Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH



Instytut Automatyki i Informatyki Stosowanej

Praca dyplomowa magisterska

na kierunku Informatyka
w specjalności Inteligetne Systemy

Porównanie wydajności i możliwości współczesnych silników gier komputerowych

inż. Krzysztof Rudnicki
Numer albumu 307585

promotor
dr inż. Michał Chwesiuk

WARSZAWA 2026

Porównanie wydajności i możliwości współczesnych silników gier komputerowych

Streszczenie. Przez ostatnią dekadę rynek tworzenia gier komputerowych zdominowały dwa silniki: Unity oraz Unreal Engine. Niniejsza praca podejmuje się wyzwania przeprowadzenia analizy porównawczej obu tych silników pod kątem wydajności oraz procesu programowania gier. W tym celu wykorzystano narzędzie do profilowania NVIDIA Nsight, służące do oceny wydajności aplikacji graficznych, w szczególności do pomiaru czasu klatki, obciążenia GPU oraz efektywności renderowania. Przeprowadzono również wywiady środowiskowe z osobami związanymi profesjonalnie z tworzeniem gier na obu platformach, jak również opisano proces twórczy, napotkane trudności oraz przewagi każdego z silników. Na potrzeby pracy stworzono dwie gry w tym samym gatunku — *bullet hell* — słynącym z mnogości elementów na ekranie i efektów specjalnych, a jednocześnie będącym relatywnie łatwym do zaimplementowania. Jedna gra została stworzona w języku C# (Unity), druga w C++ (Unreal Engine). Następnie obie gry zostały poddane analizie w programie NVIDIA Nsight w celu oceny możliwości optymalizacji obu silników. Na podstawie wywiadów oraz analizy procesu twórczego scharakteryzowano również inne aspekty obu silników, takie jak próg wejścia, współpraca z systemami kontroli wersji oraz architektura silnika. Praca stara się wypełnić niszę w literaturze i badaniach związanych z oceną obu silników do gier.

Słowa kluczowe: Gry, Silniki do gier, C#, C++, Unreal Engine, Unity, Gry *bullet hell*, NVIDIA Nsight, Wydajność, Wywiady, Analiza porównawcza, Profilowanie, Renderowanie, Optymalizacja, Programowanie gier, Tworzenie gier, Czas klatki, Architektura silnika, Kontrola wersji, GPU

Comparison of performance and capabilities of modern computer games engines

Abstract. Over the past decade, the video game development market has been dominated by two engines: Unity and Unreal Engine. This thesis undertakes the challenge of conducting a comparative analysis of both engines in terms of performance and game programming workflows. To this end, the profiling tool NVIDIA Nsight was utilized to evaluate the performance of graphics applications, particularly for measuring frame time, GPU load, and rendering efficiency. Expert interviews were also conducted with professionals involved in game development on both platforms, and the creative process, encountered difficulties, and advantages of each engine were described. For the purposes of this study, two games were developed in the same genre — *bullet hell* — known for its abundance of on-screen elements and special effects, while being relatively straightforward to implement. One game was created in C# (Unity), the other in C++ (Unreal Engine). Both games were then analyzed using NVIDIA Nsight to assess the optimization capabilities of both engines. Drawing from the interviews and analysis of the creative process, other aspects of both engines were also characterized, such as the learning curve, integration with version control systems, and engine architecture. This thesis aims to fill a gap in the literature and research concerning the evaluation of game engines.

Keywords: Games, Game engines, C#, C++, Unreal Engine, Unity, Bullet hell games, NVIDIA Nsight, Performance, Interviews, Comparative analysis, Profiling, Rendering, Optimization, Game programming, Game development, Frame time, Engine architecture, Version control, GPU

Spis treści

1. Wstęp

1.1. Motywacja i cel pracy

Współczesny rynek gier komputerowych charakteryzuje się dynamicznym rozwojem technologicznym i rosnącymi wymaganiami zarówno twórców, jak i graczy. Wybór odpowiedniego silnika gier jest kluczową decyzją, która wpływa na cały proces tworzenia gry, jej wydajność oraz możliwości techniczne.

Celem niniejszej pracy jest kompleksowe porównanie wydajności i możliwości współczesnych silników gier komputerowych, ze szczególnym uwzględnieniem ich wpływu na proces tworzenia gier oraz końcową jakość produktu.

1.2. Zakres pracy

Praca obejmuje analizę następujących aspektów:

- Wydajność renderowania grafiki 2D i 3D
- Możliwości i funkcjonalności oferowane przez różne silniki
- Łatwość użycia i krzywa uczenia się
- Wsparcie dla różnych platform docelowych
- Ekosystem narzędzi i społeczność deweloperska

1.3. Wybór gry testowej – gatunek bullet hell

W celu przeprowadzenia praktycznych testów wydajnościowych zdecydowano się na implementację gry z gatunku **bullet hell** (dosł. „piekło pocisków”), znanego również jako **danmaku** (z jap. „kurtyna pocisków”) lub **manic shooter**.

1.3.1. Charakterystyka gatunku

Bullet hell to podgatunek gier typu shoot ’em up (strzelanka), w którym gracz steruje zwykle niewielkim statkiem kosmicznym lub postacią, mierząc się z falami przeciwników wystrzeliwujących ogromne ilości pocisków tworzących skomplikowane wzory na ekranie. Kluczowe cechy gatunku obejmują:

- **Masowa ilość pocisków** – na ekranie jednocześnie może znajdować się od kilkuset do kilku tysięcy pocisków, tworzących złożone formacje geometryczne
- **Precyzyjne hitboxy** – obszar kolizji postaci gracza jest znacznie mniejszy niż jej wizualna reprezentacja (często ograniczony do kilku pikseli), co umożliwia nawigację między gęstymi wzorami pocisków
- **Wzory pocisków** – przeciwnicy wystrzeliwują pociski według określonych algorytmów, tworząc spirale, fale, rozgałęzienia i inne formacje
- **Ciągły ruch** – gracz musi nieustannie przemieszczać się po ekranie, unikając kolizji
- **Eskalacja trudności** – wraz z postępem gry wzrasta liczba przeciwników i gęstość pocisków

Klasyczne przykłady gatunku to serie *Touhou Project*, *DoDonPachi*, *Ikaruga* oraz *Geometry Wars*.

1.3.2. Uzasadnienie wyboru gatunku

Gatunek bullet hell został wybrany jako podstawa testów wydajnościowych z następujących powodów:

1. **Intensywne wykorzystanie zasobów** – jednoczesne renderowanie setek lub tysięcy obiektów (pocisków) stanowi znaczące obciążenie dla systemu renderowania
2. **Testowanie zarządzania pamięcią** – ciągłe tworzenie i niszczenie obiektów pocisków eksponuje różnice w implementacji garbage collectora (Unity/C#) versus ręcznego zarządzania pamięcią (Unreal/C++)
3. **Wymagania systemu fizyki** – wykrywanie kolizji między graczem a setkami pocisków w każdej klatce obciąża system fizyki
4. **Prostota implementacji** – podstawowa mechanika gry jest stosunkowo prosta koncepcyjnie, co pozwala skupić się na porównaniu wydajności, a nie złożoności logiki gry
5. **Skalowalność testu** – łatwo kontrolować poziom obciążenia poprzez modyfikację liczby aktywnych pocisków i przeciwników
6. **Reprezentatywność dla gier 2D** – gatunek jest typowym przedstawicielem gier 2D, co pozwala ocenić wsparcie silników dla tego segmentu rynku
7. **Wymuszenie optymalizacji** – ze względu na ekstremalną liczbę obiektów, implementacja bullet hell wymusza stosowanie technik optymalizacyjnych (object pooling, spatial partitioning), których efektywność może różnić się między silnikami

1.3.3. Parametry gry testowej

Zaimplementowana gra testowa charakteryzuje się następującymi parametrami:

- Czas rozgrywki: 90 sekund (tryb przetrwania)
- Eskalacja trudności: liniowy wzrost częstotliwości spawnu przeciwników
- Typy przeciwników: 3 warianty z różnymi wzorami strzelania
- Maksymalna liczba jednoczesnych pocisków: do 500 obiektów
- System punktacji oparty na eliminacji przeciwników
- Object pooling dla pocisków (eliminacja alokacji w runtime)

Te parametry zapewniają wystarczające obciążenie systemu do ujawnienia różnic wydajnościowych między silnikami, pozostając jednocześnie w granicach typowych dla gier indie z tego gatunku.

1.4. Struktura pracy

Praca składa się z następujących rozdziałów:

1. **Wstęp** – wprowadzenie do tematyki, motywacja, cel i zakres pracy
2. **Przegląd literatury** – analiza istniejących badań porównawczych silników gier
3. **Charakterystyka silników** – szczegółowy opis Unity i Unreal Engine
4. **Metodologia** – opis metodyki badawczej i kryteriów porównania
5. **Analiza wywiadów** – wyniki badań jakościowych z deweloperami
6. **Implementacja gry testowej** – doświadczenia z tworzenia gry w obu silnikach

7. **Narzędzia profilowania** – opis NVIDIA Nsight i metodyki pomiarów
8. **Testy wydajności** – wyniki pomiarów wydajnościowych
9. **Analiza możliwości** – porównanie funkcjonalności silników
10. **Porównanie wyników** – synteza i analiza zebranych danych
11. **Podsumowanie** – wnioski i rekomendacje

1.5. Metodologia

W pracy zastosowano metodologię badawczą łączącą podejście ilościowe z jakościowym:

- **Testy wydajnościowe** – obiektywne pomiary z wykorzystaniem NVIDIA Nsight Graphics, zapewniające porównywalność wyników między silnikami
- **Wywiady z deweloperami** – badania jakościowe dostarczające kontekstu praktycznego użytkowania silników
- **Implementacja porównawcza** – stworzenie identycznej gry w obu silnikach, dokumentując różnice w procesie deweloperskim
- **Analiza dokumentacji** – przegląd oficjalnej dokumentacji i materiałów edukacyjnych

Takie wieloaspektowe podejście pozwala na kompleksową ocenę silników, uwzględniającą zarówno mierzalne parametry techniczne, jak i subiektywne doświadczenia użytkowników.

