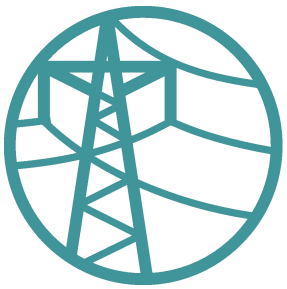


WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Kacper Rudź

Porównanie silników Unreal Engine i Unity pod kątem tworzenia gier

Praca dyplomowa magisterska

Opiekun pracy:
dr inż. Sławomir Samolej prof. PRz

Rzeszów, 2024

Spis treści

Wykaz symboli, oznaczeń i skrótów (opcjonalny)

1 ÷ 2 stron wykaz ważniejszych symboli i oznaczeń (jeśli jest potrzebny).

1. Wstęp

Silniki graficzne są technologią, która rozwijała się bardzo szybko. Pierwsze gry, które używały silników graficznych z trójwymiarową grafiką pojawiły się w drugiej połowie lat 90 ubiegłego wieku. Wcześniej silniki graficzne były używane w branży filmowej w celu stworzenia prostych efektów specjalnych. Silniki graficzne w tamtych czasach były bardzo proste i nie pozwalały na wiele. Zwykle ograniczeniem były możliwości obliczeniowe komputerów graczy. Drużyny tworzące gry zwykle składały się z kilkunastu osób, a budżety gier były małe w porównaniu do budżetów branży filmowej.

Wraz z biegiem czasu rozwój silników graficznych przyśpieszał, do czego przyczynił się rozwój procesorów oraz kart graficznych. Z każdym rokiem silniki graficzne były coraz bardziej zaawansowane i pozwalały na obsługę tekstur większej rozdzielczości, wyższej jakości efektów cząsteczkowych, ulepszono oświetlenie wprowadzając oświetlenie globalne czy ray tracing. Równolegle rozwijały się techniki animacji szkieletowych oraz programy do modelowania, teksturowania oraz animowania. W tym całym procesie nie zapomniano o poprawianiu wydajności bibliotek graficznych co przełożyło się na wydajność silników graficznych, które na nich operowały.

Rozwój grafiki komputerowej musiał spowolnić po pewnym czasie. Już w roku 2016 pewne gry takie jak 'Wiedźmin 3' studia CD projekt Red, czy 'Forza Horizon 3' studia Playground Games były uznawane za bliskie fotorealizmu. Jednak za najbardziej realistyczną grę pod względem grafiki jest gra 'Red Dead Redemption 2' studia Rockstar Games z roku 2018.

W między czasie firmy tworzące gry rosły. Powstawały nowe studia, a inne były wchłaniane przez największe korporacje, ale nadal powstawały gry tworzone przez małe grupy osób. Aktualnie zespoły tworzące gry można podzielić na kilka grup:

- Niezależni twórcy – jedna lub kilka osób tworzy gry i wydaje je zwykle samodzielnie. Często korzystają z gotowych silników graficznych lub używają swoich bardzo prostych silników. Gry wykonane przez twórców niezależnych: Factorio, Stardew Valley i pierwsze wersje gry Minecraft

- Małe studia gier – niewielkie firmy zatrudniające do kilkunastu osób. Skończony produkt jest wydawany z pomocą wydawcy, rzadko samodzielnie. Bardzo rzadko te firmy korzystają z własnych silników graficznych. Przykładowe gry zrobione

przez małe studia: Celeste, Ori and The Blind forest, Terraria

- Średnie studia gier – już większe firmy zatrudniające nawet kilkaset osób. W przeszłości spora część korzystała z własnych silników graficznych. Przykładowymi grami stworzonymi przez średnie studia są: Biomutant, Baldurs Gate 3.

- Duże i bardzo duże studia gier – posiadają nawet do kilku tysięcy pracowników. Bardzo często tworzą gry na własnych silnikach graficznych. Studia takie potrafią pracować nad kilkoma grami jednocześnie. Przykładowe gry stworzone przez duże studia: Wiedźmin 3 Dziki Gon, gry serii Far Cry i Assassins Creed.

Większość twórców używa własnych silników graficznych lub silników komercyjnych – do modelowania, animacji i teksturowania zwykle są używane narzędzia dostępne na rynku. Małe studia i średnie studia bardzo rzadko używały własnych rozwiązań i zwykle używały gotowych rozwiązań oferowanych przez firmy zajmujące się tworzeniem i rozwojem silników graficznych. Duże studia posiadały własne silniki graficzne, na których były tworzone gry. Jednak duże studia zaczynają przenosić się na silniki innych twórców. Powodem takiej sytuacji jest koszt utrzymywania i rozwoju własnych rozwiązań.

Gry w dzisiejszych czasach są tworzone pod wiele platform. Platformą w tym przypadku są rodziny systemów operacyjnych i konsole. Twórcy konsol co kilka lat muszą wydać nowe konsole, ze względu na postęp technologiczny związany z podzespołami. Zmiana jest na tyle spora, że wymaga znacznych zmian w systemach operacyjnych konsol, co w konsekwencji ma wpływ na wydajność i stabilność gier. Twórcy silników graficznych muszą aktualizować swoje silniki graficzne w celu minimalizacji wpływu zmian na stabilność i wydajność gier.

2. Cel pracy

Tematem pracy jest porównanie silników Unreal Engine i Unity pod kątem tworzenia gier. W tym celu należy wskazać różnice w budowie silnika. Zostały wskazane różnice w pisaniu skryptów, użyte języki programowania oraz różnice w architekturze silników i różnice związane z kolejnością wykonywania metod i funkcji.

Ważnym elementem gier jest stabilność stworzonej aplikacji oraz ilość zasobów wymaganych do prawidłowego funkcjonowania gry. W tym celu został przeprowadzony test polegający na rozłożeniu w silniku kilku tysięcy kopii jednej postaci i poruszaniu się kamery po ustalonej wcześniej ścieżce przez kilkadziesiąt minut. Kopie postaci wykonują zapętlone animacje – będzie 10 animacji i każda kopia będzie wykonywała jedną z tych animacji. W trakcie testu zostały zebrane informacje na temat ilości klatek na sekundę, zapotrzebowaniu na pamięć RAM oraz pamięci VRAM i ile czasu w danej klatce aplikacja używała procesora i procesora graficznego. Dane zostały zebrane z kilku różnych komputerów – komputery osobiste i laptopy – o różnych specyfikacjach, a następnie dane zostały przeanalizowane. Analizowane dane zostały ocenione pod względem różnych kryteriów.

3. Teoretycznie o silnikach graficznych

Silniki graficzne nie są wymagane przy tworzeniu gier komputerowych, ale znacząco przyśpieszają one proces twórczy. Częścią wspólną większości gier komputerowych są dźwięk, elementy graficzne, takie jak modele, tekstury, animacje, elementy interfejsu użytkownika oraz wszelkiego rodzaju skryptów i sztucznej inteligencji. Dźwięk, animacje, modele i tekstury są opracowywane w wyspecjalizowanych programach.

Silniki graficzne są używane głównie przy tworzeniu modeli, animacji i tekstur oraz jako solidna podstawa pod kod gry. Przykładowo program do grafiki komputerowej 3D „Blender" posiada 3 silniki graficzne – EEVEE, Cycles i Workbench. Każdy z nich jest wyspecjalizowany w własnej dziedzinie – EEVEE jako renderer casu rzeczywistego, Cycles jest używany do renderowania obrazów w wysokiej jakości, Workbench jest używany do podglądów, modelowania i animacji. Jako podstawę do kodu gry można użyć silnika graficznego podobnego do Unity lub Unreal Engine.

3.1. Założenia silników graficzncyh używanych w grach

Silniki graficzne używane w grach można podzielić na kilka większych wyspecjalizowanych modułów [1]:

- Moduł obsługi urządzeń wejścia – problemem urządzeń wejścia jest ich różnorodność. Jest wiele urządzeń, dzięki którym można sterować postaciami w grach, a jednymi z najczęściej używanych są klawiatury, myszki oraz pady. Poza tym są jeszcze touchpady, czujniki pokroju żyroskopów i wiele innych. Zadaniem tego modułu jest udostępnienie API, które pozwoli w prosty sposób oprogramować przetwarzanie wejść. Wczytywanie danych z wejść może odbywać się na dwa różne sposoby – pollin jako statystyka ze zmiany wejść i przerwania jako reakcja programu na zmianę stanu wejścia.

- Matematyka – matematyka w silnikach graficznych polega głównie na optymalizacji obliczeń macierzy 4x4 lub 3x3, obliczeń związanych z wektorami i kwaternionami. Macierze są wykorzystywane do reprezentacji transformacji – rotacja, skala i pozycja – względem pewnego układu odniesienia. Obliczenia wektorowe są potrzebne do obliczeń związanych z fizyką i położeniem obiektu. Kwaterniony są używane zamiast macierzy transformacji, z racji zjawiska gimbal lock, które

polega na nałożeniu się na siebie osi rotacji.

