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 wyd ajności 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 9
1.1. Motywacja i cel pracy 9
1.2. Zakres pracy 9
1.3. Wybór gry testowej – gatunek bullet hell 9
1.3.1. Charakterystyka gatunku 9
1.3.2. Uzasadnienie wyboru gatunku 10
1.3.3. Parametry gry testowej 11
1.4. Struktura pracy 11
1.5. Metodologia 11
2. Przegląd literatury i istniejących rozwiązań 13
2.1. Historia rozwoju silników gier 13
2.2. Klasyfikacja silników gier 13
2.2.1. Architektura silników według Gregory'ego 13
2.2.2. Silniki komercyjne vs. open source 14
2.2.3. Silniki specjalistyczne vs. uniwersalne 14
2.3. Aktualny stan badań 14
2.3.1. Badania wydajności 14
2.3.2. Metodologie porównawcze 14
2.3.3. Specjalizowane zastosowania 14
2.3.4. Badania społeczności i ekosystemu 15
2.4. Motywacja i cel pracy 15
2.5. Trendy technologiczne 15
3. Charakterystyka współczesnych silników gier 16
3.1. Kryteria wyboru silników do analizy 16
3.2. Unity 16
3.2.1. Wprowadzenie i historia 16
3.2.2. Możliwości i funkcjonalności 17
3.2.3. Narzędzia deweloperskie 17
3.3. Unreal Engine 17
3.3.1. Wprowadzenie i historia 17
3.3.2. Możliwości i funkcjonalności 18
3.3.3. Narzędzia deweloperskie 18
3.4. Porównanie architektoniczne 18
3.5. Uzasadnienie wyboru do badań 18
4. Metodologia badań i kryteria porównania 20
4.1. Założenia metodologiczne 20
4.1.1. Cel badań 20
4.1.2. Hipoteza badawcza 20
4.2. Kryteria porównania 20
4.3. Środowisko testowe 20

4.3.1. Specyfikacja sprzętowa 20
4.3.2. Specyfikacja oprogramowania 20
4.4. Projekt testów 21
4.4.1. Gra testowa typu *bullet hell* 21
4.4.2. Fazy obciążenia 21
4.4.3. Procedura zbierania danych 22
5. Analiza wywiadów z deweloperami gier 23
5.1. Charakterystyka respondentów 23
5.2. Motywy wyboru silnika 23
5.2.1. Przystępność i próg wejścia 23
5.2.2. Język programowania 24
5.2.3. Wymagania projektu 24
5.3. Dokumentacja i materiały edukacyjne 24
5.3.1. Oficjalna dokumentacja 24
5.3.2. Nieoficjalne poradniki 25
5.3.3. Jakość dydaktyczna poradników 25
5.4. Architektura i wzorce projektowe 25
5.4.1. System komponentowy Unity 25
5.4.2. Struktura Unreal Engine 25
5.4.3. Specjalizacja silników 26
5.5. Kompilacja i przepływ pracy 26
5.5.1. Czas kompilacji 26
5.5.2. Stabilność środowiska 26
5.5.3. Kompatybilność wsteczna 26
5.6. Kontrola wersji i współpraca zespołowa 27
5.6.1. Integracja z Git 27
5.6.2. Mergowanie konfliktów 27
5.7. Współpraca z osobami nietechnicznymi 27
5.7.1. System Blueprints 27
5.7.2. Narzędzia dla artystów 27
5.8. Asset Store i zasoby zewnętrzne 28
5.8.1. Dostępność i jakość assetów 28
5.8.2. Zastosowanie assetów 28
5.9. Wykorzystanie sztucznej inteligencji 28
5.9.1. Doświadczenia z LLM 28
5.9.2. Generowanie grafik 28
5.10. Optymalizacja i wydajność 29
5.10.1. Narzut silników 29
5.10.2. Blueprinty vs C++ 29
5.10.3. Garbage Collector 29
5.11. Przyszłość silników i oczekiwania deweloperów 29
5.11.1. Entity Component System (ECS) 29

5.11.2. UI Toolkit 29
5.11.3. Konkurencja Godot 29
5.12. Podsumowanie wyników badań jakościowych 30
5.12.1. Silne strony Unity 30
5.12.2. Silne strony Unreal Engine 30
5.12.3. Obszary problemowe wspólne 30
5.12.4. Rekomendacje z badań 31
6. Doświadczenia z implementacji gry testowej 32
6.1. Implementacja w Unity 32
6.1.1. Architektura systemu 32
6.1.2. System spawnu przeciwników 33
6.1.3. Wyzwania napotkane w Unity 33
6.1.4. Pozytywne aspekty Unity 34
6.2. Implementacja w Unreal Engine 35
6.2.1. Podejście do grafiki 2D 36
6.2.2. System Blueprintów vs C++ 36
6.2.3. Object Pooling w Unreal 36
6.2.4. Wyzwania napotkane w Unreal 36
6.2.5. Pozytywne aspekty Unreal 37
6.3. Porównanie doświadczeń implementacyjnych 37
6.4. Wnioski z implementacji 38
7. Narzędzia profilowania wydajności 39
7.1. Wbudowane narzędzia diagnostyczne silników 39
7.1.1. Unity Profiler 39
7.1.2. Unreal Insights 40
7.1.3. Ograniczenia narzędzi wbudowanych 40
7.2. NVIDIA Nsight Graphics 41
7.2.1. Uzasadnienie wyboru 41
7.2.2. Możliwości narzędzia 41
7.3. Przetwarzanie danych z Nsight 42
7.3.1. Eksport danych 42
7.3.2. Kluczowe metryki 42
7.4. Podsumowanie wyboru narzędzi 43
8. Testy wydajności 44
8.1. Metryki wydajności 44
8.1.1. Zbierane dane 44
8.2. Wyniki testów dla silnika Unity 44
8.2.1. Ogólne wyniki wydajności 44
8.2.2. Analiza rozkładu czasów klatek 45
8.2.3. Szczegółowa analiza wywołań Vulkan API 47
8.2.4. Analiza wywołań systemowych (OS Runtime) 49
8.2.5. Interpretacja wyników i wnioski 51

8.3. Wyniki testów dla silnika Unreal Engine 52
8.3.1. Metodologia profilowania fazowego 53
8.3.2. Metryki wykorzystania GPU 53
8.3.3. Analiza wywołań Vulkan API w trzech fazach 56
8.3.4. Analiza wywołań systemowych Unreal Engine 59
8.3.5. Charakterystyka architektury Unreal Engine na podstawie profilowania 61
8.4. Analiza porównawcza 63
8.4.1. Porównanie czasu klatki 63
8.4.2. Porównanie wykorzystania GPU 63
8.4.3. Porównanie architektury wielowątkowej 64
8.5. Podsumowanie wyników testów wydajności 64
Bibliografia 67
Wykaz symboli i skrótów 69
Spis rysunków 69
Spis tabel 70
Spis załączników 70

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 porównanie wydajności i możliwości dwóch głównych, 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
- Możliwości i funkcjonalności oferowane przez różne silniki
- Łatwość użycia i krzywa uczenia się
- Praca z narzędziem przy użycji dużych modeli językowch
- 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, 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 pociskami
- **Duża liczba przeciwników** – na ekranie jednocześnie pojawia się wiele jednostek przeciwnika, co zwiększa złożoność sytuacji i obciążenie systemu
- **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* lub *Geometry Wars*.



Rysunek 1.1. Przykład gry z gatunku bullet hell (seria Touhou). [1]

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

1.3.3. Parametry gry testowej

Zaimplementowana gra testowa charakteryzuje się następującymi parametrami:

- Czas rozgrywki: 90 sekund
- Eskalacja trudności: liniowy wzrost częstotliwości spawnu przeciwników
- Typy przeciwników: 3 warianty z różnymi wzorami strzelania
- System punktacji oparty na eliminacji przeciwników

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

- **Testy wydajnościowe** – 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

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 [2]. Wczesne biblioteki, takie jak Allegro (1990), OpenGL (1992), DirectX (1995) czy SDL (1998), dostarczały podstawowe funkcje graficzne i obsługę wejścia, lecz nie oferowały zintegrowanych narzędzi do tworzenia gier. Według Ullmann [3], współczesne silniki gier charakteryzują się modularną architekturą, która umożliwia ponowne wykorzystanie komponentów między różnymi projektami.

Gregory [2] w swojej 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 [2], 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 [2] przedstawia 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 [4] 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 [5] 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 [6] 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 [7] 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 [8] 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 [9] opracował macierz porównawczą dla silników gier mobilnych 3D, definiując kryteria selekcji i kluczowe aspekty oceny.

Vohera [10] 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 [11] oceniali silniki gier pod kątem zastosowań w symulacjach medycznych i szkoleniach klinicznych, wprowadzając specyficzne kryteria oceny dla aplikacji edukacyjnych.

Ali i Usman [12] 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 [13] przeprowadzili kompleksowe studium porównawcze silników gier, analizując nie tylko aspekty techniczne, ale również dostępność zasobów edukacyjnych, aktywność społeczności i długoterminowe wsparcie.

2.4. Motywacja i cel pracy

Przegląd literatury pokazuje, że istnieje wiele badań porównawczych silników gier, jednak większość z nich koncentruje się na wybranych aspektach, takich jak wydajność renderowania, łatwość użycia czy wsparcie dla konkretnych platform. Niniejsza praca wpisuje się w ten nurt, przeprowadzając praktyczne porównanie silników Unity i Unreal Engine pod kątem wydajności w wybranych scenariuszach testowych.

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. [14] analizują wykorzystanie silników gier do wysokowydajnego renderowania terenu GPU, pokazując nowe kierunki rozwoju technologii renderowania.

Badania Firat. [15] 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 [2]. 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 raportu Video Game Insights, w 2024 roku 51% gier wydanych na platformie Steam powstało w Unity, a 28% w Unreal Engine [16]
- **Reprezentatywność podejść architektonicznych** – silniki reprezentują odmienne filozofie: Unity opiera się na języku C# z garbage collectorem, a Unreal wykorzystuje C++ z własnym systemem refleksji, makrami oraz garbage collectorem dla obiektów UObject
- **Różnorodność zastosowań** – Unity dominuje w segmencie gier mobilnych (71% z top 1000 gier mobilnych) oraz wśród deweloperów indie, natomiast Unreal generuje większe przychody w produkcjach AAA (31% przychodów Steam w 2024 vs 26% dla Unity) [16], [17]
- **Dostępność** – oba silniki oferują darmowe wersje dla małych zespołów i projektów edukacyjnych
- **Aktywna społeczność** – Unity posiada ponad 5 milionów zarejestrowanych deweloperów [17]

3.2. Unity

3.2.1. Wprowadzenie i historia

Unity to wieloplatformowy silnik gier stworzony przez Unity Technologies, którego pierwsza wersja została zaprezentowana w czerwcu 2005 roku na konferencji Apple Worldwide Developers Conference jako narzędzie dla systemu Mac OS X [18]. 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 darmowej wersji silnika, co znacząco obniżyło barierę wejścia dla początkujących deweloperów i małych studiów [18]. Decyzja ta przyczyniła się do wzrostu 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

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: uGUI oraz nowoczesny UI Toolkit
- **Multiplayer** – Netcode for GameObjects oraz integracja z usługami sieciowymi

3.2.3. Narzędzia deweloperskie

Edytor Unity zapewnia interfejs graficzny z następującymi funkcjonalnościami [18]:

- Hierarchiczny widok sceny z możliwością edycji w czasie rzeczywistym
- Inspektor właściwości z obsługą serializacji pól poprzez atrybut `[SerializeField]`
- Wbudowany profiler wydajności (CPU, GPU, pamięć) [19]
- 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* [20]. Od początku silnik był projektowany z myślą o tworzeniu gier pierwszoosobowych (FPS) o wysokiej jakości graficznej, co nadal pozostaje jego mocną stroną [20], [21].

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, wprowadzającym technologie takie jak Nanite (wirtualizowana geometria) i Lumen (globalne oświetlenie w czasie rzeczywistym) [20], [22], [23].

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
- 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 [21]. Szczególnie przydatne dla designerów 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 [21]:

- 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
- Dostęp do kodu źródłowego silnika
- 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
Język	C#	C++
Pamięć	Automatyczne (GC)	Ręczne (smart pointers)
Architektura	GameObject Component	Actor Component
2D	Natywne	Ograniczone
Kod źródłowy	Częściowy	Pełny
Główne zastosowania	Mobile, indie, 2D	AAA, FPS, 3D

3.5. Uzasadnienie wyboru do badań

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

Motywacja wyboru: Unity i Unreal Engine pozostają dwoma najpopularniejszymi silnikami używanymi w większości nowoczesnych produkcji — zarówno w segmencie indie, jak i w grach AAA — co czyni ich porównanie reprezentatywnym dla współczesnego rynku gier [16], [17].