2. Przegląd literatury i istniejących rozwiązań

2.1. Historia rozwoju silników gier

Silniki gier ewoluowały znacząco od prostych bibliotek graficznych lat 80. i 90. XX wieku po współczesne, kompleksowe środowiska deweloperskie. Według Ullmann et al. **ullmann2022game**, współczesne silniki gier charakteryzują się modularną architekturą, która umożliwia ponowne wykorzystanie komponentów między różnymi projektami.

Gregory **gregory2018game** w swojej fundamentalnej pracy "Game Engine Architecture"przedstawia kompleksowy przegląd ewolucji silników gier, definiując je jako "oprogramowanie zaprojektowane specjalnie do tworzenia gier". Jego analiza pokazuje, że współczesne silniki gier składają się z kilku kluczowych warstw: warstwy platformy (platform layer), warstwy podstawowych systemów (core systems), warstwy zasobów (resource manager), warstwy renderingu (rendering engine), systemów animacji, fizyki oraz gameplay. Ta architektura warstwowa umożliwia modularność i ponowne wykorzystanie komponentów.

Pierwsze silniki gier były ściśle powiązane z konkretnym sprzętem i grami, jak np. silniki do gier id Software (Doom, Quake). Według Gregory'ego **gregory2018game**, przełomem było zrozumienie, że oddzielenie logiki gry od podstawowej infrastruktury technicznej pozwala na tworzenie bardziej uniwersalnych rozwiązań. Przełomem było wprowadzenie pierwszych uniwersalnych silników, które mogły być adaptowane do różnych rodzajów gier. Dzisiejsze silniki oferują zintegrowane środowiska deweloperskie z edytorami wizualnymi, systemami skryptowymi i zaawansowanymi narzędziami do debugowania.

2.2. Klasyfikacja silników gier

2.2.1. Architektura silników według Gregory'ego

Gregory **gregory2018game** przedstawia szczegółową taksonomię architektur silników gier, wyróżniając kilka kluczowych typów organizacji:

- **Silniki obiektowe** - bazujące na hierarchii obiektów gry z dziedziczeniem
- **Silniki komponentowe** - wykorzystujące systemy entity-component-system (ECS)
- **Silniki hybrydowe** - łączące elementy różnych podejść architektonicznych

Autor podkreśla, że wybór architektury ma fundamentalny wpływ na wydajność, skalowalność i łatwość rozwoju gier. Systemy ECS zyskują na popularności ze względu na lepszą wydajność cache procesora i większą elastyczność w definiowaniu zachowań obiektów gry.

2.2.2. Silniki komercyjne vs. open source

Analiza literatury pokazuje wyraźne różnice między rozwiązaniami komercyjnymi a otwartymi. Christopoulou i Xinogalos **christopoulou2017overview** wskazują, że silniki komercyjne jak Unity czy Unreal Engine oferują lepsze wsparcie techniczne i dokumentację, podczas gdy rozwiązania open source zapewniają większą elastyczność i kontrolę nad kodem źródłowym.

Sharif i Ameen **sharif2021game** podkreślają, że wybór między rozwiązaniem komercyjnym a open source zależy głównie od budżetu projektu i wymagań dotyczących dostosowania silnika do specyficznych potrzeb.

2.2.3. Silniki specjalistyczne vs. uniwersalne

Pavkov et al. **pavkov2017comparison** przedstawiają podział na silniki dedykowane konkretnym gatunkom gier (np. silniki do gier strategicznych czasu rzeczywistego) oraz rozwiązania uniwersalne mogące obsługiwać różnorodne typy gier. Silniki specjalistyczne oferują zoptymalizowane funkcjonalności dla określonego zastosowania, podczas gdy uniwersalne zapewniają większą wszechstronność kosztem specjalizacji.

2.3. Aktualny stan badań

2.3.1. Badania wydajności

Messaoudi et al. **messaoudi2017performance** przeprowadzili kompleksową analizę wydajności silnika Unity na urządzeniach mobilnych i stacjonarnych, koncentrując się na zużyciu CPU i optymalizacji logiki gry. Ich badania pokazują znaczące różnice w wydajności między platformami mobilnymi a desktop.

Abramowicz i Borczuk **abramowicz2024comparative** porównali wydajność Unity i Unreal Engine w grach 3D, skupiając się na renderowaniu, systemach fizyki i zarządzaniu pamięcią. Wyniki wskazują na przewagę Unreal Engine w renderowaniu zaawansowanej grafiki 3D, podczas gdy Unity wykazuje lepszą wydajność na urządzeniach o ograniczonych zasobach.

2.3.2. Metodologie porównawcze

Pattrasitidecha **pattrasitidecha2014comparison** opracował macierz porównawczą dla silników gier mobilnych 3D, definiując kryteria selekcji i kluczowe aspekty oceny. Ta metodologia została szeroko adoptowana w późniejszych badaniach.

Vohera et al. **vohera2021game** przedstawili architekturę silników gier i przeprowadzili studium porównawcze Unity, GameMaker, Unreal Engine i CryEngine, koncentrując się na parametrach wydajności, funkcjonalności i łatwości użycia.

2.3.3. Specjalizowane zastosowania

Marks et al. **marks2008evaluation** oceniali silniki gier pod kątem zastosowań w symulacjach medycznych i szkoleniach klinicznych, wprowadzając specyficzne kryteria oceny dla aplikacji edukacyjnych.

Ali i Usman **ali2016framework** opracowali framework do selekcji silników gier dla zastosowań w gamifikacji i serious games, uwzględniając specyficzne wymagania tych dziedzin.

2.3.4. Badania społeczności i ekosystemu

Barczak i Woźniak **barczak2019comparative** przeprowadzili kompleksowe studium porównawcze silników gier, analizując nie tylko aspekty techniczne, ale również dostępność zasobów edukacyjnych, aktywność społeczności i długoterminowe wsparcie.

2.4. Identyfikacja luk badawczych

Analiza dostępnej literatury ujawnia kilka istotnych luk badawczych:

1. **Brak kompleksowych badań wielokryterialnych** - większość istniejących prac koncentruje się na pojedynczych aspektach (wydajność, funkcjonalność) bez holistycznego podejścia
2. **Ograniczone badania długoterminowe** - brakuje analiz wpływu aktualizacji silników na stabilność i wydajność projektów
3. **Niewystarczające dane o współczesnych silnikach** - wiele badań koncentruje się na starszych wersjach silników, nie uwzględniając najnowszych możliwości
4. **Brak standaryzacji metodologii** - różne badania stosują odmienne kryteria oceny, co utrudnia porównanie wyników
5. **Ograniczone badania cross-platform** - niewiele prac analizuje wydajność silników na różnych platformach docelowych w sposób systematyczny

Niniejsza praca ma na celu wypełnienie tych luk poprzez przeprowadzenie kompleksowej analizy porównawczej współczesnych silników gier z zastosowaniem ustandaryzowanej metodologii i wielokryterialnego podejścia do oceny.

2.5. Trendy technologiczne

Ostatnie badania wskazują na rosnące znaczenie technologii ray tracing, sztucznej inteligencji w grach oraz wsparcia dla rzeczywistości wirtualnej i rozszerzonej. Masood et al. **masood2022high** analizują wykorzystanie silników gier do wysokowydajnego renderowania terenu GPU, pokazując nowe kierunki rozwoju technologii renderowania.

Badania Firat et al. **firat2022sound** dotyczące przestrzennego dźwięku 3D w silnikach gier wskazują na rosnące znaczenie immersyjnych doświadczeń audio jako czynnika różnicującego poszczególne rozwiązania.

3. Charakterystyka współczesnych silników gier

3.1. Kryteria wyboru silników do analizy

Rynek silników gier komputerowych oferuje szeroki wachlarz rozwiązań, od prostych frameworków 2D po zaawansowane środowiska do tworzenia fotorealistycznych produkcji AAA. W ramach niniejszej pracy zdecydowano się na dogłębną analizę dwóch silników: **Unity** oraz **Unreal Engine**. Wybór ten podyktowany był kilkoma kluczowymi czynnikami:

- **Dominacja rynkowa** – według danych z 2024 roku, Unity i Unreal Engine wspólnie obsługują ponad 70% globalnego rynku gier komputerowych i mobilnych
- **Reprezentatywność podejść architektonicznych** – silniki reprezentują odmienne filozofie: Unity opiera się na języku C# z garbage collectorem, a Unreal wykorzystuje natywny C++ z ręcznym zarządzaniem pamięcią
- **Różnorodność zastosowań** – Unity tradycyjnie dominuje w segmencie gier mobilnych i indie, natomiast Unreal jest preferowany w produkcjach AAA i projektach wymagających fotorealistycznej grafiki
- **Dostępność** – oba silniki oferują darmowe wersje dla małych zespołów i projektów edukacyjnych, co czyni je dostępnymi dla szerokiego grona deweloperów
- **Bogata dokumentacja** – zarówno Unity jak i Unreal dysponują rozbudowaną dokumentacją oficjalną oraz aktywnymi społecznościami

3.2. Unity

3.2.1. Wprowadzenie i historia

Unity to wieloplatformowy silnik gier stworzony przez Unity Technologies, którego pierwsza wersja została wydana w 2005 roku jako ekskluzywne narzędzie dla systemu macOS. Od tego czasu silnik przeszedł znaczącą ewolucję, stając się jednym z najpopularniejszych rozwiązań do tworzenia gier na świecie.