- Fizyka – moduł ten obejmuje głownie kinematykę. Zwykle fizyka nie jest idealna i w swoich obliczeniach używa wielu uproszczeń. Jednak prawa fizyki działające na obiekt w grze powinny być edytowalne – zmiana parametrów takich jak tarcie, prędkość, przyśpieszenie, prędkość odśrodkowa itp. Jako obliczenia fizyczne zalicza się również kolizje. Kolizje powinny być zdarzeniami i użytkownik silnika graficznego powinien mieć możliwość wprowadzenia własnego kodu w odpowiedzi na zdarzenie. [2]

- Obsługa plików i tekstu – może wydawać się rzeczą banalną, jednak silniki graficzne używane do tworzenia gier powinny być w stanie obsługiwać różne formaty plików oraz tekst. Głównymi formatami plików są formaty związane z modelami graficznymi i teksturami – format fbx jest używany do wyeksportowania modelu wraz z animacjami, png jest używany do przenoszenia tesktur. Obsługa tekstu dotyczy obsługi różnych czcionek – systemowych, ale też dostarczanych przez użytkownika silnika – różnych rozmiarów czcionek i innych właściwości, które może ustawić użyt- kownik silnika graficznego

- Obsługa elementów graficznych i rendering – przez elementy graficzne można rozumieć: elementy UI, wszelkiego rodzaju bryły wraz z animacjami, efekty cząsteczkowe i oświetlenie. Każdy z tych elementów musi zostać uwzględniony w procesie renderingu. Bryły w silnikach graficznych składają się z trójkątów, ponieważ każde dowolne 3 punkty w przestrzeni są współpłaszczyznowe. Do stworzenia trójkąta potrzeba współrzędnych trzech punktów. Poza tym na bryłę mogą być nałożone tekstury – mapy koloru, normalnych itp. Każda bryła może być pod wpływem zjawisk fizycznych zaimplementowanych w silniku.

- Udźwiękowienie – silniki graficzne używane w grach powinny pozwalać na dodanie dźwięku do gry. W tym celu sam silnik powinien obsługiwać różne formaty plików dźwiękowych jak i codeki. Najbardziej powszechnymi rodzajami codeków są: H.264/AVC, AVI, VP9 oraz HEVC (H.256). Dodatkowo silnik graficzny powinien pozwalać na używanie dźwięków przestrzennych oraz dźwięki statyczne – dźwięk, który nie zmienia się w zależności od zmiany położenia gracza. Dźwięk przestrzenny powinien zmieniać się w zależności od położenia gracza od źródła

dźwięku i ta zmiana powinna być odzwierciedlana w urządzeniach audio – o ile te urządzenia obsługują tę funkcję.

- Narzędzia do debugowania i rozwoju – nowoczesne komercyjne silniki graficzne powinny dawać możliwość dodawania własnych narzędzi do silnika. Często zdarza się, że szybciej jest stworzyć narzędzie, które zrobi daną rzecz za twórcę, niż tworzenie rzeczy ręcznie. Silnik graficzny w tym celu powinien udostępniać interfejs, który pozwoli na tworzenie takich narzędzi lub modyfikowanie starych. Kolejną ważną rzeczą są możliwości debugowania kodu. Wśród narzędzi do debugowania powinny się znaleźć narzędzia do sprawdzania zużycia zasobów przez grę. Błędy i wycieki pamięci mogą się zdarzyć nawet najbardziej wprawnym programistom nawet w językach, które teoretycznie temu zapobiegają. Błędy w grach często są ciężkie do znalezienia przy testach – często jest to związane z niedopatrzeniami w obliczeniach fizycznych.

Twórca gier powinien mieć możliwość używania każdego modułu bez ograniczeń. W ten sposób użytkownik silnika graficznego jest w stanie stworzyć grę będąc ograniczonym tylko umiejętnościami. Przykładem używania wielu modułów jednocześnie jest wpływanie przez zjawiska fizyczne na szkielet postaci. Taka postać może wydawać dźwięki w pewncyh punktach animacji. Jest to jeden z wielu przykładów użycia wielu różnych komponentów w tym samym czasie. Ważnym aspektem jest udostępnienie możliwości nanoszenia zmian do wymienionych modułów silnika graficznego przez twórców gier.

3.2. Architektura silnika Unity

Unity Engine jest silnikiem stworzonym i rozwijanym od 2005 roku. Silnik graficzny zdominował rynek małych, średnich gier, gier na telefon oraz jest używany w symulacjach. W Unity do programowania skryptów używa się języka C#, ale istnieje możliwość pisania skryptów za pomocą 'Visual Scripting'. Skrypty w silniku, które używają metod Unity muszą dziedziczyć z 'Monobehaviur' lub klas pochodnych.

Unity posiada bardzo prostą strukturę, a sam silnik pozwala twórcom na nadpisywanie tylko określonych metod, ale jest ich na tyle dużo, że nie wpływa to na elastyczność silnika. Cały silnik jest podzielony na kilka części[3]:

- Scena – w Unity poziomy zostały nazwane scenami. Przy przejściu między scenami obiekty na scenie są usuwane. Jedynym wyjątkiem od reguły jest użycie metody 'DontDestroyOnLoad()', która sprawi, że obiekt zostanie przeniesiony między scenami. Poza tym dane między poziomami mogą być przenoszone poprzez zmienne globalne i pola statyczne.

- Obiekt – obiekty są elementami, które można ustawić na scenie. W obiektach znajdują się komponenty, które mogą mieć różne funkcje. Obiekt na scenie ma podstawowe transformacje takie jak rotacja, skala i przesunięcie. Jeden obiekt może mieć wiele obiektów potomków, które mogą dziedziczyć transformacje.

- Komponenty – komponentem w silniku Unity można podzielić na wiele kategorii: Rigidbody, Collider, Script, Mesh Rendere, AudioSource, Light itp. Każdy element ma swoją funkcję od odtwarzania dźwięku po sterowanie animacjami.

W Unity funkcjonują tzw. 'prefab'y'. Są to szablonowe obiekty, które można umieścić na mapie lub w skrypcie. Jest to mechanika pozwalająca poprzez zmianę szablonu zmienić wszystkie elementy w silniku. Szablony obiektów mogą być bardzo złożone i mogą korzystać z innych obiektów szablonów. Ich stosowanie bardzo przyśpiesza proces tworzenia gier i pozwala na płynną naprawę błędów. W silniku funkcjonuje system pozwalający na wywoływanie metod po pewnym czasie nazwany 'Coroutine'. Mechanizm ten pozwala na odroczenie wykonywania kodu po pewnym czasie lub później w danej klatce. Odroczenie zaczyna się zwykle słowem kluczowym 'yield'.

Kolejnym aspektem, który należy przedstawić w Unity jest 'Unity Flowchart', który przedstawia w jakiej kolejności są wywoływane metody w komponentach i skryp-

tach w silniku. Jest to mechanizm bardzo ważny, ponieważ pozwala programiście na ustalenie kolejności wykonywania programu. Na rysunku 3.1 przedstawiono kolejność wykonywania działań w silniku Unity. Metody i funkcje zostały podzielone na 3 kategorie:

- Metody i funkcje nadpisywalne przez programistę oznaczono kolorem białym
- Funkcje wewnętrzne zaznaczone kolorem szarym z zaokrąglonymi rogami
- Funkcje wewnętrzne wielowątkowe zaznaczone kolorem szarym z ostrymi rogami

Życie obiektu zostało podzielone na kilka etapów:

- Inicjalizacja – grupa metod, które są wywoływane, gdy obiekt zostanie zainicjalizowany, wzbudzony lub zresetowany.
- Fizyka – metody odpowiedzialne za obliczenia związane z fizyką oraz animacjami, które są zależne od fizyki. Blok został podzielony na dwa mniejsze bloki. W pierwszym bloku komponent animacji przetwarza przejścia miedzy animacjami. W drugim bloku są obliczane transformacje kości animacji szkieletowych. Między blokami zachodzi proces przeprowadzania obliczeń fizycznych związanych z kinematyką. Etap zakończony jest metodami związanymi z kolizjami oraz metodą, gdzie jest odroczona metoda 'FixedUpdate'. Pętla fizyczna może zostać wywołana kilka razy w trakcie jednej klatki.
- Zdarzenia wejścia – w tym etapie wywoływane są metody, które odpowiadają za zdarzenia wciśnięcia klawiszy lub ruchu myszą.
- Logika Gry – pierwszą metodą w tym etapie życia obiektu jest metoda 'Update', która jest najczęściej używana do prostych obliczeń logiki gry. Następnie są wywoływane metody, które zostały odroczone. Następnie w etapie jest umieszczony blok odpowiedzialny za aktualizację animacji – zmiana stanów, uruchomienie wydarzeń w trakcie animacji i przeprowadzenie animacji. Na końcu etapu znajduje się metoda 'LateUpdate'
- Render Sceny – pierwszy etap generacji klatki. Grupa metod zawartych w tym etapie służy głownie do wprowadzenia efektów graficznych.

- Render interfejsu użytkownika i debugowania – interfejs użytkownika musi być generowany po reszcie obiektów, aby był widoczny. Dodatkowo przed interfejsem użytkownika są generowane efekty stworzone przez narzędzia debugowania.

