Politechnika Wrocławska

Wydział Informatyki i Zarządzania
kierunek studiów: Informatyka
specjalność: Systemy Mobilne

Praca dyplomowa - magisterska

Metody tworzenia gier wykorzystujących Google Cardboard

Przemysław Drgas

słowa kluczowe:
Google Cardboard,
wirtualna rzeczywistość.
Unity, Unreal Engine, gry, mobilne

krótkie streszczenie:
Analiza procesu tworzenia gier
na platformę Google Cardboard.
Porównanie wydajności silników,
jakości renderowania oraz
złożoności procesu implementacji.

opiekun pracy dyplomowej	Dr inż. Marek Kopel	.....	.....
	Tytuł/stopień naukowy/imię i nazwisko	*ocena*	*podpis*

Ostateczna ocena za pracę dyplomową			
Przewodniczący Komisji egzaminu dyplomowego	.....	.....	.....
	Tytuł/stopień naukowy/imię i nazwisko	*ocena*	*podpis*

*Do celów archiwalnych pracę dyplomową zakwalifikowano do:**
a) kategorii A (akta wieczyste)
b) kategorii BE 50 (po 50 latach podlegające ekspertyzie)
** niepotrzebne skreślić*

pieczątka wydziałowa

Wrocław 2017

Streszczenie

Praca skupia się na procesie tworzenia gier na platformę Google Cardboard. Pierwszą jej częścią jest wprowadzenie teoretyczne dotyczące wirualnej rzeczywistości, w szczególności w obszarze urządzeń mobilnych. Zawiera zarówno przegląd dostępnych konfiguracji sprzętowych, jak i silników graficznych i narzędzi. W części badawczej zaprezentowanych zostało sześć aplikacji wykonanych przy użyciu silników Unity i Unreal Engine (po trzy aplikacje na silnik). Na podstawie danych zebranych podczas testowania opracowano analizy, które mają na celu porównanie wydajności, a także stopnia wykorzystania zasobów urządzenia. Parametry używane w analizie to przede wszystkim ilość wyświetlanych klatek na sekundę (FPS – frames per second). Ta wielkość pokazuje jak płynny jest obraz, co w największym stopniu przekłada się na komfort użytkowania aplikacji opartej o wirtualną rzeczywistość. Pozostałe miary to procentowe użycie procesora oraz ilość rezerwowanej pamięci RAM urządzenia mobilnego. Pozwoliły one określić, który silnik wymaga bardziej wydajnych podzespołów do obsługi aplikacji, a co za tym idzie – nowszej konfiguracji. Ostatnim elementem jest subiektywna ocena warstwy wizualnej aplikacji, zarówno ze strony autora pracy jak i grupy użytkowników. Dotyczy ona przede wszystkim jakości krawędzi, cieniowania, oświetlenia, a także ogólnego odbioru. Wnioski jakie zostały wyciągnięte to w szczególności dużo lepsze dostosowanie silnika Unity do platformy Google Cardboard. Objawia się to między innymi lepszą wydajnością, a zarazem mniejszym zapotrzebowaniem na zasoby. Co więcej rozmiar aplikacji (ważny dla urządzeń mobilnych), a także czasy kompilacji są dużo krótsze niż w przypadku Unreal Engine. Również przygotowanie silnika do pracy z Google Cardboard jest znacznie prostsze dla Unity.

Abstract

This thesis focuses on the process of creating games for Google Cardboard platform. The first part contains theoretical introduction to virtual reality, especially in the area of mobile devices. It includes both the overview of available hardware, graphic engines and tools. In the research part there were six applications presented, developed using both - Unity and Unreal Engine (three aplcations for each). Based on the data collected during testing, the analysis have been performed, which aim to compare performance and the utilization of the device. Parameter used in the analysis is primarily frames per second (fps). This factor shows how smoothly the scene is being displayed, which is most likely connected with the comfort of using a virtual reality application. The remaining measures are mobile device CPU utilization and amount of reserved RAM. They allow to determine which engine requires more efficient components to handle the application, hence - better configuration. The final element is the subjective assessment of the visual layer of the application, both from the author and the user group. It mainly concerns edges quality, shading, lighting, and general reception. The lesson that have been learned is in particular a much better alignment of the Unity engine to the Google Cardboard platform. This includes, among other things, better performance and less demand for resources. What's more, the size of the application (important for mobile devices) and the compilation time are much shorter than in Unreal Engine. Also preparing your engine to work with Google Cardboard is much simpler with Unity engine.

Spis treści

Streszczenie 2
Abstract 2
1. Wstęp 5
2. Wprowadzenie teoretyczne 6
2.1 Wirtualna rzeczywistość 6
2.1.1 Definicja 6
2.1.2 Zastosowania 7
2.1.3 Rynek wirtualnej rzeczywistości 9
2.2 Zestawy wirtualnej rzeczywistości 10
2.2.1 Sony PlayStation VR 10
2.2.2 HTC Vive 11
2.2.3 Oculus Rift 11
2.2.4 Samsung Gear VR 12
2.2.5 Google Cardboard 12
2.3 Silniki gier 14
2.3.1 Definicja, przeznaczenie silników gier 14
2.3.2 Unity 14
2.3.3 Unreal Engine 15
2.3.4 CryEngine 16
2.3.5 Rockstar Advanced Game Engine (RAGE) 17
2.3.6 Cocos2d 17
2.3.7 Clausewitz Engine 18
2.3.8 Podsumowanie 19
2.4 Renderowanie 20
2.4.1 Shadows 20
3. Przygotowanie silników 22
3.1 Wybór silników do analizy i uzasadnienie 22
3.2 Instalacja, konfiguracja narzędzi 23
3.2.1 Android 23
3.2.2 Unity 23
3.2.3 Unreal Engine 24
3.2.4 Podsumowanie 25
4. Aplikacje 25
4.1 Aplikacje nr 1.1 i 1.2 26
4.1.1 Obiekty startowe 27

4.1.2 Implementacja rotacji obiektu ... 27
4.1.3 Implementacja generatora obiektów ... 28
4.1.4 Implementacja licznika klatek na sekundę ... 30
4.1.5 Kompilacja projektu i uruchomienie ... 32
4.2 Aplikacja nr 2 ... 33
4.2.1 Obiekty startowe ... 34
4.2.2 Implementacja rotacji obiektu ... 34
4.2.3 Implementacja licznika klatek na sekundę ... 34
4.2.4 Kompilacja i uruchomienie ... 34
5. Badania ... 35
5.1 Urządzenie testowe ... 35
5.2 Aplikacja nr 1.1 (z cieniowaniem) ... 35
5.2.1 Unity ... 36
5.2.2 Unreal Engine ... 37
5.2.3 Porównanie ... 38
5.3 Aplikacja nr 1.2 (bez cieniowania) ... 40
5.3.1 Unity ... 40
5.3.2 Unreal Engine ... 41
5.3.3 Porównanie ... 42
5.4 Aplikacja nr 2 ... 44
5.4.1 Unity ... 44
5.4.2 Unreal Engine ... 45
5.4.3 Porównanie ... 46
5.5 Wyniki zbiorcze ... 48
5.6 Wnioski ... 49
6. Podsumowanie ... 50
Bibliografia ... 51

1. Wstęp

Wraz z ciągłym, dynamicznym rozwojem technologii mobilnych, a zwłaszcza wydajności samych urządzeń, poszerza się zakres ich zastosowania. Coraz nowsze funkcjonalności systematycznie eliminują konieczność używania komputerów w wielu jego dotychczasowych zastosowaniach. Smartfony zapewniają już nie tylko komunikację i dostęp do multimediów, ale także szereg aplikacji wspomagających bezpieczeństwo, transport, zdrowie, czy proste codzienne czynności. Jednym z najnowszych, a jednocześnie najszybciej rozwijających się zastosowań jest wykorzystanie telefonu jako prostego środka do wejścia w świat wirtualnej rzeczywistości.

Wirtualna rzeczywistość to technologia, której nie można przypisać jednego kierunku w kontekście jej zagospodarowania. Pozwala ona usprawnić, a często rozszerzyć dotychczasowe rozwiązania i to zarówno w zakresie rozrywki, o której słyszymy najczęściej, ale także edukacji, lotnictwie, medycynie, komunikacji, turystyce, sporcie i wielu innych. Ze względu na coraz bardziej realistyczne odwzorowanie rzeczywistości w połączeniu z bardzo naturalnymi sposobami interakcji, możliwe jest precyzyjne symulowanie zjawisk, sytuacji, czy wydarzeń przy jednoczesnym zachowaniu poczucia „egzystowania" w obserwowanym świecie, co bezpośrednio wpływa na stopień emocjonalnego zaangażowania, koncentracji użytkownika.

W tej pracy skupię się na wirtualnej rzeczywistości w grach mobilnych wykonanych pod kątem Google Cardboard przy użyciu silników Unity oraz Unreal Engine 4. Google Cardboard to platforma zapewniająca prosty, szybki i ekonomiczny dostęp do technologii VR, przede wszystkim dzięki bardzo przystępnej cenie gogli, a nawet możliwości samodzielnego ich wykonania. Wymienione silniki to najbardziej popularne, najczęściej wykorzystywane silniki w branży gier, wokół których skupione są szerokie społeczności, zapewniające wsparcie oraz przydatne materiały. Główną motywacją jest ustalenie, który z testowanych silników pozwala stworzyć bardziej wydajną grę na Google Cardboard, w większym stopniu spełniającą wymagania urządzeń mobilnych, a także jak skomplikowany jest sam proces produkcji.

Zakres pracy obejmuje przedstawienie podstaw teoretycznych na temat wirtualnej rzeczywistości, obecnych rozwiązań, silników gier i procesu renderowania grafiki. W dalszej części zaprezentowane zostaną opracowane aplikacje oraz szczegółowa analiza całego procesu ich wykonania, od instalacji środowisk, przez implementację, aż po kompilację. Głównym celem implementacji jest uzyskanie trzech, jak najbardziej zbliżonych do siebie wizualnie gier, a co za tym idzie – porównywalnych w wielu aspektach. Następnie opisane zostaną testy wydajnościowe i wizualne. Zebrane rezultaty będą uwzględniały również ogólny odbiór aplikacji przez użytkowników. Tak obszerna analiza pozwoli zebrać szereg informacji pozwalających z dużą pewnością odpowiedzieć na podstawowe pytanie – który silnik lepiej sprawdza się w przypadku gier na platformę Google Cardboard?

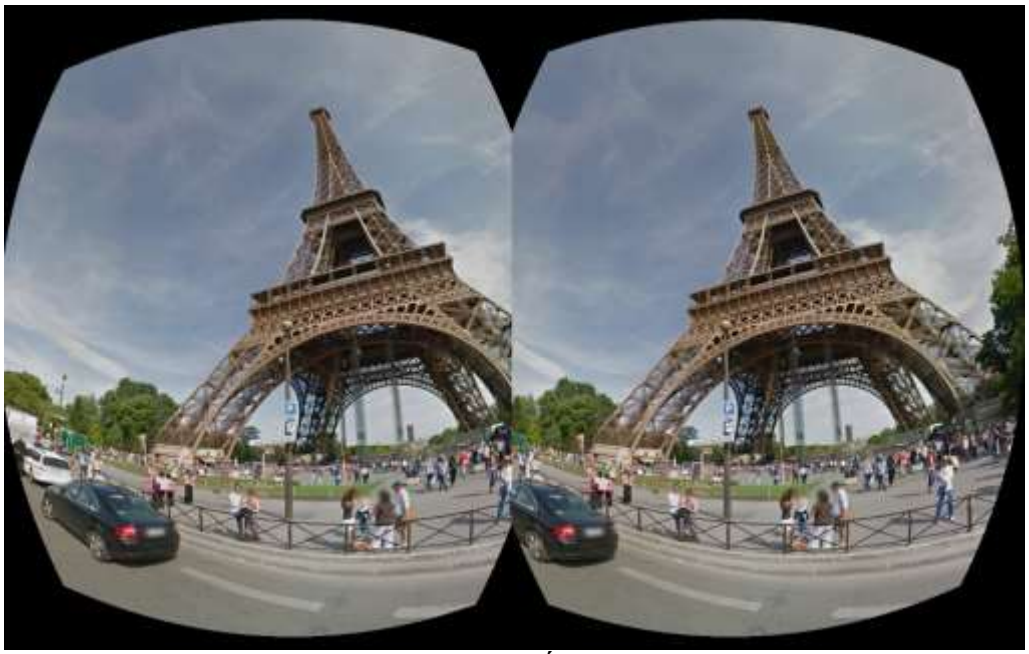
2. Wprowadzenie teoretyczne

W tym rozdziale przedstawione zostaną podstawowe informacje niezbędne do zrozumienia dalszej części pracy. Będzie to przede wszystkim definicja samej wirtualnej rzeczywistości, przegląd dostępnych rozwiązań sprzętowych, silników gier, jak również systemów i urządzeń mobilnych. Bardzo ważnym elementem będzie prezentacja platformy Google Cardboard – jej zalet, ograniczeń i wymagań.

2.1 Wirtualna rzeczywistość

2.1.1 Definicja

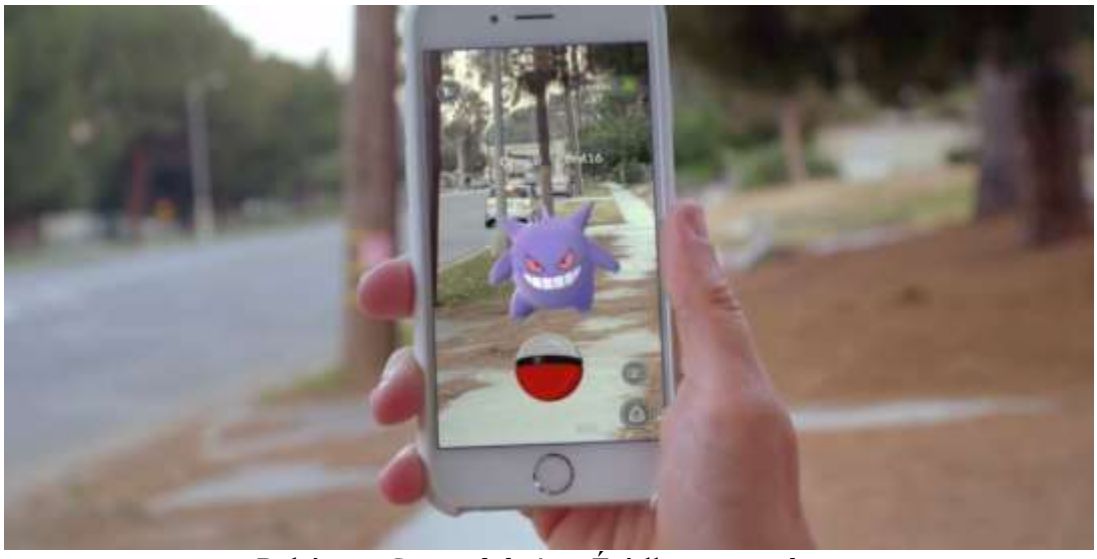
Wirtualna rzeczywistość to w najprostszym ujęciu „sposób użycia technologii komputerowej w tworzeniu efektu interaktywnego, trójwymiarowego świata, w którym obiekty dają wrażenie przestrzennej obecności.” [1]. Aktualnie na taki efekt składa się przeważnie połączenie obrazu i dźwięku, chociaż pojawiają się rozwiązania oferujące również efekty zapachowe, smakowe czy dotykowe. Poza samym zestawem odpowiadającym za wyświetlanie obrazu i dostarczanie dźwięku, często możliwe jest również użycie zewnętrznych kontrolerów, pomagających w sterowaniu ruchem tj. klawiatura, mysz, gamepady, joysticki, piloty, czujniki i inne [2]. Podstawowa funkcjonalność wyróżniająca wirtualną rzeczywistość to bezpośrednie przełożenie ruchów głowy użytkownika na ruch obrazu zestawu VR (Virtual Reality), co powoduje wrażenie bycia częścią wyświetlanej sceny – istoty VR. Należy również zwrócić uwagę na rzeczywiste wymiary obserwowanych obiektów (Rys. 2.1), a także coraz większe możliwości interakcji z nimi, dodatkowo potęgujące efekt.



Rys. 2.1 Widok VR. Źródło: stv.re

Inną formą omawianej technologii, niejako jej pochodną, jest rozszerzona rzeczywistość – AR (Augmented Reality). Jest to połączenie rzeczywistego obrazu, wyświetlanego na żywo np. poprzez kamerę z obrazem generowanym komputerowo (Rys. 2.2). Mogą to być zarówno informacje w formie tekstu nakładane na warstwę rzeczywistą, jak również obrazy

2D/3D będące w jakiś sposób powiązane z lokalizacją, działaniem, czy funkcją użytkownika.



Rys. 2.2 Pokémon Go, widok AR. Źródło: venturebeat.com

Sceny wykonywane do użycia w wirtualnej rzeczywistości mogą być projektowane w pełni komputerowo jako połączenie obiektów 2D i 3D, a także poprzez nagrywanie materiałów wideo za pomocą specjalnych kamer rejestrujących obraz w 360 stopniach.

2.1.2 Zastosowania

Zastosowania VR są dużo szersze niż szeroko pojęty świat rozrywki. Oczywiście najbardziej popularnymi i najbardziej znanymi są gry komputerowe i filmy, jednak technologia ta pozwala na bardzo poważne usprawnienia jeśli chodzi o bardziej specjalistyczne i ważniejsze z punktu widzenia życia codziennego obszary.
Przykładem mogą być generalnie różnego rodzaju szkolenia. Ćwiczenie operacji chirurgicznych może odbywać się w bardzo realistycznych warunkach bez ponoszenia ryzyka, czy nadmiernych kosztów, a do tego pozwala precyzyjnie ocenić i przeanalizować każdy element szkolenia (Rys. 2.3) [3] [4]. Podobne rozwiązania stosowane są w innych dziedzinach medycyny, na przykład w stomatologii [5].

Rys. 2.3 VR w szkoleniu chirurgów. Źródło:fundamentalvr.com

Symulacja lotów treningowych dla pilotów w bardzo drastyczny sposób zmniejsza finansowe nakłady na ich realizację i w rzeczywistości pozwala na duży szerszy zakres wykonywanych zadań, ponieważ możliwe jest odtworzenie praktycznie każdego scenariusza (Rys. 2.4). VR jest stosowane także w przypadku szkoleń inspektorów lotniczych, żołnierzy jednostek lądowych, czołgistów, astronautów, spadochroniarzy, a coraz częściej również operatorów dźwigów i innych maszyn, czy pojazdów [6] [7] [8].

Rys. 2.4 VR w symulatorze lotu DCS. Źródło:youtube.com

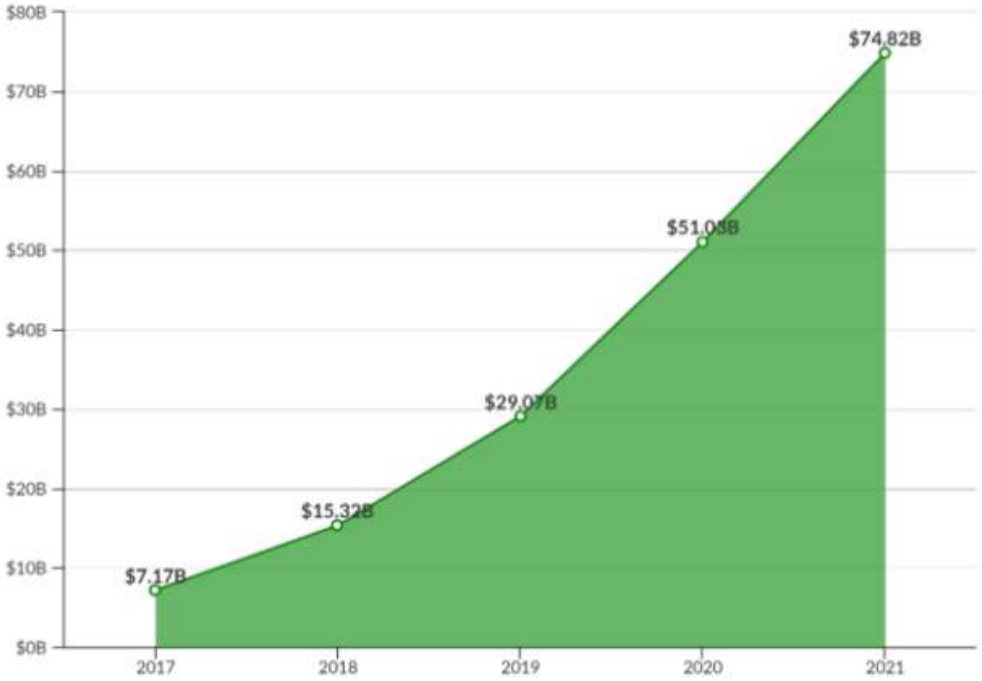
Poza zastosowaniami szkoleniowymi VR sprawdza się również jako element strategii marketingowych. Wizualizacje produktów w skali 1:1 pozwalają lepiej przyjrzeć się szczegółom, zauważyć aspekty trudno dostrzegalne na zdjęciach i w większym stopniu oddziałują na doznania potencjalnego klienta [9]. Filmy prezentowane w formie spacerów po parkach rozrywki, ciekawych obszarach miast, obiektach sportowych, galeriach sztuki, czy nawet biurach zachęcają do odwiedzenia tych miejsc. W podobny sposób można reklamować praktycznie każdy produkt i usługę. Film 360 stopni ukazujący konflikt w Syrii został użyty w celu zwrócenia uwagi na to wydarzenie oraz zbiórkę środku na ofiary [10].
W muzyce pojawiają się teledyski kompatybilne z wirtualną rzeczywistością, a także koncerty które można oglądać za pośrednictwem urządzeń VR. Podobne trendy dotykają świat filmu, który po porażce technologii opartej na okularach 3D skłania się ku nowym rozwiązaniom.
Architektura stosuje VR do wizualizowania swoich projektów, co pozwala także na odtwarzanie obiektów historycznych i stosowanie ich w placówkach kulturowych, turystyce.
Wirtualna rzeczywistość znajduje zastosowanie również w psychologii. Pozwala na oswojenie się z różnymi zjawiskami powodującymi lęki, wspomagają terapeutów w odtwarzaniu różnych wydarzeń i całym procesie leczenia [7] [11] [12].
Wraz z powstaniem VR powstał nowy kierunek w sztuce [9] skupiający się na tworzeniu instalacji umożliwiających interakcję i bardziej intensywne doświadczanie sztuki (Rys. 2.5). Również niektóre muzea zaczęły prezentować swoje zasoby za pośrednictwem tej technologii, również dostępnych w Internecie.