Kluczowym momentem w historii Unity było wprowadzenie w 2010 roku darmowej wersji silnika (Unity Free), co znacząco obniżyło barierę wejścia dla początkujących deweloperów i małych studiów. Decyzja ta przyczyniła się do eksplozji popularności silnika w segmencie gier mobilnych oraz indie.

Unity wykorzystuje język programowania **C#** działający na platformie .NET/Mono, co zapewnia:

- Automatyczne zarządzanie pamięcią poprzez garbage collector
- Bezpieczeństwo typów i obsługę wyjątków
- Bogatą bibliotekę standardową
- Stosunkowo łagodną krzywą uczenia dla programistów znających Javę lub podobne języki

Architektura Unity opiera się na wzorcu *GameObject-Component*, gdzie każdy obiekt w scenie (GameObject) może posiadać dowolną liczbę komponentów definiujących jego

zachowanie. Podejście to promuje kompozycję nad dziedziczeniem i ułatwia tworzenie modularnego kodu.

3.2.2. Możliwości i funkcjonalności

Unity oferuje kompleksowy zestaw narzędzi do tworzenia gier 2D i 3D:

- **Rendering** – wsparcie dla wielu pipeline'ów renderowania: Built-in, Universal Render Pipeline (URP) dla platform mobilnych oraz High Definition Render Pipeline (HDRP) dla wysokiej jakości grafiki
- **Fizyka** – integracja z silnikami PhysX (3D) i Box2D (2D)
- **Animacja** – system Mecanim z obsługą maszyn stanów i blendingu animacji
- **Audio** – wbudowany system dźwięku przestrzennego
- **UI** – dwa systemy interfejsu użytkownika: legacy uGUI oraz nowoczesny UI Toolkit
- **Multiplayer** – Netcode for GameObjects oraz integracja z usługami sieciowymi

3.2.3. Narzędzia deweloperskie

Edytor Unity zapewnia intuicyjny interfejs graficzny z następującymi funkcjonalnościami:

- Hierarchiczny widok sceny z możliwością edycji w czasie rzeczywistym
- Inspektor właściwości z obsługą serializacji pól poprzez atrybut `[SerializeField]`
- Wbudowany profiler wydajności (CPU, GPU, pamięć)
- Asset Store – marketplace z gotowymi zasobami i rozszerzeniami
- Obsługa hot reload – możliwość edycji kodu podczas działania gry

3.3. Unreal Engine

3.3.1. Wprowadzenie i historia

Unreal Engine to silnik gier stworzony przez Epic Games, którego historia sięga 1998 roku, kiedy to zadebiutował wraz z grą *Unreal*. Od początku silnik był projektowany z myślą o tworzeniu gier pierwszoosobowych (FPS) o wysokiej jakości graficznej, co nadal pozostaje jego mocną stroną.

Przełomowym momentem było wydanie Unreal Engine 4 w 2014 roku na licencji royalty-free (5% od przychodów powyżej $1 miliona), a następnie Unreal Engine 5 w 2022 roku, wprowadzającego rewolucyjne technologie takie jak Nanite (wirtualizowana geometria) i Lumen (globalne oświetlenie w czasie rzeczywistym).

Unreal Engine wykorzystuje język programowania **C++** z rozszerzeniami specyficznymi dla silnika (makra UE), co zapewnia:

- Maksymalną wydajność dzięki kompilacji do kodu natywnego
- Pełną kontrolę nad zarządzaniem pamięcią
- Dostęp do kodu źródłowego silnika (po uzyskaniu licencji)
- Strome krzywe uczenia, szczególnie dla programistów bez doświadczenia w C++

Dodatkowo Unreal oferuje system **Blueprints** – wizualny język skryptowy pozwalający na tworzenie logiki gry bez pisania kodu. Blueprinty są szczególnie przydatne dla desi-

gnerów i artystów, choć dla złożonych systemów mogą być mniej wydajne niż natywny C++.

3.3.2. Możliwości i funkcjonalności

Unreal Engine wyróżnia się zaawansowanymi możliwościami graficznymi:

- **Rendering** – fotorealistyczna grafika z obsługą ray tracingu, Nanite i Lumen
- **Fizyka** – silnik Chaos Physics z obsługą destrukcji i symulacji ciał miękkich
- **Animacja** – Control Rig, Animation Blueprints, IK Retargeting
- **Landscape** – zaawansowane narzędzia do tworzenia dużych terenów
- **Niagara** – system efektów cząsteczkowych nowej generacji
- **Sequencer** – narzędzie do tworzenia cinematików i cutscen

3.3.3. Narzędzia deweloperskie

Unreal Editor oferuje rozbudowane środowisko deweloperskie:

- Edytor poziomów z obsługą streamingu i Level of Detail (LOD)
- Blueprint Visual Scripting – programowanie wizualne
- Material Editor – węzłowy edytor materiałów
- Wbudowany profiler z analizą GPU/CPU i pamięci
- Marketplace – sklep z zasobami i pluginami
- Live Coding – eksperymentalne wsparcie dla hot reload w C++

3.4. Porównanie architektoniczne

Tabela 3.1. Porównanie kluczowych cech Unity i Unreal Engine

Cecha	Unity	Unreal Engine
Język programowania	C#	C++ / Blueprints
Zarządzanie pamięcią	Automatyczne (GC)	Ręczne / Smart pointers
Architektura	GameObject-Component	Actor-Component
Natywne wsparcie 2D	Tak	Nie (symulowane)
Kod źródłowy	Częściowo dostępny	Pełny dostęp
Rozmiar pustego projektu	~100 MB	~1-2 GB
Krzywa uczenia	Łagodna	Stroma
Główne zastosowania	Mobile, indie, 2D	AAA, FPS, fotorealizm

3.5. Uzasadnienie wyboru do badań

Wybór Unity i Unreal Engine jako przedmiotu porównania pozwala na analizę dwóch fundamentalnie różnych podejść do tworzenia gier:

1. **Produktywność vs wydajność** – C# w Unity oferuje szybszy rozwój kosztem pewnego narzutu wydajnościowego, podczas gdy C++ w Unreal wymaga więcej pracy, ale zapewnia maksymalną kontrolę
2. **Dostępność vs specjalizacja** – Unity celuje w szeroki rynek z niskim progiem wejścia, Unreal koncentruje się na produkcjach premium

3. **Elastyczność vs integracja** – Unity pozwala na większą swobodę w doborze zewnętrznych narzędzi, Unreal oferuje bardziej zintegrowane rozwiązania

Analiza tych dwóch silników dostarcza kompleksowego obrazu współczesnego stanu technologii do tworzenia gier i pozwala na sformułowanie praktycznych rekomendacji dla deweloperów.

4. Metodologia badań i kryteria porównania

4.1. Założenia metodologiczne

4.1.1. Cel badań

Głównym celem badań jest obiektywne porównanie wydajności i możliwości wybranych silników gier w kontrolowanych warunkach.

4.1.2. Hipotezy badawcze

1. Silniki komercyjne oferują lepszą wydajność niż rozwiązania open source
2. Kompleksowość funkcjonalności wpływa negatywnie na wydajność
3. Łatwość użycia jest odwrotnie proporcjonalna do możliwości konfiguracji

4.2. Kryteria porównania

4.2.1. Wydajność

- Szybkość renderowania (FPS)
- Zużycie pamięci RAM
- Obciążenie procesora
- Zużycie pamięci karty graficznej
- Czas ładowania scen

4.2.2. Funkcjonalność

- Wsparcie dla różnych typów renderingu
- Systemy fizyki
- Systemy audio
- Wsparcie dla VR/AR
- Możliwości skryptowania

4.2.3. Użyteczność

- Intuicyjność interfejsu
- Jakość dokumentacji
- Dostępność tutoriali
- Wsparcie społeczności
- Czas potrzebny na naukę

4.3. Środowisko testowe

4.3.1. Specyfikacja sprzętowa

Wszystkie testy wydajnościowe przeprowadzono na komputerze o następującej specyfikacji:

- **Procesor**: AMD Ryzen 9 7900X3D 12-Core Processor (24 rdzenie, 48 wątków)
- **Karta graficzna**: NVIDIA GeForce RTX 3090

- **Pamięć GPU**: 24 GB GDDR6X
- **Sterowniki NVIDIA**: wersja 590.48.01
- **Pamięć RAM**: 32 GB
- **System operacyjny**: Arch Linux (jądro Linux 6.18.5-arch1-1)
- **Dysk**: SSD o pojemności 3,6 TB

4.3.2. Specyfikacja oprogramowania

W badaniach wykorzystano następujące wersje oprogramowania:

- **Unity**: 6.0 (6000.0.58f2) LTS
- **Unreal Engine**: 5.5.3
- **NVIDIA Nsight Systems**: 2025.5.2

Wybór wersji LTS silnika Unity podyktowany był stabilnością oraz długoterminowym wsparciem, co jest istotne z punktu widzenia powtarzalności badań. W przypadku Unreal Engine wybrano najnowszą dostępną wersję stabilną w momencie rozpoczęcia badań.