- Zakończenie i wstrzymanie klatki 0 w tym etapie znajduje się dwie metody. Jedna z nich jest korutyną w silniku unity, która wznawia działanie na koniec klatki. Druga metoda jest wywoływana tylko wtedy, gdy w silnku zajdzie zdarzenie 'Pauza'.

- Likwidacja obiektu – grupa metod, która jest wywoływana, gdy obiekt został wstrzymany, obiekt został usunięty lub aplikacja została wyłączona.

Każdy etap jest ważny jednak z widzenia programisty należy zastanowić się nad funkcjami poszczególnych bloków, które można nadpisać. Pierwszym etapem jest inicjalizacja, w której znajdują się metody:

- Awake() – metoda wywoływana, przy stworzeniu obiektu lub komponentu. Metoda nie zostanie wywołana, kiedy obiekt nie jest 'przebudzony'. Metoda może zostać wywołana tylko raz.

- OnEnable() – w przeciwieństwie do 'Awake' metoda może być wywoływana wiele razy w trakcie życia obiektu lub komponentu, kiedy pole 'enabled' zmienia wartość z 'false' na 'true'.

- Start() – metoda wywoływana tylko raz w czasie życia obiektu lub metody. Jest ostatnią metodą w etapie inicjalizacji obiektu.

Kolejną rzeczą niewspomnianą w silniku Unity jest konstruktor z języka C#. Jest on wywołany w trakcie tworzenia obiektu przed metodą 'Awake'. Kolejnym etapem życia obiektu w silniku unity jest pętla fizyczna. W niej znajdują się następujące bloki:

- FixedUpdate() – pierwsza metoda w pętli fizycznej, która pozwala na stworzenie własnego kodu, który będzie używał części silnika odpowiedzialnego za fizykę. Przykładem użycia może być zmiana prędkości obiektu w taki sposób, aby ten orbitował wokół innego obiektu.

- Internal physics update – w tym momencie silnik przeprowadza obliczenia fizyczne dla wszystkich aktywnych obiektów i komponentów. W tym czasie obliczane są kolizje, dla których metody zostaną wywołane.

- OnTrigger – zdarzenie w silniku unity podczas kolizji ‘collider-trigger’.
- OnCollision – zdarzenie w silniku unity podczas kolizji typu ‘collider-collider’
- yield Wait on FixedUpdate – kontynuacja działania metody ‘FixedUpdate’ po jej odroczeniu

W trakcie pętli fizycznej znajdują się dwa duże bloki, które są używane w animacjach. W tych blokach znajdują się następujące metody:

- State machine update – silnik przeprowadza aktualizacje maszyny stanów dla kontrolerów animacji. Zmiana stanów maszyn jest przeprowadzana wielowątkowo.
- OnStateMachine Ender/Exit – metody wywoływane podczas zmiany stanów w maszynie stanów. Pozwala programistom na wprowadzenie własnego kodu.
- ProcessGraph – blok odpowiedzialny za przetworzenia grafów używanych w animatorze. Obliczany jest czas, w jakim zajdzie przejście między klatkami animacji. W tym czasie mogą znajdować się wcześniej zdefiniowane zdarzenia, co musi zostać wywołane.
- Fire animation events - system, który pozwala na wywołanie metod, które mogą być wywołane w trakcie wykonywania animacji.
- StateMachineBehaviour callbacks - wywołuje metody w trakcie przejścia między stanami.
- OnAnimationMove – metoda wywołana w przypadku ruchu obiektu na bazie animacji. Ruch ten nazwano ‘Root Motion’, gdzie ‘Root’ oznacza nazwę kości, która odpowiada za pozycję całego obiektu.
- ProcessAnimation – system, który odpowiada za przetworzenie animacji, tak aby była dobrze wyświetlona.
- OnanimationIK - metoda pozwalająca na wstrzyknięcie własnego kodu przed przetworzeniem kości, które ulegają obliczeniom kinematyki odwrotnej.
- WriteTransform - Proces odpowiedzialny za aktualizację macierzy transformacji obiektów oraz kości animacji szkieletowej.

- WriteProperties - proces odpowiedzialny za zapisanie zmian obiektów i kości w trakcie animacji.

Między blokami znajduje się grupa metod wewnętrznych silnika, które pozwalają na przeprowadzenie obliczeń fizycznych. Obliczenia te biorą pod uwagę tarcie między obiektami, grawitację, pęd obiektów, kolizję i kila innych czynników.

Pętlę fizyczną kończą dwie metody wywoływane podczas kolizji oraz metoda odroczenia metody 'FizedUpdate'. Następnym blokiem jest 'GameLogic'. W bloku logiki gry można znaleźć blok aktualizacji animacji, co zostało omówione wcześniej. Jako metody logiki gry zalicza się metody:

- Update - metoda wywoływana co klatkę, pozwala użytkownikowi na stworzenie kodu, który pozwala na aktualizację pól elementów interfejsu graficznego, reakcji obiektów na sygnały wejściowe, obliczenia sztucznej inteligencji itp.

- yield null/WaitForSeconds/WWW/StartCoroutine – grupa odroczeń lub oczekiwań na otrzymanie danych. W tym miejscu jest kontynuowanie wykonywania odroczonego kodu, po czasie lub po otrzymaniu danych.

- LateUpdate – metoda, którą można nadpisać. Jej funkcją są reakcja na wewnętrzną aktualizację animacji w trakcie bloku logicznego lub implementacja kodu, który musi zostać wykonany po animacjach.

Następnym blokiem są metody wywoływane podczas procesu generowania klatki. Metody w bloku są nadpisywane głownie w przypadku generowania efektów graficznych, post procesu lub zmian w shaderach. Metody zawarte w bloku 'Scene Rendering':

- OnPreCull – metoda użytkownika wywoływana przed renderem widoku kamery.

- OnBecameVisible – metoda użytkownika wywoływana, gdy obiekt jest widziany przez dowolną kamerę.

- OnWillRenderObject – metoda wywoływana, przed rozpoczęciem procesu renderu obiektu. Używana głównie dla obiektów normalnie niewidocznych.

- OnPreRender – metoda wywoływana, przed procesem rendeingu, używana do ustawienia pól obiektu.

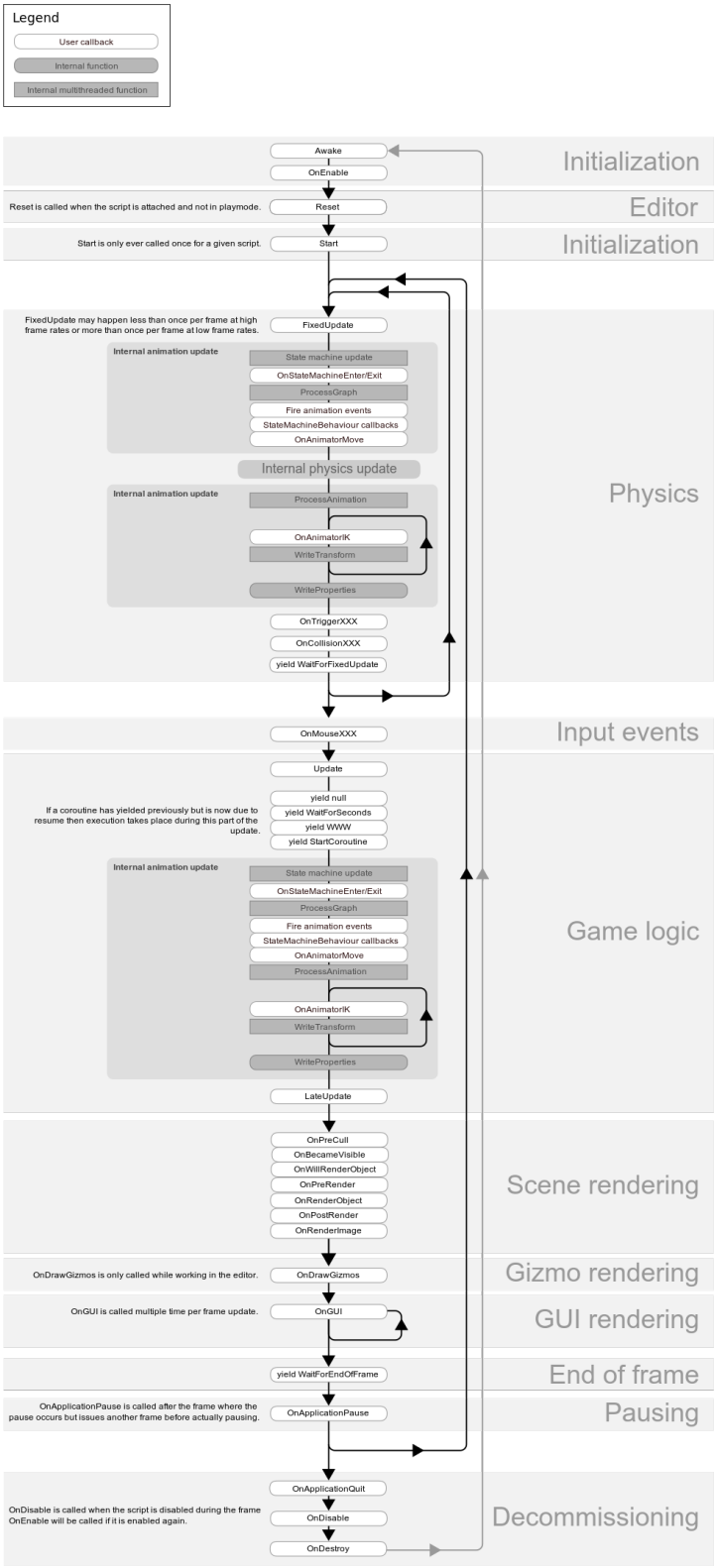
- OnREnderObject – metoda wywoływana podczas renderu obiektu.

