodaj drugi skrypt, który porusza Sphere w górę i w dół przy użyciu Mathf.Sin(Time.time)
2. Dodaj pusty obiekt Empty o nazwie SceneRoot i uporządkuj pod nim wszystkie obiekty sceny.
3. Utwórz różne wartości predkoscObrotu dla instancji prefabrykatu i porównaj efekt.
4. Dodaj prosty materiał emisyjny do jednego obiektu i sprawdź, jak zmienia się jego wygląd.
5. Wstaw Debug.Log() w metodzie Start() i Update(), a następnie zobacz w Console, jak często są wywoływane.

Zadanie bonusowe

Zaprojektuj własną mini-instalację kinetyczną z prymitywów 3D.
trzy obiekty, trzy materiały, trzy różne prędkości obrotu".

9. Najczęstsze problemy i szybka diagnostyka

Problem	Prawdopodobna przyczyna	Co sprawdzić
Skrypt nie pojawia się w Add Component	Błąd kompilacji lub niezgodna nazwa klasy i pliku.	Otwórz Console, popraw błędy, sprawdź nazwę ObrotObiektu.
Obiekt nie obraca się	Skrypt nie jest dołączony, predkoscObrotu wynosi 0 albo nie uruchomiono Play Mode.	Sprawdź Inspector obiektu Cube i wartość pola.
Scena jest pusta w Game View	Kamera patrzy w złą stronę albo obiekty są poza kadrem.	Zaznacz Main Camera i skontroluj Position oraz Rotation.
Materiały nie są widoczne	Materiał nie został przypisany albo obiekt nie ma MeshRenderer.	Przeciągnij materiał na obiekt i sprawdź komponenty.
Zmiany zniknęły po Play	Parametry były zmienione w trybie Play.	Zanotuj wartość i ustaw ją ponownie po zatrzymaniu symulacji.
Prefab nie aktualizuje instancji	Zmieniono tylko lokalną instancję albo nadpisano override.	Otwórz Prefab Mode i sprawdź, czy edytujesz źródłowy prefab.

Podsumowanie

Po tym laboratorium masz pierwszy kompletny mikro-pipeline pracy w Unity: projekt, porządek w zasobach, scena, obiekty, materiały, kamera, światło, skrypt, prefabrykat i dokumentacja. To niewielki przykład, ale obejmuje fundamenty, na których buduje się większe systemy interaktywne.