4.3.3. Konfiguracja testowa

Przed wykonaniem każdego testu zastosowano następującą procedurę przygotowawczą:

1. Zamknięto wszystkie aplikacje działające w tle
2. Wyłączono automatyczne aktualizacje systemu
3. Ustawiono tryb wydajności zasilania (performance mode)
4. Odczekano 5 minut na stabilizację termiczną systemu
5. Wykonano 3 pomiary dla każdego scenariusza testowego, z których obliczono wartość średnią

4.4. Projekt testów

4.4.1. Gra testowa typu *bullet hell*

Na potrzeby badań porównawczych zaimplementowano identyczną grę w gatunku *bullet hell* w obu silnikach. Gra charakteryzuje się następującymi cechami:

- Sterowany przez gracza statek kosmiczny
- System generowania przeciwników z progresywnie rosnącym obciążeniem
- Generowanie wzorców pocisków
- Wykrywanie kolizji między obiektami
- Tryb przetrwania trwający 90 sekund

Wybór gatunku *bullet hell* podyktowany był możliwością generowania dużej liczby obiektów na ekranie (pociski, przeciwnicy, efekty wizualne), co pozwala na skuteczne obciążenie silnika graficznego i zbadanie jego wydajności w warunkach ekstremalnych.

4.4.2. Fazy obciążenia

Gra testowa została zaprojektowana tak, aby w ciągu 90 sekund rozgrywki przechodziła przez trzy fazy o różnym poziomie obciążenia:

Faza 1: Niskie obciążenie (0–30 sekund)

- Liczba pocisków na ekranie: 50–100
- Aktywni przeciwnicy: 2–3
- Oczekiwana stabilna liczba klatek: 60 FPS
- Oczekiwane wykorzystanie GPU: poniżej 50%

Faza 2: Średnie obciążenie (30–60 sekund)

- Liczba pocisków na ekranie: 200–300
- Aktywni przeciwnicy: 5–7
- Oczekiwane wykorzystanie GPU: 50–70%
- Możliwe niewielkie spadki liczby klatek

Faza 3: Wysokie obciążenie (60–90 sekund)

- Liczba pocisków na ekranie: ponad 500
- Aktywni przeciwnicy: ponad 10
- Oczekiwane wykorzystanie GPU: powyżej 70%
- Potencjalne znaczące spadki liczby klatek

4.4.3. Metryki i wskaźniki

W ramach testów wydajnościowych mierzono następujące parametry:

1. **Czas klatki** (ang. *frame time*) – czas potrzebny na wyrenderowanie pojedynczej klatki, wyrażony w milisekundach
2. **Liczba klatek na sekundę** (FPS) – wartość pochodna od czasu klatki
3. **Wykorzystanie GPU** – procentowe obciążenie karty graficznej
4. **Zużycie pamięci** – pamięć VRAM oraz RAM wykorzystywana przez aplikację
5. **Liczba wywołań rysowania** (ang. *draw calls*) – liczba wywołań API graficznego na klatkę
6. **Liczba wierzchołków** – całkowita liczba wierzchołków renderowanych na klatkę

4.4.4. Procedura zbierania danych

Dla każdego silnika przeprowadzono trzy pełne przebiegi gry (90 sekund każdy), co dało łącznie 6 sesji pomiarowych. Narzędzie NVIDIA Nsight Systems rejestrowało dane wydajnościowe przez cały czas trwania rozgrywki, umożliwiając późniejszą analizę poszczególnych faz obciążenia.

Punkty pomiaru dla każdej fazy:

- **Faza 1**: 10s (wczesne niskie obciążenie), 20s (stabilne niskie obciążenie), 25s (koniec fazy)
- **Faza 2**: 35s (przejście do średniego obciążenia), 45s (stabilne średnie obciążenie), 55s (koniec fazy)
- **Faza 3**: 65s (przejście do wysokiego obciążenia), 75s (szczytowe obciążenie), 85s (koniec fazy)

5. Analiza wywiadów z deweloperami gier

W ramach badań jakościowych przeprowadzono osiem pogłębionych wywiadów z deweloperami gier posiadającymi doświadczenie w pracy z silnikami Unity i Unreal Engine. Celem badania było zebranie praktycznych spostrzeżeń dotyczących użyteczności, wydajności oraz przepływu pracy w obu silnikach z perspektywy osób aktywnie je wykorzystujących.

5.1. Charakterystyka respondentów

Respondenci zostali dobrani według kryterium posiadania co najmniej rocznego doświadczenia amatorskiego lub profesjonalnego w jednym z badanych silników. Profil uczestników przedstawia się następująco:

- **Respondent 1**: Około 6-10 lat doświadczenia amatorskiego w Unity, semestr zajęć z Unreal Engine, 10-20 projektów w Unity
- **Respondent 2**: 7 lat doświadczenia amatorskiego w Unity, pół roku profesjonalnego, 15-20 projektów
- **Respondent 3**: 1,5 roku amatorskiego doświadczenia w Unity, 4 projekty zakończone
- **Respondent 4**: 2 lata profesjonalne w Unreal, 2 miesiące w Unity (z przerwami przez kilka lat), projekty w obu silnikach
- **Respondent 5**: 9 lat doświadczenia zawodowego (od 2012 Unity amatorsko, od 2016 profesjonalnie; od 2019 Unreal profesjonalnie), 10-30 projektów w Unity, 5-6 w Unreal
- **Respondent 6**: Dekada doświadczenia amatorskiego w Unity, kilka projektów game jamowych
- **Respondent 7**: 9 lat hobbystycznego doświadczenia w Unity, 2 lata profesjonalnego; 1-1,5 roku amatorskiego w Unreal
- **Respondent 8**: 2 lata amatorsko w Unity, 1,5 roku profesjonalnie + pół roku stażu w Unreal, kilkanaście projektów w obu silnikach

Łącznie badana próba reprezentuje szerokie spektrum doświadczeń – od osób skupionych wyłącznie na Unity, przez deweloperów wykorzystujących oba silniki, po profesjonalistów pracujących głównie w Unreal Engine.

5.2. Motywy wyboru silnika

5.2.1. Przystępność i próg wejścia

Dominującym motywem wyboru Unity jako pierwszego silnika była jego **przystępność dla początkujących**. Respondenci wskazywali, że Unity oferuje mniejszą liczbę gotowych mechanik widocznych na starcie – silnik nie narzuca użytkownikowi wbudowanych rozwiązań, jeżeli ten nie wybierze specjalnego szablonu projektu. Było to postrzegane jako zaleta dydaktyczna, ponieważ nowicjusze nie byli przytłaczani złożonością interfejsu.

Jednocześnie respondenci podkreślali, że Unreal Engine w przeszłości (około 2018 roku) charakteryzował się znacznie wyższym progiem wejścia niż obecnie. W tamtym

okresie dostępnych było również więcej materiałów edukacyjnych dla Unity, co dodatkowo wpływało na wybór tego silnika przez początkujących.

Paradoksalnie, mniejsza liczba wbudowanych funkcjonalności w Unity była postrzegana jako zaleta dydaktyczna – silnik nie przytłaczał nowicjuszy złożonością interfejsu i pozwalał na stopniowe poznawanie kolejnych mechanizmów.

5.2.2. Język programowania

Wybór C# jako głównego języka skryptowania w Unity stanowił istotny czynnik decyzyjny dla osób z wcześniejszym doświadczeniem w tym języku. Respondenci z backgroundem w C# określali przejście do Unity jako naturalne i intuicyjne. Język ten był opisywany jako wysokopoziomowy, niewymagający ręcznego zarządzania pamięcią, co znacząco obniża barierę wejścia dla początkujących programistów.

Niektórzy respondenci zwracali uwagę, że C++ używany w Unreal Engine różni się od standardowego C++ – jest rozszerzony o makra i mechanizmy specyficzne dla silnika, co może być zaskakujące dla programistów przyzwyczajonych do klasycznego C++.

5.2.3. Wymagania projektu

Wybór Unreal Engine często był podyktowany specyfiką projektu lub wymaganiami rynku pracy. Respondenci wskazywali, że projekty wymagające wysokiej jakości grafiki naturalnie kierowały ich w stronę Unreal Engine. Dodatkowo, część osób rozpoczęła naukę Unreal ze względu na większą liczbę ofert pracy wymagających znajomości tego silnika, szczególnie w segmencie gier AAA i większych studiów deweloperskich.

5.3. Dokumentacja i materiały edukacyjne

5.3.1. Oficjalna dokumentacja

W zakresie dokumentacji oficjalnej respondenci wyraźnie faworyzowali Unity. Dokumentacja tego silnika była opisywana jako dogłębna i szczegółowa – praktycznie wszystkie klasy, metody i właściwości są dokładnie opisane, a dodatkowo często zawierają działające przykłady kodu, które można bezpośrednio skopiować i uruchomić w projekcie.

Dokumentacja Unreal Engine była oceniana znacznie gorzej. Respondenci określali ją jako szkieletową lub wręcz nieistniejącą w praktycznym sensie. Wiele stron dokumentacji zawiera jedynie nazwę funkcji i nazwy parametrów, bez jakiegokolwiek opisu działania. Jeden z respondentów porównał czytanie dokumentacji Unreal do przeglądania plików nagłówkowych (header files), gdzie użytkownik musi samodzielnie domyślać się, co dana funkcja robi.

Jako pozytywny aspekt ekosystemu Unreal wskazywano fora deweloperskie, gdzie profesjonalni użytkownicy dzielą się rozwiązaniami. Problemem jest jednak to, że część najbardziej wartościowych zasobów znajduje się w zamkniętych sekcjach forum, dostępnych tylko dla wybranych firm po uzyskaniu specjalnych uprawnień od Epic Games.

5.3.2. Nieoficjalne poradniki

W przypadku materiałów nieoficjalnych (YouTube, blogi, fora) Unity również dominowało ilościowo. Respondenci szczególnie wyróżniali kanał Brackeys jako kluczowe źródło wiedzy dla początkujących i średniozaawansowanych użytkowników Unity.