Rys. 2.5 instalacja w East Galley, NUA. Źródło:nua.ac.uk/thegallery/

Jak widać, wirtualna rzeczywistość jest wykorzystywana w bardzo wielu dziedzinach i usprawnia w zauważalny sposób wiele procesów dotychczas przeprowadzanych w znacznie bardziej kosztowny lub mniej efektywny sposób. Jeśli natomiast chodzi o aspekty rozrywkowe to sam fakt poczucia fizycznego uczestnictwa w wirtualnym świecie jest bardzo dużą zaletą tej technologii, ponieważ emocje towarzyszące rozgrywce są o wiele bardziej intensywne i realne, niż w przypadku tradycyjnych form. Oczywiście opisane zastosowania są jedynie ogólnym spojrzeniem na temat, ze względu na fakt, iż VR może znaleźć swoje miejsce w jeszcze wielu innych obszarach, a z czasem na pewno będą się pojawiały rozwiązania idące o wiele dalej w swoim zaawansowaniu, aniżeli obecnie.

2.1.3 Rynek wirtualnej rzeczywistości

Ważnym aspektem VR jest rynek związany z tą technologię, ponieważ jej szybki wzrost gwarantuje także ciągły rozwój. Dochody generowane przez wirtualną rzeczywistość sukcesywnie rosną (Rys. 2.6). W 2016 roku było to 5,2 mld $, podczas gdy na rok 2017 prognozowany jest wzrost o około 37% do 7,17 mld $. W kolejnych latach wzrost ma być jeszcze bardziej dynamiczny i z końcem 2021 roku dochód ma osiągnąć aż 74,2 mld $ [13]. Inne źródła podają, że zysk ten może osiągnąć nawet 162 mld $ i to już w 2020 roku [14]. Większość dochodów generuje sprzedaż sprzętu VR, jednak w przyszłości coraz znaczniejsza część będzie rezultatem sprzedaży zawartości.
Prognozy jednoznacznie wskazują na bardzo szybki wzrost całej branży, a co za tym idzie, ilości usług adaptujących technologię wirtualnej rzeczywistości. Będzie ona obiektem coraz większego zainteresowania, jak również wykorzystywana na coraz to nowe sposoby.



Rys. 2.6 Prognozowany dochód w obszarze VR w latach 2017-2021. Źródło:variety.com

2.2 Zestawy wirtualnej rzeczywistości

Na rynku można znaleźć bardzo wiele rozwiązań jeśli chodzi o zestawy VR. Można je podzielić na dwie kategorie: wymagające połączenia z komputerem lub konsolą i mobilne, wykorzystujące smartfony. Chciałbym się skupić na obu tych kategoriach, aby ukazać ogólny przekrój dostępnych zestawów. Oczywiście opiszę tylko najbardziej popularne, najlepsze z nich, ponieważ ze względu na spore ilości wciąż pojawiającego się sprzętu nie sposób przedstawić wszystkie.

2.2.1 Sony PlayStation VR

Zestaw wirtualnej rzeczywistości od Sony wymaga oczywiście konsoli PlayStation (Rys. 2.7). Obsługuje nie tylko gry dostosowane do VR, ale pozwala także na tryb, w którym wyświetlany jest „wirtualny monitor”, przez co użytkownik może używać gogli do wszystkich funkcji konsoli. Pole widzenia to 100 stopni, częstotliwość odświeżania – 120Hz, rozdzielczość – 960 x 2160px. Jedynym wymaganiem sprzętowym jest posiadanie PlayStation 4, sterowanie obsługiwane jest przy pomocy kontrolerów PlayStation Move i kamery PlayStation Camera.

Rys. 2.7 Zestaw PlayStation VR. Źródło:playstation.com

2.2.2 HTC Vive

HTC Vive poza reagowaniem na ruchy głową, przy zastosowaniu dodatkowych czujników (stacji bazowych) wykrywa także przemieszczanie się użytkownika (w polu ok. 4,5m x 4,5m), co wyróżnia go spośród innych zestawów (Rys. 2.8). Jego pole widzenia to 110 stopni, częstotliwość odświeżania – 90Hz, rozdzielczość 1080 x 2400px. Sterowanie ułatwiają dwa kontrolery ruchu (po jednym dla każdej ręki) wyposażone w joystick oraz przyciski. Wymaga podłączenia do komputera PC. Rekomendowana specyfikacja to procesor Intel Core i5, 4GB RAM, HDMI 1.4, jeden port USB 2.0 lub nowszy, 64 bitowy system operacyjny Windows 7 lub nowszy oraz kartę graficzną GeForce GTX 1060.

Rys. 2.8 Zestaw HTC Vive. Źródło:roadtovr.com

2.2.3 Oculus Rift

Oculus Rift to gogle stworzone przez Oculus VR, obecnie część Facebook'a (Rys. 2.9). Pierwsze gogle zostały sfinansowane jako projekt na portalu Kickstarter. Wymagają podłączenia do komputera PC. Pole widzenia to 110 stopni, częstotliwość odświeżania – 90Hz, rozdzielczość 1080x2400px. Sterowanie odbywa się za pośrednictwem dwóch kontrolerów ruchu (po jednym dla każdej ręki) wyposażonych w joystick oraz przyciski. Gogle posiadają także zintegrowane słuchawki. Rekomendowane wymagania sprzętowe to procesor Intel Core

i3, 16GB RAM, HDMI 1.3, dwa porty USB 3.0, jeden USB 2.0, 64 bitowy system operacyjny Windows 8 lub nowszy oraz kartę graficzną GeForce GTX 960.

Rys. 2.9 Zestaw Oculus Rift. Źródło:oculus.com

2.2.4 Samsung Gear VR

Rozwiązanie Samsunga współpracujące tylko ze smartfonami tego producenta, aktualnie od modelu Galaxy S6 do S8 (łącznie 8). Gogle jako ekran wykorzystują wyświetlacz smartfona, natomiast same funkcjonują jako kontroler, sterując polem widzenia za pomocą akcelerometru, żyroskopu oraz czujnika zbliżeniowego (Rys. 2.10). Na goglach umieszczony jest touchpad i przyciski wspomagające sterowanie, dostępny jest również zewnętrzny kontroler ułatwiający poruszanie się. Ze względu na użycie telefonu zamiast komputera, czy konsoli, wydajność zależy bezpośrednio od podzespołów posiadanego urządzenia. Pole widzenia w Gear VR to 101 stopni, częstotliwość odświeżania zależy od parametrów telefonu.

Rys. 2.10 Zestaw Samsung Gear VR. Źródło:samsung.com

2.2.5 Google Cardboard

Google Cardboard to bardzo oryginalne rozwiązanie bazujące na goglach możliwych do samodzielnego wykonania lub zakupienia (Rys. 2.11). Obie opcje są bardzo przystępne cenowo, ponieważ gogle są wykonane z kawałka kartonu, dwóch soczewek, magnesu, rzepu, gumki i opcjonalnego tagu NFC. Podobnie jak w przypadku Samsung Gear VR, Cardboard wymaga telefonu o maksymalnej przekątnej 6 cali. Dodatkowo musi on być wyposażony w żyroskop, który wykrywa ruchy głowy i precyzyjnie steruje obrazem. Plusem zestawu Google

jest brak ograniczenia obsługi do telefonów konkretnego producenta, jest on dostępny także na system iOS. Cardboard nie wykorzystuje zewnętrznych kontrolerów.

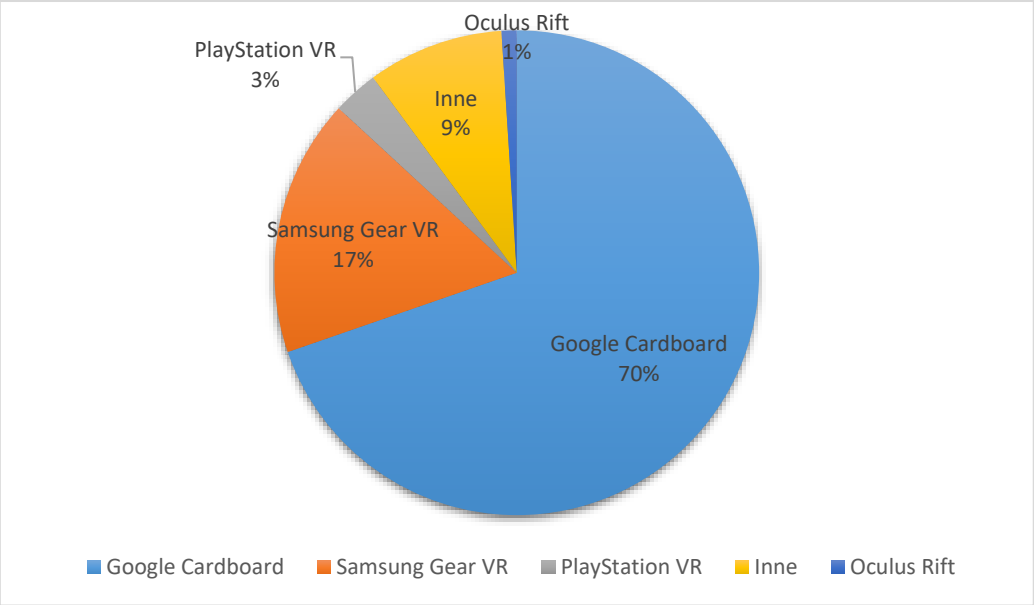
Rys. 2.11 zestaw Google Cardboard. Źródło:vr.google.com

Poniższa tabela (Tabela 2.1) podsumowuje opisane wyżej zestawy, porównując ich podstawowe specyfikacje.

Czynnik	PlayStation VR	HTC Vive	Oculus Rift	Samsung Gear VR	Google Cardboard
Rodzaj zestawu	Stacjonarny	Stacjonarny	Stacjonarny	Mobilny	Mobilny
Rozdzielczość [piksele]	960 x 2160	1080 x 2400	1080 x 2400	Taka jak telefonu	Taka jak telefonu
Częstotliwość odświeżania [Hz]	120	90	90	Taka jak telefonu	60
Pole widzenia [stopnie]	100	110	110	101	-
Platforma sprzętowa	PlayStation 4	PC	PC	Android (Samsung)	-
System operacyjny	PlayStation 4	Windows, Linux	Windows	Android, iOS	Android
Zewnętrzne kontrolery	Tak	Tak	Tak	Tak	Nie
Cena	629$	800$	600$	95$	15$

Tabela 2.1 Porównanie zestawów VR. Opracowanie własne

Zestawy stacjonarne charakteryzują się dużo lepszymi parametrami, niż zestawy mobilne. Jednak jest to możliwe tylko ze względu na korzystanie z zasobów komputera o dużej wydajności. Wadą tego rozwiązania jest „przywiązanie" użytkowania do konkretnego miejsca, w którym znajduje się komputer. Dużą przewagą zestawów mobilnych jest możliwość korzystania z nich w praktycznie każdym miejscu. Również różnice w cenach między tymi grupami produktów są znaczące.



Rys. 2.12 Ilościowy udział w rynku zestawów VR. Źródło: Opracowanie własne, na podstawie https://www.strategyanalytics.com

Te elementy przekładają się na udział zestawów mobilnych w ich ogólnej liczbie (87%), gdzie Google Cardboard jest zdecydowanym liderem (Rys. 2.12). Jest to przede wszystkim zasługa dostępności tej platformy. Jednak mimo tak dużego udziału ilościowego, rozwiązania mobilne generują jedynie 23% ogólnych zysków z wirtualnej rzeczywistości [15].

2.3 Silniki gier

2.3.1 Definicja, przeznaczenie silników gier

Silnik gry to w ogólności oprogramowanie służące do tworzenia gier. Składa się ono z komponentów odpowiadających za poszczególne elementy gry tj. silnik renderujący grafikę 2D lub 3D, silnik odpowiedzialny za fizykę, animacje, dźwięk, sztuczną inteligencję, zarządzanie zasobami, zadania sieciowe, czy skrypty [16]. Istotą silników jest umożliwienie sprawnego, wydajnego i niezawodnego przebiegu produkcji, a także dostarczenie elastycznych narzędzi pozwalających na zrealizowanie jak największej ilości koncepcji. Ze względu na dużą różnorodność rodzajów gier i obsługujących ich platform przez lata pojawiają się coraz to nowe silniki, kładące nacisk na odmienne elementy. Przykładowo część silników oferuje bardzo realistyczne i rozbudowane możliwości graficzne, wymagające wydajnych jednostek obliczeniowych, podczas gdy inne skupiają się na umożliwieniu płynnej rozgrywki sieciowej na dużą skalę [16]. Wielu producentów gier posługuje się własnymi, stworzonymi na potrzeby wewnętrzne silnikami, zazwyczaj nie udostępnianymi do sprzedaży, czy darmowego wykorzystania przez niezależnych twórców [16].
Wraz ze wzrostem znaczenia platform mobilnych, duża część silników, choć nie wszystkie, zaczęła wspierać również tego typu urządzenia. Taka sama tendencja pojawia się w przypadku wirtualnej rzeczywistości, zdobywającej coraz większą popularność. W następnych podrozdziałach opisane zostaną wybrane silniki, ich wady i zalety, główne obszary zainteresowania, dostępność oraz używane technologie.

2.3.2 Unity

Silnik Unity został napisany w językach C i C++ (aktualna wersja 5), umożliwia tworzenie gier, symulacji, wizualizacji na systemach Windows, macOS i Linux. Docelowe platformy to między innymi Android, iOS, Tizen, Windows Phone, Google Cardboard, HTC Vive, Oculus Rift, PlayStation, Windows, macOS i wiele innych, obejmując właściwie wszystkie najważniejsze obecnie platformy, co stało się jednym z głównych haseł pod którym reklamowane jest Unity. Skrypty obsługujące tworzoną aplikację można pisać w językach C# or JavaScript (UnityScript), możliwe jest również używanie języka Boo, jednak od pojawienia się wersji piątej silnika, Boo nie jest już wspierane zarówno w dokumentacji, jak i w samym edytorze, ponieważ był używany w mniej niż 0,5% skryptów.
Dostępne są cztery wersje silnika zapewniające różne warunki użytkowania, a tym samym różne ceny. Podstawowa, darmowa wersja daje dostęp do wszystkich funkcji. Ograniczenia dotyczą braku dostępu do kodu źródłowego i pomocy technicznej, obsługi mniejszych rozmiarów gier typu multiplayer, braku możliwości modyfikacji startowego okna gry, wyglądu edytora oraz ograniczenia przychodów z tworzonych aplikacji do 100 tysięcy dolarów. Kolejne wersje przewidują coraz wyższe możliwości w zakresie wymienionych funkcjonalności i miesięczne opłaty za każdą instalację (Plus-35$/miesiąc, Pro-125$/miesiąc, Enterprise - ustalana odrębnie).
Przy użyciu Unity powstały takie gry jak np. Pillars of Eternity, Endless Legend (Rys. 2.13), Prey for the Gods, czy Pokemon GO.

Rys. 2.13 Endless Legend – gra stworzona w Unity. Źródło:store.steampowered.com

2.3.3 Unreal Engine

Unreal Engine to silnik wyprodukowany przez Epic Games, przy użyciu języka C++, C#, GLSL, HLSL (aktualna wersja 4) [17]. Podobnie jak w przypadku Unity, zapewnia obsługę wszystkich głównych systemów mobilnych, desktopowych, konsolowych, HTML5 i systemów VR [17]. Unreal Engine może być używany na systemie Windows, Linux oraz macOS. Elementem wyróżniającym silnik jest programowanie graficzne oparte na schematach (Blueprints). Łączenie dostępnych bloków (zaprogramowanych w C++) pozwala na uzyskanie oczekiwanej logiki gry. Możliwe jest również samodzielne tworzenie bloków używając języka C++. Pierwsza wersja silnika powstała w 1998 roku, przez ten czas dodanych zostało bardzo wiele nowych funkcji, co przekłada się na możliwość modyfikacji praktycznie każdego aspektu projektu.

Unreal Engine 4 jest dostępny w jednej, pełnej wersji, całkowicie za darmo [17]. Udostępniony jest również kod źródłowy, który można dowolnie modyfikować. Wprowadzone zmiany są analizowane przez Epic Games i w razie pozytywnej weryfikacji poprawki są wprowadzane do głównej wersji silnika. Opłaty pobierane są wyłącznie w przypadku osiągnięcia zysku powyżej 3000 $ na kwartał, jest to stawka w wysokości 5% dochodu [17].
Unreal Engine pomógł stworzyć takie gry jak Gears of War 4, polskie The Vanishing of Ethan Carter (Rys. 2.14), a także Unreal Tournament, który jest dziełem producenta silnika, firmy Epic Games.

Rys. 2.14 Vanishing of Ethan Carter – gra stworzona w Unreal Engine 4.
Źródło:store.steampowered.com

2.3.4 CryEngine

CryEngine został stworzony przez firmę Crytek przy użyciu C++, C# i Lua. W przeciwieństwie do dwóch wcześniej zaprezentowanych silników CryEngine wspiera mniejszą ilość platform. Umożliwia on tworzenie gier dla systemów Windows, macOS, Linux, PlayStation, Xbox, Wii, Android, iOS, Oculus Rift, Open Source VR, PlayStation VR oraz HTC Vive. Językami skryptowymi są C++ i C#. CryEngine w najnowszej wersji wprowadził również system Schematyc (funkcja beta), w połączeniu z Flow Graphs działający na podobnej zasadzie co Blueprints w Unreal Engine 4 [18].
Silnik od Crytek jest dostępny zupełnie bezpłatnie, a twórcy nie są zobowiązani do ponoszenia jakichkolwiek opłat za gry, z których uzyskali zysk. Opłaty są w pełni opcjonalne, przyjmując formę dobrowolnego wsparcia dla rozwoju oprogramowania. Od 2016 roku dostępny jest również kod źródłowy [18]. W tym samym roku na bazie CryEngine, przez firmę Amazon wydany został silnik Amazon Lumberyard, jako jego zmodyfikowana wersja [19].
Przy użyciu CryEngine powstały takie gry jak Far Cry (Rys. 2.15), Crysis, Sniper: Ghost Warrior 2. Pierwsze dwa tytuły, jak wskazują ich nazwy, zostały wyprodukowane przez autora silnika – Crytek [18].

Rys. 2.15 Scena stworzona w CryEngine. Źródło : cryengine.com

2.3.5 Rockstar Advanced Game Engine (RAGE)

Nazwa silnika pochodzi od jego producenta, czyli firmy RAGE Technology. Na jego temat nie ma zbyt wielu informacji, ponieważ nie jest dostępny do pobrania, ani do zakupu. Jest on używany wyłączne do wewnętrznych projektów studia. Gry wyprodukowany przy jego użyciu trafiają na platformy Windows, PlayStation, Wii oraz Xbox. Są to zaawansowane, rozbudowane tytuły o bardzo wysokich walorach wizualnych, dlatego wymagają bardzo wydajnego sprzętu, co eliminuje chociażby platformy mobilne. Najpopularniejszą grą korzystającą z tego silnika jest jedna z najbardziej dochodowych serii - Grand Theft Auto (Rys. 2.16).