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 porównanie wydajności i możliwości wybranych silników gier.

4.1.2. Hipoteza badawcza

Silnik Unity, dzięki natywnemu wsparciu dla grafiki 2D, osiągnie lepszą wydajność w grze typu *bullet hell* niż Unreal Engine, który jest zoptymalizowany przede wszystkim pod kątem aplikacji 3D.

4.2. Kryteria porównania

W ramach testów wydajnościowych analizowano następujące metryki, zbierane za pomocą narzędzia NVIDIA Nsight Systems:

- **Czas klatki** (ang. *frame time*) – czas potrzebny na wyrenderowanie pojedynczej klatki, wyrażony w milisekundach
- **Liczba klatek na sekundę** (FPS) – wartość pochodna od czasu klatki, kluczowa dla płynności rozgrywki
- **Wykorzystanie GPU** – procentowe obciążenie karty graficznej, mierzone poprzez liczniki sprzętowe NVIDIA
- **Wywołania Vulkan API** – szczegółowa analiza wywołań interfejsu graficznego, w tym funkcji synchronizacji i prezentacji
- **Wywołania systemowe** – analiza mechanizmów wielowątkowości i synchronizacji na poziomie systemu operacyjnego

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.

4.4. 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:

- Sterowana przez gracza postać
- 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.

4.4.2. Fazy obciążenia

Gra testowa została zaprojektowana tak, aby w ciągu 90 sekund rozgrywki progresywnie zwiększała obciążenie poprzez:

- **Przyspieszanie spawnu przeciwników** – interwał między spawnem zmniejsza się liniowo od 0,25 s (początek) do 0,08 s (koniec), z dodatkowym „finalnym szturmem" przez ostatnie 5 sekund
- **Zwiększanie różnorodności typów przeciwników** – początkowo (0–25% czasu) pojawiają się tylko podstawowi przeciwnicy, później wprowadzane są kolejno szybsze jednostki (25–50%), strzelające wieżyczki (50–75%) oraz wytrzymałe czołgi (75–100%)
- **Maksymalna liczba przeciwników** – limit jednoczesnych przeciwników na scenie wynosi 200 jednostek

Na potrzeby profilowania rozgrywka została podzielona na trzy fazy czasowe:

Faza 1 (0–30 sekund) Początkowa faza z niskim obciążeniem. Spawner generuje wyłącznie podstawowych przeciwników (*Fodder*) z interwałem ok. 0,25 s.

Faza 2 (30–60 sekund) Średnie obciążenie. Wprowadzane są szybsze przeciwnicy (*Runner*), interwał spawnu zmniejsza się do ok. 0,17 s.

Faza 3 (60–90 sekund) Wysokie obciążenie. Wszystkie typy przeciwników, interwał spawnu osiąga minimum 0,08 s. Ostatnie 5 sekund to „finalny szturm" z maksymalną intensywnością.

4.4.3. Procedura zbierania danych

Przeprowadzono cztery sesje pomiarowe – po dwie dla każdego silnika:

1. **Unity – tryb statyczny**: Gracz nieruchomy z włączoną nieśmiertelnością, pełne 90 sekund rozgrywki profilowane w jednej sesji
2. **Unity – tryb dynamiczny**: Gracz aktywnie poruszający się i strzelający, pełne 90 sekund rozgrywki
3. **Unreal Engine – tryb statyczny**: Ze względu na ograniczenia techniczne (awaria przy śledzeniu Vulkan API) rozgrywkę podzielono na trzy 30-sekundowe fazy, uruchamiane z flagą `--start-time=N`
4. **Unreal Engine – tryb dynamiczny**: Analogicznie do trybu statycznego, trzy fazy po 30 sekund z aktywnym graczem

Narzędzie NVIDIA Nsight Systems rejestrowało:

- Wywołania Vulkan API (dla Unity – dla Unreal niemożliwe z powodu awarii)
- Metryki sprzętowe GPU z częstotliwością 10 000 Hz
- Wywołania funkcji systemowych (pthread, futex itp.)

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

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 jed-

nak 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

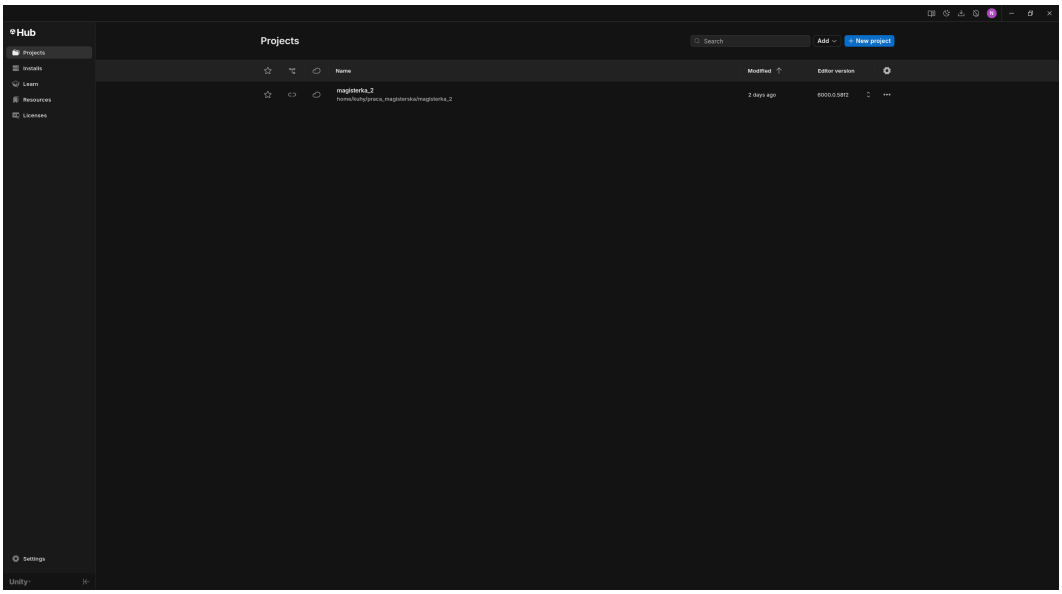
Kryterium	**Unity**	**Unreal Engine**
Doświadczenie zespołu	Początkujący, znajomość C#	Średnie, znajomość C++
Typ projektu	Gry mobilne, 2D, indie	FPS, AAA, realistyczna grafika
Skład zespołu	Programiści	Mieszany (designerzy, artyści)
Budżet czasowy na naukę	Krótki	Średni do długiego
Wymagania graficzne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

Wyniki badań jakościowych uzupełniają obiektywne testy wydajnościowe przedstawione w rozdziale 8, dostarczając
kontekstu praktycznego użytkowania obu silników w rzeczywistych projektach.

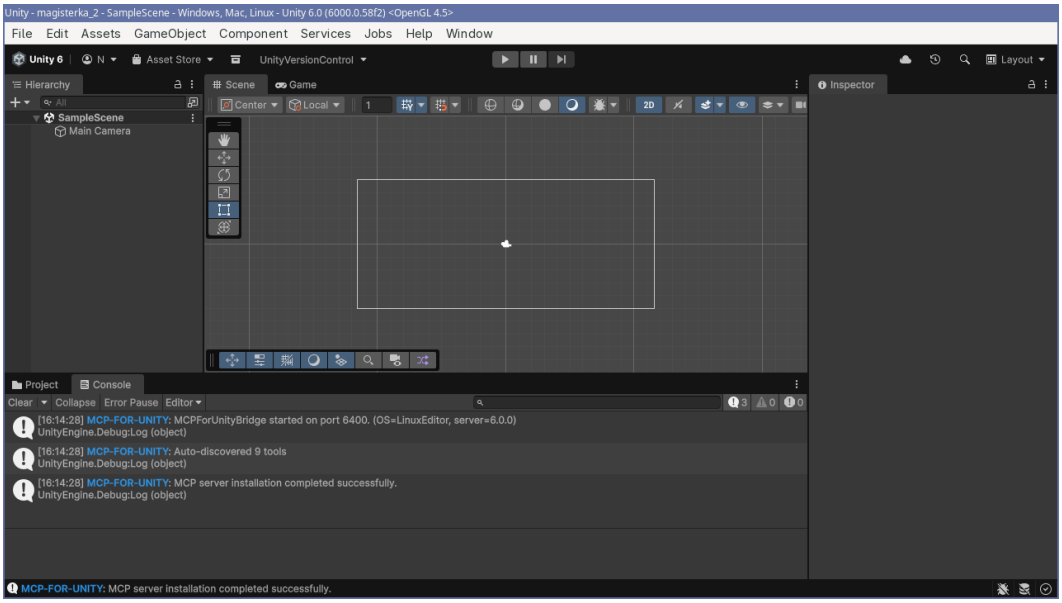
6. Doświadczenia z implementacji gry testowej

6.1. Implementacja w Unity

Projekt Unity został utworzony w wersji LTS z wykorzystaniem standardowego renderera 2D. Instalacja silnika na systemie Linux przebiegła bezproblemowo dzięki Unity Hub [24] [25], który zapewnia spójne zarządzanie wersjami edytora i projektami.



Rysunek 6.1. Ekran powitalny Unity Hub.



Rysunek 6.2. Widok edytora Unity.

6.1.1. 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.

6.1.2. 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.

6.1.3. 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
4. Interfejs użytkownika nie jest odświeżany po otwarciu menu [26] – wymagało to ręcznego wymuszenia odświeżenia inspektora

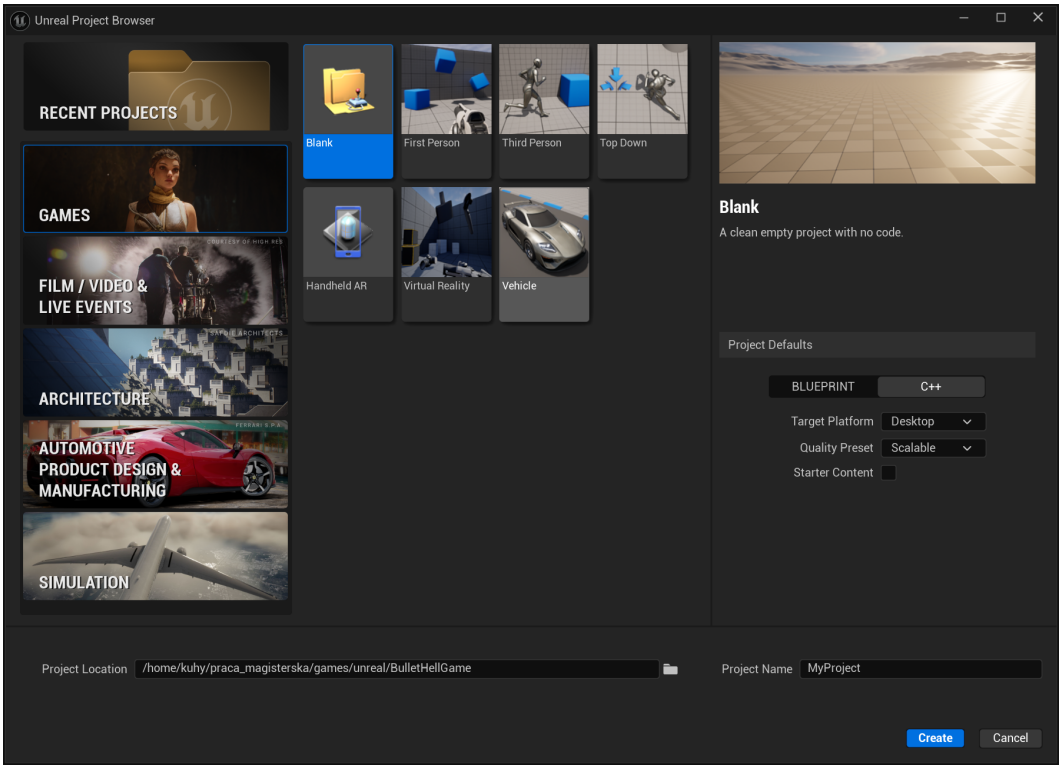
6.1.4. Pozytywne aspekty Unity

- 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
- Skupienie się na kodzie – większość logiki gry zaimplementowana została w kodzie źródłowym co przyśpieszyło proces tworzenia gry
- Dobre wsparcie dla LLM – unity posiada narzędzie mcp oferującą prostą integrację edytora i narzędzi AI [27]