Poradniki do Unreal Engine były oceniane jako:

- Mniej liczne niż dla Unity
- Często nieaktualne – dotyczące starszych wersji silnika (np. Unreal 4), które mogą, ale nie muszą działać w nowszych wersjach
- Zbyt skoncentrowane na systemie Blueprints kosztem programowania w C++

5.3.3. Jakość dydaktyczna poradników

Respondenci zwracali uwagę na wspólny problem poradników do obu silników – koncentrację na implementacji konkretnych funkcji kosztem dobrych praktyk programistycznych. Większość dostępnych materiałów skupia się na pokazaniu, jak zaimplementować pojedynczą mechanikę, bez wyjaśniania szerszego kontekstu architektonicznego czy zasad rozszerzalności kodu.

Ten brak holistycznego podejścia sprawia, że początkujący deweloperzy potrafią zaimplementować poszczególne funkcje, ale mają trudności z połączeniem ich w spójną całość lub późniejszym rozwojem projektu.

5.4. Architektura i wzorce projektowe

5.4.1. System komponentowy Unity

Architektura Unity oparta na komponentach była oceniana pozytywnie pod względem elastyczności. Respondenci doceniali możliwość dzielenia funkcjonalności na małe, niezależne moduły (komponenty), które następnie można łączyć w większe całości.

Jednocześnie wskazywano na problemy wynikające z długu technologicznego Unity. Silnik jest bardzo monolityczny, z głęboką hierarchią dziedziczenia podstawowych konceptów. Niektóre obiekty bazowe zajmują tak dużo pamięci, że nie mieszczą się w pojedynczej linii cache procesora, co na współczesny hardware stanowi istotny problem wydajnościowy.

5.4.2. Struktura Unreal Engine

Architektura Unreal Engine wymusza bardziej uporządkowany styl pracy. Respondenci zauważali, że nawet podstawowe projekty tworzone w Unreal mają tendencję do bycia lepiej zorganizowanymi, ponieważ silnik narzuca określoną strukturę.

Struktura aktor-komponent w Unreal (level zawiera aktorów, aktorzy zawierają komponenty) została opisana jako bardziej restrykcyjna niż prefaby w Unity. Próby tworzenia zagnieżdżonych struktur (aktor w aktorze) często prowadzą do problemów, podczas gdy w Unity hierarchie prefabów są bardziej elastyczne.

5.4.3. Specjalizacja silników

Respondenci zauważyli, że Unreal Engine jest wyraźnie zoptymalizowany pod gry typu first-person shooter. Tworzenie gier FPS w Unreal jest niezwykle proste – wystarczy zaznaczyć odpowiednie opcje. Natomiast projekty odbiegające od tego wzorca (np. gry z rozbudowanym interfejsem użytkownika, gry turowe) wymagają znacznie więcej pracy i często sprowadzają się do obchodzenia domyślnych mechanizmów silnika.

5.5. Kompilacja i przepływ pracy

5.5.1. Czas kompilacji

Czas kompilacji w Unity był identyfikowany jako znaczący problem przy większych projektach. W miarę rozrastania się bazy kodu, czas potrzebny na rekompilację po każdej zmianie rośnie.

Unity oferuje mechanizm Assembly Definitions jako rozwiązanie tego problemu. Bez podziału projektu na osobne assemblies każda zmiana w kodzie powoduje rekompilację całego projektu. Podział na mniejsze moduły pozwala kompilować tylko zmienione fragmenty, znacząco skracając czas iteracji.

5.5.2. Stabilność środowiska

Istotną różnicą między silnikami jest obsługa błędów krytycznych. W Unity gra uruchomiona w edytorze działa jako osobny proces – gdy wystąpi błąd krytyczny, zamyka się tylko ten proces, a edytor pozostaje stabilny. W Unreal Engine silnik i gra działają jako jeden proces, więc crash w grze powoduje utratę całego edytora wraz z ewentualnymi niezapisanymi zmianami.

Ta różnica architekturalna ma istotne konsekwencje dla produktywności, szczególnie przy debugowaniu. Przy dużych projektach, gdzie uruchomienie silnika może trwać kilkanaście minut, każdy crash oznacza znaczną stratę czasu.

5.5.3. Kompatybilność wsteczna

Unreal Engine był krytykowany za problemy z kompatybilnością między wersjami. Respondenci wskazywali, że rozpoczęcie projektu w określonej wersji silnika może skutkować problemami, jeśli ta wersja okaże się zawierać fundamentalne błędy. Epic Games nie backportuje poprawek do starszych wersji w takim stopniu jak Unity robi to dla wersji LTS.

5.6. Kontrola wersji i współpraca zespołowa

5.6.1. Integracja z Git

Współpraca z systemem Git była oceniana lepiej dla Unity ze względu na tekstową serializację assetów. Pliki scen i prefabów w Unity są zapisywane w formacie YAML, co teoretycznie umożliwia ich mergowanie. Nowoczesne narzędzia (np. merge w Rider) potrafią automatycznie rozwiązywać niektóre konflikty na scenach.

Pliki binarne w Unreal Engine stanowią znaczące wyzwanie. Respondenci zwracali uwagę, że nawet pliki Blueprintów, które ewidentnie mają serializację tekstową, są zapisywane na dysku jako binaria. To znacznie utrudnia współpracę wielu programistów nad tym samym projektem.

5.6.2. Mergowanie konfliktów

Konflikty na scenach i prefabach stanowią problem w obu silnikach. Gdy dwie osoby edytują tę samą scenę, rozwiązanie konfliktu często sprowadza się do wybrania jednej wersji i ręcznego przeniesienia zmian z drugiej.

Jako rozwiązanie wskazywano praktykę lockowania plików (preferowana przy użyciu Perforce) lub podział pracy na oddzielne sceny, gdzie każdy deweloper pracuje we własnym środowisku. Unity ułatwia takie podejście dzięki elastycznemu systemowi scen, podczas gdy Unreal silniej promuje architekturę z jedną główną sceną.

5.7. Współpraca z osobami nietechnicznymi

5.7.1. System Blueprints

Blueprinty w Unreal Engine były postrzegane jako skuteczne narzędzie ułatwiające współpracę z osobami nietechnicznymi. System wizualnego programowania pozwala designerom i artystom na tworzenie logiki gry bez pisania kodu tekstowego. Respondenci zauważali, że osoby niebędące programistami często nie zdają sobie sprawy, że faktycznie programują, korzystając z Blueprintów.

Jednocześnie integracja Blueprintów z kodem C++ nie jest idealna. Przejście między oboma systemami wymaga dodatkowej pracy, a wystawianie funkcji C++ do Blueprintów nie zawsze działa bezproblemowo.

5.7.2. Narzędzia dla artystów

Unity wymaga więcej pracy przy tworzeniu narzędzi dla osób nietechnicznych. Respondenci wskazywali, że w Unreal Engine osoby nietechniczne mają lepsze wsparcie „out of the box", podczas gdy w Unity zazwyczaj trzeba przeprowadzać szkolenia lub tworzyć dedykowane narzędzia edytorowe, aby umożliwić artystom i designerom samodzielną pracę.

5.8. Asset Store i zasoby zewnętrzne

5.8.1. Dostępność i jakość assetów

Asset Store Unity był oceniany jako lepiej zarządzany i bogatszy. Respondenci wskazywali na silniejsze wsparcie społeczności i większe szanse na znalezienie potrzebnych zasobów.

Interesującą obserwacją było to, że najlepsze produkty z Asset Store mają tendencję do opuszczania platformy – twórcy zakładają własne strony internetowe po osiągnięciu określonego poziomu popularności.

Unreal Marketplace przeszedł niedawno transformację w platformę Fab, co według respondentów pogorszyło doświadczenie użytkownika i zwiększyło liczbę kroków potrzebnych do pobrania darmowych zasobów.

5.8.2. Zastosowanie assetów

Assety były rekomendowane głównie do prototypowania, nie do produkcji komercyjnej. Respondenci podkreślali, że niespójny styl graficzny wynikający z łączenia assetów od różnych twórców jest gorszy niż jednolity, nawet jeśli prosty styl graficzny.

5.9. Wykorzystanie sztucznej inteligencji

5.9.1. Doświadczenia z LLM

Większość respondentów miała ograniczone doświadczenia z wykorzystaniem AI w pracy z silnikami gier. Główna obserwacja dotyczyła niskiej jakości generowanego kodu – naprawianie błędów w kodzie wygenerowanym przez ChatGPT często zajmowało więcej czasu niż napisanie rozwiązania od podstaw.

Jednocześnie AI było wykorzystywane skuteczniej jako substytut dokumentacji dla Unreal Engine. Pomimo częstych konfabulacji, modele językowe potrafiły naprowadzić na właściwe słowa kluczowe lub nazwy funkcji, które następnie można było zweryfikować w kodzie źródłowym silnika.

5.9.2. Generowanie grafik

Pozytywne doświadczenia zgłoszono w zakresie generowania placeholderów graficznych podczas game jamów. AI pozwala szybko uzyskać przyzwoicie wyglądające grafiki do prototypów, choć do wersji finalnych produktów nadal preferowana jest praca profesjonalnych grafików.

5.10. Optymalizacja i wydajność

5.10.1. Narzut silników

Respondenci wskazywali, że Unity ma mniejszy narzut wydajnościowy niż Unreal dla prostych projektów. Czas ładowania projektów w Unity jest znacznie krótszy, co respondenci przypisywali domyślnie niższym rozdzielczościom tekstur i prostszym ustawieniom graficznym.