- OnPostRender – metoda jest wywoływana po wyrenderowaniu obiektu, pozwala na wprowadzenie post-procesu dla obiektu

- OnRenderImage – metoda wywoływana podczas generowania obrazu.

Następną metodą wywoływaną podczas czasu życia obiektu jest metoda 'OnDrawGizmos'. Jest ona wywoływana podczas pracy w edytorze. Głównym zastosowaniem tej metody jest stworzenie narzędzi do debugowania. Po niej wywoływana jest metoda 'OnGui', która może być wywoływana wiele razy w klatce, a jej przeznaczeniem jest generowanie elementów interfejsu użytkownika – głownie przyciski, paski itp. Przed zakończeniem klatki wznawiane są odroczenia o nazwie 'WaitForEndOfFrame', a po niej następuje wywołanie metody 'OnApplicationPause'. Metoda 'OnApplicationPause' jest wywoływane w następnej klatce po wprowadzeniu stanu pauzy.

Ostatnią częścią życia obiektu w silniku Unity są metody z bloku 'decommisioning', gdzie obiekt jest usuwany lub zatrzymywany. Metoda 'OnDisable' jest jedyną metodą w bloku, która nie usunie obiektu. Jest ona wywoływana, gdy obiekt zostanie wyłączony. 'OnDisable' może być wywołany wielokrotnie w trakcie życia obiektu. Metoda 'OnDestroy' musi być wywołana przez użytkownika w celu usunięcia obiektu. Podobnie jest z metodą 'OnApplicationQuit' i jedyną różnicą jest to, że 'OnApplicationQuit' rozpocznie zakończenie programu.



Rysunek 3.1: Unity Flowchart[4]

3.3. Architektura silnika Unreal Engine

Silnik Unreal Engine jest powstałym w 1991 roku rozwijanym do dzisiaj. Silnik aktualnie jest jednym z najbardziej rozwiniętych silników graficznych. W silniku można pisać kod na dwa sposoby:

- Kod c++ – Unreal Engine posiada możliwość pisania kodu za pomocą języka c++. W tym przypadku twórca może tworzyć własne klasy za pomocą dziedziczeniu z klas zaimplementowanych w silniku. Gry tworzone w ten sposób będą działać szybciej niż te napisane za pomocą „Blueprint'ów" kosztem dłuższego czasu kompilacji.

- Blueprint'y – jest to programowanie obiektowe za pomocą bloków. Programowanie w ten sposób działa podobnie do programowania za pomocą c++ – jest obsługiwane tworzenie programistycznych interfejsów, dziedziczenie i przysłanianie metod. Zaletą tego rozwiązania jest szybkikrótki czas kompilacji.

Twórcy silnika Unreal Engine nie narzucają w jaki sposób należy używać ich narzędzia. Obie metody programowania są ze sobą kompatybilne i można ich używać naprzemiennie. Cały silnik jest bardzo plastyczny i jasno podzielony na wiele klas, które można nadpisać. [5].

Silnik został podzielony na 4470 różnych klas, które można nadpisać i dowolnie używać, ale najczęściej używane to:

- GameInstance — Istnieje tylko jedna instancja tej klasy w całym silniku, która działa od rozpoczęcia programu do jego wyłączenia. Obiekt klasy może być użyty w celu przechowywania danych między poziomami, zapisywania i wczytywania danych np. zapisu gry lub ustawień gracza oraz do przechowywania funkcji, które muszą być dostępne globalnie.

- GameMode — Klasa, która definiuje logikę w danym trybie gry. W grze może być zaimplementowanych wiele trybów, ale tylko jeden obiekt typu GameMode może istnieć. Obiekt tej klasy jest zawarty w obiekcie GameInstance. GameMode jest odpowiedzialny za obsługę wejścia, kolizji oraz zmian poziomów, stworzenie obiektu gracza i systemy zarządzania punktami. Zwykle każdy poziom ma swój obiekt typu GameMode.

- GameState — klasa używana do przechowywania i zarządzania danymi pokroju obecne punkty, pozostały czas, zdrowie gracza i inne ważne dane gry. Obiekt jest zwykle zarządzany przez obiekt GameMode i w przypadku gry sieciowej jest replikowany z serwera do klientów. Obiekt tej klasy jest łatwo dostępny przez inne obiekty w programie pokroju kontrolerów gracza i obiektów klasy Actor. Najważniejszą funkcją obiektów tej klasy jest synchronizacja stanu gry dla wszystkich graczy.

- LevelScriptActor — specjalny typ aktora w silniku, który pozwala na stworzenie logiki, która będzie wykonywana w czasie ładowania poziomu. Obiekt klasy pozwala na ustawienie początkowego ustawienia klasy GameState, stworzenie i ustawienie obiektów, aktywacja wydarzeń i wiele innych. Klasa jest nadpisywana w celu stworzenia skryptów, które będą używane w zależności od poziomu.

- Controller — kontrolery dzielą się na dwie kategorie: kontrolery AI i kontrolery gracza. Te obiekty tej klasy są odpowiedzialne za przetwarzanie i przekazanie sygnałów wejścia do kontrolowanego obietku. Kontorlery determinują sposób nawigacji postaci przez poziom, reakcję postaci w różnych sytuacjach oraz interakcje z innymi obiektami w poziomie.

- Pawn/Actor/Chracter – Actor jest najbardziej podstawowym obiektem, który można postawić na poziomie. W obiekcie tego typu znajdują się różne komponenty typu oświetlenie, siatka czy kamera. Obiekty tego typu są bardzo proste i mogą zostać użyte do implementacji własnych zachowań. Pawn jest klasą pochodną, która posiada implementację złożonej fizyki, może oddziaływać na obiekty. Obiekty typu Pawn są używane do stworzenia zachowań imitujących pojazdy lub zwierzęta. Character jest klasą dziedziczącą z klasy Pawn. Obiekty klasy character w porównaniu do dwóch poprzednich klas posiadają własną implementację ruchu, komponentów animacji i zmiany wejścia na ruch fizyczny. Character jest klasą używaną do stworzenia obiektu imitującego człowieka lub zwierzę.

- Komponenty — komponentami w UnrealEngine są wszystkie klasy, które dziedziczą z klasy UActorComponent, a jest ich ponad 400. Komponenty zwykle odpowiadają za jedną funkcjonalność np. oświetlenie, kolizje, fizyka, kamera,

siatka itp. W ten sposób w silniku został stworzony stopień modularności, który pozwala na wysoką elastyczność i skalowalność projektu. Jeden komponent może zostać użyty w wielu różnych klasach. Komponenty mogą zostać użyte tylko w klasach dziedziczących z klasy Actor lub innych komponentach. KOlejną metodą jest

Kolejną rzeczą, która jest ważna w programowaniu gier jest kolejność wykonywania kolejnych części programu. W silniku Unreal każdy element jest inny i twórca gier ma możliwość zmiany każdej części silnika. Poza tym silnik posiada wiele metod, które są wspólne dla wielu elementów. Rysunek 3.2 przedstawia cykl życia obiektu w silniku UnrealEngine. Na rysunku przedstawione różne ścieżki. Jeżeli Aktor musiał być wczytany, ponieważ był częścią mapy, to cykl życia zaczyna się od punktu 'LoadMap'. Funkcja 'AddToWorld' jest częścią działania 'LoadMap'. Kolejnym ważnym punktem na tej ścieżce jest metoda 'PostLoad', która jest wywoływana po wczytaniu mapy.