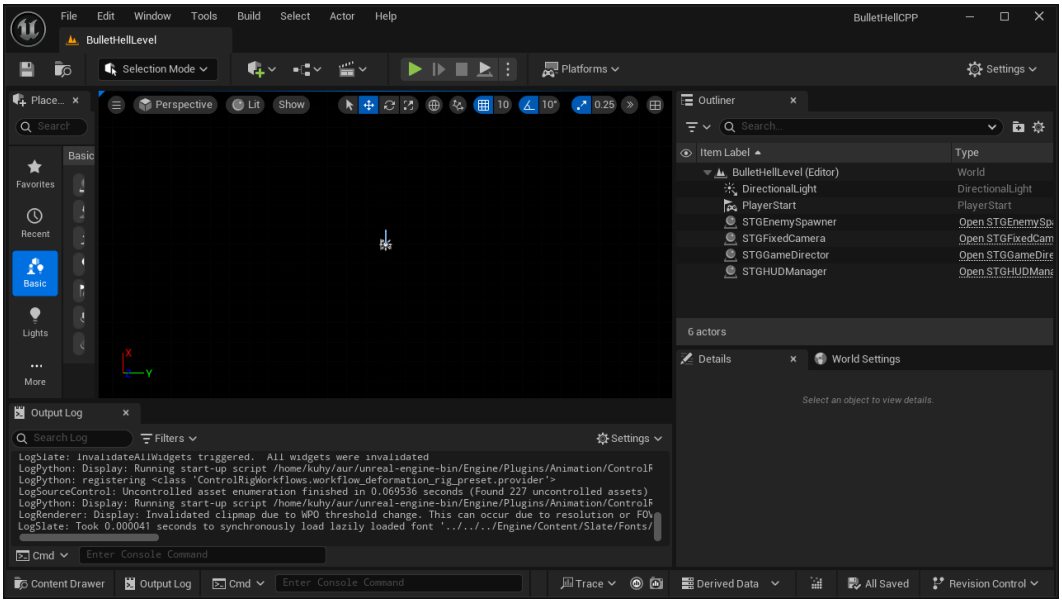
6.2. Implementacja w Unreal Engine

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 [28]
2. Pobranie prekompilowanej wersji binarnej [29]



Rysunek 6.3. Wybór projektu w Unreal Engine.



Rysunek 6.4. Widok edytora Unreal Engine.

6.2.1. 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

6.2.2. System Blueprintów vs C++

Unreal oferuje dwa podejścia do programowania logiki gry:

Blueprinty – wizualny system skryptowy, który w teorii pozwala na szybkie prototypowanie bez pisania kodu.

W praktyce korzystanie z nich dla osoby która zna tradycyjne programowanie okazazało się frustrujące ze względu na:

- Problemy z kontrolą wersji
- Trudności w debugowaniu
- Ograniczoną czytelność i skalowalność
- Ograniczenie w funkcjach z którycj można korzytsać

C++ – dla bardziej wydajnościowo krytycznych elementów (jak system object poolingu) zalecane jest użycie C++. użycie go ostatecznie okazało się bardziej intujcywne i prostsze dla projektu z uwagi na doświadczenie w pisaniu programów w tradycyjny sposób

6.2.3. 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.

6.2.4. 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

6.2.5. 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

6.3. 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

6.4. 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

7.1. Wbudowane narzędzia diagnostyczne silników

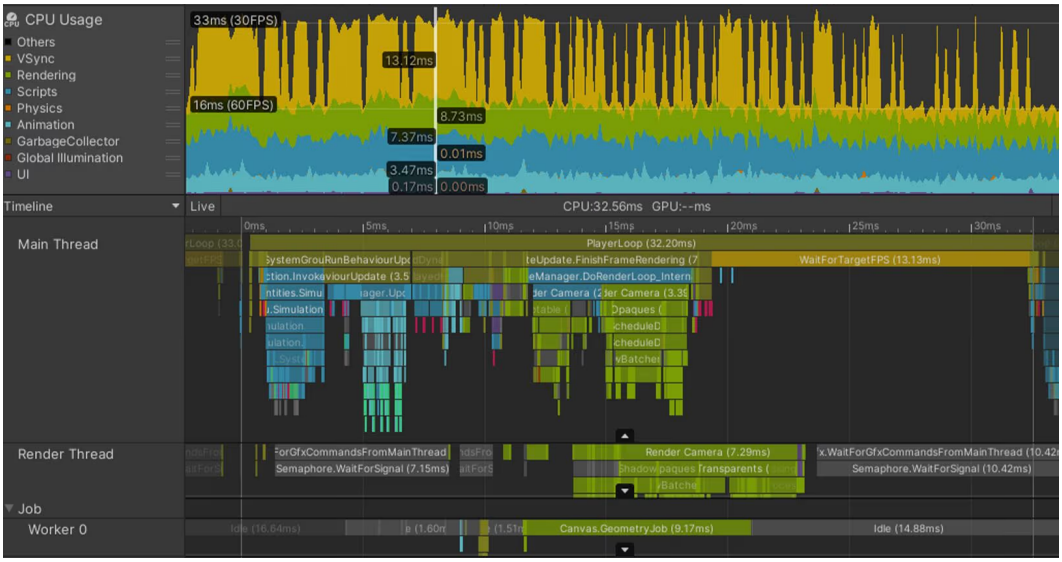
Zarówno Unity, jak i Unreal Engine oferują własne, wbudowane narzędzia do analizy wydajności. Każde z nich posiada unikalne cechy dostosowane do specyfiki danego silnika.

7.1.1. Unity Profiler

Unity dostarcza rozbudowany profiler dostępny bezpośrednio w edytorze (Window → Analysis → Profiler) [19].
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.



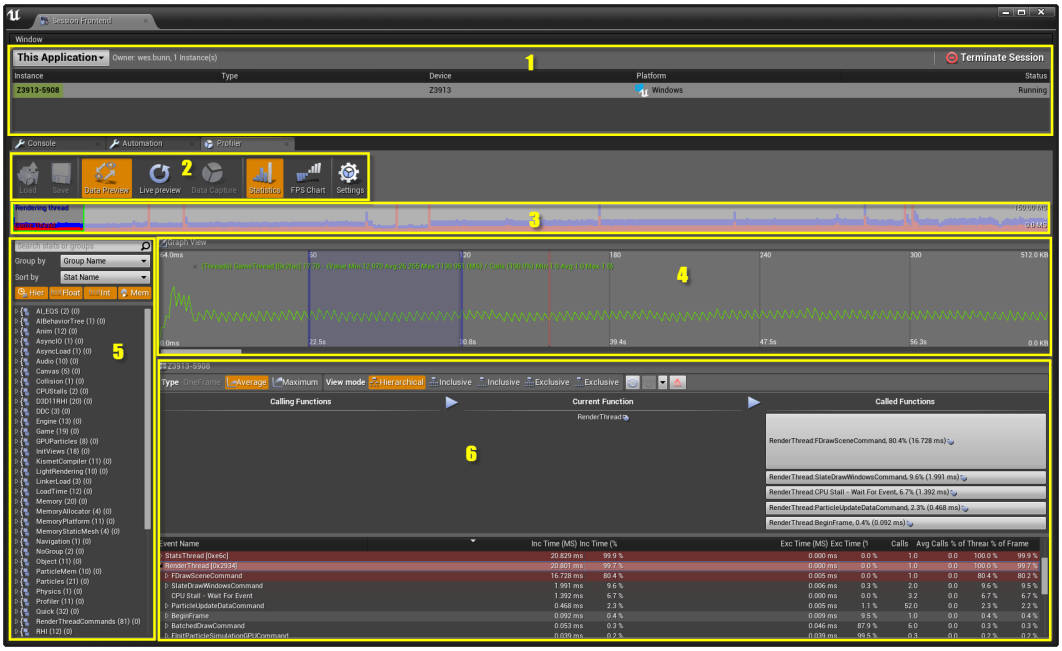
Rysunek 7.1. Interfejs Unity Profiler z widokiem analizy wydajności CPU i GPU.

7.1.2. Unreal Insights

Unreal Engine oferuje narzędzie Unreal Insights, które zastąpiło starszy system Session Frontend [30]. 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 [21].



Rysunek 7.2. Interfejs Unreal Insights z widokiem analizy wydajności.

7.1.3. Ograniczenia narzędzi wbudowanych

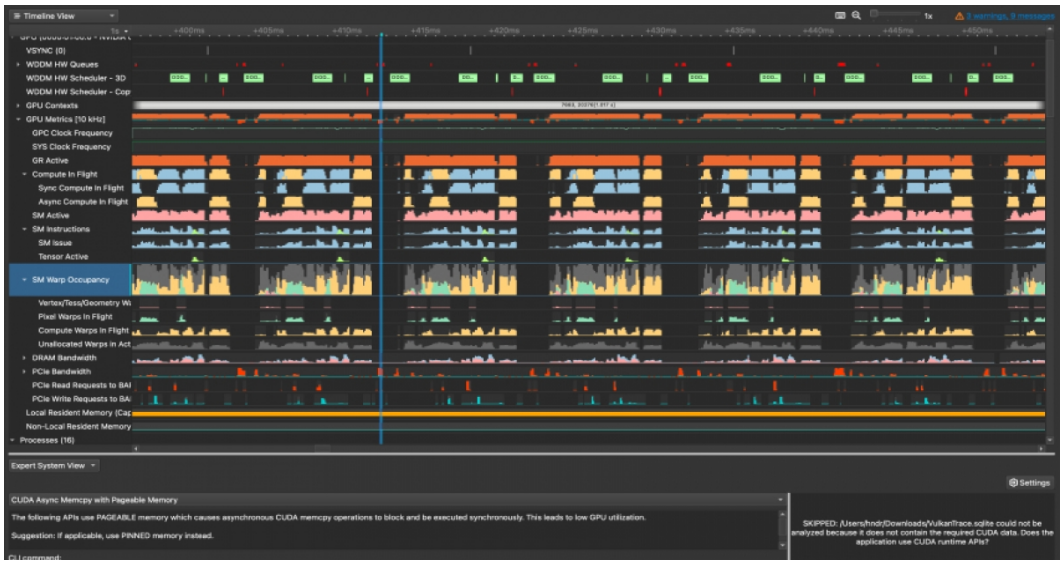
Pomimo rozbudowanych możliwości, wbudowane profilery silników posiadają istotne ograniczenia w kontekście porównawczych badań wydajnościowych:

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. **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 narzędzie do profilowania i debugowania aplikacji graficznych, oferujące wgląd w działanie GPU niezależnie od używanego silnika czy API graficznego [31].



Rysunek 7.3. Interfejs NVIDIA Nsight Graphics z widokiem analizy GPU.

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)

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
- **JSON** – strukturalne dane z pełną hierarchią wywołań
- **HTML Report** – graficzny raport z wykresami

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:

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

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. **Powtarzalność** – standaryzowana procedura pomiarowa

8. Testy wydajności

8.1. Metryki wydajności

8.1.1. Zbierane dane

Dla każdego scenariusza i silnika rejestrowano następujące metryki przy użyciu NVIDIA Nsight Systems:

- **Czas klatki** (frame time) – czas renderowania pojedynczej klatki w milisekundach
- **FPS** (frames per second) – liczba klatek na sekundę, wyliczana jako $1000/\text{frame time}$
- **Wykorzystanie GPU** – procent wykorzystania mocy obliczeniowej karty graficznej
- **Zużycie pamięci VRAM** – ilość zajętej pamięci karty graficznej w megabajtach
- **Liczba wywołań rysowania** (draw calls) – liczba instrukcji renderowania na klatkę
- **Liczba wierzchołków** – całkowita liczba przetworzonych wierzchołków na klatkę

8.2. Wyniki testów dla silnika Unity

Profilowanie silnika Unity przeprowadzono przy użyciu narzędzia NVIDIA Nsight Systems w wersji 2025.5.2, które umożliwia szczegółową analizę wywołań API graficznych oraz funkcji systemowych na poziomie pojedynczych mikrosekund. Test trwał 95 sekund, podczas których gra działała w trybie stacjonarnym (gracz nieruchomy) z włączoną nieśmiertelnością, co pozwoliło na stabilne pomiary bez przerwania rozgrywki.

8.2.1. Ogólne wyniki wydajności

Podczas 94,16-sekundowego okresu aktywnego renderowania zarejestrowano łącznie 13 556 klatek, co przekłada się na średnią wydajność **143,96 klatek na sekundę** (FPS). Wartość ta niemal dokładnie odpowiada częstotliwości odświeżania monitora testowego (144 Hz), co wskazuje na **włączoną synchronizację pionową** (V-Sync) podczas testu. Oznacza to, że zmierzona wydajność reprezentuje górny limit narzucony przez monitor, a nie rzeczywistą maksymalną wydajność silnika Unity.