5.10.2. Blueprinty vs C++

Istotną różnicę wydajnościową w Unreal stanowi wybór między Blueprintami a kodem C++. Blueprinty są interpretowane w czasie wykonania jako dane, a nie kompilowane do kodu maszynowego. W praktyce oznacza to, że logika napisana w Blueprintach jest znacznie wolniejsza niż równoważny kod C++.

5.10.3. Garbage Collector

Problem garbage collectora w Unity był wielokrotnie wspominany jako znany problem, przed którym ostrzegają doświadczeni deweloperzy. Cykliczne uruchamianie garbage collectora może powodować zauważalne zacięcia w grze. Co ciekawe, wielu respondentów wspominało o tym problemie jako o teoretycznym zagrożeniu, nie mając bezpośrednich negatywnych doświadczeń – prawdopodobnie dzięki stosowaniu praktyk takich jak object pooling.

5.11. Przyszłość silników i oczekiwania deweloperów

5.11.1. Entity Component System (ECS)

Nowy system DOTS/ECS w Unity był oczekiwaną funkcjonalnością, która w momencie przeprowadzania wywiadów została już oficjalnie wydana. System ten pozwala na pisanie wysoce wydajnego, zorientowanego na dane kodu, kosztem większej złożoności programistycznej.

5.11.2. UI Toolkit

Nowy system UI w Unity (UI Toolkit) był wskazywany jako obszar wymagający poprawy. Respondenci wyrażali nadzieję na jego dalszy rozwój w kierunku zbliżonym do technologii frontendowych, co ułatwiłoby pracę osobom z doświadczeniem w tworzeniu aplikacji webowych.

5.11.3. Konkurencja Godot

Część respondentów wyraziła zainteresowanie silnikiem Godot jako alternatywą dla Unity i Unreal. Główne przyczyny to:

- Model licencyjny royalty-free (brak opłat od przychodów)
- Otwarte źródła umożliwiające modyfikację silnika
- Mniejsza złożoność ułatwiająca naukę
- Kontrowersje związane z próbą zmiany modelu licencyjnego Unity w 2023 roku

Respondenci przewidywali, że jeśli Unity nie poprawi swojego wizerunku i oferty, Godot może w przyszłości stać się poważną konkurencją w segmencie gier indie.

5.12. Podsumowanie wyników badań jakościowych

Na podstawie przeprowadzonych wywiadów można sformułować następujące wnioski:

5.12.1. Silne strony Unity

- Wysoka jakość oficjalnej dokumentacji
- Bogaty ekosystem materiałów edukacyjnych
- Niższy próg wejścia dla początkujących
- Lepsza integracja z systemami kontroli wersji (tekstowa serializacja)
- Przystępny język programowania (C#)
- Elastyczna architektura komponentowa

- Mniejszy narzut wydajnościowy dla prostych projektów

5.12.2. Silne strony Unreal Engine

- Wymuszona struktura projektu promująca dobre praktyki
- System Blueprints ułatwiający współpracę z osobami nietechnicznymi
- Więcej gotowych funkcjonalności „out of the box”
- Lepsze wsparcie dla projektów wysokobudżetowych (grafika, multiplayer)
- Dostęp do kodu źródłowego silnika
- Lepsza integracja z zewnętrznymi narzędziami graficznymi (np. Blender)

5.12.3. Obszary problemowe wspólne

- Trudności z mergowaniem assetów graficznych w systemach kontroli wersji
- Poradniki koncentrujące się na implementacji kosztem dobrych praktyk
- Problemy z kompatybilnością między wersjami silników

5.12.4. Rekomendacje z badań

Na podstawie wywiadów można zasugerować następujące kryteria wyboru silnika:

Tabela 5.1. Rekomendacje wyboru silnika w zależności od kontekstu projektu

Kryterium	**Unity**	**Unreal Engine**
Doświadczenie zespołu	Początkujący, znajomość C#	Średniozaawansowany, znajomość C++
Typ projektu	Gry mobilne, 2D, indie	FPS, AAA, realistyczna grafika
Skład zespołu	Programiści	Mieszany (designerzy, artyści)
Budżet czasowy na naukę	Krótki	Średni do długiego
Wymagania graficzne	Standardowe	Wysokie

Wyniki badań jakościowych uzupełniają obiektywne testy wydajnościowe przedstawione w rozdziale **??**, dostarczając kontekstu praktycznego użytkowania obu silników w rzeczywistych projektach.

6. Doświadczenia z implementacji gry testowej

W ramach praktycznej części badań zaimplementowano grę typu bullet-hell w obu porównywanych silnikach. Gatunek ten został wybrany ze względu na jego wymagania wydajnościowe – jednoczesne renderowanie setek pocisków na ekranie stanowi doskonały test możliwości graficznych oraz efektywności zarządzania pamięcią przez silnik.

6.1. Opis projektu testowego

Zaimplementowana gra to klasyczny przedstawiciel gatunku bullet-hell, w którym gracz steruje statkiem kosmicznym i musi przetrwać przez określony czas (90 sekund), unikając pocisków wrogów i eliminując przeciwników. Kluczowe mechaniki gry obejmują:

- System spawnu wrogów z eskalującą trudnością – częstotliwość pojawiania się przeciwników wzrasta wraz z upływem czasu
- System pocisków z object poolingiem – optymalizacja pozwalająca na obsługę setek aktywnych pocisków
- System zdrowia i kolizji dla gracza oraz przeciwników
- Dynamiczne tło z efektem paralaksy
- System punktacji i warunki zwycięstwa/porażki

6.2. Implementacja w Unity

6.2.1. Środowisko i konfiguracja projektu

Projekt Unity został utworzony w wersji LTS z wykorzystaniem standardowego renderera 2D. Instalacja silnika na systemie Linux przebiegła bezproblemowo dzięki Unity Hub, który zapewnia spójne zarządzanie wersjami edytora i projektami.

Struktura projektu została zorganizowana według wzorca przestrzeni nazw, co pozwoliło na czytelną organizację kodu i uniknięcie konfliktów nazw.

6.2.2. Architektura systemu

Implementacja Unity wykorzystuje kilka kluczowych wzorców projektowych:

Wzorzec Bootstrap Klasa `GameBootstrap` wykorzystuje atrybut `[RuntimeInitializeOnLoadMethod]` do zapewnienia, że obiekt `GameInitializer` istnieje w scenie przed rozpoczęciem gry. Jest to eleganckie rozwiązanie problemu inicjalizacji singletonów w Unity.

Object Pooling System `BulletPool` stanowi rdzeń optymalizacji wydajnościowej. Zamiast ciągłego tworzenia i niszczenia obiektów pocisków (co generowałoby znaczące obciążenie garbage collectora), pociski są recyklingowane z puli:

Listing 1. Fragment implementacji object poolingu w Unity

```
public Bullet Spawn(Vector2 position, Vector2 direction,
                    float speed, float damage)
```

```
{
    Bullet bullet = _pool.Count > 0
        ? _pool.Dequeue()
        : Bullet.Create(this, bulletColor, faction);
    _liveBullets.Add(bullet);
    bullet.gameObject.SetActive(true);
    bullet.transform.position = position;
    bullet.Configure(direction, speed, damage, faction);
    return bullet;
}
```

Pula jest wstępnie rozgrzewana (*warm capacity*) podczas inicjalizacji, co eliminuje alokacje podczas rozgrywki.

Singleton Pattern Klasy `GameDirector` i `EnemySpawner` wykorzystują wzorzec Singleton z właściwością `Instance`, zapewniając globalny punkt dostępu do kluczowych systemów gry.

6.2.3. System spawnu przeciwników

`EnemySpawner` implementuje system eskalującej trudności poprzez interpolację czasu między spawnami:

Listing 2. Interpolacja trudności w Unity

```
float t = _elapsed / totalDuration;
float delay = Mathf.Lerp(spawnDelayStart, spawnDelayEnd, t);
```

Przeciwnicy są definiowani przez strukturę `EnemyBlueprint`, która zawiera parametry takie jak prędkość, zdrowie, wzorce strzelania i zachowania. To podejście data-driven pozwala na łatwe tworzenie różnorodnych typów wrogów.

6.2.4. Wyzwania napotkane w Unity

Podczas implementacji napotkano następujące wyzwania:

1. **Garbage Collection** – początkowa implementacja bez object poolingu powodowała zauważalne spadki klatek przy dużej liczbie pocisków
2. **Kolejność inicjalizacji** – konieczność użycia wzorca Bootstrap wynikała z nieprzewidywalnej kolejności wywoływania metod `Awake()` i `Start()`
3. **Serializacja** – atrybuty `[SerializeField]` wymagały starannego rozplanowania, które pola powinny być edytowalne w inspektorze

6.2.5. Pozytywne aspekty Unity

- Natywne wsparcie dla 2D – dedykowany tryb 2D z odpowiednimi komponentami fizyki (`Rigidbody2D`, `Collider2D`)
- Hot reload – możliwość edycji kodu i natychmiastowego testowania zmian
- Intuicyjny inspektor – łatwa konfiguracja parametrów gry bez rekompilacji

- Bogata dokumentacja C# i społeczność

6.3. Implementacja w Unreal Engine

6.3.1. Środowisko i konfiguracja projektu

Instalacja Unreal Engine na systemie Linux okazała się znacznie bardziej skomplikowana niż w przypadku Unity. Dostępne są dwie ścieżki:

1. Uzyskanie dostępu do oficjalnego repozytorium GitHub Epic Games i samodzielna kompilacja silnika ze źródeł
2. Pobranie prekompilowanej wersji binarnej

Należy zauważyć, że Unreal Engine nie oferuje wersji LTS (Long Term Support), co może stanowić wyzwanie dla długoterminowych projektów.