Rys. 2.16 GTA V – gra stworzona w RAGE. Źródło : rockstargames.com

2.3.6 Cocos2d

Cocos2d jest silnikiem obsługującym szeroki zakres platform: Windows, macOS, Linux, iOS, Android, Tizen, Windows Phone, HTML5, Xbox, Samsung Gear VR, Oculus VR, Google Cardboard, Google Daydream [20]. W zależności od używanego języka programowania i

docelowej platformy możemy wybrać odpowiednią wersję oprogramowania, na przykład Cocos2d-ObjC wykorzystuje Objective-C i Swift, z kolei Cocos2d-xna używa C# [20]. Silnik jest wciąż rozwijany, jako że kod źródłowy poszczególnych wersji jest dostępny w serwisie GitHub. Trwają prace nad opracowaniem wersji oferującej wsparcie dla jak największej ilości platform [20].
Ważną cechą silnika jest możliwość tworzenia jedynie gier i aplikacji 2D (Rys. 2.17). Mimo to jest on bardzo popularny ze względu na wydajność, optymalizację, oszczędność zasobów oraz wspomnianą wcześniej wieloplatformowość. Nie bez znaczenia jest również szeroka społeczność rozwijająca projekt od prawie 10 lat, co niesie za sobą duże ilości tutoriali, dyskusji i wsparcia na różnorodnych forach.
Przykładem gry wykonanej na silniku Cocos2d jest chociażby Angry Birds, czy aplikacja AnTuTu Benchmark [20].

Rys. 2.17 Badland – gra stworzona w Cocos2d. Źródło:cocos2d-x.org

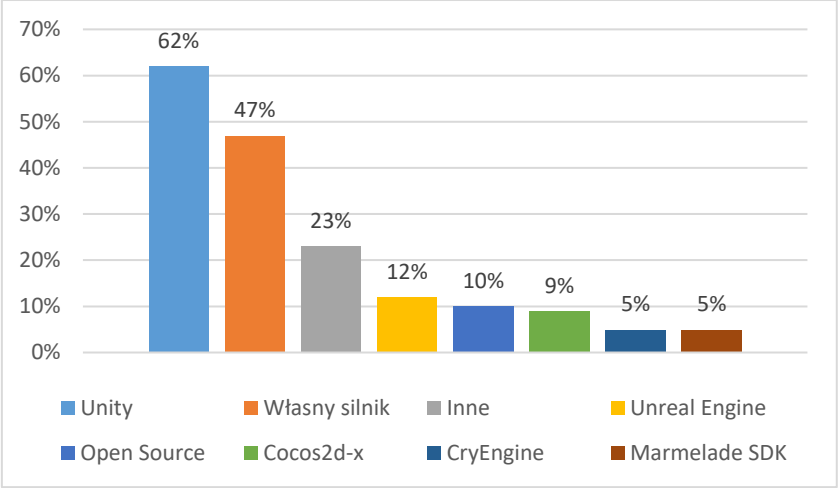
2.3.7 Clausewitz Engine

Silnik firmy Paradox Interactive stworzony w 2007 roku dla użytku wewnętrznego. Clausewitz został opracowany pod kątem gier typu 4X (eksploracja, ekspansja, eksterminacja, eksploatacja), z których słynie samo studio. Są to produkcje bardzo złożone pod względem rozgrywki, ponieważ dysponują bardzo dużą ilością informacji i rozbudowanymi mechanikami, natomiast trójwymiarowa warstwa wizualna jest zachowana w przystępnej i przejrzystej formie, nie stanowiąc głównej siły gry (Rys. 2.18). Silnik Paradoxu koncentruje się jedynie na platformach desktopowych tj. Windows, macOS oraz Linux. Dostęp do kodu źródłowego pozwala na wprowadzanie dowolnych modyfikacji, co umożliwia tworzenie modyfikacji (modów) przez szeroką społeczność skupioną wokół gier studia. Najbardziej znane z nich to na przykład seria Hearts of Iron, Europa Universalis, czy Crusader Kings.

Rys. 2.18 Europa Universalis 4 – gra stworzona w Clausewitz Engine.
Źródło:paradoxplaza.com

2.3.8 Podsumowanie

Jak widać, nie istnieje jeden idealny silnik, którego stosowanie zapewni najwyższą wydajność we wszystkich typach gier i na wszystkich platformach. Z tego właśnie względu wielu producentów gier decyduje się na opracowanie własnego silnika lub modyfikację któregoś z dostępnych rozwiązań, dopasowując je do własnych potrzeb. Pokazuje to poniższy wykres (Rys. 2.19).



Rys. 2.19 Odsetek deweloperów używających poszczególne silniki, Wielka Brytania, 2014.
Źródło: Opracowanie własne, na podstawie:
https://thenextweb.com/gaming/2016/03/24/engine-dominating-gaming-industry-right-now/

Wybór silnika jest szczególnie uzależniony od założeń dotyczących rozgrywki. Gra cechująca się zaawansowaną, realistyczną grafiką i wymagającymi efektami wizualnymi potrzebuje silnika, który oferuje szeroką gamę narzędzi i funkcji umożliwiających głęboką ingerencję w najmniejszy aspekt grafiki, a także silnik renderujący pozwalający na osiągnięcie oczekiwanej jakości obrazu. Z kolei dwuwymiarowa gra mobilna wymaga przede wszystkim małych rozmiarów i płynnej rozgrywki na słabszych urządzeniach.

Zrzuty ekranów zaprezentowane pod opisem każdego kolejnego silnika najlepiej prezentują ogólny przekrój produkowanych gier, co bezpośrednio przekłada się na różnorodność samych silników.

2.4 Renderowanie

Renderowanie jest procesem automatycznego generowania obrazu na podstawie utworzonego modelu. W jego trakcie algorytm określa wszystkie parametry elementów sceny takich jak cienie, oświetlenie, odbicia świateł, refrakcja, głębia i wiele innych. Poniżej przedstawione zostaną niektóre, ważne z punktu widzenia tematu pracy składowe całego procesu.

2.4.1 Shadows

Cienie renderowane w scenie to efekt symulujący blokowanie przez obiekty promieni świetlnych, czyli . Kształt cienia, tak jak w rzeczywistości jest uzależniony od położenia źródła światła względem oświetlanego elementu. Istnieje bardzo wiele różnych technik realizujących efekt cieniowania, jest to proces wymagający zaangażowania dużej ilości zasobów podzespołów, co stanowi sporą przeszkodę przy konieczności renderowania cieni w czasie rzeczywistości.

Rys. 2.20 Wygładzanie krawędzi mapy cienia.
Źródło: https://developer.nvidia.com/gpugems/GPUGems/gpugems_ch11.html

Mnogość metod towarzyszących cieniowaniu wynika z ilości aspektów jakie pojawiają się wraz z poprawą jakości wizualnej efektu. Jednym z nich jest chociażby wygładzanie krawędzi cienia, które jest stosowane jako pewnego rodzaju kompromis między jego rozdzielczością, a jakością (wysoka rozdzielczość cienia jest dużym obciążeniem dla podzespołów, niska rozdzielczość zaniża ogólny efekt wizualny) (Rys. 2.20). Kolejnym przykładem może być rozmywanie cienia wraz ze zwiększaniem się odległości między obiektem, a powierzchnią na którą jest rzucany. Wszystkie te elementy składają się na konieczność ciągłego usprawniania algorytmów sterujących efektem. W przypadku urządzeń mobilnych, technika cieniowania wykorzystywana przez silnik ma duże znaczenie ze względu na mniej wydajne podzespoły, niż w przypadku konsol czy komputerów PC.

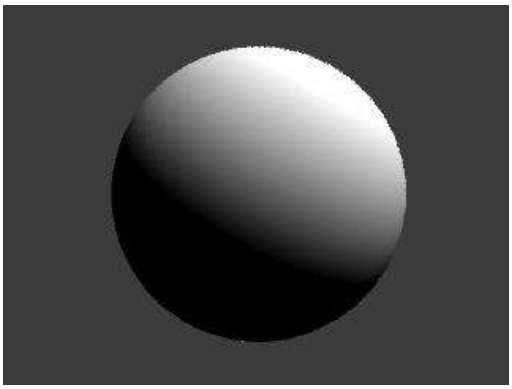
2.4.2 Motion blur

Efekt odpowiadający za rozmycie obrazu powodujący zwiększenie odczucia dynamiki ruchu w scenie. Jest on stosowany w animacjach w celu uzyskania jak najbardziej realistycznego obrazu, odpowiadającego na przykład obrazowi z kamery. Podstawową metodą symulacji efektu jest aplikowanie rozmycia do całego widoku kamery (im bliżej krawędzi tym bardziej intensywne rozmycie) na podstawie szybkości jej przemieszczania. Innym, bardziej

zaawansowanym rozwiązaniem jest selektywne rozmywanie konkretnych obiektów, jeśli poruszają się one wystarczająco szybko w stosunku do kamery.

2.4.3 Shading

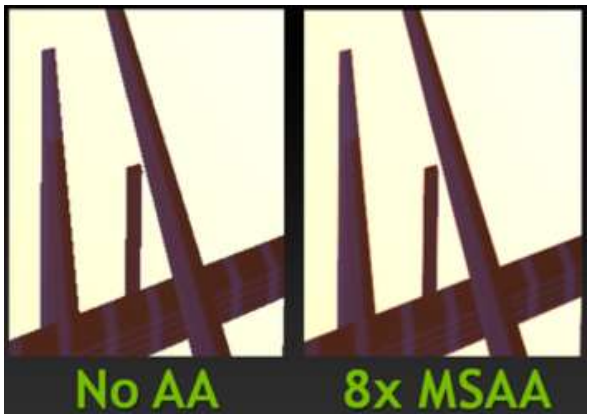
Odpowiada za zróżnicowanie jasności powierzchni obiektu w zależności od odległości i pozycji w stosunku do źródła światła, a także pozycji kamery i materiału zastosowanego dla obiektu (materiał posiada swoje własne właściwości regulujące rodzaj i stopień oddziaływania światła na obiekt). Im większa ilość poziomów jasności, tym efekt jest bardziej realistyczny, jednak wymaga też większej ilości zasobów. Dzięki temu efektowi symulowane jest wrażenie głębi.



Rys. 2.21 Shading – przykład.
Źródło:https://accad.osu.edu/~aprice/courses/752/shading_models_files/ballD.jpg

2.4.4 Anti-aliasing

Metoda wygładzania krawędzi obiektów. Zapobiega efektowi schodkowania krawędzi, który powoduje, iż obraz staje się nienaturalny i zdecydowanie obniża ogólny odbiór wizualny sceny. Najbardziej znane techniki anti-aliasingu to między innymi FSAA, TXAA, FXAA, czy MSAA. Techniki zasadniczo nie różnią się między sobą jeśli chodzi o główną ideę działania, którą jest generowanie płynnego przejścia kolorystycznego między obiektami. W dalszej części pracy stosowana będzie metoda MSAA.



Rys. 2.22 Anti-aliasing metodą MSAA.
Źródło: http://cdn.overclock.net/a/a5/500x1000px-LL-a5bf4676_TXAA.png

2.4.5 Kompresja tekstur

Metoda kompresji tekstur jest bardzo istotna z punktu widzenia płynności obrazu, częstotliwości wyświetlania klatek. Formaty popularne dla zdjęć, takie jak PNG czy JPEG są zbyt obszerne przez co uniemożliwiają osiągnięcie zadowalającej w kontekście gier wideo wydajności. Z tego powodu niezbędne jest kompresowanie tekstur, czyli zmniejszenie ich rozmiaru poprzez zapis z wykorzystaniem mniejszej liczby bitów, kosztem mniejszej ilości niesionych informacji. Istnieje wiele metod kompresji, w dalszej części pracy wykorzystywana będzie metoda ETC2 ze względu na jej wsparcie przez urządzenie testowe.

Rys. 2.23 Wybrane metody kompresji tekstur.
Źródło: http://tvtropes.org/pmwiki/pmwiki.php/UsefulNotes/TextureCompression

Zmniejszenie rozmiaru tekstur pozwala na zwolnienie pamięci karty graficznej a tym samym szybsze renderowanie. Jak widać, tekstury PNG mają blisko czterokrotnie większy rozmiar niż te same tekstury po kompresji do formatu ETC2 (Rys. 2.23).

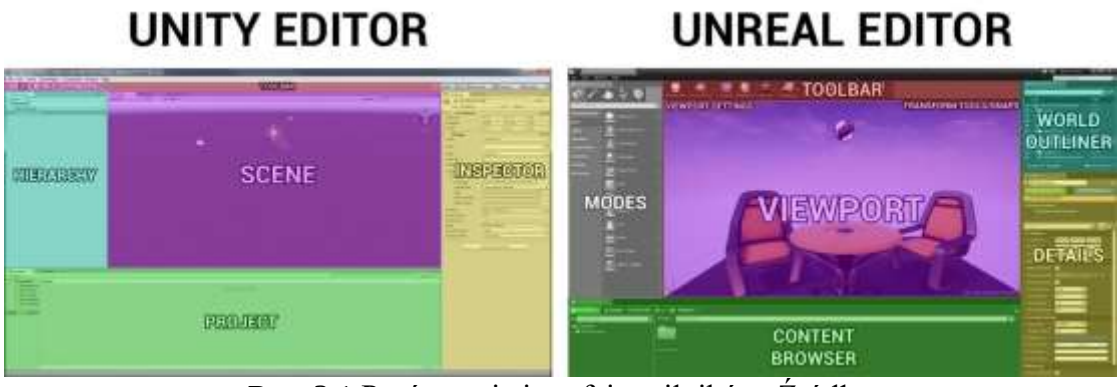
3. Przygotowanie silników

Jak wcześniej wspomniano, badania będą opierały się na wykonaniu sześciu implementacji na platformę Cardboard wykorzystując przy tym dwa różne silniki (po trzy dla każdego z nich). Celem jest uzyskanie jak najbardziej zbliżonych do siebie aplikacji. Pozwoli to porównać wydajność silników, a także jakość wizualną renderowanych scen. Ponadto opisane zostaną kolejne etapy procesu implementacji – jego złożoność oraz ewentualne problemy z nim związane.

3.1 Wybór silników do analizy i uzasadnienie

Wybrane do części badawczej silniki to Unity oraz Unreal Engine. Powodem takiego wyboru jest kilka czynników.
Pierwszy z nich to dokumentacja udostępniona przez Google, która skupia się właśnie na tych narzędziach. Opisano w niej poszczególne kroki pozwalające na tworzenie aplikacji pod platformę Cardboard. Żaden inny silnik nie został uwzględniony w tej dokumentacji. Kolejnym argumentem jest popularność silników. Spośród rozwiązań renderujących grafikę trójwymiarową to właśnie Unreal Engine i Unity są najczęściej wykorzystywane i oferują najwięcej zasobów w postaci tutoriali, społeczności i szczegółowej dokumentacji. Ponadto w dokumentacji silnika od Epic Games dostępne są materiały porównujące oba edytory pod względem budowy oraz nazewnictwa (Rys. 3.1). Pozwala to odnieść się do odpowiedników danych elementów w Unity. Poza tym oba silniki są często zestawiane jako główni konkurenci,

jednak brakuje obiektywnych porównań Unity i Unreal Engine w kontekście platformy Google Cardboard.



Rys. 3.1 Porównanie interfejsu silników. Źródło: https://docs.unrealengine.com/latest/INT/GettingStarted/FromUnity/

3.2 Instalacja, konfiguracja narzędzi

Pierwszym etapem jest instalacja narzędzi, i silników oraz skonfigurowanie ich pod kątem platformy Google Cardboard, umożliwiając tym samym stworzenie kompatybilnej aplikacji.

3.2.1 Android

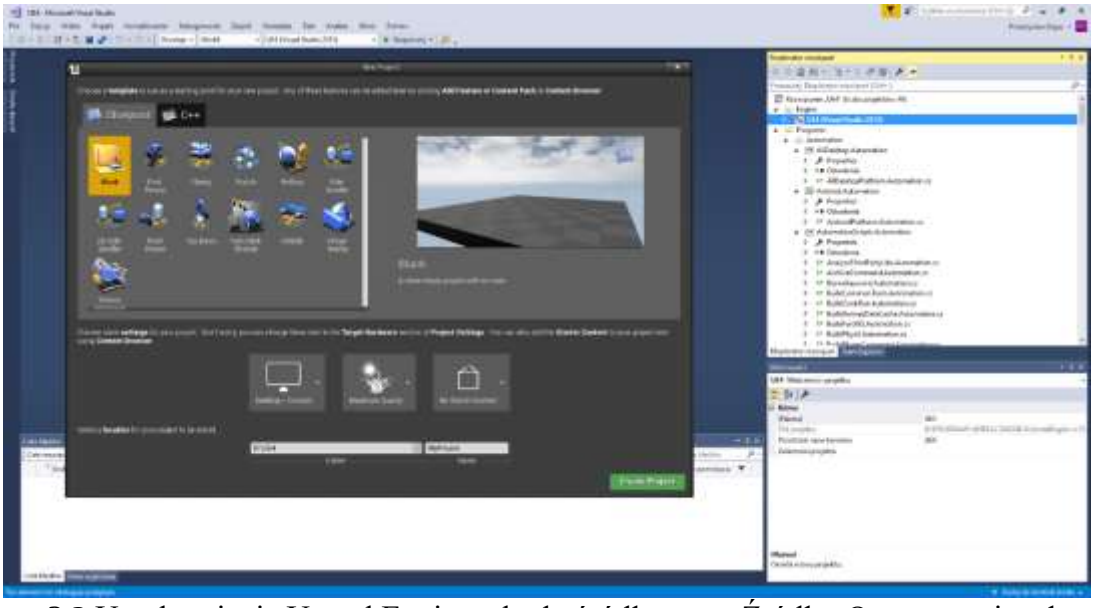
Podstawą jest instalacja zestawu narzędzi wspierających programowanie aplikacji na system Android. Są to Android SDK (Software Development Kit), JDK (Java Development Kit) oraz NDK (Native Development Kit). Zapewniają one niezbędne biblioteki, kompilatory, debugery oraz wsparcie dla bibliotek napisanych w C/C++.
Ten etap został uwzględniony dla zachowania kompletności procesu, mimo że opisane narzędzia są niezbędnym krokiem w produkcji każdego rodzaju aplikacji na system Android, nie tylko wskazanych silników.

3.2.2 Unity

Proces instalacji i konfiguracji Unity został opisany w dokumentacji Google. Ma to na celu dostosowanie go do wymagań wirtualnej rzeczywistości oraz Google Cardboard.
Wiąże się to z pobraniem i instalacją silnika w stabilnej, obsługującej Cardboard wersji (w czasie pisania tej pracy jest to wersja 5.6.0) z uwzględnieniem komponentu Android Build Support, który pozwala Unity kompilować aplikacje na ten system. Następnym krokiem jest pobranie Google VR SDK for Unity, które należy następnie w pełni zaimportować do utworzonego w Unity projektu jako pakiet zasobów (Assets). W ustawieniach kompilacji Unity konieczna jest zmiana docelowej platformy (Android), natomiast w ustawieniach gracza wybór wsparcia dla wirtualnej rzeczywistości, dodanie Cardboard jako jej SDK oraz ustawienie minimalnej wspieranej wersji API (Android 4.4 Kit Kat – API level 19). Należy również wskazać lokalizację SDK, NDK i JDK w ustawieniach narzędzi zewnętrznych.
Od tego momentu projekt jest dostosowany do Google Cardboard.
Cały proces jest przejrzysty, dlatego przebiega sprawnie i stosunkowo szybko. W trakcie realizacji tej pracy nie pojawiły się żadne problemy.

3.2.3 Unreal Engine

W przypadku silnika Unreal Engine sytuacja jest bardziej skomplikowana. Pierwszym krokiem jest pobranie z serwisu GitHub kodu źródłowego silnika w wersji zintegrowanej z Google VR. Następnie przy użyciu Visual Studio (wersja nie starsza niż 2015) należy skompilować pobrany wcześniej kod źródłowy jako Development Editor dla Win64. Po ukończeniu kompilowania uruchamiamy nową instancję programu (Start new instance) (Rys. 3.2). Aby umożliwić pracę z system Android konieczne jest zainstalowanie i uruchomienie NVIDIA CodeWorks for Android (obecnie wersja 1R6), jest to narzędzie, które automatycznie pobiera i instaluje niezbędne narzędzia (SDK, NDK, sterowniki Tegra itd.). Program jest pobierany przez uruchomienie pliku Setup.bat będącego częścią pobranego kodu źródłowego.