Kolejną metodą jest 'RouteActorInitialize', który odpowiada za inicjalizacje ścieżki poruszania się obiektu po świecie jeżeli została ustalona. Następnie znajdują się 3 metody wywoływane podczas inicjalizacji komponentu. Powodem stworzenia tych metod jest problem używania komponentów przez inne komponenty. W 'PreInitializeComponents' powinno się inicjalizować zmienne, tworzyć komponenty i zmieniać ustawienia komponentu, który wywołał tę metodę. 'InitializeComponents' najczęściej jest używany podczas uzyskiwania dostępu do innych komponentów. Ta metoda jest używana do uzyskania dostępu do innych komponentów, punkcie czasu są one zainicjalizowane, ale nie można uzyskać dostępu do niektórych pól. 'PostInitializeComponents' jest wywoływana jako ostatni etap inicjalizacji. W tym momencie komponent skończył inicjalizację i twórca kodu może w tym momencie wczytywać dane z innych komponentów należących do tego samego pionka lub aktora bez błędu związanego z nieistnieniem obiektu. Kolejną ważną i często używaną metodą jest metoda 'BeginPlay', która jest wywoływana przed pierwszą klatką, kiedy dany pionek został stworzony lub po wczytaniu poziomu. Metoda nie zostanie wywołana, jeżeli gra jest zapauzowana. Przed omówieniem metody 'Tick' należy opisać drogę od 'SpawnActor' i 'SpawnActorDeffered'. Obie te ścieżki są podobne i główna różnica polega na odroczeniu stworzenia obiektu. 'SpawnActor' zacznie tworzyć aktora w momencie wywołania metody 'SpawnActor'. 'SpawnActorDeffered' rozpocznie tworzyć obiekt po pewnym czasie lub przy spełnieniu pewnych warunków. Konstruktor obiektu z języka C++ zostanie wywołany

dopiero w bloku ‘ExecuteConstruction‘. W między czasie jest wywołana metoda ‘On-Construction‘ i ‘PostConstruction‘. Metoda ‘OnActorSpawned‘ zostaje wywołana po zakończeniu tworzenia obiektu.

Najczęściej używaną metodą, która jest wywoływana co klatkę jest metoda ‘Tick’ na rysunku 3.2. Tę metodę posiada każdy komponent i pionek oraz każdy z tych obiektów może mieć wiele metod typu tick zgodnie z dokumentacją [7]. Dodatkowo według tej samej dokumentacji te metody mogą być wywoływane w różnym punkcie obliczeń silnika. W dokumentacji wyróżniono 6 grup, kiedy metoda ‘Tick‘ może zostać wywołana, ale nie wszystko zostało udokumentowane. Opisane grupy to:

- TG_PrePhysics – pierwsza grupa, najczęściej odpowiada za aktualizację stanu obiektów, na które wpływa fizyka, ale nie zmienia położenia, prędkości i przyśpieszenia obiektów. Najlepszy moment na zebranie sygnałów wejściowych i przekazanie ich do komponentów zajmujących się ruchem.

- TG_DuringPhysics – wszystkie komponenty używające kolizji lub fizyki muszą zostać zaktualizowane. Obiekty znajdujące się w tej grupie obliczają kolizje i inne elementy fizyczne. Twórcy gier mogą zmienić ustawienia silnika aby animacje były aktualizowane w tej grupie. Obiekty w tej grupie mogą być aktualizowane kilka razy w ciągu jednej iteracji pętli ze względu na możliwość implementacji symulacji fizyki w silniku.

- TG_PostPhysics – domyślnie do tej grupy należą wszystkie komponenty zajmujące się animacjami, elementy interfejsu użytkownika i elementy graficzne.

- TG_POostUpdateWork – do tej grupy powinny należeć obiekty, które wymagają dużej ilości obliczeń lub nie są istotne z punktu widzenia działania gry. Obiekty w tej grupie są aktualizowane rzadziej niż te w innych grupach, więc jest to idealna grupa dla sztucznej itenligencji, która wymaga dużej ilości obliczeń.

Zakończenie cyklu życia aktora w silniku Unreal Engine może rozpocząć się w kilku przypadkach:

- Wywołanie metody ‘Destroy‘ - ze względu na architekturę silnika, programista go używający nie może zniszczyć obiektów dziedziczących z klas komponentów

lub klasy aktora nie mogą być zniszczone wywołaniem zwykłego destruktora. Metoda ‘Destroy‘ najpierw bezpiecznie usunie obiekt z elementów silnika, a następnie wywoła destruktor obiektu.

- Czas życia Aktora się skończył – w trakcie działania gry tworzonych i usuwanych jest wiele obiektów. W silniku zaimplementowano mechanizm, który pozwala na usuwanie obiektów po jakimś czasie ustalonym przez twórcę gry. Czas życia obiektu można ustawić za pomocą metody ‘SetLifeSpan(float time)’, która rozpocznie niszczenie obiektu po czasie podanym w sekundach.

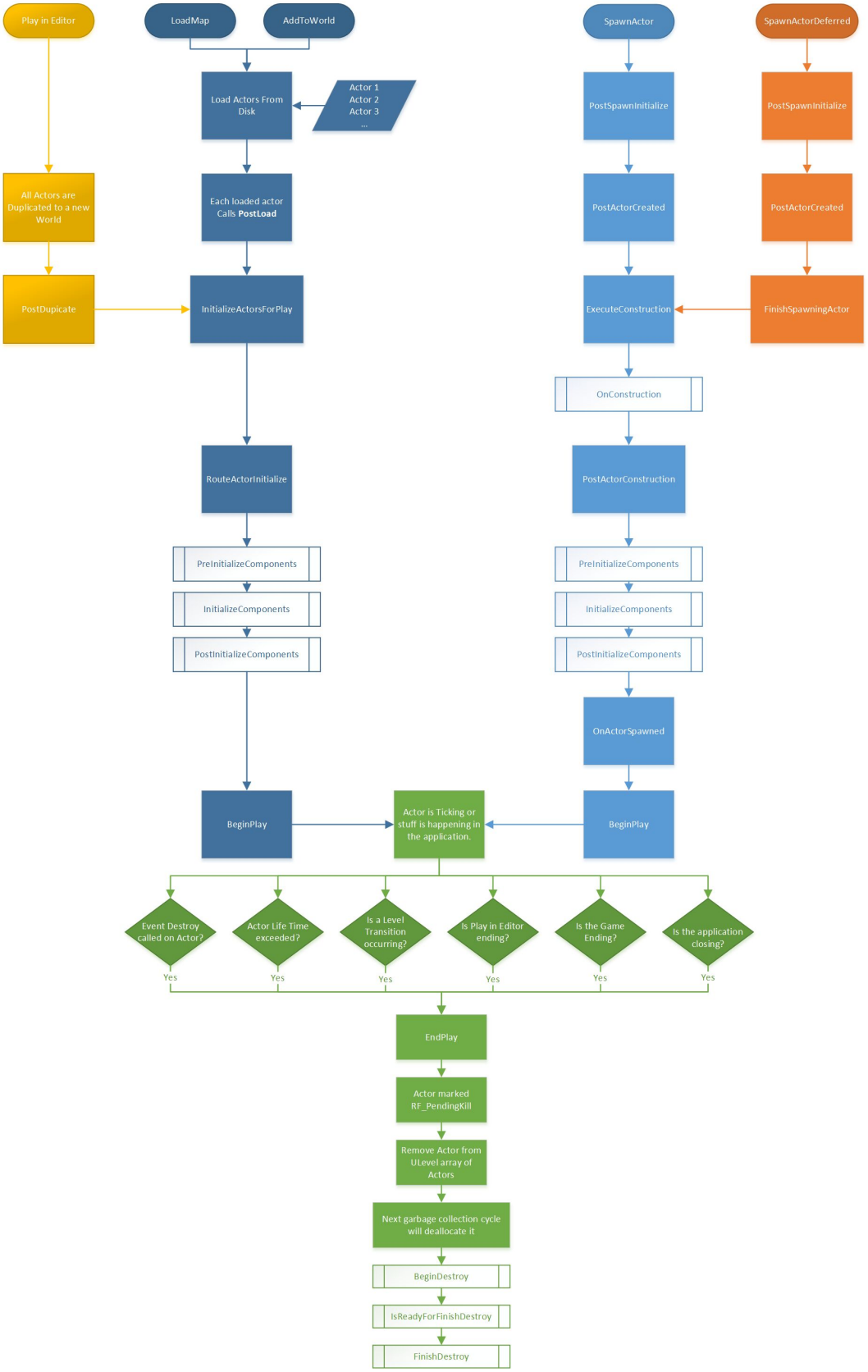
- Zmiana poziomu – gry mogą się składać z wielu poziomów i zabezpieczenie programu przed wyciekami pamięci przy zmianie poziomów jest bardzo ważną rzeczą. Przy zmianie poziomów wszystkie obiekty w poziomie zostają zniszczone i zostaje wczytany nowy poziom.

- Zakończenie gry w Edytorze – testowanie funkcji w edytorze jest ważnym elementem tworzenia elementów gry w każdym silniku. Gra w edytorze może być włączana nawet kilkaset razy, co bez usuwania obiektów może doprowadzić do wycieku pamięci i w konsekwencji wyłączenie się edytora.

- Zakończenie gry – wywołane w przypadku zakończenia rundy, utraty połączenia lub w przypadku osiągnięcia końca gry i wyświetlenia napisów końcowych.

- Zakończenie programu

Jeżeli jeden z tych warunków zostanie spełniony zostanie wywołana metoda ‘EndPlay‘. Metoda ma na celu zatrzymanie obiektu, zatrzymanie wywoływania metod typu ‘Tick‘, wysłanie sygnału o usuwaniu obiektu do obiektów ‘GameState‘i innych obiektów w grze. Obiekt zostanie oznaczony flagą ‘RI_PendingKill‘, co zakolejkuje go do usunięcia za pomocą mechanizmu ‘Garabage Collector‘. Przy usuwaniu zostaną wywołane metody ‘BeginDestroy‘, ‘IsReadyForFInishDestroy‘ i ‘Finishdestroy‘, w których powinna być zawarta implementacja usuwania elementów alokowanych dynamicznie.