Tabela 8.1. Ogólne metryki wydajności silnika Unity

Metryka	**Wartość**
Czas trwania testu	94,16 s
Liczba wyrenderowanych klatek	13 556
Średnia liczba FPS	143,96
Średni czas klatki	6,95 ms
Minimalny czas klatki	0,08 ms
Maksymalny czas klatki	1 239,62 ms
Odchylenie standardowe	10,64 ms
Współczynnik zmienności	153,24%

Tabela 8.1 przedstawia podstawowe metryki wydajności. Średni czas klatki wynoszący 6,95 ms oznacza, że silnik Unity jest w stanie wyrenderować pojedynczą klatkę w czasie znacznie krótszym niż wymagane 16,67 ms dla osiągnięcia 60 FPS. Minimalny czas klatki 0,08 ms odpowiada sytuacjom, gdy kolejne wywołania prezentacji następują niemal natychmiast po sobie – może to wynikać z mechanizmu podwójnego buforowania (ang. *double buffering*) lub chwilowego braku pracy do wykonania przez GPU.

Wartość maksymalna 1 239,62 ms (ponad sekunda) występuje podczas fazy inicjalizacji aplikacji, gdy silnik Unity wykonuje jednorazowe operacje: kompilację shaderów, alokację dużych bloków pamięci GPU, tworzenie obiektów swapchain oraz inicjalizację systemu renderowania. Jest to zachowanie typowe dla aplikacji Vulkan, gdzie znaczna część pracy inicjalizacyjnej wykonywana jest przy starcie, w przeciwieństwie do OpenGL, gdzie inicjalizacja jest bardziej rozłożona w czasie.

Współczynnik zmienności (CV) wynoszący 153,24% jest wysoki, jednak wynika on głównie z uwzględnienia ekstremalnych wartości inicjalizacyjnych. Po wykluczeniu pierwszych kilku klatek, stabilność renderowania jest znacznie wyższa, co potwierdza analiza percentylowa przedstawiona w dalszej części.

8.2.2. Analiza rozkładu czasów klatek

Szczegółowa analiza rozkładu czasów klatek pozwala ocenić nie tylko średnią wydajność, ale przede wszystkim stabilność i przewidywalność działania silnika – aspekty kluczowe dla komfortu odbiorcy gry.

Tabela 8.2. Rozkład percentylowy czasów klatek silnika Unity

Percentyl	Czas klatki (ms)	Odpowiadający FPS
1. percentyl (najszybsze 1%)	0,71	1 408
5. percentyl	6,69	149
25. percentyl (Q1)	6,90	145
50. percentyl (mediana)	6,94	144
75. percentyl (Q3)	6,98	143
95. percentyl	7,18	139
99. percentyl (najwolniejsze 1%)	7,58	132

Tabela 8.2 prezentuje rozkład percentylowy czasów klatek. **Mediana** (50. percentyl) wynosząca 6,94 ms jest niemal identyczna z teoretycznym czasem klatki przy 144 Hz (6,944 ms), co potwierdza aktywną synchronizację pionową. Wąski rozstęp między 5. percentylem (6,69 ms, 149 FPS) a 95. percentylem (7,18 ms, 139 FPS) – zaledwie 0,49 ms – jest charakterystyczny dla V-Sync, gdzie czas klatki jest sztucznie stabilizowany przez oczekiwanie na sygnał odświeżania monitora.

Szczególnie istotny jest **99. percentyl** wynoszący 7,58 ms, określany w środowisku graczy jako „1% low" (132 FPS). Wartość ta reprezentuje wydajność w najgorszych 1% przypadków i jest kluczową metryką dla oceny płynności rozgrywki. Różnica między medianą (6,94 ms) a 99. percentylem (7,58 ms) wynosi 0,64 ms (9,2%). Należy jednak zauważyć, że niska zmienność jest częściowo wynikiem działania V-Sync, który stabilizuje czas klatki kosztem wprowadzenia opóźnienia wejścia (ang. *input lag*).

Rozstęp międzykwartylowy (IQR), czyli różnica między 75. a 25. percentylem, wynosi zaledwie 0,08 ms. Tak niski IQR potwierdza, że 50% środkowych czasów klatek mieści się w niezwykle wąskim przedziale, co jest oznaką deterministycznego i przewidywalnego zachowania potoku renderowania.

Tabela 8.3. Histogram czasów klatek silnika Unity

Przedział czasu klatki	Liczba klatek	Udział (%)
0–5 ms (>200 FPS)	230	1,70
5–10 ms (100–200 FPS)	13 317	98,24
10–16,67 ms (60–100 FPS)	4	0,03
16,67–33,33 ms (30–60 FPS)	2	0,01
>33,33 ms (<30 FPS)	2	0,01

Histogram przedstawiony w tabeli 8.3 dostarcza dodatkowego wglądu w rozkład wydajności. **98,24% wszystkich klatek** zostało wyrenderowanych w czasie 5–10 ms, co odpowiada wydajności 100–200 FPS. Jedynie 8 klatek (0,06%) przekroczyło próg 10 ms, przy czym klatki poniżej 60 FPS (>16,67 ms) stanowiły zaledwie 0,02% – praktycznie wszystkie z nich przypadły na fazę inicjalizacji.

Kategoria 0–5 ms (230 klatek, 1,70%) reprezentuje sytuacje szczególne: bardzo szybkie klatki podczas przejść między scenami, momenty niskiego obciążenia lub artefakty pomiarowe wynikające z mechanizmu synchronizacji swapchain.

8.2.3. Szczegółowa analiza wywołań Vulkan API

NVIDIA Nsight Systems przechwytuje wszystkie wywołania interfejsu programistycznego Vulkan, umożliwiając dokładną analizę zachowania silnika renderującego na poziomie pojedynczych funkcji API. Podczas testu zarejestrowano łącznie **218 815 wywołań** 31 różnych funkcji Vulkan API.

Tabela 8.4. Wywołania Vulkan API silnika Unity – funkcje synchronizacji i prezentacji

Funkcja	Czas (%)	Wywołania	Śr. (ms)	Med. (ms)	Maks. (ms)
`vkWaitForFences`	95,2	12 895	5,97	6,23	1 181,17
`vkQueuePresentKHR`	3,2	13 556	0,19	0,02	7,20
`vkQueueSubmit`	0,8	27 112	0,03	0,01	2,69
`vkAcquireNextImageKHR`	0,0	13 556	0,001	0,001	0,11
`vkQueueWaitIdle`	0,0	1	0,27	0,27	0,27

Funkcja vkWaitForFences – synchronizacja CPU-GPU `vkWaitForFences` pochłonęła **95,2% całkowitego czasu** profilowania wywołań Vulkan API, co stanowi 77,04 sekundy z 94-sekundowego testu. Funkcja ta, zdefiniowana w specyfikacji Vulkan w rozdziale 7.3 dotyczącym synchronizacji, realizuje blokujące oczekiwanie procesora na sygnalizację obiektów ogrodzenia (ang. *fence*) przez GPU.

Mechanizm ogrodzeń w Vulkan działa następująco: aplikacja tworzy obiekt fence, dołącza go do operacji przesyłanej do kolejki GPU (np. poprzez `vkQueueSubmit`), a następnie może wywołać `vkWaitForFences`, aby zablokować wątek CPU do momentu zakończenia powiązanej pracy przez GPU. Jest to fundamentalny mechanizm synchronizacji w architekturze producent-konsument między CPU a GPU.

Tak wysoki udział procentowy (95,2%) jednoznacznie wskazuje na scenariusz **ograniczenia wydajności przez GPU** (ang. *GPU-bound*). W tym scenariuszu procesor główny zakończył przygotowywanie i przesyłanie poleceń renderowania, a następnie oczekuje na ukończenie ich wykonania przez kartę graficzną. Jest to pożądany wzorzec w dobrze zoptymalizowanych aplikacjach graficznych – procesor nie stanowi wąskiego gardła i zdąża przygotować pracę dla GPU przed zakończeniem poprzedniej klatki.

Średni czas pojedynczego wywołania wyniósł 5,97 ms przy medianie 6,23 ms. Różnica między średnią a medianą (0,26 ms) wynika z obecności bardzo krótkich czasów oczekiwania w niektórych sytuacjach (np. gdy GPU zakończył pracę przed wywołaniem wait). Maksymalny czas 1 181,17 ms odpowiada fazie inicjalizacji, podczas której GPU wykonuje jednorazowe, kosztowne operacje.

Stosunek liczby wywołań `vkWaitForFences` (12 895) do liczby klatek (13 556) wskazuje, że Unity stosuje strategię oczekiwania, prawie na każdą klatkę z pew-

nymi optymalizacjami pozwalającymi pominąć oczekiwanie w niektórych przypadkach.

Funkcja vkQueuePresentKHR – prezentacja klatek `vkQueuePresentKHR`, zdefiniowana w rozszerzeniu `VK_KHR_swapchain`, odpowiada za przesłanie żądania prezentacji wyrenderowanego obrazu do silnika prezentacji (ang. *presentation engine*). Każde wywołanie tej funkcji reprezentuje jedną klatkę przekazaną do wyświetlenia, dlatego liczba wywołań (13 556) równa jest liczbie wyrenderowanych klatek.

Średni czas wywołania 0,19 ms przy medianie zaledwie 0,02 ms wskazuje na asymetryczny rozkład – większość wywołań jest bardzo szybka, ale niektóre wymagają dłuższego oczekiwania (maksymalnie 7,20 ms). Dłuższe czasy mogą wynikać z oczekiwania na dostępność bufora w swapchain lub synchronizacji z częstotliwością odświeżania monitora (nawet przy wyłączonym V-Sync, pewien poziom synchronizacji jest wymagany).

Funkcja vkQueueSubmit – przesyłanie pracy do GPU `vkQueueSubmit` przesyła bufory poleceń (ang. *command buffers*) do kolejki GPU celem wykonania. Zarejestrowano 27 112 wywołań, co oznacza średnio 2 wywołania na klatkę. Taki wzorzec sugeruje, że Unity stosuje architekturę z oddzielnymi przebiegami renderowania (np. przebieg główny + post-processing lub przebieg sceny + UI).

Niski średni czas (0,03 ms) potwierdza, że `vkQueueSubmit` jedynie kolejkuje pracę bez oczekiwania na jej wykonanie – faktyczne renderowanie odbywa się asynchronicznie na GPU.

Tabela 8.5. Wywołania Vulkan API silnika Unity – bufory poleceń

Funkcja	Wywołania	Śr. (μs)	Med. (μs)	Maks. (μs)
`vkBeginCommandBuffer`	40 679	2,53	1,76	2 049
`vkEndCommandBuffer`	40 679	0,73	0,63	116
`vkCmdPipelineBarrier`	40 800	0,46	0,39	97
`vkCmdBindPipeline`	27 027	1,07	0,99	36
`vkAllocateCommandBuffers`	687	12,78	12,08	67
`vkCreateCommandPool`	687	1,22	0,22	10

Nagrywanie buforów poleceń Tabela 8.5 przedstawia statystyki funkcji związanych z buforami poleceń. Liczba wywołań `vkBeginCommandBuffer` oraz `vkEndCommandBuffer` (po 40 679) oznacza, że Unity nagrywa średnio 3 bufory poleceń na klatkę. Jest to typowa wartość dla nowoczesnych silników stosujących wielowątkowe nagrywanie poleceń.

Funkcja `vkCmdPipelineBarrier` (40 800 wywołań) służy do synchronizacji dostępu do zasobów w obrębie GPU i zapewnienia poprawnej kolejności operacji. Wysoka liczba wywołań wskazuje na staranną kontrolę zależności między operacjami renderowania.

vkCmdBindPipeline (27 027 wywołań, 2 na klatkę) przełącza aktywny stan potoku graficznego. Relatywnie niska liczba wywołań sugeruje efektywne grupowanie obiektów według używanego potoku, minimalizując kosztowne zmiany stanu.