Rys. 3.2 Uruchomienie Unreal Engine z kodu źródłowego. Źródło: Opracowanie własne

Po wykonaniu tych kroków możemy utworzyć nowy projekt w edytorze Unreal Engine. W ustawieniach projektu należy wskazać lokalizację narzędzi dla Androida (SDK, NDK, JDK, Apache ANT), oraz skonfigurować projekt pod kątem tej platformy. Niezbędne jest ustawienie minimalnej wspieranej wersji API (Android 4.4 Kit Kat – API level 19), a także trybu GoogleVR (Cardboard). Po wykonaniu tych czynności środowisko jest gotowe do użycia, a projekt kompatybilny z platformą Cardboard.
Niestety w praktyce cały proces jest dużo bardziej czasochłonny i problematyczny niż mogłoby się wydawać. Pierwszą niedogodnością jest bardzo duży rozmiar silnika kompilowanego z kodu. W przypadku wersji instalowanej za pośrednictwem menadżera Epic Games jest to około 16GB, natomiast przy kompilacji z kodu potrzebujemy aż 40GB wolnej przestrzeni dyskowej, co przy realizacji tej pracy okazało się sporym problemem, ponieważ informacja o rozmiarze instalacji nie jest podana w dokumentacji. Ponadto sama instalacja trwa bardzo długo, bo około czterech godzin, czym oczywiście w pewnym stopniu można obarczyć konfigurację sprzętową, ale wciąż jest to niewątpliwie pewien problem. Kolejny kłopot wiąże się z programem CodeWorks for Android, który podczas pobierania i instalacji napotkał kilka błędów przez co proces musiał zostać parokrotnie powtórzony. Jednak po przebrnięciu przez proces instalacji dostosowanie projektu do wirtualnej rzeczywistości i platformy Cardboard przebiega bardzo sprawnie i w tym zakresie nie można mieć żadnych zastrzeżeń.

3.2.4 Podsumowanie

Podsumowanie powyższych informacji w formie tabeli prezentuje jak bardzo, z pozoru podobne narzędzia, różnią się od siebie, jeśli chodzi o proces instalacji i dostosowania do wymogów docelowej platformy (Tabela 3.1).

	Unity	Unreal Engine
Dostosowanie silnika do Google VR	Pakiet dodatków	Osobna wersja silnika
Sposób instalacji	Standardowy instalator	Kompilacja z kodu źródłowego
Rozmiar silnika	około 6GB	Około 40GB
Potrzebne programy i narzędzia	-Android SDK, NDK -JDK	-Android SDK, NDK -JDK -Apache ANT -Visual Studio 2015 (lub nowsze) -Nvidia CodeWorks for Android
Łączny czas instalacji i konfiguracji środowisk	Około 40 minut	Około 5 godzin

Tabela 3.1 Porównanie procesu instalacji i konfiguracji silnika. Źródło: Opracowanie własne

4. Aplikacje

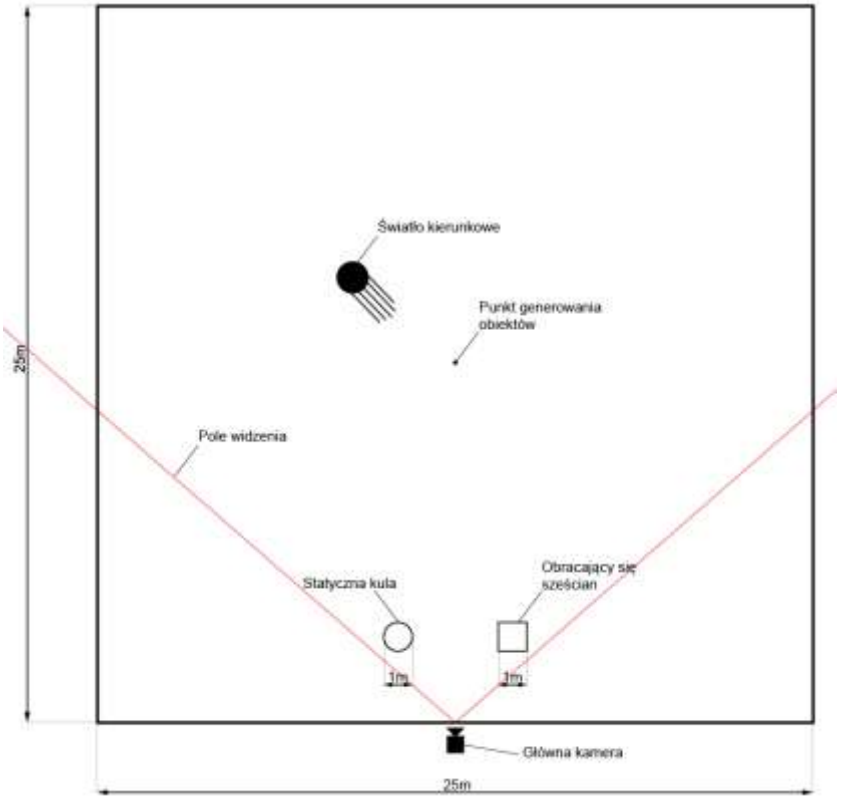
Wykonane zostało sześć aplikacje przy użyciu których możliwa będzie główna część badań, czyli analizy wydajnościowe oraz jakościowe. Przy okazji opisana zostanie złożoność operacji koniecznych do osiągnięcia efektu końcowego, a więc aplikacji zgodnej z założeniami projektu. Kolejne podrozdziały skupią się właśnie na procesie implementacji jako wstępie do głównej analizy. Podstawowe ustawienia dla obu silników prezentuje poniższa tabela (Tabela 4.1).

HDR (High Dynamic Range)	tak
Maximum number of CSM cascades	4
Mobile MSAA	8x MSAA
Occlusion culling	tak
Reflection capture resolution	128
Bloom	tak
Motion blur	nie
Support Distance Field Shadows	tak
Support Movable Directional Lights	tak

Tabela 4.1 Ustawienia renderowania dla Unity i Unreal Engine. Źródło: Opracowanie własne

4.1 Aplikacje nr 1.1 i 1.2

Pierwsza aplikacja (wersja 1.1) ma za zadanie sprawdzić wydajność silnika przy generowaniu dużej ilości obiektów. Same obiekty poruszając się wymuszają dynamiczne renderowanie cieni, ze względu na zmianę położenia. Poszczególne elementy sceny mogą być również porównane na przykład ze względu na jakość krawędzi, czy właśnie cieni.
Druga wersja tej samej aplikacji (wersja 1.2) nie będzie uwzględniała cieni rzucanych przez obiekty na płaszczyznę. Przełoży się to na mniejsze obciążenie urządzenia i pozwoli zbadać wydajność przy mniej wymagającym renderingu, skupiając się na procesowaniu fizyki oraz podstawowych aspektów graficznych.

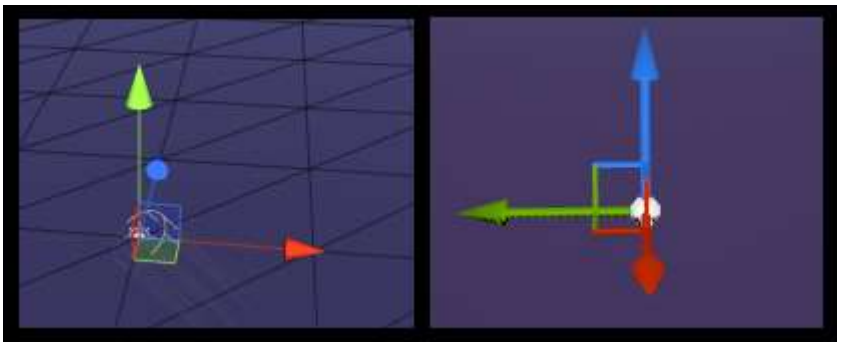


Rys. 4.1 Projekt aplikacji nr 1, położenie obiektów. Źródło: Opracowanie własne

Powyższy schemat (Rys. 4.1) prezentuje zaprojektowany rozkład obiektów w scenie dla obu wersji aplikacji. Elementy położone najbliżej kamery mają na celu umożliwienie obserwacji jakości renderowania cieni, rotujący sześcian dodatkowo pozwala analizować jakość renderowania dynamicznych obiektów. Punkt generowania obiektów jest położony na środku sceny, 10 metrów nad powierzchnią płaszczyzny i ma być w nim generowane dziesięć obiektów (sześciany 1m x 1m) na sekundę. Światło kierunkowe pada na płaszczyznę pod kątem 50 stopni względem osi x oraz 60 stopniu względem osi y. Płaszczyzna o wymiarach 25m x 25m ma za zadanie utrzymywać obiekty w polu widzenia kamery, aby liniowo zwiększać ich ilość, a tym samym konsekwentnie zwiększać zapotrzebowanie na zasoby urządzenia. Dodatkowym elementem aplikacji jest licznik wyświetlający aktualną liczbę klatek na sekundę, będący elementem interfejsu.
Opis implementacji powyższego projektu w obu silnikach będzie przedstawiał kroki od momentu ukończenia konfiguracji, co zostało przybliżone w rozdziale 3.2 (Instalacja, konfiguracja narzędzi).

4.1.1 Obiekty startowe

Pierwszym krokiem jest utworzenie i dostosowanie obiektów startowych, które będą obecne w scenie od samego początku. Są to wszystkie elementy przedstawione na schemacie (Rys. 4.1) za wyjątkiem „punktu generowania obiektów", który został zaprezentowany na schemacie jedynie w celu przybliżenia konstrukcji sceny, nie jest on fizycznym obiektem, a jedynie oznaczeniem lokalizacji.
W obu silnikach obiekty tworzymy w ten sam sposób. Spośród gotowych, podstawowych obiektów wybieramy te, które są nam potrzebne po czym zostają one dodane do sceny i okna zawierającego wszystkie obiekty w scenie. Warto wspomnieć, że oba narzędzia używają innych układów współrzędnych, w Unity wysokość określa położenie na osi y, natomiast w Unreal Engine jest to oś z (Rys. 4.2).



Rys. 4.2 Układy współrzędnych w Unity (z lewej) i Unreal Engine (z prawej).
Źródło: Opracowanie własne

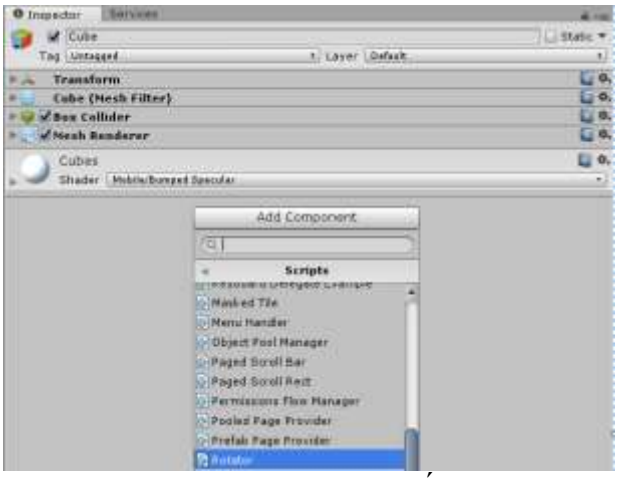
4.1.2 Implementacja rotacji obiektu

W Unity ciągłą rotację programujemy za pomocą skryptu C#. Dokumentacja Unity w szczegółach opisuje wszystkie elementy API wraz z przykładami prezentującymi sposoby ich wykorzystania.

```
1 using UnityEngine;
2
3 public class Rotator : MonoBehaviour {
4     private void Update()
5     {
6         transform.Rotate(new Vector3(15, 30, 45) * Time.deltaTime);
7     }
8 }
```

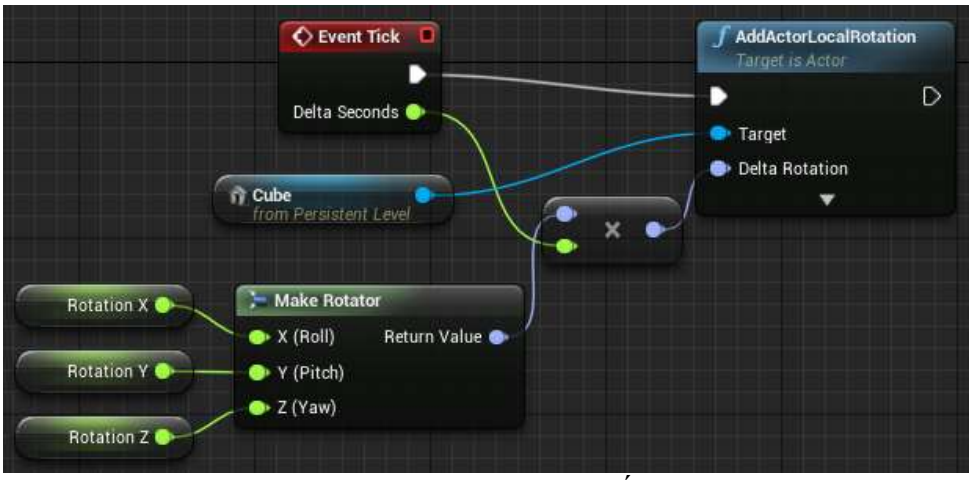
Rys. 4.3 Skrypt rotujący obiekt Unity. Źródło: Opracowanie własne

Powyższy skrypt (Rys. 4.3) powoduje obrót obiektu w trzech osiach jednocześnie (odpowiednio x,y,z). Zmienna *Time.deltaTime* zapewnia niezależność kąta obrotu od ilości klatek na sekundę, co sprawia że obiekt obraca się co sekundę o ustalone wartości. Metoda *Update* jest wywoływana co klatkę, dzięki czemu uzyskujemy ciągły obrót. Jak widać skrypt nie zawiera bezpośredniego odwołania do konkretnego obiektu. Przydzielenie zaprogramowanej mechaniki do obiektu odbywa się poprzez dodanie do niego skryptu jako komponentu (Rys. 4.4).



Rys. 4.4 Dodanie skryptu do obiektu „Cube”. Źródło: Opracowanie własne

W Unreal Engine taki sam efekt osiągamy innymi środkami. Jak wcześniej wspomniano silnik ten funkcjonuje w oparciu o schematy (Blueprints). Również scena jest traktowana jako schemat, w którym możemy definiować jej mechaniki (Rys. 4.5).



Rys. 4.5 Schemat rotujący obiekt Unreal Engine. Źródło: Opracowanie własne

Event Tick uruchamia cały zaimplementowany schemat co klatkę. *AddActorLocalRotation* to metoda rotująca wskazany obiekt *Cube* o kąt, który jest iloczynem zdefiniowanego przez zmienne wektora *Make Rotator* i czasu między kolejnymi klatkami *Delta Seconds*, który pozwala na uniezależnienie zakresu obrotu od ilości klatek na sekundę. Warto nadmienić, iż istnieje możliwość definiowania własnych funkcji poprzez napisanie odpowiedniego skryptu i w ten sposób zredukowania ilości bloków w schemacie. Jednak takie rozwiązanie jest zazwyczaj używane w przypadku mechanik trudnych do skonstruowania za pomocą dostępnych bloków.

Jak widać, tą samą mechanikę uzyskujemy w podobny sposób jeśli chodzi o samą logikę działania, jednak system zaproponowany w Unreal Engine wymaga zdecydowanie większej ilości operacji, niż ma to miejsce w przypadku silnika Unity, gdzie wystarczająca okazuje się jedna linia kodu.

4.1.3 Implementacja generatora obiektów

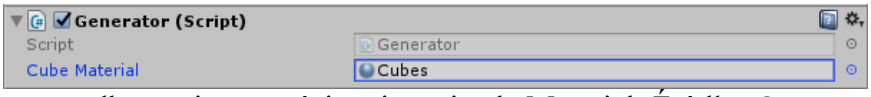
Ta mechanika ma za zadanie generowanie obiektów w określonym punkcie dziesięć razy na sekundę. Początek działania jest określony na piątą sekundę od uruchomienia.

Unity do tego zadania wykorzystuje skonstruowany skrypt (Rys. 4.6).

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

public class Generator : MonoBehaviour {
    public Material cubeMaterial;

    private void Start()
    {
        InvokeRepeating("createCube", 5.0f, 0.1f);
    }

    private void createCube()
    {
            GameObject cube = GameObject.CreatePrimitive (PrimitiveType.Cube);
            cube.GetComponent<MeshRenderer> ().material = cubeMaterial;
            Rigidbody gameObjectsRigidBody = cube.AddComponent<Rigidbody> ();
            gameObjectsRigidBody.mass = 1;
            gameObjectsRigidBody.collisionDetectionMode = CollisionDetectionMode.ContinuousDynamic;
            cube.transform.localScale = new Vector3(1, 1, 1);
            cube.transform.position = new Vector3 (0, 10, 0);
            cube.transform.rotation = Quaternion.Euler(45, 45, 45);
    }
}
```

Rys. 4.6 Skrypt generujący obiekty. Źródło: Opracowanie własne

Metoda *createCube* najpierw tworzy nowy obiekt *cube* (sześcian), następnie zmienia jego standardowy komponent *material* na *cubeMaterial*. Jako, że *cubeMaterial* został zadeklarowany jako zmienna publiczna, odpowiadający mu materiał jest wybierany z poziomu edytora przez przyporządkowanie utworzonego wcześniej materiału *Cubes* jako wartości zmiennej (Rys. 4.7).



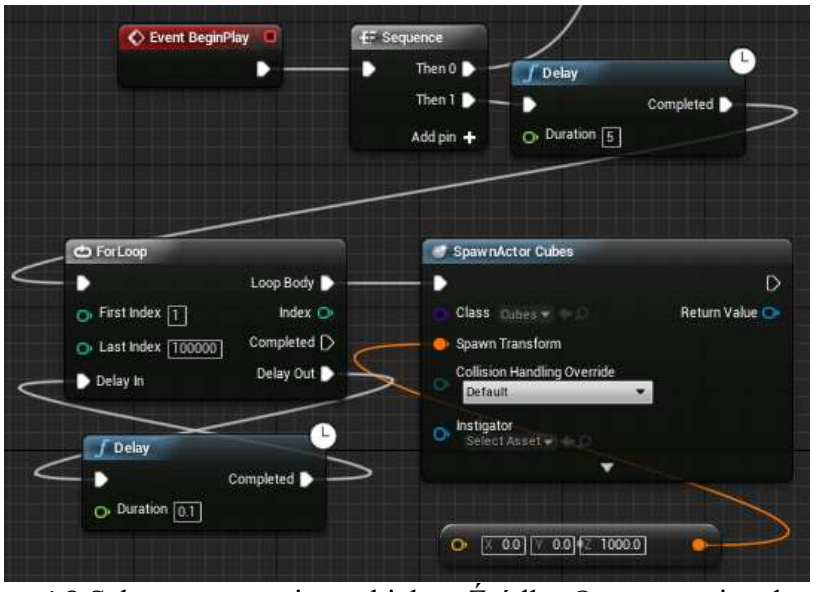
Rys. 4.7 Przyporządkowanie wartości zmiennej cubeMaterial. Źródło: Opracowanie własne

Kolejny krok to dodanie do obiektu komponentu Rigidbody, dzięki któremu symulowana jest fizyka obiektu umożliwiająca między innymi jego spadanie. W atrybutach Rigidbody ustawiana jest masa elementu (1) oraz tryb ciągłego wykrywania kolizji między obiektami (CollisionDetectionMode.ContinuousDynamic), aby nie dopuścić do sytuacji w której obiekty przenikają przez inne elementy sceny. Na koniec definiowana jest początkowa wielkość, lokalizacja oraz rotacja tworzonego sześcianu w postaci wektorów odpowiadających przestrzeni trójwymiarowej. Metoda InvokeRepeating odpowiada za wywoływanie metody createCube co dziesiątą część sekundy zaczynając od piątej sekundy od uruchomienia.

Unreal Egine ponownie wymaga skonstruowania odpowiedniego schematu. Utworzenie nowego obiektu w scenie wiąże się z utworzeniem nowej klasy (nazwanej *Cubes*) definiującej obiekt, która będzie wykorzystywana później przez generator. W tym przypadku będzie to po prostu sześcian o ustalonych wymiarach oraz rotacji. Ponadto należy określić atrybut Mobility obiektu (Rys. 4.8) jako mobilny (Movable), czyli taki który może przemieszczać się w obrębie sceny. Aby umożliwić symulowanie fizyki obiektu konieczne jest zaznaczenie opcji Simulate Physics co pozwala zadać masę obiektu (MassInKg). Materiał zostaje przyporządkowany do obiektu przez wybranie wcześniej utworzonego materiału CubesMat. Tym samym klasa Cubes jest gotowa.

Rys. 4.8 Właściwości schematu obiektu. Źródło: Opracowanie własne

Następnym krokiem jest edycja schematu sceny (*Level Blueprint*, Rys.4.9), gdzie należy zaimplementować generowanie obiektów zgodnych z utworzoną klasą *Cubes*. Do bloku *Event BeginPlay*, który jest uruchamiany przy starcie aplikacji dołączany jest blok Delay opóźniający wykonywanie dalszych elementów o określony czas (5 sekund). Główna funkcja wywołująca w pętli metodę *SpawnActor Cubes*. Tworzy ona nowy obiekt zgodny z *Cubes*, opóźniając każdą iterację o 0,1 sekundy blokiem *Delay*, w lokalizacji sprecyzowanej przez dołączony wektor. W taki sposób otrzymujemy działający generator.



Rys. 4.9 Schemat generujący obiekty. Źródło: Opracowanie własne

4.1.4 Implementacja licznika klatek na sekundę

Licznik ma za zadanie wyświetlanie ilości klatek na sekundę, co pomoże w ocenie wydajności aplikacji, będzie jednym ze źródeł pozyskiwania danych do analizy.

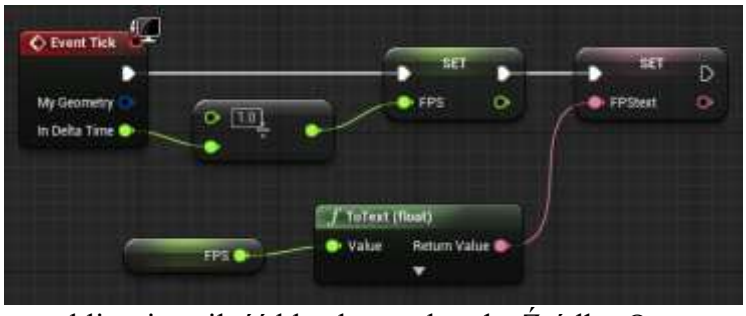
W Unity procedura wygląda podobnie jak w przypadku generatora. Potrzebny jest skrypt obliczający ilość klatek oraz text, w którego postaci obliczona wartość będzie wyświetlana na ekranie. Skrypt wygląda następująco (Rys. 4.10).