6.3.2. Podejście do grafiki 2D

Fundamentalna różnica między Unity a Unreal w kontekście gier 2D polega na tym, że Unreal traktuje 2D jako „fałszywe 2D" – w rzeczywistości jest to scena 3D z zablokowaną trzecią osią i kamerą ortograficzną. Unity natomiast oferuje dedykowany tryb 2D z wyspecjalizowanymi komponentami.

Ta różnica ma praktyczne konsekwencje:

- W Unreal konieczne jest ręczne konfigurowanie kamery ortograficznej
- Fizyka 2D w Unreal wykorzystuje te same komponenty co 3D, z ograniczeniami na odpowiednich osiach
- Sprite'y w Unreal są renderowane jako płaskie meshe w przestrzeni 3D

6.3.3. System Blueprintów vs C++

Unreal oferuje dwa podejścia do programowania logiki gry:

Blueprinty – wizualny system skryptowy, który pozwala na szybkie prototypowanie bez pisania kodu. Dla prostych mechanik bullet-hell Blueprinty okazały się wystarczające i intuicyjne.

C++ – dla bardziej wydajnościowo krytycznych elementów (jak system object poolingu) zalecane jest użycie C++. Jednak próg wejścia jest znacznie wyższy niż w przypadku C# w Unity.

6.3.4. Object Pooling w Unreal

Implementacja object poolingu w Unreal wymaga innego podejścia niż w Unity. Zamiast prostego `SetActive(true/false)`, Unreal wykorzystuje:

- `SetActorHiddenInGame()` – kontrola widoczności
- `SetActorEnableCollision()` – kontrola kolizji
- `SetActorTickEnabled()` – kontrola aktualizacji logiki

Ta granularność daje większą kontrolę, ale wymaga więcej kodu do osiągnięcia tego samego efektu.

6.3.5. Wyzwania napotkane w Unreal

1. **Brak natywnego 2D** – konieczność "symulowania" środowiska 2D w silniku 3D
2. **Czas kompilacji** – kompilacja projektów C++ jest znacznie wolniejsza niż kompilacja C# w Unity
3. **Rozmiar projektu** – nawet prosty projekt Unreal zajmuje wielokrotnie więcej miejsca na dysku
4. **Dokumentacja** – dla mniej popularnych zastosowań (jak gry 2D) dokumentacja jest ograniczona
5. **Blueprinty i kontrola wersji** – pliki Blueprintów są binarne, co utrudnia merge'owanie i code review

6.3.6. Pozytywne aspekty Unreal

- Potężny system materiałów i efektów wizualnych
- Wbudowane zaawansowane narzędzia profilowania
- Blueprinty umożliwiają szybkie prototypowanie przez osoby nietechniczne
- Doskonałe wsparcie dla grafiki 3D i fotorealizmu

6.4. Porównanie doświadczeń implementacyjnych

Tabela 6.1. Porównanie doświadczeń z implementacji gry bullet-hell

Aspekt	**Unity**	**Unreal Engine**
Czas instalacji (Linux)	~30 min	~2-4 h
Wsparcie natywne 2D	Tak	Nie (symulowane)
Język programowania	C#	C++ / Blueprinty
Próg wejścia	Niski	Średni/Wysoki
Czas kompilacji	Szybki	Wolny (C++)
Object pooling	Prosty	Bardziej złożony
Hot reload	Tak	Ograniczony
Rozmiar projektu	Mały	Duży

6.5. Wnioski z implementacji

Doświadczenia z implementacji gry bullet-hell potwierdzają, że wybór silnika powinien być uzależniony od typu projektu:

1. **Dla gier 2D** – Unity oferuje znacznie lepsze wsparcie natywne, niższy próg wejścia i szybszy cykl iteracji
2. **Dla gier 3D AAA** – Unreal Engine dysponuje lepszymi narzędziami do tworzenia fotorealistycznej grafiki
3. **Dla prototypowania** – Unity pozwala na szybsze testowanie koncepcji dzięki hot reloadowi i prostszej konfiguracji
4. **Dla zespołów mieszanych** – Blueprinty Unreal mogą być wartościowe dla współpracy z designerami, choć problemy z kontrolą wersji stanowią wyzwanie

Implementacja gry bullet-hell w Unity zajęła około 60% czasu potrzebnego na implementację analogicznej funkcjonalności w Unreal Engine, głównie ze względu na natywne wsparcie 2D i prostszy system object poolingu.

7. Narzędzia profilowania wydajności

Obiektywne porównanie wydajności silników gier wymaga zastosowania odpowiednich narzędzi pomiarowych. W niniejszym rozdziale przedstawiono analizę dostępnych rozwiązań oraz uzasadnienie wyboru NVIDIA Nsight jako głównego narzędzia profilowania.

7.1. Wbudowane narzędzia diagnostyczne silników

Zarówno Unity, jak i Unreal Engine oferują własne, wbudowane narzędzia do analizy wydajności. Każde z nich posiada unikalne cechy dostosowane do specyfiki danego silnika.

7.1.1. Unity Profiler

Unity dostarcza rozbudowany profiler dostępny bezpośrednio w edytorze (Window → Analysis → Profiler). Narzędzie to oferuje:

- **CPU Profiler** – analiza czasu wykonania poszczególnych funkcji, z podziałem na kategorie (rendering, skrypty, fizyka, animacje)
- **GPU Profiler** – pomiar czasu renderowania na karcie graficznej
- **Memory Profiler** – szczegółowa analiza alokacji pamięci, wykrywanie wycieków
- **Audio Profiler** – monitorowanie obciążenia systemu dźwiękowego
- **Physics Profiler** – analiza wydajności silnika fizyki
- **Frame Debugger** – krokowa analiza procesu renderowania pojedynczej klatki

Unity Profiler umożliwia również zdalne profilowanie aplikacji uruchomionej na urządzeniu docelowym (np. smartfonie), co jest szczególnie przydatne przy optymalizacji gier mobilnych.

7.1.2. Unreal Insights

Unreal Engine oferuje narzędzie Unreal Insights, które zastąpiło starszy system Session Frontend. Kluczowe funkcjonalności obejmują:

- **Timing Insights** – precyzyjny pomiar czasu wykonania poszczególnych systemów silnika
- **Asset Loading Insights** – analiza czasu ładowania zasobów
- **Memory Insights** – monitorowanie alokacji i dealokacji pamięci
- **Animation Insights** – profilowanie systemu animacji
- **Network Insights** – analiza ruchu sieciowego w grach multiplayer

Dodatkowo Unreal Engine udostępnia komendy konsolowe (np. `stat fps`, `stat unit`, `stat gpu`) pozwalające na szybki podgląd podstawowych metryk wydajności podczas rozgrywki.

7.1.3. Ograniczenia narzędzi wbudowanych

Pomimo rozbudowanych możliwości, wbudowane profilery silników posiadają istotne ograniczenia w kontekście porównawczych badań wydajnościowych:

1. **Brak standaryzacji metryk** – każdy silnik definiuje i mierzy parametry w odmienny sposób, co utrudnia bezpośrednie porównania

2. **Różna granularność danych** – poziom szczegółowości raportów różni się między silnikami
3. **Narzut profilowania** – wbudowane profilery same generują obciążenie, które może być różne dla każdego silnika
4. **Brak dostępu do danych niskopoziomowych** – profilery silnikowe operują na poziomie abstrakcji silnika, nie hardware'u
5. **Nieporównywalność formatów wyjściowych** – dane eksportowane przez różne profilery mają odmienne struktury

Z powyższych powodów zdecydowano się na zastosowanie zewnętrznego, niezależnego od silnika narzędzia profilowania.

7.2. NVIDIA Nsight Graphics

NVIDIA Nsight Graphics to profesjonalne narzędzie do profilowania i debugowania aplikacji graficznych, oferujące głęboki wgląd w działanie GPU niezależnie od używanego silnika czy API graficznego.

7.2.1. Uzasadnienie wyboru

Wybór NVIDIA Nsight jako głównego narzędzia pomiarowego podyktowany był następującymi czynnikami:

- **Niezależność od silnika** – Nsight analizuje aplikację na poziomie wywołań API graficznego (DirectX, Vulkan, OpenGL), co zapewnia porównywalność wyników między Unity a Unreal Engine
- **Standaryzowane metryki** – narzędzie dostarcza zunifikowany zestaw metryk sprzętowych (GPU utilization, memory bandwidth, shader throughput)
- **Minimalny narzut** – profilowanie na poziomie sterownika generuje mniejsze zakłócenia niż profilery działające wewnątrz silnika
- **Dostęp do danych niskopoziomowych** – możliwość analizy poszczególnych wywołań draw call, shaderów, transferów pamięci
- **Spójny format danych** – wyniki z obu silników mają identyczną strukturę, co ułatwia automatyzację analizy

7.2.2. Możliwości narzędzia

NVIDIA Nsight Graphics oferuje szereg funkcjonalności istotnych dla badań wydajnościowych:

Frame Profiler Główny moduł analizy wydajności, umożliwiający:

- Przechwycenie i analizę pojedynczej klatki (frame capture)
- Hierarchiczny widok wszystkich wywołań GPU
- Pomiar czasu wykonania każdego etapu renderowania
- Identyfikację wąskich gardeł (bottlenecks)
- Analizę wykorzystania jednostek obliczeniowych GPU

GPU Trace Moduł do długoterminowej analizy wydajności:

- Rejestrowanie metryk przez określony czas (nie tylko pojedyncza klatka)
- Wykrywanie spadków wydajności i ich przyczyn
- Analiza zmienności czasów klatek (frame time variance)
- Korelacja obciążenia GPU z wydarzeniami w grze

Shader Profiler Narzędzie do optymalizacji shaderów:

- Analiza wydajności poszczególnych shaderów
- Identyfikacja nieefektywnych instrukcji
- Pomiar occupancy (wykorzystania jednostek obliczeniowych)
- Sugestie optymalizacyjne

7.2.3. Konfiguracja środowiska pomiarowego

Przed przeprowadzeniem pomiarów skonfigurowano środowisko w następujący sposób:

1. Wyłączenie V-Sync w obu silnikach (eliminacja sztucznego ograniczenia FPS)
2. Ustawienie identycznej rozdzielczości renderowania (1920×1080)
3. Wyłączenie dynamicznego skalowania rozdzielczości
4. Ustawienie stałej częstotliwości zegara GPU (eliminacja power throttlingu)
5. Zamknięcie zbędnych procesów w tle
6. Oczekiwanie na ustabilizowanie temperatury GPU przed pomiarem

7.3. Przetwarzanie danych z Nsight

Dane zebrane przez NVIDIA Nsight wymagają odpowiedniego przetworzenia w celu uzyskania porównywalnych metryk.