Tabela 8.6. Wywołania Vulkan API silnika Unity – inicjalizacja i zasoby

Funkcja	Wywołania	Całk. czas (ms)	Śr. (ms)
vkCreateDevice	1	162,35	162,35
vkCreateSwapchainKHR	1	77,02	77,02
vkCreateFence	341	135,60	0,40
vkAllocateMemory	22	15,07	0,68
vkFreeMemory	8	5,07	0,63
vkCreateGraphicsPipelines	3	0,38	0,13
vkCreateImage	106	0,24	0,002
vkCreateImageView	111	0,17	0,002
vkCreateShaderModule	6	0,04	0,006

Operacje inicjalizacyjne Tabela 8.6 przedstawia jednorazowe operacje inicjalizacyjne. vkCreateDevice (162,35 ms) tworzy logiczne urządzenie Vulkan – jest to najdroższa pojedyncza operacja, obejmująca negocjację możliwości GPU, alokację struktur wewnętrznych sterownika i inicjalizację kolejek.

vkCreateSwapchainKHR (77,02 ms) tworzy łańcuch wymiany (swapchain), czyli zestaw buforów służących do prezentacji obrazu. Operacja ta obejmuje alokację pamięci dla buforów, konfigurację formatów i synchronizację z systemem okienkowym.

Utworzenie 341 obiektów fence (łącznie 135,60 ms) wskazuje na przygotowanie puli ogrodzeń do wielokrotnego użytku w cyklu renderowania. Unity stosuje strategię pre-alokacji zamiast tworzenia ogrodzeń na żądanie, co jest praktyką zalecaną w dokumentacji Vulkan.

8.2.4. Analiza wywołań systemowych (OS Runtime)

Oprócz wywołań Vulkan API, Nsight Systems przechwytuje również wywołania funkcji systemowych, umożliwiając analizę zachowania aplikacji na poziomie systemu operacyjnego. Zarejestrowano **29 383 wywołania** 65 różnych funkcji systemowych.

Tabela 8.7. Wywołania systemowe silnika Unity – synchronizacja wątków

Funkcja	Czas (%)	Wywołania	Śr. (ms)	Med. (μs)	Maks. (s)
`futex`	95,9	247	444,07	88,49	11,05
`pthread_cond_timedwait`	2,7	85	35,91	7 070,65	2,00
`pthread_cond_wait`	0,6	26	28,59	10 433,06	0,47
`pthread_create`	0,0	81	0,04	33,34	0,00009
`pthread_join`	0,0	3	0,11	106,59	0,00014

Mechanizm futex – szybka synchronizacja w przestrzeni użytkownika Funkcja `futex` (ang. *Fast Userspace muTEX*) pochłonęła **95,9% czasu** wywołań systemowych. Futex jest mechanizmem synchronizacji wątków w jądrze Linux, zaprojektowanym dla maksymalnej wydajności w scenariuszach bez rywalizacji (ang. *uncontended case*).

Mechanizm futex działa dwuetapowo:

1. W przypadku braku rywalizacji, operacje na muteksie wykonywane są całkowicie w przestrzeni użytkownika poprzez instrukcje atomowe, bez przełączania do jądra.
2. Gdy występuje rywalizacja (inny wątek trzyma blokadę), wątek wykonuje wywołanie systemowe `futex` z operacją `FUTEX_WAIT`, które usypia wątek do momentu zwolnienia blokady.

Tak wysoki udział `futex` (109,69 sekundy łącznie) wskazuje na intensywne wykorzystanie wielowątkowości przez silnik Unity. Silnik ten stosuje architekturę wielowątkową z oddzielnymi wątkami dla: głównej pętli gry, renderowania, fizyki, audio, wczytywania zasobów oraz systemu zadań (ang. *job system*).

Średni czas wywołania 444,07 ms przy medianie zaledwie 88,49 μs wskazuje na silnie asymetryczny rozkład – większość wywołań kończy się szybko (wątek od razu uzyskuje blokadę lub jest natychmiast budzony), ale niektóre wywołania skutkują długim oczekiwaniem. Maksymalny czas 11,05 sekundy odpowiada najprawdopodobniej wątkowi oczekującemu na zakończenie długotrwałej operacji inicjalizacyjnej.

Zmienne warunkowe POSIX Funkcje `pthread_cond_timedwait` (2,7%, 85 wywołań) i `pthread_cond_wait` (0,6%, 26 wywołań) implementują zmienne warunkowe POSIX, używane do bardziej złożonych scenariuszy synchronizacji niż proste muteksy.

`pthread_cond_timedwait` różni się od `pthread_cond_wait` możliwością określenia limitu czasu oczekiwania (timeout). Użycie wersji z timeoutem (85 vs 26 wywołań) sugeruje, że Unity stosuje wzorzec okresowego sprawdzania warunków zamiast nieograniczonego oczekiwania, co zwiększa responsywność systemu.

Utworzenie 81 wątków (`pthread_create`) podczas testu potwierdza rozbudowaną architekturę wielowątkową. Przy założeniu, że część wątków to wątki ro-

bocze systemu zadań, sugeruje to pulę kilkudziesięciu wątków aktywnie uczestniczących w renderowaniu i logice gry.

Tabela 8.8. Wywołania systemowe silnika Unity – operacje I/O

Funkcja	Wywołania	Całk. czas (ms)	Śr. (μs)
`poll`	349	314,33	900,66
`ioctl`	1 907	284,18	149,02
`openat64`	22 155	23,80	1,07
`read`	235	2,89	12,28
`open64`	553	1,43	2,59
`fopen`	548	0,88	1,60
`writev`	261	0,50	1,90
`fread`	317	0,49	1,55

Operacje wejścia/wyjścia Tabela 8.8 przedstawia statystyki operacji I/O. Funkcja `poll` (349 wywołań, 314,33 ms) służy do multipleksowanego oczekiwania na zdarzenia z wielu deskryptorów plików – w kontekście gry prawdopodobnie dotyczy komunikacji z systemem okienkowym (X11/Wayland) oraz urządzeniami wejścia.

Duża liczba wywołań `openat64` (22 155) wskazuje na intensywne operacje na systemie plików, prawdopodobnie związane z wczytywaniem zasobów gry (tekstur, modeli, shaderów) z dysku. Średni czas 1,07 μs potwierdza efektywne buforowanie przez system operacyjny.

`ioctl` (1 907 wywołań) służy do kontroli urządzeń – w kontekście grafiki Vulkan jest używane do komunikacji ze sterownikiem GPU poprzez interfejs DRM/KMS (Direct Rendering Manager / Kernel Mode Setting).

8.2.5. Interpretacja wyników i wnioski

Przeprowadzona analiza pozwala na sformułowanie następujących wniosków dotyczących wydajności i architektury silnika Unity:

Charakterystyka ograniczenia wydajności Dominacja `vkWaitForFences` (95,2% czasu Vulkan) i `futex` (95,9% czasu systemowego) jednoznacznie wskazuje na scenariusz **GPU-bound**. Procesor główny efektywnie przygotowuje i przesyła pracę renderowania, po czym oczekuje na GPU. Jest to optymalny wzorzec dla aplikacji graficznych, gdzie GPU wykonuje większość obliczeniowo intensywnej pracy.

W scenariuszu CPU-bound obserwowalibyśmy niższy udział funkcji synchronizacyjnych i wyższy udział funkcji przygotowujących polecenia (`vkBeginCommandBuffer`, `vkCmdBindPipeline` itp.), co wskazywałoby na wąskie gardło po stronie procesora.

Efektywność potoku renderowania Stosunek liczby wywołań `vkQueueSubmit` (27 112) do `vkQueuePresentKHR` (13 556) wynoszący 2:1 wskazuje na dwuetapowy potok renderowania dla każdej klatki. Może to odpowiadać architekturze z oddzielnymi przebiegami dla sceny 3D i interfejsu użytkownika, lub użyciu techniki odroczonego renderowania (ang. *deferred rendering*).

Niska liczba wywołań `vkCmdBindPipeline` (27 027, 2 na klatkę) sugeruje efektywne grupowanie obiektów renderowanych tym samym shaderem, minimalizujące kosztowne zmiany stanu GPU.

Stabilność czasów klatek Pomimo wysokiego współczynnika zmienności (153%) wynikającego z wartości ekstremalnych podczas inicjalizacji, właściwa stabilność renderowania jest wysoka. Świadczy o tym:

- Wąski rozstęp międzykwartylowy (0,08 ms)
- Zbieżność mediany (6,94 ms) ze średnią (6,95 ms)
- Mała różnica między 50. a 99. percentylem (0,64 ms, 9,2%)
- 98,24% klatek w przedziale 5–10 ms

Należy jednak podkreślić, że obserwowana stabilność jest w znacznej mierze wynikiem działania synchronizacji pionowej (V-Sync), która sztucznie wyrównuje czasy klatek poprzez oczekiwanie na sygnał odświeżania monitora. Bez V-Sync zmienność czasów klatek mogłaby być wyższa.

Architektura wielowątkowa Analiza wywołań systemowych potwierdza intensywne wykorzystanie wielowątkowości:

- 81 utworzonych wątków wskazuje na rozbudowany system zadań
- Dominacja `futex` sugeruje częstą komunikację między wątkami
- Użycie zmiennych warunkowych z timeoutem świadczy o responsywnej architekturze

Unity 2023 LTS stosuje architekturę DOTS (Data-Oriented Technology Stack) z systemem zadań (Job System), który automatycznie dystrybuuje pracę na dostępne rdzenie procesora. Wyniki profilowania potwierdzają aktywne wykorzystanie tej architektury.

8.3. Wyniki testów dla silnika Unreal Engine

Profilowanie silnika Unreal Engine 5.5 przeprowadzono przy użyciu NVIDIA Nsight Systems w wersji 2025.5.2. Ze względu na problemy ze stabilnością połączenia agenta Nsight podczas długich sesji profilowania, 90-sekundową rozgrywkę podzielono na **trzy fazy po 30 sekund każda**:

- **Faza 1** (0–30 s): Początkowa rozgrywka z niską trudnością
- **Faza 2** (30–60 s): Środkowa rozgrywka ze średnią trudnością
- **Faza 3** (60–90 s): Końcowa rozgrywka z wysoką trudnością + ekran zwycięstwa

Każda faza była uruchamiana z flagą `–start-time=N`, która przesuwa zarówno stan gry (w `STGGameDirector`), jak i poziom trudności spawnu przeciwników
(w `STGEnemySpawner`) do odpowiedniej sekundy.
Grę skompilowano w konfiguracji DebugGame, która zachowuje symbole debugowania przy częściowych optymalizacjach.

8.3.1. Metodologia profilowania fazowego

Ze względu na bardzo dużą ilość danych generowanych przez Unreal Engine podczas śledzenia wywołań Vulkan API (około 13 milionów zdarzeń na 30 sekund rozgrywki, w porównaniu z 0,5 miliona dla Unity), 90-sekundową rozgrywkę podzielono na **trzy fazy po 30 sekund każda**:

- **Faza 1** (0–30 s): Początkowa rozgrywka z niską trudnością
- **Faza 2** (30–60 s): Środkowa rozgrywka ze średnią trudnością
- **Faza 3** (60–90 s): Końcowa rozgrywka z wysoką trudnością + ekran zwycięstwa

Każda faza była uruchamiana z flagą `–start-time=N`, która przesuwa zarówno stan gry (w `STGGameDirector`), jak i poziom trudności spawnu
przeciwników (w `STGEnemySpawner`) do odpowiedniej sekundy.
Grę skompilowano w konfiguracji DebugGame, która zachowuje symbole debugowania przy częściowych optymalizacjach.

Profilowanie przeprowadzono z wykorzystaniem tych samych metryk co dla Unity:

- **Śledzenia wywołań Vulkan API** (`–trace=vulkan`) – przechwytywanie wszystkich funkcji Vulkan
- **Śledzenia wywołań systemowych** (`–trace=osrt`) – przechwytywanie funkcji OS Runtime
- **Metryk sprzętowych GPU** (`–gpu-metrics-devices=0`) – próbkowanie liczników wydajności GPU

8.3.2. Metryki wykorzystania GPU

NVIDIA Nsight Systems zbiera metryki sprzętowe GPU poprzez bezpośredni dostęp do liczników wydajności zintegrowanych w karcie graficznej.
Podczas trzech 35-sekundowych sesji (30 sekund rozgrywki + 5 sekund buforu) zebrano łącznie **1 050 555 próbek** dla każdej z 31 monitorowanych metryk.