```
using UnityEngine;
using System.Collections;

public class FPSDisplay : MonoBehaviour
{
    float deltaTime = 0.0f;
    public TextMesh fpsText;

    void Start()
    {
        if (Camera.main != null) {
            transform.SetParent(Camera.main.GetComponent<Transform>(), true);
        }
    }

    void Update()
    {
        deltaTime += (Time.deltaTime - deltaTime) * 0.1f;
        float fps = 1.0f / deltaTime;
        fpsText.text = fps.ToString();
    }
}
```

Rys. 4.10 Skrypt obliczający ilość klatek na sekundę. Źródło: Opracowanie własne

Zmienna *fpsText* tak jak w przypadku wcześniejszego skryptu jest przyporządkowywana do obiektu z poziomu edytora. Ponadto pozycja tekstu jest powiązana z pozycją kamery dzięki czemu jest on cały czas widoczny. Metoda *Update* co klatkę aktualizuje wartość zmiennej obliczanej jako stosunek jednostki czasu do różnicy czasu między kolejnymi klatkami. Współczynnik wygładzający równy 0,1 przy obliczaniu *deltaTime* zapobiega drastycznym wahaniom tej wartości.

W Unreal Engine ponownie użyta została klasa oparta na metodzie Event Tick, która co klatkę ustawia wartość utworzonej zmiennej FPS (float) jako stosunek jednostki czasu do czasu między kolejnymi klatkami. Następnie jest ona konwertowana na typ string jako zmienna FPStext i w tej formie zwracana przez klasę o nazwie FPS (Rys. 4.11).



Rys. 4.11 Schemat obliczający ilość klatek na sekundę. Źródło: Opracowanie własne

Dzięki metodzie Create Widget tworzony jest element interfejsu oparty na klasie FPS, a wartość przez nią zwracana jest przekazywana do widoku gracza (Rys. 4.12). Dzięki temu ilość wyświetlanych klatek mieści się nieprzerwanie w polu widzenia kamery (użytkownika).

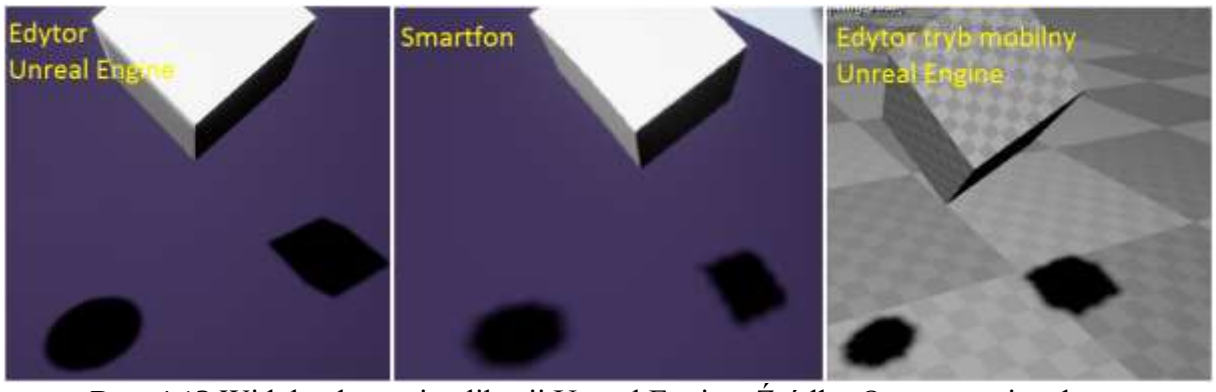
Rys. 4.12 Schemat przekazujący obliczaną wartość do widoku gracza. Źródło: Opracowanie własne

Ponownie ilość operacji koniecznych do wykonania w silniku Unreal Engine jest zdecydowanie większa, co być może nie wynika bezpośrednio z opisu, jednak w praktyce wykonanie poszczególnych czynności trwa zauważalnie dłużej, niż ma to miejsce w przypadku pisania skryptu dla Unity. Ponadto implementacja wygładzania wartości prezentującej ilość klatek na sekundę w Unreal Engine okazała się nieskuteczna, mimo że logika rozwiązania była taka sama jak w przypadku skryptu Unity. Może to wynikać ze sposobu w jaki system realizuje skonstruowany schemat. Z kolei w Unity problemem okazało się wyświetlanie licznika jako warstwy interfejsu w obszarze kamery. Mimo, że licznik był prawidłowo wyświetlany w podglądzie edytora, to po uruchomieniu aplikacji na smartfonie nie był widoczny, co prawdopodobnie jest związane z trybem widoku stereoskopowego dostosowanego do wirtualnej rzeczywistości, który „czyści" widok kamery. Z tego względu konieczne okazało się umieszczenie licznika w scenie w postaci obiektu.

4.1.5 Kompilacja projektu i uruchomienie

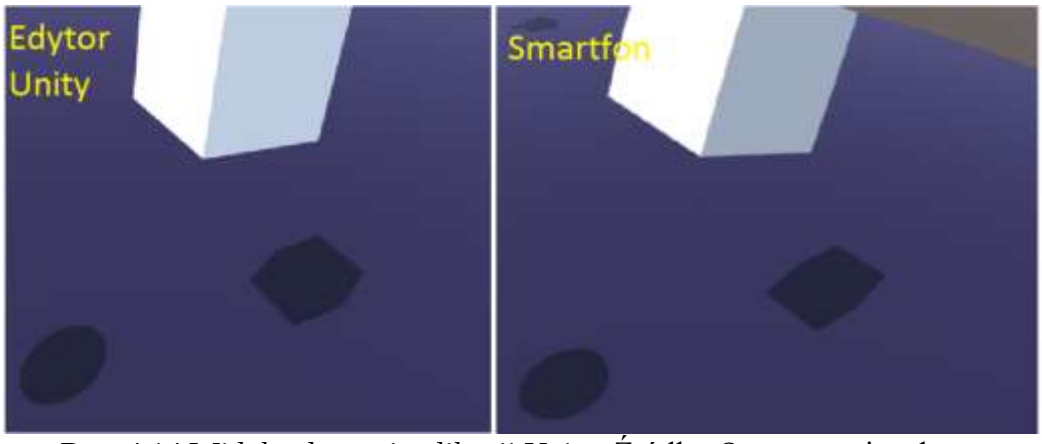
W obu silnikach ten proces odbywa się w praktycznie ten sam sposób. Wybieramy docelową platformę (Android) i metodę kompresji. Metoda kompresja powinna być zgodna z technologiami obsługiwanymi przez kartę graficzną, w tym przypadku jest to format ETC2 wspierany przez urządzenia dostosowane do OpenGL 3. Następnie uruchamiamy proces kompilacji.

Unity na cały proces potrzebuje około 50 sekund i 50MB wolnej przestrzeni dyskowej, w przypadku Unreal Engine jest to około 20 minut i 160MB. Co więcej proces kompilacji w Unreal Engine jest przerywany błędem dostępu, jeśli ścieżka do plików projektu jest dłuższa niż 260 znaków, co powoduje konieczność utworzenia nowego projektu o krótszej niż pierwotnie nazwie i migracji utworzonych wcześniej elementów, a także ponownej konfiguracji ustawień projektu. Kolejny problem to duża rozbieżność między obrazem wyświetlanym w podglądzie edytora, a tym na ekranie smartfona (Rys. 4.13).



Rys. 4.13 Widok edytora i aplikacji Unreal Engine. Źródło: Opracowanie własne

Jak widać pewnym rozwiązaniem jest uruchamianie podglądu w trybie mobilnym, jednak jest on otwierany jako nowy proces i potrzebuje około minuty na uruchomienie, a następnie jednorazowo kolejnych dziesięciu na załadowanie shaderów. Natomiast jeśli chcemy uzyskać chociażby podgląd renderowanych cieni to jest to rozwiązanie zadowalające. Dla porównania poniżej zostało zamieszczone analogiczne zestawienie dotyczące Unity (Rys. 4.14). Widać, że podgląd edytora i ostateczny efekt osiągnięty na smartfonie są praktycznie identyczne.

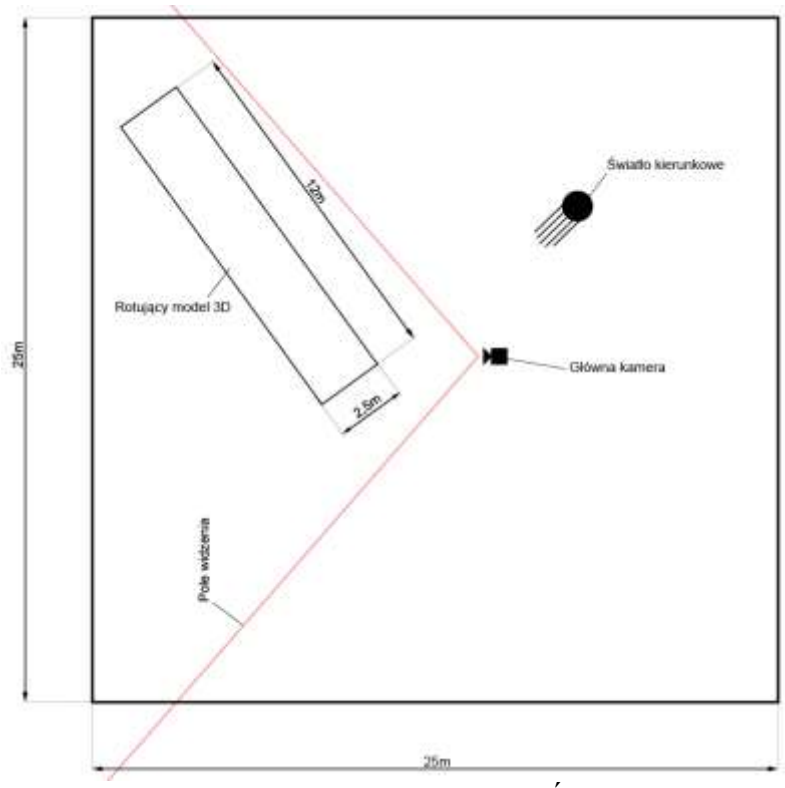


Rys. 4.14 Widok edytora i aplikacji Unity. Źródło: Opracowanie własne

Ostatni problem to startowa orientacja kamery w aplikacji skompilowanej przez Unreal Engine. Mianowicie początkowy widok kamery jest inny od tego zdefiniowanego w projekcie. Ten niewielki problem w przypadku chociażby gier PC jest znacznym utrudnieniem przy zastosowaniu wirtualnej rzeczywistości. Dla użytkownika powoduje to konieczność obrócenia się o odpowiedni kąt, który umożliwi „powrót" do pozycji wyjściowej, zakładanej w projekcie. Zdecydowanie obniża to komfort użytkowania, ponieważ fizyczna zmiana położenia gracza wiąże się często ze wstaniem z krzesła, czy obrotem w kierunku ściany.
W przypadku Unity nie pojawiły się żadne problemy związane z kompilacją, czy uruchomieniem.

4.2 Aplikacja nr 2

Druga aplikacja pokaże jak silniki radzą sobie z dużymi, skomplikowanymi modelami. Wykorzystany model będzie zmieniał swoje położenie w taki sposób, aby naprzemiennie znajdował się w polu widzenia użytkownika oraz poza nim. W ten sposób możliwe będzie zaobserwowanie jaki wpływ na wydajność ma ruch obiektu oraz jego renderowanie. Ponownie możliwa będzie ocena warstwy wizualnej.
Poniższy schemat (Rys. 4.15) prezentuje zaprojektowany rozkład obiektów w scenie. Tym razem jest to model lokomotywy [21] złożony ze 150 tysięcy wielokątów (polygons) oraz 73 tysięcy punktów (vertices), co czyni go wystarczająco skomplikowanym modelem, aby w znacznym stopniu obciążyć podzespoły urządzenia. Dodatkowo model będzie się obracał wokół własnej osi z. Światło kierunkowe pada na płaszczyznę pod kątem 30 stopni względem osi y. Płaszczyzna o wymiarach 25m x 25m ma za zadanie przysłaniać obracający się model, dzięki temu możliwa będzie obserwacja zmian wydajnościowych podczas opuszczenia przez model pola widzenia kamery. Dodatkowym elementem aplikacji jest licznik wyświetlający aktualną liczbę klatek na sekundę, będący elementem interfejsu.



Rys. 4.15 Projekt aplikacji nr 2, położenie obiektów. Źródło: Opracowanie własne.

Opis implementacji powyższego projektu (Rys. 4.15) w obu silnikach będzie przedstawiał kroki od momentu ukończenia konfiguracji, co zostało przybliżone w rozdziale 3.2. Opisy kolejnych etapów nie będą uwzględniały informacji podanych w przypadku projektu pierwszej aplikacji.

4.2.1 Obiekty startowe

Poza płaszczyzną, jedynym fizycznym obiektem jest model, który nie jest tworzony z poziomu edytora tak jak w przypadku podstawowych kształtów. Zamiast tego, pobrany wcześniej model jest importowany do obu silników.

4.2.2 Implementacja rotacji obiektu

Rotacja jest implementowana w ten sam sposób co w przypadku aplikacji nr 1. Jedyną różnicą jest zmiana wartości obrotu na 120 stopni dla osi równoległej do płaszczyzny, czyli osi *z* w przypadku Unity i osi *y* dla Unreal Engine. Obrót dla dwóch pozostałych osi przyjmuje wartość zero.

4.2.3 Implementacja licznika klatek na sekundę

Licznik jest implementowany w ten sam sposób co w przypadku aplikacji nr 1. Natomiast na potrzeby tej aplikacji jego położenie zostało nieznacznie zmienione w celu umożliwienia dokładniejszej obserwacji modelu.

4.2.4 Kompilacja i uruchomienie

Na tym etapie nie wystąpiły żadne dodatkowe problemy poza powtarzającymi się, wymienionymi dla aplikacji nr 1.

5. Badania

Badania zostały przeprowadzone przy użyciu czterech różnych narzędzi. Pierwszym z nich jest oprogramowanie GameBench służące do badania wydajności aplikacji. Podaje ono takie informacje jak użycie procesora, pamięci fizycznej, czy ilość klatek na sekundę. Kolejne dwa narzędzia to wbudowane w silniki programy do profilowania, które pozwalają zbierać dane dotyczące praktycznie każdego aspektu aplikacji, w tym takie jak ilość wyświetlanych obiektów, użycie procesora do obliczeń fizycznych, czas renderowania klatki itd. Ostatnim źródłem informacji jest licznik ilości klatek zaimplementowany we wszystkich aplikacjach. Dzięki niemu możliwe będzie potwierdzenie, iż zebrane informacje są precyzyjne, poprzez sprawdzenie zgodności danych z trzech różnych źródeł.