Rysunek 3.2: Cykl życia obiektu w silniku Unreal Engine[6]

3.4. Różnice między silnikami

Poprzednie podrozdziały pokazały różnice w budowie i funkcjonowaniu silników. Poza tym należy uwzględnić inne różnice między silnikami, które nie dotyczą ich budowy, ale są ważnym elementem podczas programowania w obu silnikach.

W tabeli 3.1 przedstawia różnice między silnikami, które można zauważyć przy korzystaniu z obu silników w dłuższym okresie czasu. Pierwszą różnicą rzucającą się w oczy jest podstawowa jednostka długości. W przypadku Unity jednostką odległości jest metr. W przypadku prędkości jest to metr na sekundę. W Unreal Engine jednostką odległości jest centymetr, a prędkości centymetr na sekundę. Z widzenia programisty jest to ważna różnica, ponieważ wymagana jest konwersja jednostek w przypadku korzystania ze stałych lub wzorów. Kolejną ważną różnicą jest oznaczenie osi, która opisuje wysokość. W Unity wysokość jest oznaczona osią Y. Jest to ważne z punktu widzenia programisty ponieważ domyślnie grawitacja jest skierowana prostopadle w dół do osi oznaczającej wysokość. W Unreal Engine wysokość jest opisana osią Z.

Następnym punktem w tabeli są wbudowane elementy w silnik, które pozwalają na zbudowanie gry. Unity posiada podstawowe funkcjonalności związane z animacjami i ich edytowaniem, obsługą dźwięku, modeli, skryptów itp. Jest to jednocześnie wada i zaleta silnika. Mała liczba gotowych elementów zmusza programistę do tworzenia brakujących elementów lub korzystania z elementów stworzonych przez społeczność. Takie podejście jest jednak czasochłonne i może doprowadzić do błędów w działaniu gry. Unreal Engine stoi po przeciwnej stronie medalu. Silnik zapewnia wiele elementów, stworzonych przez twórców silnika, które zostały przetestowane i sprawdzone, jednak takie podejście ogranicza w pewnym stopniu kreatywność.

Kolejnym punktem zapisanym w tabeli3.1 jest podział skryptów. W Unity wszystko zostało zawarte w ‘Monobehaviour’. Ze względu na charakter języka C# jest to bardzo dobre podejście, ale ogranicza się to do pisania skryptów i wyklucza tworzenie komponentów. Unreal Engine każdą klasę posiada w osobnych plikach z rozszerzeniami ‘.h’ i ‘.cpp’, co jest standardem w przypadku programowania obiektowego w języku ‘C++’. Wadą tego rozwiązania jest szukanie pewnych klas i funkcji oraz długi czas kompilacji projektu. Rozwiązaniem problemu jest użycie ‘Blueprintów’ razem z ‘C++’.’Blueprinty’ pozwalają na szybkie prototypowanie, jednak kod zapisany w ten sposób jest wolniejszy niż ten zapisany w ‘C++’.

Budowa silników została opisana w poprzednich rozdziałach. Unity ma prostą budowę składającą się z 3 elementów. Zaletą tego rozwiązania jest lekkość silnika, a wadą brak rozwiniętej infrastruktury gry.

Tabela 3.1: Znaczące różnice między silnikami Unity i Unreal Engine

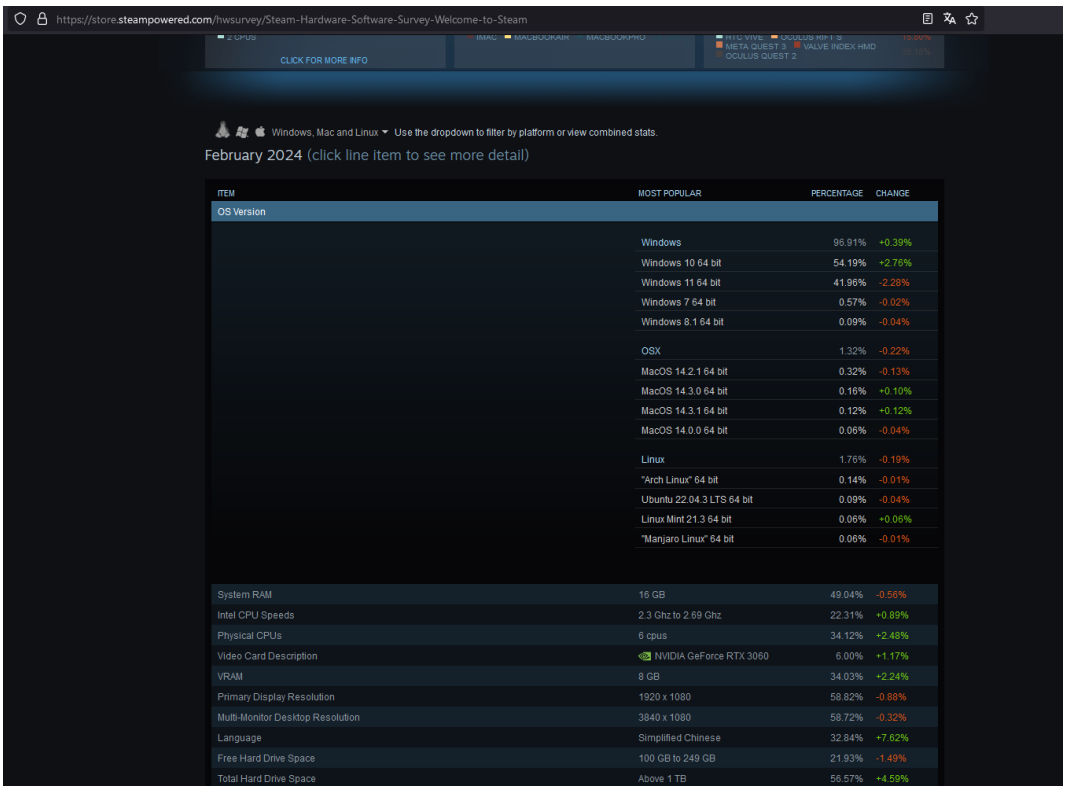
Różnica	Unity	Unreal Engine
Podstawowa jednostka długości	metr	centymetr
Oznaczenie osi	Y oznacza wysokość	Z oznacza wysokość
Wbudowane funkcjonalności	Wymaga zbudowania gry od podstaw lub używania plugin'ów	Posiada wbudowane funkcje, które występują w wielu grach
Podział skryptów	Wszystko zawarte w Monobehaviour	Wszystkie są zapisane w osobnych plikach
Podział elementów silnika	Scena, Obiekt, Komponent	Instancja silnika, GameMode, GameState, Level, Controller, Aktor, Komponent

4. Testowanie wydajności silników Unity i Unreal Engine

Głównym testem jest sprawdzenie jak silniki radzą sobie z generacją obrazu w czasie rzeczywistym. Ilość obiektów została wyznaczona metodą prób i błędów w taki sposób, aby silniki graficzne były pod obciążeniem. Obiekty są prostymi modelami z zapętlonymi animacjami wybranymi wcześniej w sposób losowy – rozstawienie, animacje i prędkość animacji zostaną wygenerowane skryptem i były takie same dla obu silników. Ruch kamery został zdefiniowany przed testem podobnie jak obiekty rozstawione w świecie.

Silniki graficzne są bardzo złożone, dlatego test obejmuje tylko jeden aspekt jakim jest renderowanie dynamicznie rozstawionych obiektów wraz z animacjami. Obiekty w świcie gry mogą być statyczne, co oznacza, że nie można ich ruszyć i nie działają na nie prawa fizyki. Silniki graficzne mogą zmniejszać wymagania obliczeniowe związane z tym obiektem przykładowo poprzez wypalania oświetlenia. Dynamicznymi obiektami są określane wszystkie obiekty, które można ruszyć, poruszają się same z siebie albo są zaanimowane.

Gry komputerowe trafiają na różne urządzenie z różną mocą obliczeniową. Dodatkowo komputery posiadają wiele programów i usług działających w tle, które mają wpływ na wynik testów. Te czynniki należy wziąć pod uwagę podczas trwania testów. Dlatego testy odbędą się na komputerach stacjonarnych i laptopach z systemem Windows 10 i Windows 11. System operacyjny został wybrany ze względu na popularność wśród graczy. Zgodnie z badaniem z lutego 2024 roku wykonanym przez platformę Steam, która jest największa aplikacją sprzedająca gry na komputery. Gracz korzystający ze sklepu najczęściej używają systemów Windows 10 i Windows 11 stanowiących odpowiednio 53,19% i 41,96% wszystkich graczy co zosstało przedstawione na rysunku 4.3



Rysunek 4.3: Statystyka użycia systemów operacyjnych przez graczy komputerowych przeprowadzona przez Steam [8]

Głównym celem testów jest oszacowanie, który silnik graficzny lepiej radzi sobie z generacją klatek w czasie rzeczywistym. Kryteriami oceniania są:

- Licznik FPS – jest to wartość równa 1/(czas wykonania programu między klatkami). Najważniejsze kryterium ze wszystkich, jednak różnica między 1-2 klatkami na sekundę nie jest znacząca. Dodatkowo można określić stabilność na podstawie różnych miar pokroju minimum, maskimum i odchylenie standardowe.