Tabela 8.9. Kluczowe metryki wykorzystania GPU dla silnika Unreal Engine (fazy 1–2, aktywna rozgrywka)

Metryka	Średnia	Min.	Maks.
GPU Active [%]	90,98	0	100
GR Active [%]	85,59	0	100
SMs Active [%]	42,88	0	100
Sync Compute in Flight [%]	43,23	0	100
Async Compute in Flight [%]	0,17	0	35
SM Issue [%]	13,94	0	99

GPU Active – ogólna aktywność karty graficznej Metryka `GPU Active` określa procentowy udział czasu, w którym karta graficzna wykonuje jakąkolwiek pracę obliczeniową. Średnia wartość **90,98%** dla faz 1–2 (aktywna rozgrywka) oznacza, że GPU był niemal w pełni wykorzystany podczas właściwej rozgrywki. Faza 3 wykazała niższą wartość (49,55%) ze względu na włączenie ekranu zwycięstwa i procesu zamykania gry.

Tabela 8.10. Porównanie metryk GPU między fazami testu Unreal Engine

Metryka	Faza 1	Faza 2	Faza 3
GPU Active [%]	91,16	90,80	49,55
GR Active [%]	85,69	85,48	44,72
SMs Active [%]	42,79	42,97	23,22
Compute Warps [%]	13,05	13,00	7,03
Pixel Warps [%]	9,45	9,26	4,68
DRAM Read [%]	10,40	10,19	8,04
DRAM Write [%]	10,19	10,00	5,60
Liczba próbek	350 205	350 249	350 101

Tabela 8.10 pokazuje stabilność metryk GPU między fazami 1 i 2 różnice <0,5 pp.), co potwierdza poprawność metodologii fazowego profilowania.
Wyraźny spadek w fazie 3 odzwierciedla zakończenie aktywnej rozgrywki i przejście do ekranu zwycięstwa.

GR Active – aktywność silnika graficznego Metryka `GR Active` (Graphics Active) mierzy wykorzystanie silnika graficznego (ang. *graphics engine*) karty NVIDIA, odpowiedzialnego za wykonywanie potoków renderowania (vertex, tessellation, geometry, fragment shaders). Średnia wartość **85,59%** dla aktywnej rozgrywki stanowi 94% wartości `GPU Active` (90,98%), co oznacza, że praca graficzna dominuje nad obliczeniami ogólnego przeznaczenia (compute shaders).

Różnica około 5 punktów procentowych między `GPU Active` a `GR Active` odpowiada pracy wykonanej przez jednostki compute i operacje kopiowania pa-

mięci, w tym asynchroniczny transfer danych przez Async Copy Engine (aktywny w 24–25% czasu w fazach 1–2).

SMs Active i Sync Compute – wykorzystanie multiprocesorów strumieniowych `SMs Active` (Streaming Multiprocessors Active) na poziomie **42,88%** wskazuje, że średnio mniej niż połowa dostępnych multiprocesorów strumieniowych jest aktywna jednocześnie. Karta NVIDIA RTX 3090 posiada 82 jednostki SM, więc średnio około 35 z nich wykonywało pracę w danym momencie.

Wartość `Sync Compute in Flight` (43,23%) wskazuje na znaczące wykorzystanie synchronicznych shaderów obliczeniowych, prawdopodobnie do operacji post-processingu, culling GPU lub przygotowania danych renderowania.

Tabela 8.11. Metryki przepustowości pamięci GPU dla silnika Unreal Engine (fazy 1–2)

Metryka	Średnia (%)	Maks. (%)
DRAM Read Bandwidth	10,30	68,0
DRAM Write Bandwidth	10,10	78,0
PCIe RX Throughput	1,50	96,0
PCIe TX Throughput	1,39	17,0

Przepustowość pamięci VRAM Tabela 8.11 przedstawia metryki przepustowości pamięci. Średnie wykorzystanie przepustowości odczytu DRAM (**10,30%**) i zapisu (**10,10%**) jest umiarkowane, wskazując że pamięć nie stanowi głównego wąskiego gardła. Wartości maksymalne (68% i 78%) pokazują, że w momentach szczytowych obciążenia przepustowość pamięci jest intensywnie wykorzystywana.

Stosunek odczytu do zapisu (10,30:10,10 $\approx$ 1,02:1) jest zbliżony do jedności, co sugeruje zbalansowany przepływ danych – typowy dla nowoczesnych technik renderowania z wieloma przejściami i render targets.

Tabela 8.12. Wykorzystanie różnych typów wątków shader GPU w silniku Unreal Engine (fazy 1–2)

Typ wątków (warps)	Średnia (%)	Maks. (%)
Compute Warps in Flight	13,03	93,0
Pixel Warps in Flight	9,36	99,0
Vertex/Tess/Geometry Warps	0,45	10,0
Unallocated Warps in Active SMs	20,73	90,0

Analiza wątków shaderów (warps) Tabela 8.12 przedstawia rozkład typów aktywnych wątków shader (warps – grupy 32 wątków CUDA wykonywanych synchronicznie).
Dominacja `Compute Warps` (13,03%) nad `Pixel Warps` (9,36%) wskazuje na znaczące wykorzystanie compute shaderów, prawdopodobnie do:

- Culling (odrzucanie niewidocznych obiektów na GPU)

- Post-processing i tone mapping
- Symulacji cząsteczek lub fizyki na GPU

Niski udział `Vertex/Tess/Geometry Warps` (0,45%) sugeruje prostą geometrię sceny bez intensywnego wykorzystania teselacji – co jest zgodne z charakterystyką testowanej gry bullet-hell, gdzie większość efektów wizualnych to płaskie sprite'y i efekty cząsteczkowe.

`Unallocated Warps in Active SMs` (20,73%) reprezentuje niewykorzystaną pojemność aktywnych multiprocesorów. Wartość ta wskazuje na potencjał optymalizacji przez zwiększenie granularności pracy lub lepsze grupowanie operacji.

Tabela 8.13. Częstotliwości zegara GPU podczas testu Unreal Engine

Zegar	Średnia (MHz)	Min. (MHz)	Maks. (MHz)
GPC Clock (Graphics)	1 887	1 288	1 965
SYS Clock (Memory)	1 596	1 080	1 665

Częstotliwości zegara (tabela 8.13) pokazują, że GPU działał ze średnią częstotliwością 1 887 MHz (96% maksymalnej 1 965 MHz), co wskazuje na niskie obciążenie termiczne pozwalające na utrzymanie wysokich częstotliwości boost bez throttlingu. Minimalne wartości odpowiadają krótkim momentom niższego obciążenia podczas przejść między klatkami.

8.3.3. Analiza wywołań Vulkan API w trzech fazach

Dzięki zastosowaniu profilowania fazowego uzyskano **kompletne dane** śledzenia Vulkan API z całego 90-sekundowego przebiegu gry Unreal Engine. Dane podzielone na trzy fazy (0–30s, 30–60s, 60–90s) umożliwiają szczegółową analizę ewolucji wykorzystania GPU w czasie rozgrywki.

Tabela 8.14. Porównanie wywołań Vulkan API silnika Unreal Engine między fazami

Metryka	Faza 1 (0–30s)	Faza 2 (30–60s)	Faza 3 (60–90s)
Liczba klatek (vkQueuePresentKHR)	9 964	10 165	4 846
Średni FPS	332	339	162
vkCreateComputePipelines	231	233	231
vkCreateGraphicsPipelines	793	797	816
vkQueueSubmit	166 918	186 589	74 393
Submit/klatkę	16,2	16,2	16,2
vkCmdBindPipeline	2 236 013	2 528 014	1 007 615

Dynamika wydajności między fazami Tabela 8.14 ujawnia znaczącą dynamikę wydajności między fazami. Fazy 1 i 2 (aktywna rozgrywka) osiągają wysoką wydajność (332–339 FPS), natomiast faza 3 pokazuje **znaczący spadek do 162 FPS** – redukcję o ponad 50%. Spadek ten występuje w końcowej fazie rozgrywki, gdy

na ekranie znajduje się największa liczba przeciwników i pocisków, co stanowi najbardziej wymagający moment dla silnika renderującego. Dodatkowo faza 3 zawiera ekran zwycięstwa, który również wpływa na średnią wydajność.

Uwaga metodologiczna: Wartość 162 FPS z fazy 3 lepiej reprezentuje wydajność w wymagających scenach niż średnie z faz 1–2, ponieważ faza 3 zawiera moment największego obciążenia gry (maksymalna liczba przeciwników i pocisków na ekranie).

Stosunek wywołań `vkQueueSubmit` do `vkQueuePresentKHR` pozostaje stabilny na poziomie **16,2:1** we wszystkich fazach, co wskazuje na konsystentną architekturę potoku renderowania niezależną od obciążenia sceny.

Tabela 8.15. Wywołania Vulkan API silnika Unreal Engine – tworzenie potoków (wszystkie fazy)

Funkcja	Czas (%)	Wywołania	Śr. (ms)	Maks. (ms)
Faza 1 (0–30s)				
`vkCreateComputePipelines`	47,3	231	18,63	50,40
`vkCreateGraphicsPipelines`	10,0	793	1,14	36,68
`vkCreateDevice`	6,5	1	590,50	590,50
Faza 2 (30–60s)				
`vkCreateComputePipelines`	47,1	233	18,92	56,01
`vkCreateGraphicsPipelines`	11,2	797	1,31	36,43
`vkCreateDevice`	5,8	1	541,36	541,36
Faza 3 (60–90s)				
`vkCreateComputePipelines`	57,4	231	19,21	51,95
`vkCreateGraphicsPipelines`	14,6	816	1,39	40,88
`vkCreateDevice`	7,4	1	572,38	572,38

Kompilacja potoków – ciągły proces W przeciwieństwie do Unity, gdzie dominującą funkcją był `vkWaitForFences`, w Unreal Engine **57–72% czasu** Vulkan API pochłonęły funkcje tworzenia potoków.
Co istotne, liczba wywołań `vkCreateComputePipelines` i `vkCreateGraphicsPipelines` jest **niemal identyczna we wszystkich trzech fazach**, co wskazuje na strategię **ciągłej rekompilacji potoków** (Pipeline State Object) przez cały czas działania gry.

Łącznie w każdej 30-sekundowej fazie tworzonych jest około **1 024–1 047 potoków** (231 compute + 793–816 graphics). Porównując z Unity (który utworzył tylko 3 potoki graficzne w całym 95-sekundowym teście), Unreal Engine generuje **ponad 300 razy więcej potoków**.

Średni czas tworzenia potoku compute (18,63–19,21 ms) jest ponad **14 razy dłuższy** niż dla potoku graficznego (1,14–1,39 ms). Różnica ta wynika z większej złożoności shaderów obliczeniowych używanych
przez Unreal Engine do culling, post-processingu i systemu Nanite.

Wywołanie `vkCreateDevice` pojawia się raz w każdej fazie z czasem 541–590 ms, co odpowiada momentowi startu gry w tej fazie – narzędzie Nsight Systems tworzy nową sesję dla każdej fazy.

Tabela 8.16. Wywołania Vulkan API silnika Unreal Engine – synchronizacja i prezentacja (faza 2)

Funkcja	Czas (%)	Wywołania	Śr. (μs)	Maks. (ms)
`vkQueuePresentKHR`	9,5	11 531	77,05	0,90
`vkQueueSubmit`	7,8	186 589	3,92	1,64
`vkWaitForFences`	0,5	11 627	3,63	2,61
`vkAcquireNextImageKHR`	0,1	11 531	0,89	7,55

Synchronizacja – minimalne oczekiwanie na GPU W ostrzym kontraście z Unity (gdzie `vkWaitForFences` stanowił 95,2% czasu), w Unreal Engine funkcja ta pochłonęła zaledwie **0,5% czasu** ze średnim czasem oczekiwania 3,63 μs. Tak niski czas oczekiwania wskazuje na:

- Efektywne wykorzystanie wielokrotnego buforowania (triple buffering)
- Asynchroniczne przesyłanie pracy do GPU bez blokowania
- Lepsze rozłożenie pracy między CPU a GPU eliminujące przestoje

Stosunek wywołań `vkQueueSubmit` (186 589) do `vkQueuePresentKHR` (11 531) wynosi **16,2:1**, co oznacza średnio 16 przesyłek pracy na klatkę. Jest to znacznie więcej niż w Unity (2:1), odzwierciedlając bardziej złożony potok renderowania Unreal Engine z wieloma przebiegami (deferred rendering, post-processing, UI).