5.1 Urządzenie testowe

Urządzeniem na którym będą prowadzone testy aplikacji jest smartfon Huawei P10 Lite. Jest to telefon średniej klasy spełniający wymagania Google Cardboard odnośnie niezbędnych czujników. Jego parametry istotne dla pomiarów wydajności prezentuje poniższa tabela (Tabela 5.1).

Wyświetlacz	IPS TFT, 16M kolorów, 1080x1920 (5,2”)
Pamięć RAM	3GB
System operacyjny	Android 7.0 Nougat
Procesor	HiSilicon Kirin 658, 2.1Ghz, 8 rdzeni
Karta graficzna	ARM Mali-T830 MP2

Tabela 5.1 Parametry urządzenia testowego. Źródło: Opracowanie własne.

Są to parametry ważne, ponieważ przy innej konfiguracji, głównie jeśli chodzi o procesor i kartę graficzną, wyniki mogą się istotnie różnić, jednak ogólna tendencja powinna być zachowana.
Podczas wykonywania testów wszystkie dodatkowe usługi urządzenia były wyłączone (takie jak bluetooth, wi-fi, transmisja danych, synchronizacja itd.). Uruchomione były jedynie podstawowe, niezbędne procesy systemu android oraz aplikacja zbierająca dane wydajnościowe. Przed każdym testem weryfikowana była ilość dostępnych zasobów urządzenia, aby każdy kolejny pomiar był wykonywany w jak najbardziej zbliżonych warunkach.

5.2 Aplikacja nr 1.1 (z cieniowaniem)

Niestety już przy pierwszym pomiarze okazało się, że narzędzie do profilowania od Unity w bardzo dużym stopniu obciąża profilowane urządzenie, co skutkuje niską wiarygodnością zbieranych danych. Przy wyłączonym przekazywaniu danych do narzędzia aplikacja wyświetlała około 38 klatek na sekundę, natomiast po uruchomieniu komunikacji wartość ta spadała do około 20 klatek. Tak duża różnica całkowicie eliminuje możliwość wykorzystania narzędzia do dalszych badań.
Podobna sytuacja ma miejsce w przypadku narzędzia wbudowanego w Unreal Engine, jednak jego wpływ na wydajność urządzenia jest mniejsze, ponieważ komunikacja z komputerem odbywa się w inny sposób. Mimo to jego użycie mija się z celem, ze względu na wykluczenie odpowiednika od Unity.

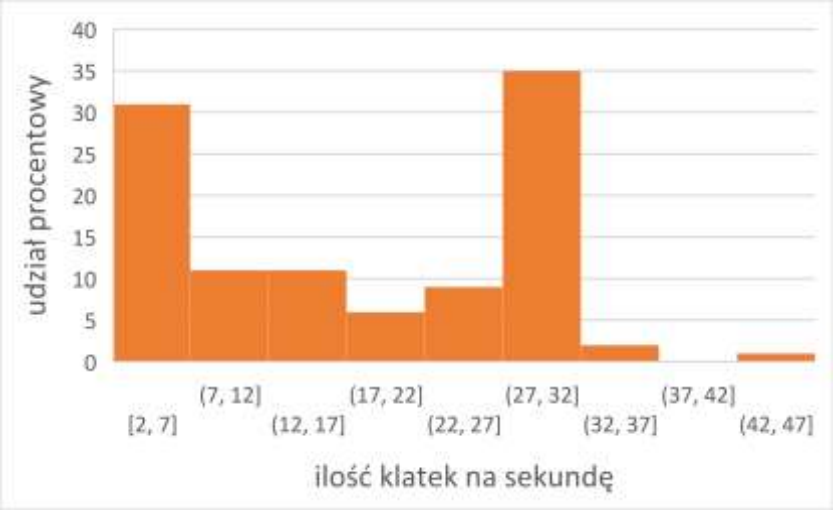
Dlatego wszystkie kolejne pomiary będą w głównej mierze oparte o GameBench, jako że gwarantuje to jednakowe techniki pomiarowe, a tym samym możliwość wiarygodnych porównań uzyskanych wyników. Dodatkowo użycie tego narzędzia w równym stopniu będzie obciążało urządzenie na którym prowadzone są badania. Licznik FPS z aplikacji posłuży do weryfikacji zgromadzonych danych.
Dla przypomnienia należy odnotować, że od piątej sekundy w wykonanej scenie pojawia się coraz więcej obiektów w tempie dziesięciu na sekundę. Jest to kluczowa mechanika dla analizy wydajnościowej.

5.2.1 Unity

Aplikacja Unity prezentuje wysoki poziom wizualny. Krawędzie obiektów są ostre i nie występują problemy z rozdzielczością cieniowania (Rys. 5.1). W początkowym okresie od uruchomienia (około 1 minuty) obraz jest wystarczająco płynny i utrzymuje się na zadowalającym poziomie.

Rys. 5.1 Widok aplikacji Unity 1.1. Źródło: Opracowanie własne.

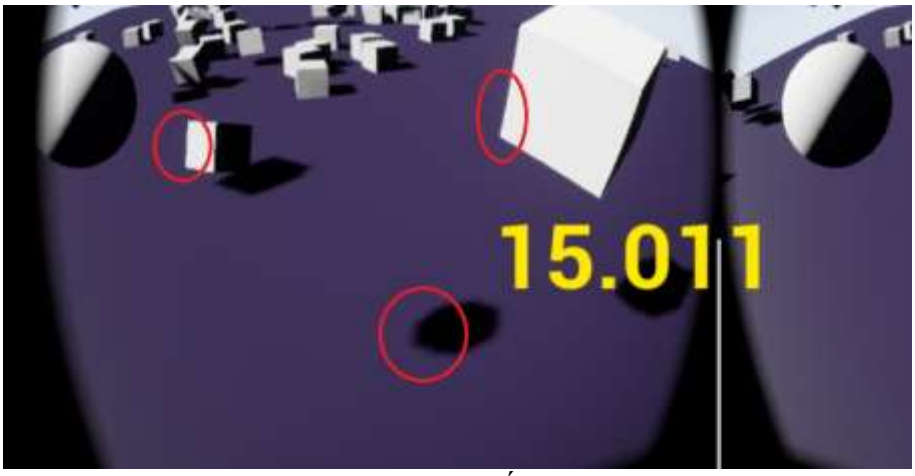
W późniejszym etapie ilość wyświetlanych klatek znacząco spadła, co skutecznie uniemożliwiło jakiekolwiek użytkowanie aplikacji. W trakcie całego okresu działania zanotowano naprawdę spore odchylenia wartości tego parametru, co prezentuje poniższy histogram (Rys. 5.2).



Rys. 5.2 Histogram ilości klatek na sekundę Unity 1.1. Źródło: Opracowanie własne.

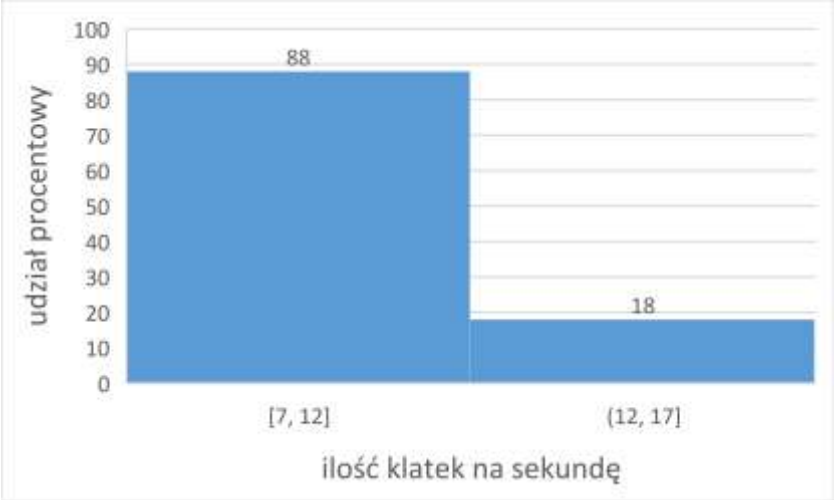
5.2.2 Unreal Engine

W przypadku Unreal Engine do warstwy wizualnej można mieć kilka zastrzeżeń. Po pierwsze jakość cieniowania nie stoi na zbyt wysokim poziomie, mimo wielokrotnych prób dostosowania ustawień renderowania w tym zakresie.



Rys. 5.3 Widok aplikacji Unity 1.1. Źródło: Opracowanie własne.

Niestety ograniczenia dla urządzeń mobilnych dotyczące dopuszczalnej ilości kaskad cieni (do dwóch) narzucone przez silnik są powodem słabych rezultatów w tym zakresie. Próba ustawienia wyższych wartości skutkuje zupełnym wyłączeniem cieniowania. Drugim problemem jest niska jakość wyświetlanych krawędzi, na których widać dużą ilość nieregularności, przez co krawędź nie jest gładka, ma strukturę „schodkową”.

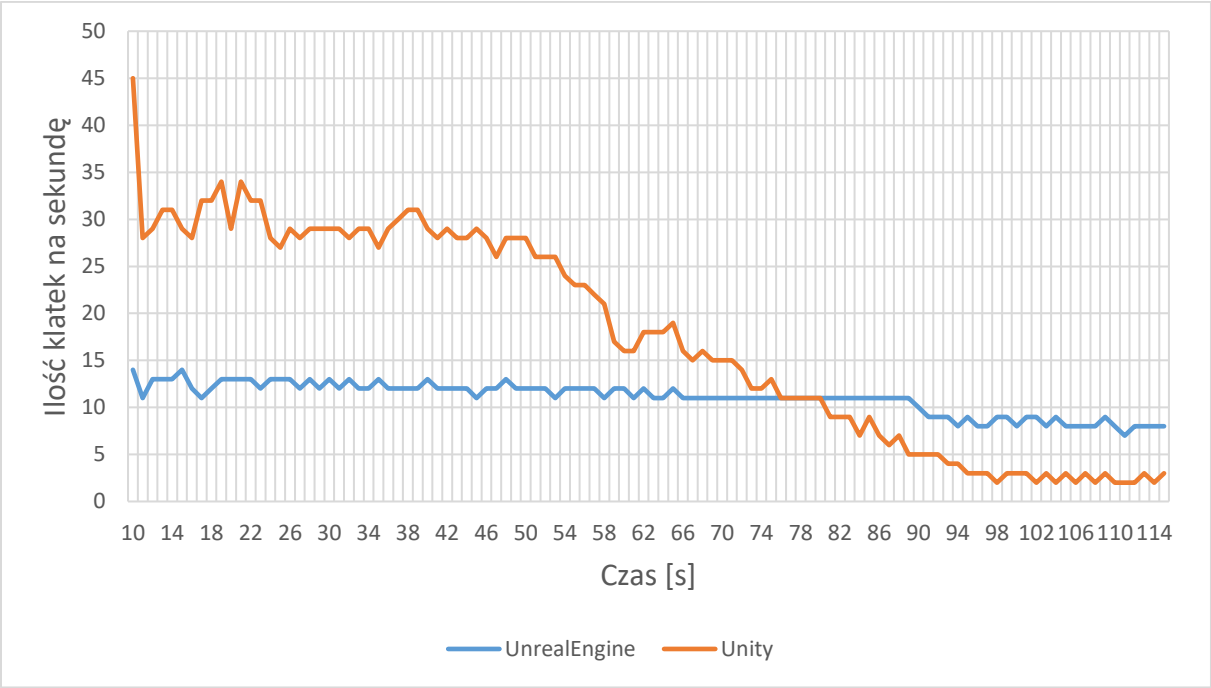


Rys. 5.4 Histogram ilości klatek na sekundę Unreal Engine 1.1. Źródło: Opracowanie własne.

Jeśli chodzi o rozkład ilości wyświetlanych klatek to w przypadku silnika EpicGames parametr ten utrzymuje się przez cały czas na bardzo podobnym poziomie (Rys. 5.4) i nie spada poniżej siedmiu klatek, co świadczy o stabilności, jednak górna wartość (17) nie jest zadowalająca i nie zapewnia płynności obrazu.

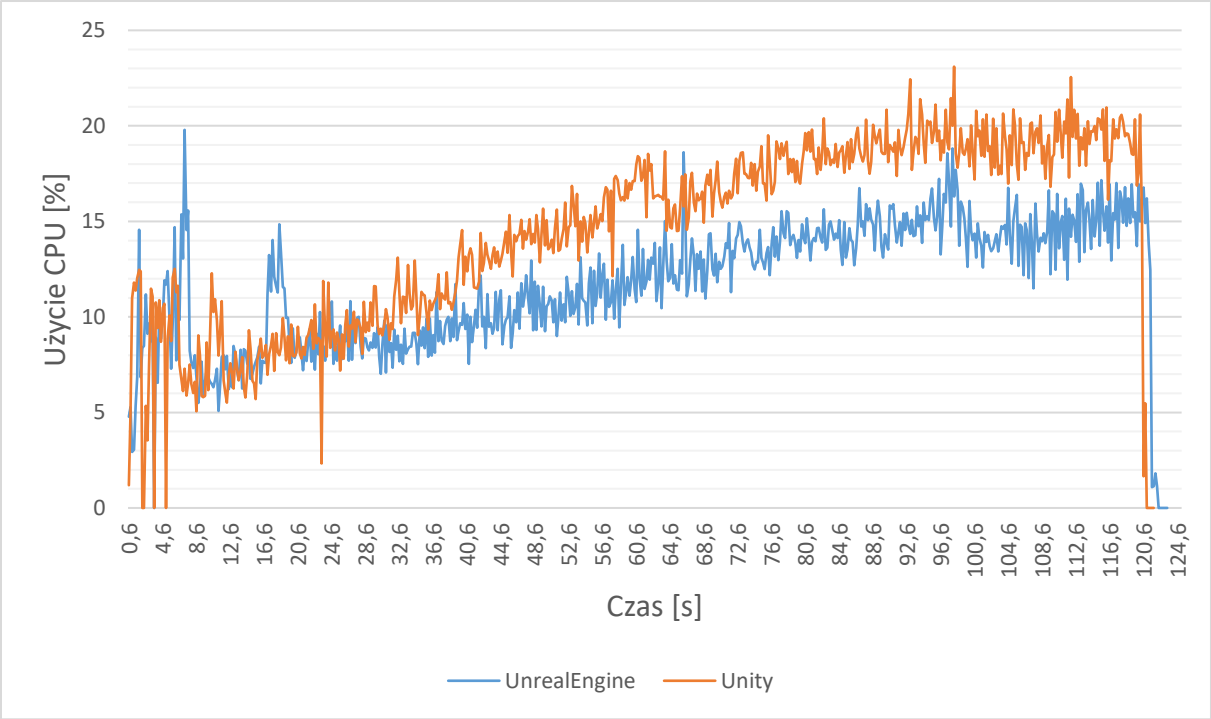
5.2.3 Porównanie

W tej części porównane zostaną aplikacje wykonane przy użyciu obu silników ze względu na trzy parametry – ilość wyświetlanych klatek na sekundę, użycie procesora, użycie pamięci RAM.



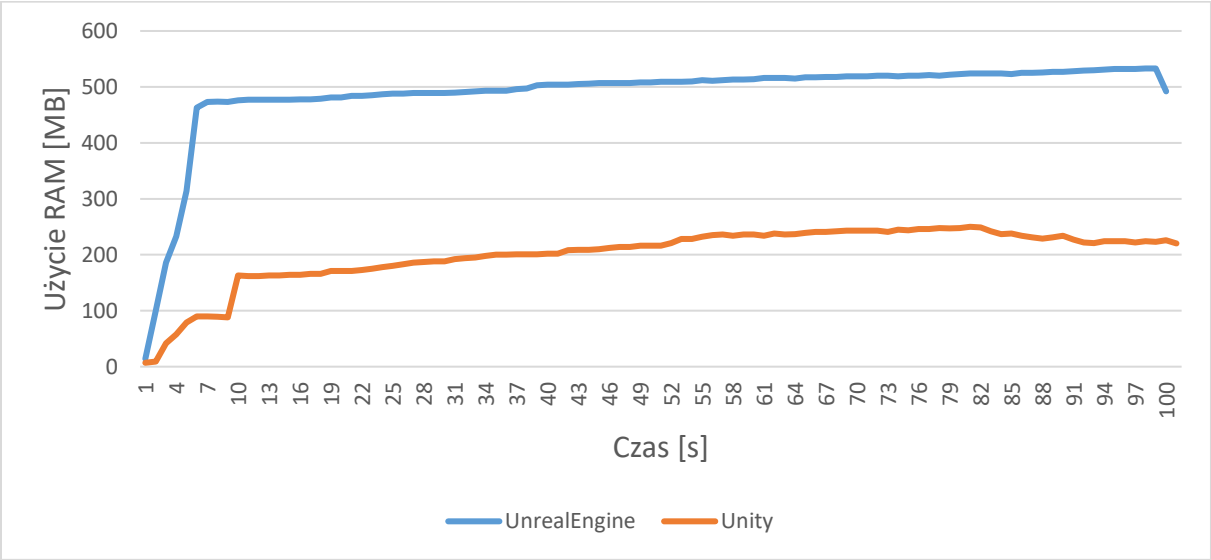
Rys. 5.5 Ilość klatek na sekundę, aplikacja nr 1.1. Źródło: Opracowanie własne.

Zaraz po uruchomieniu aplikacji widać dużo wyższe wartości fps dla silnika Unity. Różnice są spore, wynoszą około 15 klatek na sekundę. Taki poziom utrzymuje się mniej więcej do pięćdziesiątej sekundy, kiedy ilość klatek dla Unity zaczyna konsekwentnie spadać, aż osiąga wartość 2-3 klatek i pozostaje na nim do końca działania aplikacji (Rys. 5.5). Ciekawym zjawiskiem jest fakt, iż aplikacja Unreal Engine przez cały okres działania zachowuje stabilność i nawet przy dłuższych testach ilość klatek nie spadała poniżej siedmiu.



Rys. 5.6 Użycie procesora, aplikacja nr 1.1. Źródło: Opracowanie własne.

Jeśli chodzi o użycie procesora to bardziej obciąża go aplikacja Unity. W początkowej fazie obie aplikacje potrzebują około 10% mocy obliczeniowej, natomiast w późniejszym etapie zauważalny jest szybszy jej przyrost dla Unity, który ostatecznie stabilizuje się na poziomie około 20%, podczas gdy Unreal Engine wymaga około 5 punktów procentowych mniej (Rys. 5.6).



Rys. 5.7 Użycie pamięci RAM, aplikacja nr 1.1. Źródło: Opracowanie własne.

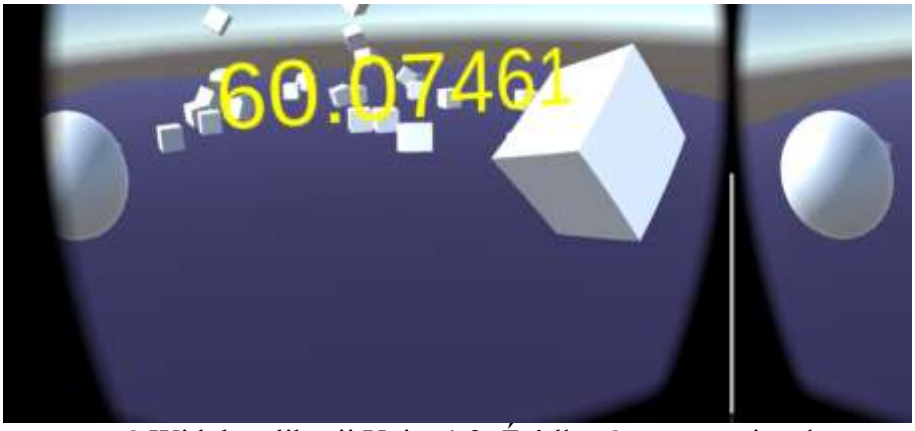
W przypadku pamięci RAM różnica między oboma silnikami jest bardzo znacząca. Unity oscyluje wokół użycia na poziomie 200MB, nie przekraczając punktu 250MB. Unreal Engine praktycznie od samego początku rezerwuje blisko 500MB pamięci RAM w szczytowym momencie osiągając wartość 533MB. Dla obu silników nie występują silne wahania wartości, utrzymują się one przez cały okres uruchomienia aplikacji na zbliżonym szczeblu.

5.3 Aplikacja nr 1.2 (bez cieniowania)

Brak renderowania cieni spowodował istotny wzrost wydajności dla obu silników. Miało to również naturalne przełożenie na stopień wykorzystania zasobów, co zostanie zaprezentowane w dalszej części przy pomocy wykresów. Jeśli natomiast chodzi o kwestie graficzne, to aspekty opisane w odniesieniu do pierwszej wersji aplikacji nie uległy zmianie, poza oczywistym brakiem możliwości analizy cieni.

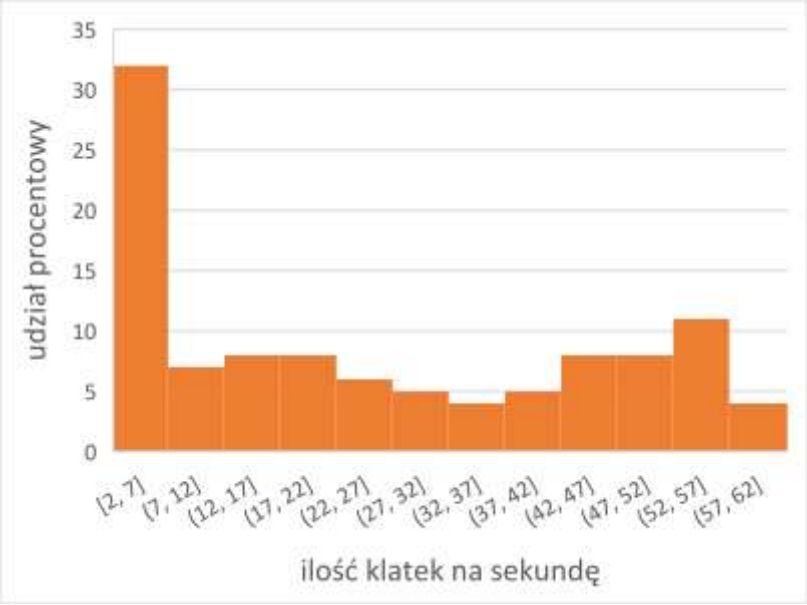
5.3.1 Unity

Jak widać na poniższym zrzucie ekranu (Rys. 5.8), aplikacja wygląda dokładnie tak samo jak jej pierwsza wersja z pominięciem renderowania cieni obiektów na płaszczyźnie. Zarówno obiekty w oddali, jak również te blisko kamery nie rzucają cieni na podłoże.



Rys. 5.8 Widok aplikacji Unity 1.2. Źródło: Opracowanie własne.

Różnica w ilości wyświetlanych klatek, między uruchomieniem aplikacji z cieniowaniem i bez cieniowania jest widoczna na pierwszy rzut oka. Wyświetlany obraz jest ewidentnie bardziej płynny, co widać przy wykonywaniu szybkich ruchów kamerą. Klatki są renderowane bez efektu asynchronizacji akcji.

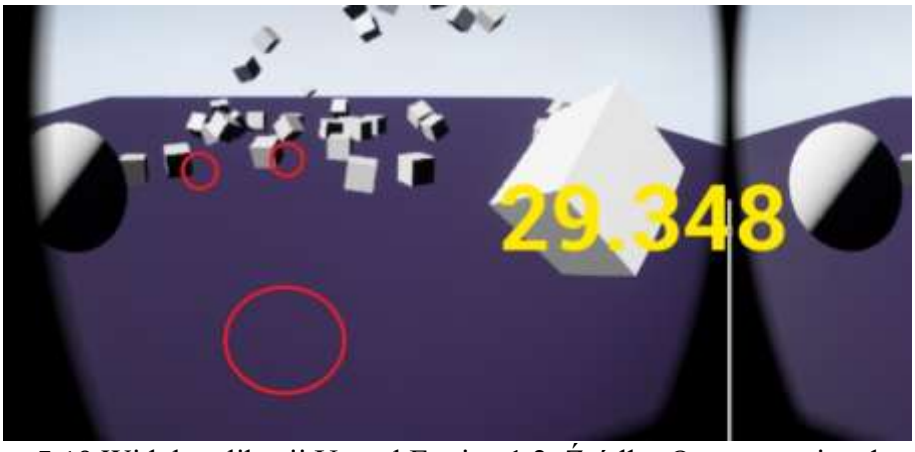


Rys. 5.9 Histogram ilości klatek na sekundę Unity 1.2. Źródło: Opracowanie własne.

Zmiana polega przede wszystkim na przesunięciu się górnej granicy wartości fps (Rys. 5.9). Poprzednio był to poziom około 32 fps, tym razem osiągnięto maksymalną wartość sześćdziesięciu klatek na sekundę (w standardowym trybie jest to wartość docelowa, górna granica dla urządzeń mobilnych). Podobnie jak wcześniej, na ostatnim etapie działania częstotliwość spadła do 2 fps, a w trakcie działania zanotowano znaczące wahania tego parametru.

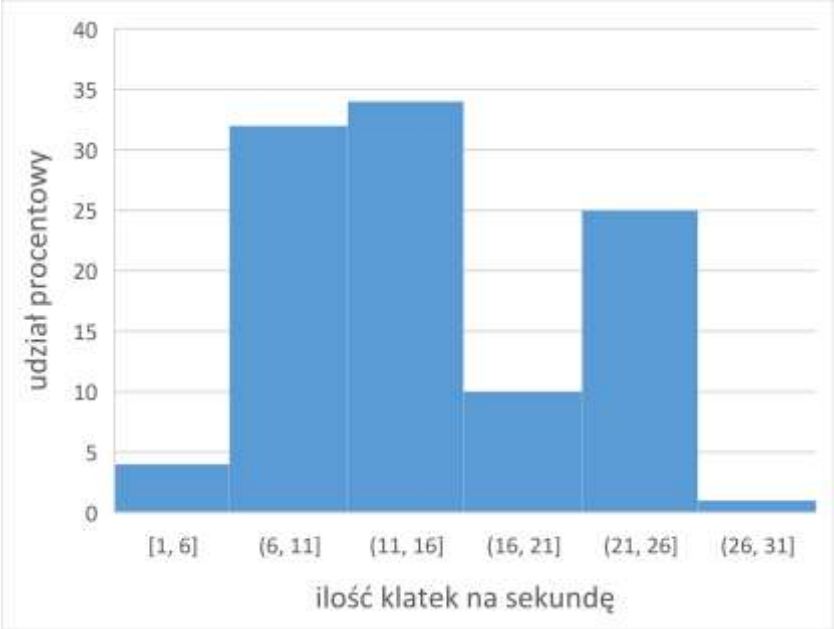
5.3.2 Unreal Engine