7.3.1. Eksport danych

Nsight umożliwia eksport danych w kilku formatach:

- **CSV** – tabularyczne dane liczbowe, idealne do dalszej analizy
- **JSON** – strukturalne dane z pełną hierarchią wywołań
- **HTML Report** – czytelny raport z wykresami (mniej przydatny do automatyzacji)

W niniejszej pracy wykorzystano format CSV ze względu na łatwość importu do narzędzi analizy statystycznej.

7.3.2. Kluczowe metryki

Z danych eksportowanych przez Nsight wyodrębniono następujące metryki:

7.3.3. Metodyka pomiarów

Dla każdej konfiguracji testowej przeprowadzono serię pomiarów według następującego protokołu:

1. Uruchomienie aplikacji i oczekiwanie 30 sekund na stabilizację
2. Rozpoczęcie rejestracji GPU Trace (czas trwania: 60 sekund)
3. Przechwycenie 10 pojedynczych klatek w równych odstępach czasu

Tabela 7.1. Kluczowe metryki wydajnościowe z NVIDIA Nsight

Metryka	**Jednostka**	**Opis**
Frame Time	ms	Całkowity czas renderowania klatki
GPU Duration	ms	Czas pracy GPU (bez CPU overhead)
Draw Calls	liczba	Ilość wywołań rysowania na klatkę
Triangles Rendered	liczba	Liczba wyrenderowanych trójkątów
GPU Memory Used	MB	Zużycie pamięci VRAM
SM Occupancy	%	Wykorzystanie jednostek obliczeniowych
Memory Bandwidth	GB/s	Przepustowość pamięci GPU

4. Zakończenie rejestracji i eksport danych
5. Powtórzenie procedury 3 razy dla każdej konfiguracji

Wyniki uśredniono, odrzucając wartości odstające (outliers) zidentyfikowane metodą IQR (InterQuartile Range).

7.3.4. Automatyzacja analizy

W celu zapewnienia powtarzalności i eliminacji błędów ludzkich, proces analizy danych został częściowo zautomatyzowany za pomocą skryptów Python. Główne etapy obejmowały:

- Parsowanie plików CSV eksportowanych z Nsight
- Agregację danych z wielu sesji pomiarowych
- Obliczanie statystyk opisowych (średnia, mediana, odchylenie standardowe)
- Generowanie wykresów porównawczych
- Eksport wyników do formatu LaTeX (tabele)

7.4. Podsumowanie wyboru narzędzi

Zastosowanie NVIDIA Nsight jako głównego narzędzia profilowania zapewnia:

1. **Obiektywność** – pomiary wykonywane na tym samym poziomie abstrakcji dla obu silników
2. **Porównywalność** – identyczne metryki i format danych
3. **Wiarygodność** – niskopoziomowe pomiary eliminują artefakty wprowadzane przez profilery silnikowe
4. **Powtarzalność** – standaryzowana procedura pomiarowa

Wbudowane profilery Unity i Unreal Engine pozostają cennym narzędziem podczas procesu optymalizacji, jednak do celów badawczych wymagających bezpośredniego porównania między silnikami, zewnętrzne narzędzie oferuje znaczące przewagi metodologiczne.

8. Testy wydajności

8.1. Metodyka przeprowadzania testów

8.1.1. Przygotowanie środowiska testowego

Przed każdą sesją pomiarową wykonano następujące czynności przygotowawcze:

1. Zamknięto wszystkie zbędne aplikacje działające w tle
2. Zweryfikowano, że system jest w stanie bezczynności
3. Uruchomiono NVIDIA Nsight Systems w trybie profilowania
4. Skonfigurowano Nsight do przechwytywania danych z uruchamianej gry

8.1.2. Standaryzacja warunków testowych

W celu zapewnienia porównywalności wyników między silnikami Unity i Unreal Engine:

- Obie gry uruchamiano w tej samej rozdzielczości (1920×1080)
- Wyłączono synchronizację pionową (V-Sync) w obu implementacjach
- Zastosowano identyczną mechanikę rozgrywki i parametry generowania przeciwników
- Każdy test powtórzono trzykrotnie, obliczając wartości średnie

8.2. Wyniki testów dla silnika Unity

8.2.1. Faza niskiego obciążenia (0–30 sekund)

8.2.2. Faza średniego obciążenia (30–60 sekund)

8.2.3. Faza wysokiego obciążenia (60–90 sekund)

8.3. Wyniki testów dla silnika Unreal Engine

8.3.1. Faza niskiego obciążenia (0–30 sekund)

8.3.2. Faza średniego obciążenia (30–60 sekund)

8.3.3. Faza wysokiego obciążenia (60–90 sekund)

8.4. Analiza porównawcza

8.4.1. Porównanie czasu klatki

8.4.2. Porównanie wykorzystania GPU

8.4.3. Porównanie zużycia pamięci

8.4.4. Porównanie liczby wywołań rysowania

8.5. Podsumowanie wyników testów wydajności

9. Analiza możliwości i funkcjonalności

9.1. Metodyka oceny funkcjonalności

9.1.1. Kryteria oceny

9.1.2. Proces walidacji

9.2. Analiza możliwości renderingu

9.2.1. Wsparcie dla różnych technik renderingu

9.2.2. Systemy materiałów i shaderów

9.2.3. Systemy oświetlenia

9.3. Systemy fizyki i symulacji

9.3.1. Rigid body physics

9.3.2. Soft body physics

9.3.3. Systemy cząstek

9.4. Systemy audio

9.4.1. Wsparcie formatów audio

9.4.2. Przestrzenny dźwięk 3D

9.4.3. Efekty audio i DSP

9.5. Narzędzia deweloperskie

9.5.1. Edytory wizualne

9.5.2. Systemy debugowania

9.5.3. Profilowanie wydajności

9.6. Wsparcie dla platform docelowych

9.6.1. Platformy desktop

9.6.2. Platformy mobilne

9.6.3. Konsole

9.6.4. Platformy VR/AR

9.7. Ekosystem i rozszerzalność

9.7.1. Asset Store / Marketplace

9.7.2. Wsparcie społeczności

9.7.3. Dokumentacja i materiały edukacyjne

10. Porównanie wyników i analiza

10.1. Synteza wyników badań

10.1.1. Zestawienie wyników testów wydajności

10.1.2. Zestawienie analizy funkcjonalności

10.2. Analiza wielokryterialna

10.2.1. Macierz porównawcza

10.2.2. Analiza wag kryteriów

10.3. Przypadki użycia

10.3.1. Gry indie

10.3.2. Gry mobilne

10.3.3. Gry AAA

10.3.4. Gry VR/AR

10.4. Analiza korelacji

10.4.1. Związek między wydajnością a funkcjonalnością

10.4.2. Wpływ złożoności na użyteczność

10.5. Ograniczenia badań

10.5.1. Ograniczenia metodologiczne

10.5.2. Ograniczenia techniczne

10.5.3. Ograniczenia czasowe

10.6. Weryfikacja hipotez badawczych

10.7. Implikacje praktyczne

11. Podsumowanie i wnioski

11.1. Główne wyniki badań

11.1.1. Odpowiedzi na pytania badawcze

11.1.2. Weryfikacja hipotez

11.2. Wnioski praktyczne

11.2.1. Rekomendacje dla deweloperów

11.2.2. Wytyczne dla różnych typów projektów

11.3. Wkład naukowy

11.3.1. Nowatorskie aspekty badań

11.3.2. Znaczenie dla branży

11.4. Ograniczenia i przyszłe badania

11.4.1. Identyfikacja ograniczeń

11.4.2. Propozycje dalszych badań

11.4.3. Rozwój metodologii

11.5. Refleksje końcowe

11.6. Znaczenie wyników w kontekście rozwoju technologii

Wykaz symboli i skrótów

EiTI – Wydział Elektroniki i Technik Informacyjnych
PW – Politechnika Warszawska
WEIRD – ang. *Western, Educated, Industrialized, Rich and Democratic*

Spis rysunków

Spis tabel

Spis załączników

1. Nazwa załącznika 1 52
2. Nazwa załącznika 2 54

Załącznik 1. Nazwa załącznika 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa. Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel



Rysunek 1.1. Obrazek w załączniku.

justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

Załącznik 2. Nazwa załącznika 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris. Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at,

Tabela 2.1. Tabela w załączniku.

Kolumna 1	Kolumna 2	Liczba
cell1	cell2	60
	Suma:	123,45

tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.