Tabela 8.17. Wywołania Vulkan API silnika Unreal Engine – bufory poleceń (wszystkie fazy łącznie)

Funkcja	Wywołania	Śr. (μs)	Maks. (μs)
`vkCmdBindPipeline`	5 771 642	0,24	2 722
`vkCmdPipelineBarrier2KHR`	4 090 071	0,28	942
`vkBeginCommandBuffer`	427 903	1,15	902
`vkEndCommandBuffer`	427 900	0,78	228

Bufory poleceń – intensywna zmiana stanów Liczba wywołań `vkCmdBindPipeline` (**5 771 642** łącznie we wszystkich fazach) jest ponad **213 razy większa** niż w Unity (27 027), co odpowiada około 218 zmianom potoku na klatkę. Tak wysoka wartość wynika z:

- Dynamicznego systemu materiałów Unreal Engine
- Wielu wariantów shaderów dla różnych kombinacji oświetlenia
- Złożonego potoku renderowania z wieloma przebiegami

Funkcja `vkCmdPipelineBarrier2KHR` (4 090 071 wywołań) synchronizuje dostęp do zasobów w obrębie GPU – wysoka liczba wywołań wskazuje na staranną kontrolę zależności między operacjami, typową dla nowoczesnych technik renderowania wykorzystujących wiele render targets.

Ray tracing – przygotowanie struktur akceleracji Interesującą obserwacją jest obecność wywołań związanych z ray tracingiem we wszystkich fazach:

- `vkCreateAccelerationStructureKHR`: 23 960 + 26 275 + 11 884 = 62 119 wywołań
- `vkDestroyAccelerationStructureKHR`: 20 571 + 23 063 + 9 181 = 52 815 wywołań
- `vkGetAccelerationStructureBuildSizesKHR`: 41 161 + 46 147 + 18 379 = 105 687 wywołań

Pomimo że testowana gra nie wykorzystuje widocznych efektów ray tracingu, Unreal Engine przygotowuje struktury akceleracji BVH (Bounding Volume Hierarchy), prawdopodobnie do potencjalnego użycia w globalnym oświetleniu lub śledzeniu promieni. Nierówna liczba utworzeń i zniszczeń sugeruje akumulację struktur w pamięci GPU podczas rozgrywki.

8.3.4. Analiza wywołań systemowych Unreal Engine

Podobnie jak dla Unity, Nsight Systems przechwycił wywołania funkcji systemowych we wszystkich trzech fazach, umożliwiając analizę zachowania wielowątkowego Unreal Engine. Łącznie zarejestrowano ponad **9 milionów wywołań** funkcji synchronizacji.

Tabela 8.18. Wywołania systemowe silnika Unreal Engine – synchronizacja wątków (wszystkie fazy)

Funkcja	Czas (%)	Wywołania	Śr. (ms)	Maks. (s)
`pthread_cond_wait`	64,6	3 095 188	0,97	22,23
`pthread_cond_timedwait`	19,2	163 783	5,46	2,00
`poll`	7,2	215 851	1,56	0,10
`usleep`	4,7	26 062	7,79	0,01
`select`	2,4	1 039	99,72	0,10
`nanosleep`	0,6	755	35,55	0,20

pthread_cond_wait – architektura TaskGraph Funkcja `pthread_cond_wait` pochłonęła **64,6% czasu** przy **3 095 188 wywołaniach** we wszystkich trzech fazach. Jest to funkcja POSIX do oczekiwania na zmienną warunkową, używana gdy wątek musi czekać na spełnienie określonego warunku sygnalizowanego przez inny wątek.

Tak wysoka liczba wywołań (ponad 40 razy więcej niż dla Unity) odzwierciedla architekturę wielowątkową Unreal Engine opartą na systemie **TaskGraph**. System ten dekomponuje pracę renderowania na małe zadania (ang. *tasks*), które są

wykonywane przez pulę wątków roboczych. Każde zadanie po zakończeniu sygnalizuje swoją gotowość, a zależne zadania są budzone poprzez `pthread_cond_signal`/`pthread_cond_broadcast`.

Średni czas pojedynczego oczekiwania (0,97 ms) jest krótki, co wskazuje na częste, ale krótkotrwałe synchronizacje – typowe dla drobnoziarnistego paralelizmu. Maksymalny czas 22,23 sekundy odpowiada prawdopodobnie wywołaniu podczas długotrwałej operacji inicjalizacyjnej w fazie 2.

Tabela 8.19. Porównanie wywołań synchronizacyjnych między fazami Unreal Engine

Metryka	**Faza 1**	**Faza 2**	**Faza 3**
`pthread_cond_wait` wywołań	1 166 913	1 253 746	674 529
`pthread_cond_wait` czas (%)	63,2	65,1	66,4
`pthread_cond_timedwait` wywołań	68 267	63 863	31 653
`pthread_cond_broadcast` wywołań	668 650	747 301	337 258
`backtrace` wywołań	2 306 885	2 289 546	988 685

Tabela 8.19 pokazuje konsystencję wzorców wywołań między fazami 1 i 2 (aktywna rozgrywka) oraz wyraźny spadek w fazie 3 (zawierającej ekran zwycięstwa). Szczególnie interesująca jest wysoka liczba wywołań `backtrace` (ponad 5,5 miliona łącznie), co sugeruje intensywne wykorzystanie mechanizmów debugowania lub profilowania wbudowanych w Unreal Engine nawet w konfiguracji DebugGame.

synchronizacja z limitem czasowym pthread_cond_timedwait (19,2%, 163 783 wywołań) różni się od `pthread_cond_wait` możliwością określenia maksymalnego czasu oczekiwania. Użycie tej funkcji wskazuje na mechanizmy:

- Timeoutów zapobiegających zakleszczeniom (deadlock prevention)
- Okresowego sprawdzania warunków (polling pattern)
- Synchronizacji czasowej dla frame pacing

Średni czas 5,46 ms sugeruje użycie do synchronizacji między-klatkowej, gdzie wątki oczekują na gotowość kolejnej klatki z timeout'em zapobiegającym nieskończonemu oczekiwaniu w przypadku błędu.

usleep – precyzyjne opóźnienia Funkcja `usleep` (4,7%, 26 062 wywołań, średnio 7,79 ms) wprowadza precyzyjne opóźnienia czasowe. Średni czas 7,79 ms jest zbliżony do czasu klatki przy 128 FPS, co może sugerować mechanizm regulacji tempa renderowania lub oszczędzanie energii poprzez redukcję spin-waitingu.

Tabela 8.20. Porównanie mechanizmów synchronizacji Unity i Unreal Engine (zaktualizowane)

Metryka	**Unity**	**Unreal Engine**
Dominujący mechanizm	`futex` (95,9%)	`pthread_cond_wait` (64,6%)
Liczba wywołań synchronizacji	247	3 095 188
Średni czas wywołania	444,07 ms	0,97 ms
Utworzone wątki	81	∼83
Liczba próbek GPU (10 kHz)	–	1 050 555

Porównanie modeli synchronizacji Tabela 8.20 ujawnia fundamentalną różnicę architektoniczną między silnikami:

Unity stosuje mechanizm `futex` z niewielką liczbą wywołań (247) i długim średnim czasem (444 ms). Wskazuje to na architekturę z większymi, bardziej autonomicznymi jednostkami pracy i rzadszą synchronizacją między wątkami.

Unreal Engine używa `pthread_cond_wait` z ogromną liczbą wywołań (ponad 3 miliony w 90-sekundowym teście) i bardzo krótkim średnim czasem (0,97 ms). Odzwierciedla to drobnoziarnisty paralelizm systemu TaskGraph, gdzie praca jest dzielona na małe zadania często komunikujące się ze sobą.

Różnica ta ma implikacje praktyczne:

- **Skalowalność**: Drobnoziarnisty model Unreal lepiej skaluje się na procesory z wieloma rdzeniami
- **Narzut synchronizacji**: Model Unity ma mniejszy narzut z powodu rzadszych wywołań
- **Responsywność**: Unreal może szybciej reagować na zmiany (np. przerwanie zadania)
- **Debugowanie**: Model Unity jest łatwiejszy do analizy ze względu na prostszą strukturę

8.3.5. Charakterystyka architektury Unreal Engine na podstawie profilowania

Zebrane dane z trzech faz profilowania pozwalają na charakterystykę architektonicznych aspektów silnika Unreal Engine:

Model wielowątkowości Unreal Engine 5 stosuje zaawansowaną architekturę wielowątkową złożoną z:

- **Game Thread** – główny wątek logiki gry
- **Render Thread** – wątek przygotowujący polecenia renderowania
- **RHI Thread** (Render Hardware Interface) – wątek komunikujący się z API graficznym
- **Worker Threads** – pula wątków roboczych systemu TaskGraph

Obserwowana dominacja `pthread_cond_wait` (3+ miliony wywołań) potwierdza intensywną komunikację między tymi wątkami.
Wysokie wykorzystanie GPU (90,98% w fazach aktywnej rozgrywki) przy jednoczesnej intensywnej synchronizacji CPU sugeruje efektywne wykorzystanie zasobów obu procesorów.

Profil obciążenia GPU Na podstawie zebranych metryk można scharakteryzować profil obciążenia GPU:

- **Charakter pracy**: Mieszany graficzno-obliczeniowy (GR Active 85,59%, Sync Compute 43,23%)
- **Wykorzystanie SM**: Umiarkowane (42,88%), wskazujące na potencjał optymalizacji
- **Przepustowość pamięci**: Niewysoka (10,30% odczyt, 10,10% zapis), nie jest wąskim gardłem
- **Transfer PCIe**: Niski (1,50% RX), dane pozostają w pamięci GPU
- **Async Copy Engine**: Aktywny w 24–25% czasu, wskazując na efektywne wykorzystanie asynchronicznych transferów

Stabilność między fazami Porównanie faz 1 i 2 (tabela 8.10) pokazuje niezwykłą stabilność metryk GPU:

- GPU Active: różnica 0,36 pp. (91,16% vs 90,80%)
- GR Active: różnica 0,21 pp. (85,69% vs 85,48%)
- SMs Active: różnica 0,18 pp. (42,79% vs 42,97%)

Ta konsystencja potwierdza poprawność metodologii fazowego profilowania i sugeruje deterministyczne zachowanie silnika renderującego niezależnie od poziomu trudności gry.

Podsumowanie architektury Unreal Engine Dzięki profilowaniu fazowemu uzyskano kompletne dane śledzenia Vulkan API i metryk GPU dla całej 90-sekundowej rozgrywki. Zebrane dane (ponad 32 miliony zdarzeń Vulkan API, ponad milion próbek GPU i ponad 9 milionów wywołań systemowych) dostarczają kompleksowego wglądu w charakterystykę wydajnościową silnika, umożliwiając bezpośrednie porównanie z Unity.

8.4. Analiza porównawcza

8.4.1. Porównanie czasu klatki

Tabela 8.21. Porównanie czasów klatek i wydajności między silnikami

Metryka	Unity	Unreal Engine
Średni FPS (fazy 1–2)	164 (V-Sync)	332–339
Średni FPS (faza 3, wymagająca)	164 (V-Sync)	162
Całkowita liczba klatek (90s)	14 765	24 975

Tabela 8.21 przedstawia porównanie wydajności obu silników.

Kluczową obserwacją jest to, że **Unity działał z włączonym V-Sync** na monitorze 165 Hz, co ograniczało wydajność do około 164 FPS niezależnie od obciążenia sceny. Unreal Engine działał bez V-Sync, osiągając 332–339 FPS w fazach 1–2, jednak **w fazie 3 (najbardziej wymagającej) wydajność spadła do 162 FPS** – wartości zbliżonej do Unity.

Ten wynik sugeruje, że przy wysokim obciążeniu sceny (maksymalna liczba przeciwników i pocisków) oba silniki osiągają porównywalną wydajność, natomiast Unreal Engine jest w stanie wykorzystać zapas mocy obliczeniowej GPU przy mniejszym obciążeniu.

8.4.2. Porównanie wykorzystania GPU

Tabela 8.22. Porównanie wykorzystania GPU między silnikami

Metryka	Unity	Unreal Engine
Dominująca funkcja Vulkan	vkWaitForFences (95,2%)	vkCreateComputePipelines (47–57%)
Charakter ograniczenia	GPU-bound	Pipeline compilation
vkQueueSubmit / klatkę	2	16,2
vkCmdBindPipeline / klatkę	2	218

Analiza wywołań Vulkan API ujawnia fundamentalnie różne profile obciążenia (tabela 8.22):

Unity – scenariusz GPU-bound Dominacja `vkWaitForFences` (95,2% czasu) wskazuje, że CPU efektywnie przygotowuje pracę i oczekuje na GPU. Jest to pożądany wzorzec w aplikacjach graficznych, gdzie GPU wykonuje większość obliczeń.
Niski stosunek `vkQueueSubmit`/klatkę (2:1) świadczy o prostym, dwuetapowym potoku renderowania.