Brak wyświetlania cieni nie spowodował poprawienia jakości renderowania krawędzi (Rys. 5.10). Taki efekt został zaobserwowany na etapie implementacji w przypadku ograniczenia elementów wykorzystujących cieniowanie, co przełożyło się na wyższą jakość cieni pozostałych obiektów.



Rys. 5.10 Widok aplikacji Unreal Engine 1.2. Źródło: Opracowanie własne.

Natomiast zmiany ilości wyświetlanych klatek ponownie zachowują względną stabilność. Przez zdecydowaną większość czasu (ponad 60%) są to wartości w przedziale od sześciu do szesnastu klatek (Rys. 5.11). W początkowym etapie ze względu na brak cieniowania, górna granica fps przesunęła się do około 30.

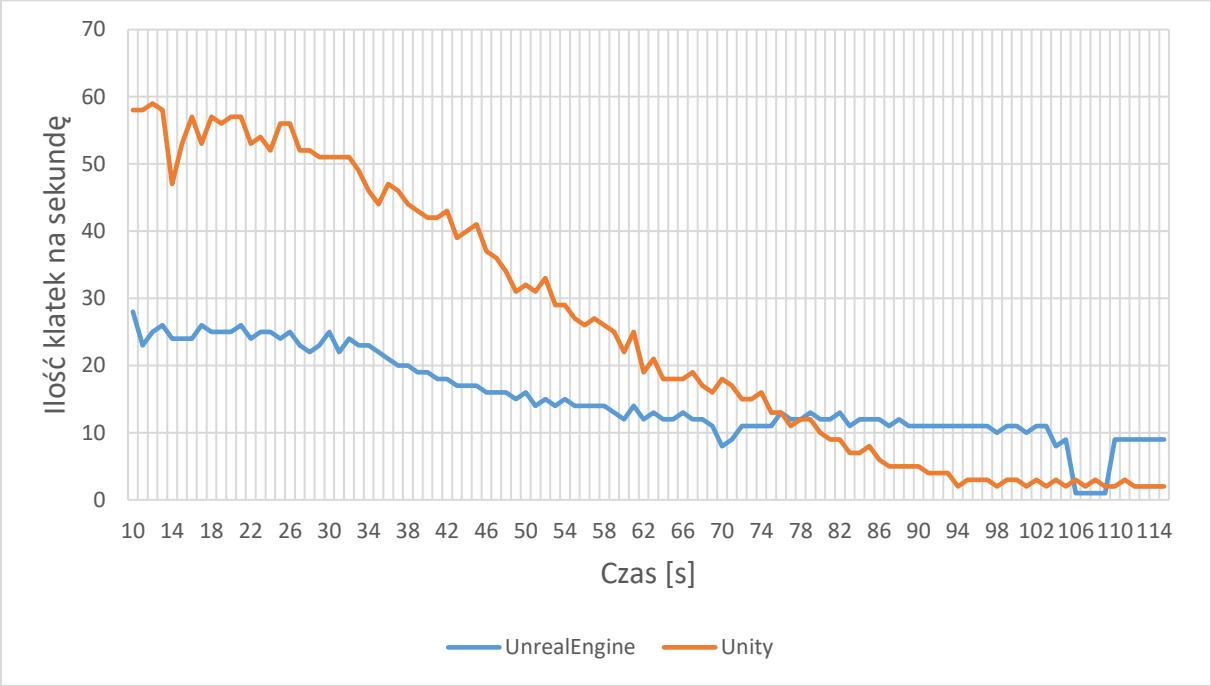


Rys. 5.11 Histogram ilości klatek na sekundę Unreal Engine 1.2. Źródło:Opracowanie własne.

Tym razem przez krótki, ciągły okres zaobserwowano spadek ilości wyświetlanych klatek poniżej sześciu, jednak poza tym incydentem wartość nie spadała poniżej 9 klatek, w związku z tym sytuacja wygląda bardzo podobnie jak w przypadku pierwszej aplikacji.

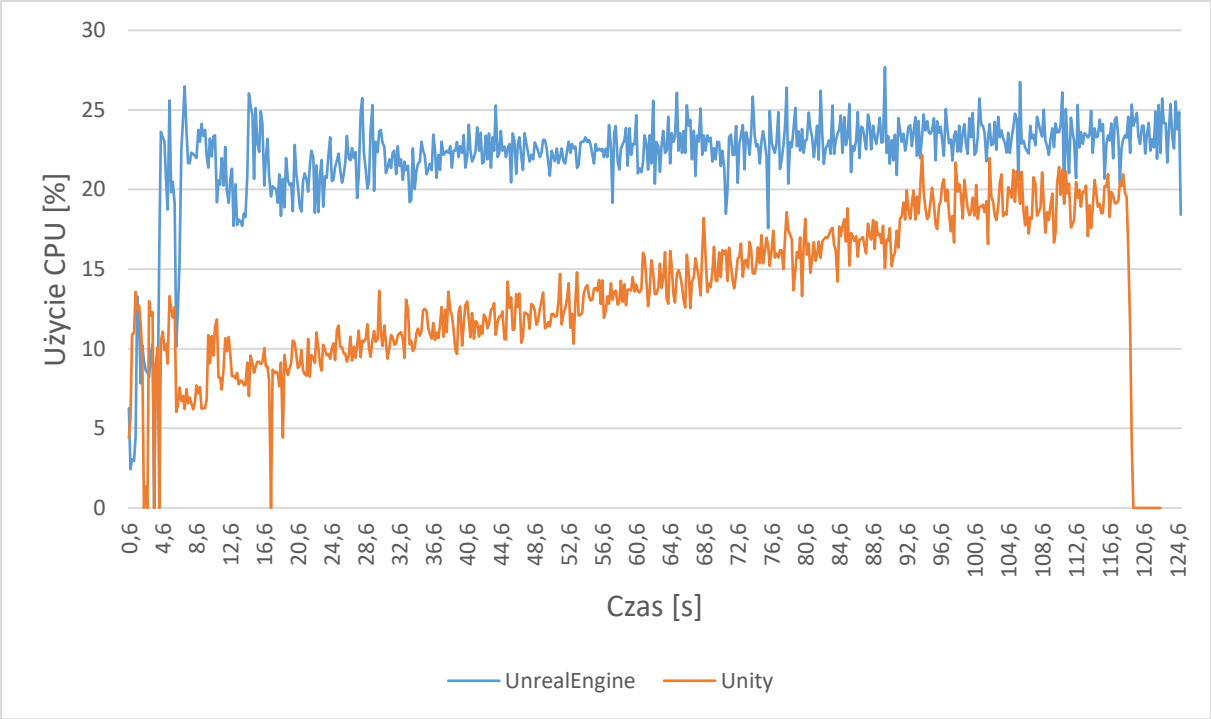
5.3.3 Porównanie

To porównanie ma na celu zaobserwowanie jak duże obciążenie dla podzespołów stanowi renderowanie cieni oraz w jakim stopniu oddziałuje ono na wydajność analizowanych silników. Ponadto możliwe będzie odniesienie otrzymanych wyników do wyników porównania dla pierwszej aplikacji.



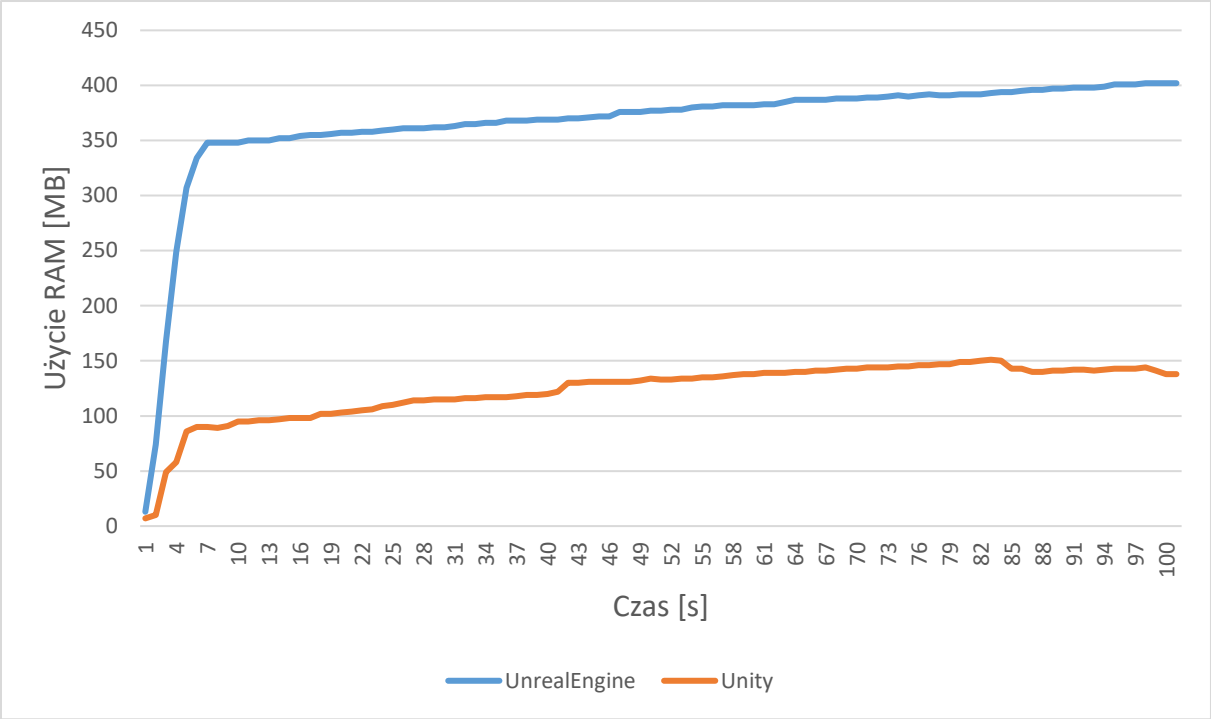
Rys. 5.12 Ilość klatek na sekundę, aplikacja nr 1.2. Źródło: Opracowanie własne.

Poprzednio zaobserwowana tendencja dla ilości wyświetlanych klatek wygląda praktycznie identycznie w przypadku aplikacji 1.2. Ponownie Unity zapewnia płynny start i co ciekawe, niższe wartości fps w stosunku do aplikacji skompilowanej przy użyciu Unreal Engine zaczęły być notowane po upływie dokładnie tego samego czasu, czyli 75 sekund (Rys. 5.12). Następnie Unity stabilizuje się na poziomie 2-3 klatek, a Unreal Engine, poza widocznym drastycznym spadkiem zasadniczo utrzymuje się na pułapie 9 klatek (Rys. 5.12).



Rys. 5.13 Użycie procesora, aplikacja nr 1.2. Źródło: Opracowanie własne.

W przypadku użycia procesora, tym razem to Unreal Engine wymagał większych zasobów (Rys. 5.13). Prawdopodobnie jest to związane z większą ilością wyświetlanych klatek, co przekłada się na konieczność bardziej intensywnego procesowania obiektów i ich zachowań. Różnicą między silnikami w tym aspekcie jest zwiększanie użycia procesora przez Unity wraz z przyrostem ilości obiektów w scenie, natomiast Unreal Engine utrzymuje stały poziom użycia. Jest to ciekawy fakt, szczególnie ze względu na ciągłe generowanie kolejnych obiektów, które powinno w naturalny sposób zwiększać użycie procesora. Zaobserwowano dużo większe wykorzystanie procesora dla Unreal Engine, niż przy pierwszym teście, co jest wyraźnie powiązane z wartością parametru fps. Natomiast dla Unity zmiany następowały w zasadzie identycznie jak w pierwszej aplikacji.



Rys. 5.14 Użycie pamięci RAM, aplikacja nr 1.2. Źródło: Opracowanie własne.

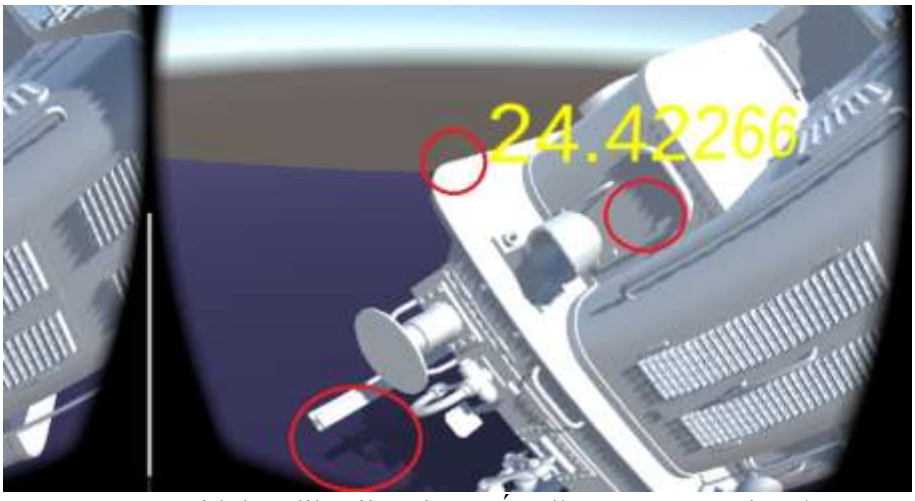
Użycie pamięci RAM dla obu silników spadło o około 100MB, co świadczy o sporym obciążeniu generowanym przez procesowanie cieni. Podobnie jak w pierwszym teście, użycie pamięci RAM utrzymuje się na stabilnym poziomie i jest dużo wyższe dla Unreal Engine (Rys. 5.14).

5.4 Aplikacja nr 2

Aplikacja spełniła swój cel i podczas jej działania można było zaobserwować oczekiwane wahania wydajności. Znika i pojawianie się modelu w polu widzenia kamery wywołało spodziewane zmiany w zakresie ilości wyświetlanych klatek, a także pozostałych badanych parametrów.

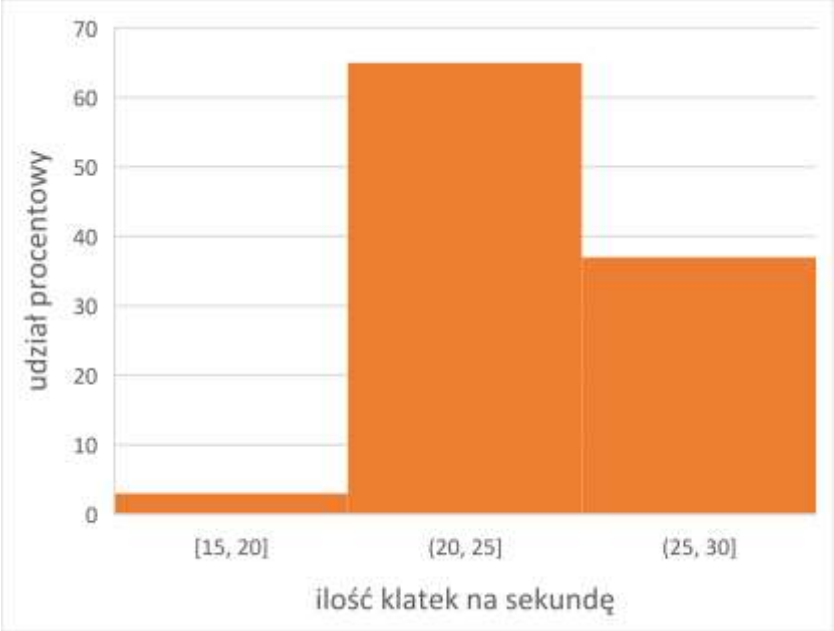
5.4.1 Unity

W przypadku silnika Unity model od strony wizualnej prezentuje się bardzo dobrze. Jego krawędzie są ciągłe, bez anomalii w postaci nierówności lub zanikania. Cienie rzucane przez model na płaszczyznę charakteryzują się wysoką rozdzielczością i wyglądają naturalnie. Problemy występują natomiast w przypadku cieniowania powierzchni obiektu, gdzie jak widać cienie są „poszarpane” co nie oddaje w realistyczny sposób efektu oświetlania prostej krawędzi (Rys. 5.15).



Rys. 5.15 Widok aplikacji Unity 2. Źródło: Opracowanie własne.

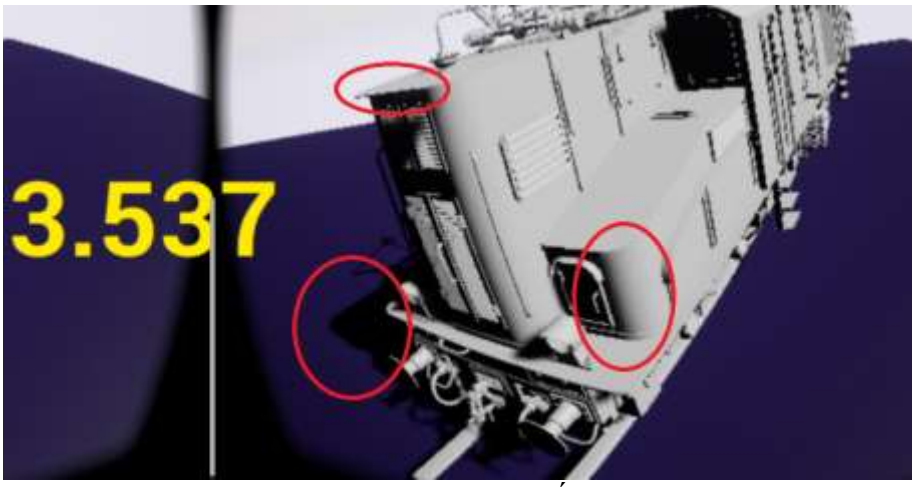
Podczas praktycznie całego czasu testu wartości fps mieściły się w przedziale od 20 do 30 (Rys. 5.16). Jako, że wykonywana była tylko jedna, powtarzalna operacja (obrót modelu), tak niewielki rozrzut wartości nie jest zaskakujący. Przy tak wysokim poziomie skomplikowania modelu (około 150 tysięcy wielokątów) osiągnięte rezultaty są zadowalające.



Rys. 5.16 Histogram ilości klatek na sekundę Unity 2. Źródło: Opracowanie własne.

5.4.2 Unreal Engine

Ponownie bardzo widoczne są nieregularne krawędzie, występuje efekt schodkowy. Jednak w przypadku jakości cieniowania rezultaty są bardzo pozytywne. Problemy z cieniowaniem dużej ilości obiektów w poprzednich testach nie miały miejsca w tym przypadku, ponieważ tym razem był to tylko jeden, duży obiekt. Cienie rzucane na płaszczyznę są wyraźne i naturalne, tak jak przy Unity. Co więcej samo cieniowanie obiektu wygląda bardzo dobrze. Zaokrąglone ściany modelu charakteryzują się płynnymi przejściami intensywności cienia. Cienie są naturalne i odpowiadające rzucającym je elementom (Rys. 5.17).



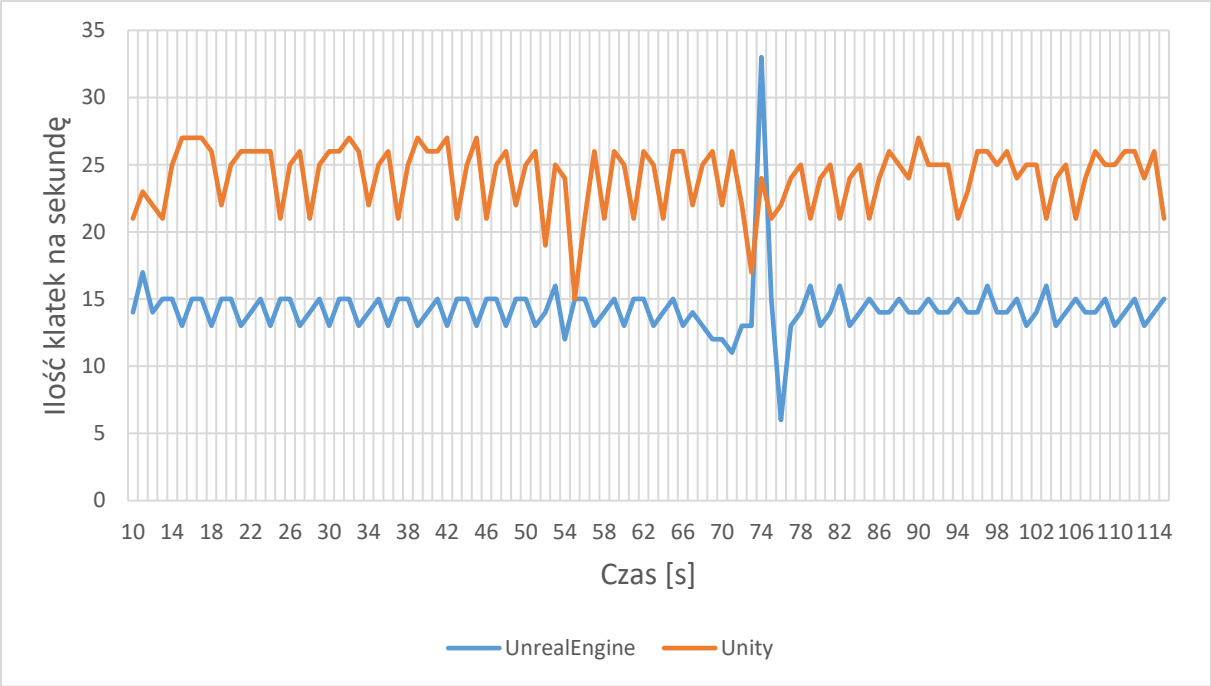
Rys. 5.17 Widok aplikacji Unreal Engine 2. Źródło: Opracowanie własne.

Jeśli natomiast chodzi o płynność wyświetlanego obrazu to ponownie zaobserwowano bardzo małe rozbieżności w ilości klatek na sekundę. Blisko przez 100% czasu mieściły się one w zakresie od 11 do 16 klatek (Rys. 5.18). Są to jednak wartości na tyle małe, że obraz bardzo wyraźnie przeskakuje między kolejnymi klatkami, co utrudnia komfortową nawigację w scenie.

Rys. 5.18 Histogram ilości klatek na sekundę Unreal Engine 2. Źródło: Opracowanie własne.

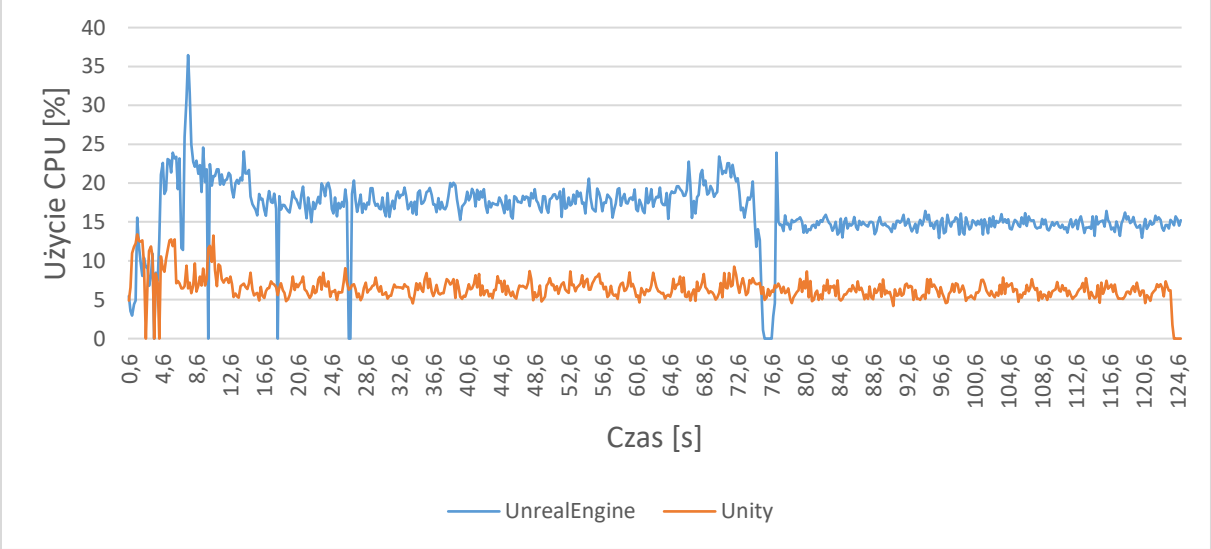
5.4.3 Porównanie

Różnice między oboma silnikami zanotowane po testach aplikacji to w głównej mierze większa płynność Unity i wyższa jakość cieniowania modelu ze strony Unreal Engine. Oczekiwanym zjawiskiem było uformowanie się wykresu ilości klatek w czasie w sposób przypominający sinusoidę. Te przewidywania wiązały się z faktem naprzemiennego pojawiania się i znikania użytego modelu w polu widzenia kamery. Rzeczywiście wartości tego parametru regularnie wzrastają i spadają, jednak w przypadku Unity wahania są blisko trzykrotnie większe, niż dla Unreal Engine (Rys. 5.19). Jest to kolejne potwierdzenie większej stabilności tego silnika, ale również istotnie słabszych osiągów.

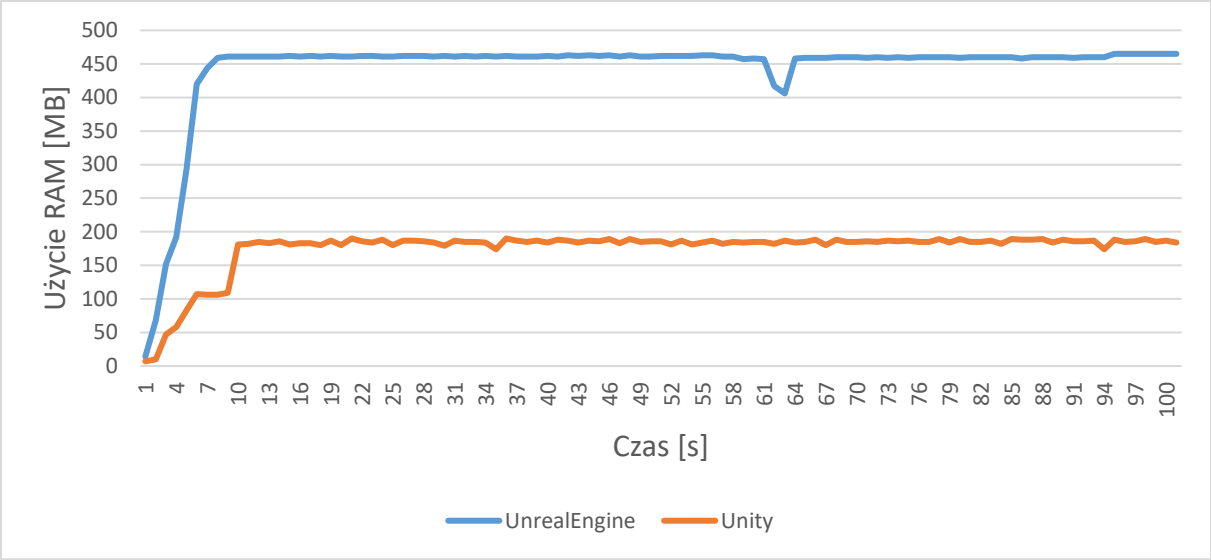


Rys. 5.19 Ilość klatek na sekundę, aplikacja nr 2. Źródło: Opracowanie własne.

Wykorzystanie procesora osiąga wyższe wartości dla Unreal Engine i jest to na przestrzeni całego czasu testu wynik około dwa razy większy niż dla Unity (Rys. 5.20). Oba silniki charakteryzuje stabilizacja użycia CPU, która wynika przede wszystkim z powtarzalnej mechaniki sceny. Dla Unreal Engine zauważalny jest również związek między ilością wyświetlanych klatek a użyciem procesora. W 76 sekundzie wzrost wartości fps powoduje spadek wykorzystania procesora do zera. Jest to o tyle ciekawe zjawisko, że dla Unity w takich sytuacjach zaobserwowano raczej utrzymanie poziomu użycia lub nawet wzrost.



Rys. 5.20 Użycie procesora, aplikacja nr 2. Źródło: Opracowanie własne.



Rys. 5.21 Użycie pamięci RAM, aplikacja nr 2. Źródło: Opracowanie własne.

Różnica w eksploatacji pamięci RAM tak jak w poprzednich testach wynosi mniej więcej 250MB. W tym aspekcie również zauważalne są większe wahania wartości dla Unity, mimo że wciąż jest to poziom zdecydowanie stabilny.

5.5 Wyniki zbiorcze

Poniższa tabela (Tabela 5.2) prezentuje zbiorcze wyniki wszystkich przeprowadzonych badań. Kolumna „Stosunek" zawiera iloraz wartości danego atrybutu dla Unity i Unreal Engine. Kolor zielony wskazuje na przewagę Unity w danym aspekcie, natomiast czerwony oznacza przewagę Unreal Engine. Kolor żółty informuje, iż oba silniki charakteryzuje podobna wydajność.

	Aplikacja nr 1.1		Stosunek	Aplikacja nr 1.2		Stosunek	Aplikacja nr 2		Stosunek
	Unreal Engine	Unity		Unreal Engine	Unity		Unreal Engine	Unity	
Mediana FPS	11	15	1,36	12	21	1,75	14	24	1,71
Stabilność FPS	76,86%	13,11%	0,17	45,16%	11%	0,24	90,55%	89,52%	0,99
Średnie użycie CPU	11,68%	14,49%	1,24	21,99%	13,49%	0,61	16,20%	6,47%	0,4
Maksymalne użycie CPU	19,79%	23,11%	1,16	27,70%	22,18%	0,80	36,47%	13,41%	0,37
Średnie użycie RAM	489MB	201MB	0,41	366MB	123MB	0,34	444MB	175MB	0,40