- Użycie czasów procesora i procesora graficznego – ważne kryterium, które pomoże ustalić, który z silników graficznych wymaga większej mocy obliczeniowej. Jest to drugorzędny wyznacznik i zostało zmierzone średnie, minimalne i maksymalne użycie.

- Zużycie pamięci RAM i VRAM – Zużycie pamięci RAM i VRAM – ostatni wyznacznik. Zbyt mała ilość pamięci RAM i VRAM może doprowadzić do nieuruchomienia się gry. Najważniejsza jest maksymalna ilość używanego zasobu

podczas testu, a średnie i minimalne użycie jest ciekawostką. Maksymalne wartości pozwolą na określenie, który silnik lepiej gospodaruje zasobami.

5. Analiza danych

W tabeli 5.2 przedstawiono 7 komputerów, które zostały użyte do przetestowania wydajności silników. Wśród komputerów 6 jest komputerami stacjonarnymi, a jeden to laptop. Wśród używanych komputerów pojawiają się różne karty graficzne. Komputerem z najwolniejszą kartą graficzną jest pc6, a z najszybszą pc4. Komputerami z największym taktowaniem pamięci RAM są pc3 i pc2. Najwolniejszą pamięć RAM posiada cp6. Najwolniejszy procesor posiada pc7. Najszybsze procesory znajduje się w komputerze pc2. PC3 zawiera najnowszy model procesora, ale jest to wersja dedykowana dla laptopów, przez co jest wolniejszy.

Tabela 5.2: Komputery użyte do testów

LP	Typ jednostki	Procesor	Pamięć RAM	Karta Graficzna	System
PC1	K. stacjonarny	Ryzen 5 3600	16GB 3200MHz	GeForce RTX 2070 8GB	Win 10
PC2	K. stacjonarny	Ryzen 5 5600	32GB 4800MHz	GeForce RTX 4060 8GB	Win 11
PC3	Laptop	Ryzen 5 6600H	16GB 4800MHz	GeForce RTX 3050 4GB	Win 10
PC4	K. stacjonarny	Ryzen 5 3600	16GB 3333MHz	Radeon RX6800 16GB	Win 10
PC5	K. stacjonarny	Ryzen 5 3600X	16GB 2133MHz	Radeon RX 580 8GB	Win 10
PC6	K. stacjonarny	i7-3770K	16GB 1333MHz	GeForce GTX 1050 Ti 4GB	Win 10
PC7	K. stacjonarny	i5 4460	16GB 1600MHz	Radeon RX6600 8GB	Win 10

W tabelach 5.3 i 5.4 przedstawiono dane statystyczne stworzone na podstawie zebranych danych dla silnika unity. Minimalna liczba klatek mieściła się w przedziale <3.00, 5.52>. W przypadkach 4 komputerów minimalna liczba FPS wynosiła 3.00 co może się wiązać z działaniem silnika. Maksymalna liczba FPS wynosiła 50.00 dla wszystkich komputerów poza PC7, dla którego wyniosła 60.06. Te wyniki zostały odnotowane podczas rozstawiania modeli w silniku. Średnia liczba klatek mieściła się w przedziale <17.77, 34.25>. Średnia liczba klatek była najniższa w komputerze z najsłabszym procesorem, a najwyższa w przypadku PC4. Odchylenie standardowe liczby klatek na sekundę mieściło się w przedziale od <3.15, 4.75>, co świadczy o sporym odchyleniu od wartości średniej. Średnia najgorszego 1% i 0.1% klatek na sekundę odbiega od wartości średniej o ponad 10 FPS w prawie wszystkich przypadkach co świadczy o dużych fluktuacjach w prędkości wyświetlanego obrazu. Wymagany minimalny czas

pracy procesora nie przekraczał 42 milisekund. Największy maksymalny czas pracy procesora został odnotowany w pc4, który posiadał najniższy minimalny czas pracy. Średnia czasu pracy procesora mieściła się w przedziale <29.29, 52.59>. Silnik Unity używał średnio ok 770MB pamięci RAM i wartość średnia była podobna dla wszystkich komputerów. Wartości maksymalne nie odbiegały znacząco od wartości średnich. Minimalne użycie pamięci RAM oscylowało wokół 700MB i najwięcej używał komputer z najsłabszym procesorem PC7. W użyciu czasu karty graficznej można zauważyć duże różnice między jednostkami, które są zależne mocy kart graficznych. Największe wartości minimalnego, maksymalnego oraz średniego czasu użycia zostały odnotowane dla komputera PC6. Najniższy średni i maksymalny czas pracy został odnotowany dla komputera pc4, a najniższy średni czas pracy został odnotowany dla komputera pc2, co świadczy o tym, że komputer z szybszym procesorem średnio używał mniej czasu karty graficznej, niż komputer z szybszą kartą graficzną. Takie zjawisko może być skorelowane z przesyłaniem danych z procesora do karty graficznej. Ostatnimi wartościami w tabelach są minimalne, maksymalne i średnie użycie pamięci VRAM karty graficznej. W teście średnio używano ok 2400MB pamięci VRAM z małymi odchyleniami w zależności od komputera i podobne zjawisko zachodzi w przypadku wartości maksymalnych, które nie odbiegają znacząco od wartości średnich. Minimalne użycie pamięci VRAM jest podobne dla wszystkich komputerów poza pc7, który używał ok. 700MB pamięci więcej.

Podsumowując użycie zasobów różnych komputerów można dojść do następujących wniosków:

- Test unity działał lepiej na PC4 osiągając średnio 34.25 klatek na sekundę z odchyleniem standardowym na poziomie 4.15. Najgorszy 1% i 0.1% różni się od siebie w znaczący sposób, co świadczy od dużych fluktuacjach.

- Laptop używany w teście nie odbiegał znacząco od komputerów stacjonarnych w wynikach poza maksymalnym użyciem czasu procesora, który wyniósł 730 ms, co było najwyższym uzyskanym wynikiem.

- Silnik unity wymaga więcej mocy procesora niż karty graficznej, o czym świadczy różnica między użyciem karty graficznej i procesora, ale najlepszy wynik został zanotowany na komputerze z 4 najlepszym procesorem i najlepszą kartą graficzną wśród komputerów.

- Unity używa ok.770MB pamięci RAM i ok 2400MB pamięci VRAM niezależnie od mocy obliczeniowej procesora i karty graficznej. Wartości maksymalne są mocno zbliżone do wartości średnich.

Tabela 5.3: Dane statystyczne zebrane z komputerów w Unity cz.1

LP	Min FPS	Max FPS	Avg FPS	Std FPS	Worst 1%	Worst 0.1%	Min CPU (ms)	Max CPU (ms)	Avg CPU (ms)
PC1	5.33	50.00	31.34	4.29	20.68	19.60	24.64	72.70	31.19
PC 2	5.52	50.00	33.48	4.55	22.22	18.54	22.88	57.00	29.29
PC 3	5.26	50.00	31.09	4.75	16.68	14.71	24.56	64.38	32.07
PC 4	3.00	50.00	34.25	4.15	19.22	5.24	18.49	730.92	30.66
PC 5	3.00	50.00	22.53	3.60	14.10	10.97	32.02	97.24	42.73
PC 6	3.00	50.00	20.47	3.21	9.55	8.33	36.41	144.98	47.93
PC 7	3.00	60.06	17.77	3.15	11.98	10.00	41.41	296.63	52.59

Tabela 5.4: Dane statystyczne zebrane z komputerów w Unity cz.2

LP	Min RAM Usage (MB)	Max RAM Usage (MB)	Avg RAM Usage (MB)	Min GPU (ms)	Max GPU (ms)	Avg GPU (ms)	Min VRA-MUsage (MB)	Max VRA-MUsage (MB)	Avg VRA-MUsage (MB)
PC1	698.84	773.10	772.16	2.37	46.20	15.12	302.44	2496.94	2452.35
PC2	698.84	777.10	776.03	1.78	40.31	13.71	265.52	2460.02	2415.44
PC3	698.84	777.10	776.03	2.91	60.89	20.58	265.52	2460.02	2415.42
PC4	698.84	779.10	775.41	0.05	32.11	16.32	265.52	2472.02	2417.40
PC5	698.84	773.10	772.21	0.79	68.11	17.85	265.52	2460.02	2415.30
PC6	697.59	775.85	774.72	3.53	100.53	33.20	265.52	2460.02	2415.37
PC7	715.47	770.60	770.45	2.11	97.06	25.22	997.93	2565.50	2533.38