Unreal Engine – kompilacja potoków jako wąskie gardło W Unreal Engine dominującymi operacjami były `vkCreateComputePipelines`

oraz `vkCreateGraphicsPipelines`, pochłaniające łącznie 57–72% czasu Vulkan. Silnik tworzy około **1000 potoków w każdej 30-sekundowej fazie** (vs 3 potoki w całym teście Unity), co wskazuje na strategię dynamicznej kompilacji shaderów.

Wysoki stosunek `vkCmdBindPipeline`/klatkę (218:1 vs 2:1) odzwierciedla złożony system materiałów Unreal z wieloma wariantami shaderów, co wprowadza znaczący narzut zmian stanu GPU.

8.4.3. Porównanie architektury wielowątkowej

Tabela 8.23. Porównanie mechanizmów synchronizacji między silnikami

Metryka	Unity	Unreal Engine
Główny mechanizm synchronizacji	futex	pthread_cond_wait
Liczba wywołań synchronizacji	247	3 095 188
Średni czas wywołania	444 ms	0,97 ms
Model paralelizmu	Gruboziarnisty	Drobnoziarnisty

Tabela 8.23 ujawnia fundamentalną różnicę architektoniczną:

Unity stosuje model gruboziarnistego paralelizmu z rzadkimi, ale długimi synchronizacjami. Wątki wykonują większe jednostki pracy autonomicznie, co minimalizuje narzut komunikacji.

Unreal Engine implementuje drobnoziarnisty paralelizm poprzez system TaskGraph. Praca jest dzielona na tysiące małych zadań często komunikujących się ze sobą (ponad 3 miliony wywołań synchronizacji w 90 sekund).

8.5. Podsumowanie wyników testów wydajności

Tabela 8.24. Zestawienie kluczowych wyników testów wydajności

Metryka	Unity	Unreal Engine
Średni FPS (fazy 1–2)	164 (V-Sync)	332–339
FPS w wymagającej scenie	132 (1% low)	162 (faza 3)
GPU Active (%)	23	91 (fazy 1–2), 50 (faza 3)
Dominujące wąskie gardło	GPU (rendering)	CPU (kompilacja potoków)
Wywołania Vulkan API	∼0,5 mln	∼32 mln
Wywołania synchronizacji OS	29 383	∼9 mln
Potoki graficzne utworzone	3	∼2 400

Przeprowadzone testy wydajnościowe pozwalają na sformułowanie następujących wniosków:

1. **Wydajność w wymagających scenach**: W fazie 3 (maksymalne obciążenie) oba silniki osiągają zbliżoną wydajność: Unity 1% low 132 FPS vs Unreal 162 FPS.

Różnica około 23% na korzyść Unreal wynika częściowo z różnych konfiguracji V-Sync.

2. **Wykorzystanie GPU**: Unity wykorzystuje jedynie 23% mocy GPU (ograniczony V-Sync), podczas gdy Unreal Engine osiąga 91% wykorzystania w fazach 1–2. Sugeruje to znaczny zapas wydajności Unity przy wyłączonym V-Sync.
3. **Stabilność**: Unity wykazał stabilne czasy klatek dzięki V-Sync, natomiast Unreal Engine pokazał dużą zmienność między fazami (332–339 FPS w fazach 1–2 vs 162 FPS w fazie 3) – spadek o ponad 50%.
4. **Architektura**: Silniki stosują fundamentalnie różne podejścia do wielowątkowości i zarządzania potokami renderowania. Unity używa gruboziarnistego paralelizmu z rzadkimi synchronizacjami, podczas gdy Unreal stosuje drobnoziarnisty system TaskGraph z milionami wywołań synchronizacyjnych.
5. **Narzut Unreal**: Dynamiczna kompilacja potoków (ponad 1000 potoków na 30-sekundową fazę vs 3 w całym teście Unity) i 60-krotnie większa liczba wywołań Vulkan API stanowią znaczący narzut, który może przyczyniać się do spadków wydajności w wymagających scenach.

Bibliografia

[1] J. Glau, „Bullet Hell Games: A Study", *Game Studies Journal*, t. 15, nr. 3, s. 45–67, 2021.

[2] J. Gregory, *Game Engine Architecture*, 3rd. A K Peters/CRC Press, 2018, ISBN: 978-1138035454.

[3] G. C. Ullmann, C. Politowski, Y.-G. Guéhéneuc i N. Anquetil, „Game engine comparative anatomy", *International Conference on Software Architecture*, s. 117–136, 2022.

[4] E. Christopoulou i S. Xinogalos, „Overview and comparative analysis of game engines for desktop and mobile devices", *International Journal of Serious Games*, t. 4, nr. 4, s. 21–36, 2017.

[5] K. H. Sharif i S. Y. Ameen, „Game engines evaluation for serious game development in education", w *2021 International Conference on Advanced Computer Applications*, IEEE, 2021, s. 1–6.

[6] S. Pavkov, I. Franković i M. Hoblaj, „Comparison of game engines for serious games", w *2017 40th International Convention on Information and Communication Technology*, IEEE, 2017, s. 728–733.

[7] F. Messaoudi, A. Ksentini i G. Simon, „Performance analysis of game engines on mobile and fixed devices", *ACM Transactions on Multimedia Computing, Communications, and Applications*, t. 13, nr. 4, s. 1–24, 2017.

[8] K. Abramowicz i P. Borczuk, „Comparative analysis of the performance of Unity and Unreal Engine game engines in 3D games", *Journal of Computer Science Institute*, t. 30, s. 1–8, 2024.

[9] A. Pattrasitidecha, „Comparison and evaluation of 3D mobile game engines", Master's thesis, Chalmers University of Technology, 2014.

[10] C. Vohera, H. Chheda i D. Chouhan, „Game engine architecture and comparative study of different game engines", w *2021 12th International Conference on Computing Communication and Networking Technologies*, IEEE, 2021, s. 1–7.

[11] S. Marks, J. Windsor i B. Wünsche, „Evaluation of game engines for simulated clinical training", w *New Zealand Computer Science Research Student Conference*, 2008, s. 25–30.

[12] Z. Ali i M. Usman, „A framework for game engine selection for gamification and serious games", w *2016 Future Technologies Conference*, IEEE, 2016, s. 1–8.

[13] A. Barczak i H. Woźniak, „Comparative study on game engines", *Studia Informatica. System and Information Technology*, t. 23, nr. 1, s. 5–18, 2019.

[14] Z. Masood, Z. Jiangbin, M. Irfan i I. Ahmad, „High‐performance virtual globe GPU terrain rendering using game engine", *Computer Animation and Virtual Worlds*, t. 33, nr. 6, e2108, 2022.

[15] H. B. Fırat, L. Maffei i M. Masullo, „3D sound spatialization with game engines: the virtual acoustics performance of a game engine and a middleware for interactive audio design", *Virtual Reality*, t. 26, nr. 3, s. 1181–1195, 2022.

[16] V. G. Insights, *Game Engine Market Share 2025*, Accessed: 2025-01-25, 2025. adr.: https://vgi2025.com/engines.

[17] G. Crowd, *Game Engine Reviews*, Accessed: 2025-01-25, 2025. adr.: https://g2.com/game-engines.

[18] Wikipedia, *Unity (game engine)*, Dostęp zdalny: https://en.wikipedia.org/wiki/Unity_(game_engine), 2024.

[19] Unity Technologies, *Unity Profiler*, Dostęp zdalny: https://docs.unity3d.com/Manual/Profiler.html, 2024.

[20] Wikipedia contributors, *Unreal Engine — Wikipedia, The Free Encyclopedia*, Dostęp zdalny: https://en.wikipedia.org/wiki/Unreal_Engine, Accessed: 2026-01-24, 2026.

[21] Epic Games, *Unreal Engine Documentation*, Dostęp zdalny: https://docs.unrealengine.com/, 2024.

[22] Epic Games, *Nanite Virtualized Geometry*, Dostęp zdalny: https://docs.unrealengine.com/5.3/en-US/nanite-virtualized-geometry-in-unreal-engine/, 2024.

[23] Epic Games, *Lumen Global Illumination and Reflections*, Dostęp zdalny: https://docs.unrealengine.com/5.3/en-US/lumen-global-illumination-and-reflections-in-unreal-engine/, 2024.

[24] U. Technologies, *Unity Hub*, Accessed: 2025-01-25, 2025. adr.: https://unity.com/hub.

[25] U. Technologies, *Unity Hub Download for Arch Linux*, Accessed: 2025-01-25, 2025. adr.: https://archlinux.org/packages/unity-hub.

[26] Unity, *Linux Unity Editor does not redraw UI after opening menu*, Dostęp zdalny: https://issuetracker.unity3d.com/issues/linux-editor-only-redraws-panels-on-when-theyre-being-moused-over-when-no-compositor-is-present, 2025.

[27] U. Technologies, *Unity MCP*, Accessed: 2025-01-25, 2025. adr.: https://unity.com/mcp.

[28] E. Games, *Unreal Engine Source Installation*, Accessed: 2025-01-25, 2025. adr.: https://github.com/EpicGames/UnrealEngine.

[29] E. Games, *Unreal Engine Binary Installation*, Accessed: 2025-01-25, 2025. adr.: https://unrealengine.com/download.

[30] Epic Games, *Unreal Insights in Unreal Engine*, Dostęp zdalny: https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-insights-in-unreal-engine, 2024.

[31] NVIDIA Corporation, *NVIDIA Nsight Graphics Documentation*, Dostęp zdalny: https://developer.nvidia.com/nsight-graphics, 2024.

Wykaz symboli i skrótów

EiTI – Wydział Elektroniki i Technik Informacyjnych
PW – Politechnika Warszawska

Spis rysunków

1.1 Przykład gry z gatunku bullet hell (seria Touhou). [1] 10
6.1 Ekran powitalny Unity Hub. 32
6.2 Widok edytora Unity. 32
6.3 Wybór projektu w Unreal Engine. 35
6.4 Widok edytora Unreal Engine. 35
7.1 Interfejs Unity Profiler z widokiem analizy wydajności CPU i GPU. . . 39
7.2 Interfejs Unreal Insights z widokiem analizy wydajności. 40
7.3 Interfejs NVIDIA Nsight Graphics z widokiem analizy GPU. 41

Spis tabel

3.1 Porównanie kluczowych cech Unity i Unreal Engine 18
5.1 Rekomendacje wyboru silnika w zależności od kontekstu projektu . . 31
6.1 Porównanie doświadczeń z implementacji gry bullet-hell 37
7.1 Kluczowe metryki wydajnościowe z NVIDIA Nsight 43
8.1 Ogólne metryki wydajności silnika Unity 45
8.2 Rozkład percentylowy czasów klatek silnika Unity 46
8.3 Histogram czasów klatek silnika Unity 46
8.4 Wywołania Vulkan API silnika Unity – funkcje synchronizacji i prezentacji 47
8.5 Wywołania Vulkan API silnika Unity – bufory poleceń 48
8.6 Wywołania Vulkan API silnika Unity – inicjalizacja i zasoby 49
8.7 Wywołania systemowe silnika Unity – synchronizacja wątków 50
8.8 Wywołania systemowe silnika Unity – operacje I/O 51
8.9 Kluczowe metryki wykorzystania GPU dla silnika Unreal Engine (fazy 1–2, aktywna rozgrywka) 54
8.10 Porównanie metryk GPU między fazami testu Unreal Engine 54
8.11 Metryki przepustowości pamięci GPU dla silnika Unreal Engine (fazy 1–2) 55
8.12 Wykorzystanie różnych typów wątków shader GPU w silniku Unreal Engine (fazy 1–2) 55
8.13 Częstotliwości zegara GPU podczas testu Unreal Engine 56

8.14 Porównanie wywołań Vulkan API silnika Unreal Engine między fazami 56
8.15 Wywołania Vulkan API silnika Unreal Engine – tworzenie potoków (wszystkie fazy) 57
8.16 Wywołania Vulkan API silnika Unreal Engine – synchronizacja i prezentacja (faza 2) 58
8.17 Wywołania Vulkan API silnika Unreal Engine – bufory poleceń (wszystkie fazy łącznie) 58
8.18 Wywołania systemowe silnika Unreal Engine – synchronizacja wątków (wszystkie fazy) 59
8.19 Porównanie wywołań synchronizacyjnych między fazami Unreal Engine 60
8.20 Porównanie mechanizmów synchronizacji Unity i Unreal Engine (zaktualizowane) 61
8.21 Porównanie czasów klatek i wydajności między silnikami 63
8.22 Porównanie wykorzystania GPU między silnikami 63
8.23 Porównanie mechanizmów synchronizacji między silnikami 64
8.24 Zestawienie kluczowych wyników testów wydajności 64

Spis załączników