Maksymalne użycie RAM	533MB	250MB	0,47	402MB	151MB	0,38	465MB	190MB	0,41
Średnie użycie baterii	542mA	560mA	1,03	549mA	493mA	0,90	639mA	669mA	1,05
Głosy użytkowników	1	19	X	1	19	X	0	20	X

Tabela 5.2 Wyniki zbiorcze. Źródło: Opracowanie własne

Stabilność FPS to parametr wskazujący procentową część całego czasu uruchomienia, w której odchylenie wartości FPS od mediany było mniejsze niż 20%. Pozostałe miary to standardowe charakterystyki wskazujące poziom obciążenia urządzenia.

5.6 Wnioski

Wbrew oczekiwaniom wyniki przeprowadzonych badań jednoznacznie wskazują jeden z testowanych silników jako wydajniejszy. Unity w każdym teście uzyskiwało lepsze rezultaty w zakresie częstotliwości wyświetlania klatek. Były to wyniki aż o 36-75% lepsze, co w przypadku tego parametru jest bardzo dużą różnicą. Dla aplikacji nr 2 Unreal Engine osiągnął medianę FPS na poziomie 14 (bardzo skokowe przejścia między kolejnymi klatkami), podczas gdy w Unity była to wartość 24 (taka wartość charakteryzuje większość filmów kinowych). W praktyce kontrast w komforcie użytkowania jest bardzo mocno widoczny.
Jeśli natomiast chodzi o odchylenia od mediany FPS, to Unreal Engine okazał się dużo bardziej stabilny podczas rosnącej ilości obiektów. Jedną z najciekawszych obserwacji jest jego zdolność do utrzymywania się powyżej pewnej dolnej granicy FPS, mimo coraz większego obciążenia. Tymczasem Unity zaraz po uruchomieniu i rozpoczęciu generowania obiektów zaczyna notować praktycznie jednostajny spadek FPS do momentu osiągnięcia wartości 2-4 klatek na sekundę.
Średnie użycie procesora to miara, dla której najtrudniej zauważyć pewną tendencję. Z analiz wynika, iż Unity charakteryzuje się dużo lepszym wykorzystaniem tego zasobu, ponieważ osiągając istotnie wyższą płynność obrazu używa zbliżonej lub dużo niższej mocy CPU. Kluczowy okazał się test aplikacji nr 2, gdzie Unreal Engine wykorzystywał średnio 16,20% mocy procesora. W tym samym teście Unity potrzebowało jedynie 6,47% CPU do uzyskania prawie dwa razy większej mediany FPS.
Wykorzystanie pamięci RAM we wszystkich testach wskazuje na bardzo wyraźną przewagę Unity. Za każdym razem średnia różnica między poziomem użycia wynosiła około 250MB, co jest równe maksymalnemu zapotrzebowaniu Unity na przestrzeni wszystkich testów. Unreal Engine osiągał użycie nawet do 533MB, ze średnimi niewiele poniżej tej wartości. Nawet przy dzisiejszych konfiguracjach urządzeń mobilnych, tak duże zapotrzebowanie jest dużym problemem, ponieważ większość z nich dysponuje mniej więcej 1GB wolnej pamięci RAM przy ograniczonym dostępie do usług i aplikacji.
Jedynym aspektem, w którym nie zaobserwowano praktycznie żadnych różnic było użycie baterii, jednak był to wynik spodziewany jeszcze przed wykonaniem badań.
Poza aspektami wydajnościowymi analizie poddano również kwestie wizualne. Występujące w pracy porównania jednoznacznie wskazują aplikację skompilowaną przy użyciu Unity jako prezentującą wyższy poziom graficzny. Zarówno jakość krawędzi, jak i renderowane cienie wyglądały po prostu lepiej. Jedynie model z aplikacji nr 2 charakteryzował się lepszym cieniowaniem dla Unreal Engine. Oceny użytkowników prawie w 100% (96,6%) wskazują aplikacje Unity jako cechującą się lepszą grafiką.
Warto również przypomnieć, że sam proces instalacji i konfiguracji silnika pod kątem Google Cardboard jest dużo prostszy i przebiega zdecydowanie sprawniej dla Unity. Co więcej ilość wątków na forach Unity odnoszących się do tej platformy jest istotnie większa. Dużym problemem Unreal Engine okazało się też sterowanie (pozycja startowa kamery ustawiona w niewłaściwym kierunku), które dla wirtualnej rzeczywistości jest elementem kluczowym.
Jeżeli chodzi o sposób implementacji mechanik rozgrywki (za pomocą skryptów czy systemu Blueprints) to jest to z pewnością kwestia indywidualna i trudno jednoznacznie wskazać lepszy. Autorowi tej pracy bardziej przypadł do gustu system oparty o skrypty (Unity), dlatego można jedynie zakładać, że osobom zajmujących się programowaniem bliżej będzie właśnie do tego

rozwiązania, natomiast pozostali prawdopodobnie lepiej będą się czuli z graficznym systemem Unreal Engine.

6. Podsumowanie

Biorąc pod uwagę wszystkie wymienione wyżej argumenty, Unity jest w przypadku platformy Google Cardboard silnikiem bardziej przystępnym, a do tego zapewniającym lepszą wydajność, mniejsze wykorzystanie zasobów, szybszy proces produkcji i testowania jak również lepsze rezultaty od strony wizualnej. Dużo większy udział Unity w rynku gier mobilnych nie jest przypadkowy, jest wynikiem bardzo dobrego przystosowania do tego typu produkcji we wszystkich aspektach. Unreal Engine jest z kolei silnikiem oferującym bardzo szerokie możliwości, w tym graficzne, które jednak stają się zauważalne przy dużych, wymagających projektach na platformy zapewniające wysokie osiągi. Takimi nie są na pewno gry mobilne na Google Cardboard.
Zgodnie z założeniami, przeprowadzona analiza pozwoliła udzielić odpowiedzi na pytanie postawione we wstępie pracy – który z silników jest lepszy? Otóż w odniesieniu do Google Cardboard lepszym silnikiem jest Unity.

Bibliografia

[1] C. P. Burdea G., Virtual Reality Technology, New Jersey: Wiley, 2003.

[2] M. K. Martel E., „Controlling VR games: control schemes and the player experience," *Entertainment Computing,* tom 21, pp. 19-31, 2017.

[3] L. M. S. P. W. P. W. M. Pflander M., „Virtual reality based simulators for spine surgery: a systematic review," 29 maj 2017. [Online]. [Data uzyskania dostępu: 24 czerwiec 2017].

[4] K. O. D. Q. F. J. R. D. D. P. O. D. T. O. T. S. Harrington C., „Development and evaluation of a trauma decision-making simulator in Oculus virtual reality," *The American Journal of Surgery,* 2017.

[5] B. M. M. G. R. Roy E., „The need for virtual reality simulators in dental education: A review," *The Saudi Dental Journal,* tom 29, pp. 41-47, 2017.

[6] Z. M., „From visual simulation to virtual reality to games," *Computer,* tom 38, nr 9, pp. 25 - 32, 2005.

[7] A. P. T. L. B. Rizzo, „Virtual Reality Goes to War: A Brief Review of the Future of Military Behavioral Healthcare," *Journal of Clinical Psychology in Medical Settings,* tom 18, nr 2, pp. 176-187, 2011.

[8] N. S. G. K. A. D. T. A. Vora J., „Using virtual reality technology for aircraft visual inspection training: presence and comparison studies," *Applied Ergonomics,* tom 33, nr 6, pp. 559-570, 2002.

[9] R. M. L., Narrative as Virtual Reality: Immersion and Interactivity in Literature and Electronic Media, Baltimore: Johns Hopkins University Press Baltimore, 2001.

[10] M. C. Arora G., Reżyser, *Clouds Over Sidra.* [Film]. Syria.2015.

[11] G. F. G. D. W. R. Z. P. Bun P., „Low – Cost Devices Used in Virtual Reality Exposure Therapy," w *ICTE,* Ryga, 2016.

[12] P.-K. R. v. R. A. H. O. v. d. G. M. V. W. Counotte J., „High psychosis liability is associated with altered autonomic balance during exposure to Virtual Reality social stressors," *Schizophrenia Research,* tom 184, pp. 14-20, 2017.

[13] R. T. V. Greenlight Insights, „Virtual Reality Industry Report," San Francisco, 2017.

[14] I. D. Corporation, „Worldwide Semiannual Augmented and Virtual Reality Spending Guide," Framingham, 2016.

[15] W. S. W. D. M. N. U. C. O. W. N. R. Raskind C., Strategy Analytics, 13 4 2016. [Online]. Available: https://www.strategyanalytics.com/strategy-analytics/news/strategy-analytics-press-releases/strategy-analytics-press-release/2016/04/13/strategy-analytics-oculus-rift-htc-vive-sony-playstation-vr-will-dominate-$895-million-virtual-reality-headset-market-i. [Data uzyskania dostępu: 15 6 2017].

[16] S. P. S. Trenholme D., „Computer game engines for developing first-person virtual environments," *Virtual Reality,* tom 12, nr 3, pp. 181-187, 2008.

[17] Epic Games, [Online]. Available: https://docs.unrealengine.com. [Data uzyskania dostępu: 15 6 2017].

[18] Crytek, [Online]. Available: https://www.cryengine.com. [Data uzyskania dostępu: 15 6 2017].

[19] N. A., „Kotaku," 2 9 2016. [Online]. Available: http://kotaku.com/amazon-releases-its-own-game-engine-for-free-1757995787. [Data uzyskania dostępu: 15 6 2017].

[20] [Online]. Available: http://www.cocos2d-x.org/. [Data uzyskania dostępu: 15 6 2017].

[21] Locomotive_works, Turbo Squid, [Online]. Available: https://www.turbosquid.com/FullPreview/Index.cfm/ID/784982. [Data uzyskania dostępu: 20 6 2017].