Tabele 5.5 i 5.6 przedstawiają statystyki opracowane na bazie danych zebranych podczas testów silnika Unreal Engine przeprowadzonych na komputerach przedstawionych w tabeli5.2. Minimalna liczba klatek na sekundę jest niska dla wszystkich jednostek i nie przekracza wartości 0.77. Najwyższe wartości średnie i maksymalne klatek w teście zostały osiągnięte przez PC1, co świadczy o tym, że na szybkość generowania klatek mają wpływ inne czynniki niż moc obliczeniowa karty graficznej i procesora. Istnieje możliwość, że pc2 osiągną słabszy wynik ze względu na system operacyjny. Nieznacznie niższy wynik średniej ilości FPS osiągną PC4 z niższą wartością odchylenia standardowego. Najgorszy wynik uzyskał pc6, z najgorszymi podzespołami. Odchylenie standardowe liczby klatek jest zależne od podzespołów komputera, ale największą wartość osiągną pc3. Wartość najgorszego 1% i 0.1% jest mniejsza od wartości średniej odpowiednio w przedziale (3,5) i (4,10) i jest zależne od mocy obliczeniowej komputera. Najniższa wartość najgorszego 0.1% klatek został odnotowany dla pc4, co świadczy o rzadkich, ale większych spadkach stabilności wyświetlania obrazu. Użycie procesora nie jest zależne od jego mocy, ponieważ komputer z najwolniejszym procesorem średnie użycie czasu pracy procesora na poziomie zbliżonym do komputerów z szybszymi procesorami. Najdłużej średnio procesor pracował w komputerze z najwolniejszą kartą graficzną, ale najdłuższy czas pracy procesora w klatce został zanotowany dla komputera PC5. Minimalne użycie czasu pracy procesora nie przekroczyło wartości

8ms i w 3 przypadkach było mniejsze niż 0.5ms. Użycie pamięci RAM było zbliżone dla wszystkich komputerów. Maksymalnie test używał 1515MB pamięci na komputerze pc5. Ten sam komputer posiadał najwyższe średnie użycie pamięci RAM, które odbiegało od drugiego najwyższego wyniku o ok. 40 MB i wynosiło 1303. Najmniej tej pamięci używał pc2. Te wyniki sugerują, że użycie RAM'u jest zależne od podzespołów komputera. W przypadku użycia czasu pracy karty graficznej można zauważyć, czas pracy tego podzespołu jest zależny od jego mocy. Karta graficzna PC4 pracowała najkrócej, a PC5 najdłużej. Minimalny czas pracy karty graficznej wyniósł dla wszystkich komputerów 0ms i było to podczas ładowania poziomu. W przypadku pamięci VRAM karty graficznej jej średnie i maksymalne zużycie jest różne dla wszystkich jednostek i oscyluje odpowiednio między wartościami <756,1375>MB i <1024, 1932>MB. Minimalne użycie pamięci VRAM było równe dla wszystkich komputerów i wynosiło 0.02MB. Ta wartość została odnotowana w czasie ładowania poziomu.

Po analizie danych statystycznych na temat użycia zasobów przez silnik Unreal Enigne można dojść do następujących wniosków:

- Na działanie programów w silniku Unreal Engine mogą mieć wpływ inne czynniki niż moc obliczeniowa karty graficznej czy procesora
- Stabilność generowania klatek jest zależna od mocy procesora i karty graficznej.
- Unreal wykorzystuje w teście więcej mocy obliczeniowej karty graficznej niż mocy procesora.
- Ilość używanej pamięci RAM i VRAM przez program nie jest równa dla wszystkich komputerów na których wykonano testy. Różnice wyniosły 400MB w przypadku pamięci RAM i ok. 900MB w przypadku VRAM'u.
- W trakcie ładowania poziomu silnik nie wykorzystuje karty graficznej.
- W przypadku komputera z systemem Windows 11 został odnotowany słabszy wynik, co może być powiązane z różnicami między systemami Windows 10 i Windows 11.

Tabela 5.5: Dane statystyczne zebrane z komputerów w Unreal Engine cz.1

LP	Min FPS	Max FPS	Avg FPS	Std FPS	Worst 1%	Worst 0.1%	Min CPU (ms)	Max CPU (ms)	Avg CPU (ms)
PC1	0.72	17.62	14.75	1.13	11.29	10.65	1.85	76.75	23.77
PC2	0.77	12.30	10.26	0.57	8.80	7.89	7.99	65.53	31.30
PC3	0.75	13.28	10.42	1.78	5.55	5.07	0.10	166.32	30.74
PC4	0.62	16.69	14.24	0.98	10.99	8.55	0.29	77.37	23.13
PC5	0.69	15.78	12.39	1.30	9.15	2.55	3.15	248.33	27.82
PC6	0.60	8.75	6.96	1.34	3.57	3.28	3.00	189.06	51.07
PC7	0.26	14.64	10.49	1.34	7.41	7.11	0.21	103.86	29.82

Tabela 5.6: Dane statystyczne zebrane z komputerów w Unreal Engine cz.2

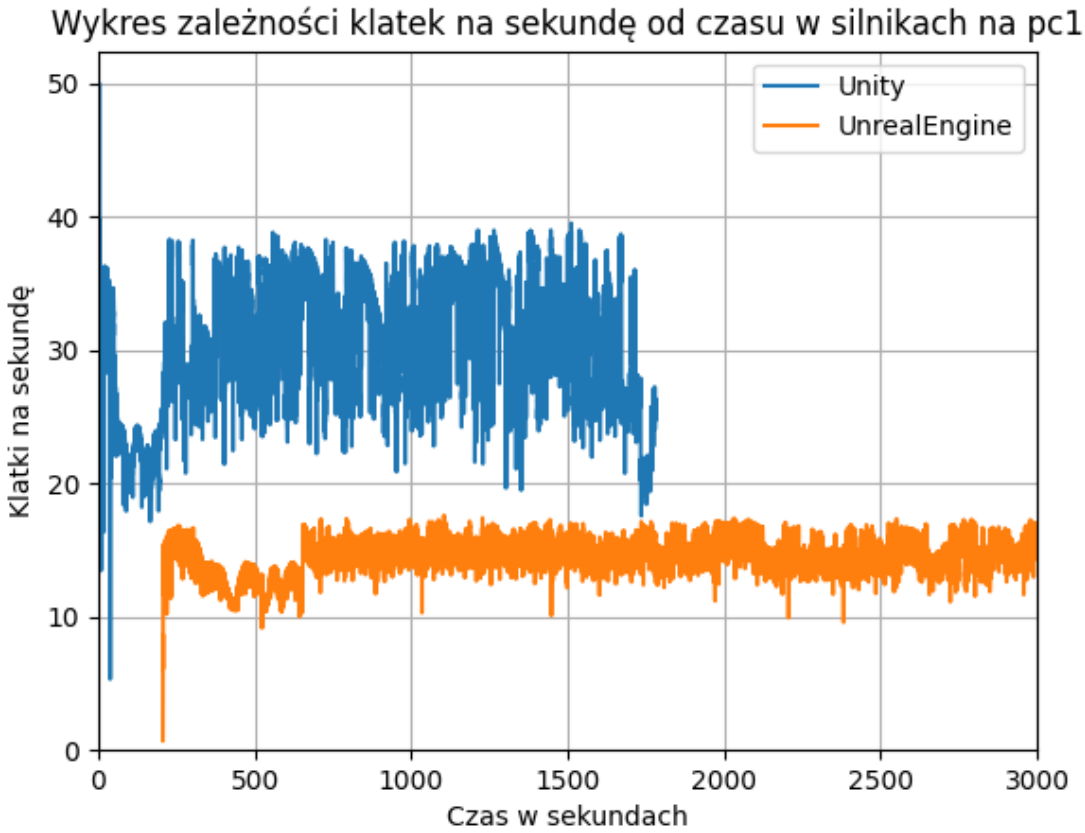
LP	Min RA-MU-sage (MB)	Max RAM Usage (MB)	Avg RAM Usage (MB)	Min GPU (ms)	Max GPU (ms)	Avg GPU (ms)	Min VRAM Usage (MB)	Max VRAM Usage (MB)	Avg VRAM Usage (MB)
PC1	891.84	1192.42	1147.32	0.00	495.69	41.44	0.02	1376.75	1347.87
PC2	895.72	1107.62	1059.57	0.00	275.53	30.45	0.02	1042.42	756.09
PC3	967.32	1270.13	1219.99	0.00	916.77	77.82	0.02	1552.02	993.64
PC4	914.39	1339.31	1159.13	0.00	179.04	14.95	0.02	1605.20	1373.53
PC5	868.34	1432.96	1264.48	0.00	511.32	48.57	0.02	1932.48	1343.09
PC6	870.37	1515.11	1303.42	0.00	1408.78	127.18	0.02	1733.08	1375.39
PC7	820.81	1175.81	1129.66	0.00	257.55	22.68	0.00	1752.53	906.55

Rysunki 5.4, 5.5, 5.6, 5.7, 5.8, 5.9, 5.10 przedstawiają wykresy klatek na sekundę od czasu przebiegu testów dla różnych komputerów. Na wykresach kolorem niebieskim oznaczono dane zebrane w czasie testu silnika unity, a na pomarańczowo dla silnika Unreal Engine. Wykresy zostały ograniczone do 3000 sekund trwania testu. Wykres dla silnika Unreal Enigne został przesunięty ze względu na działanie narzędzia, które zapisuje czas.

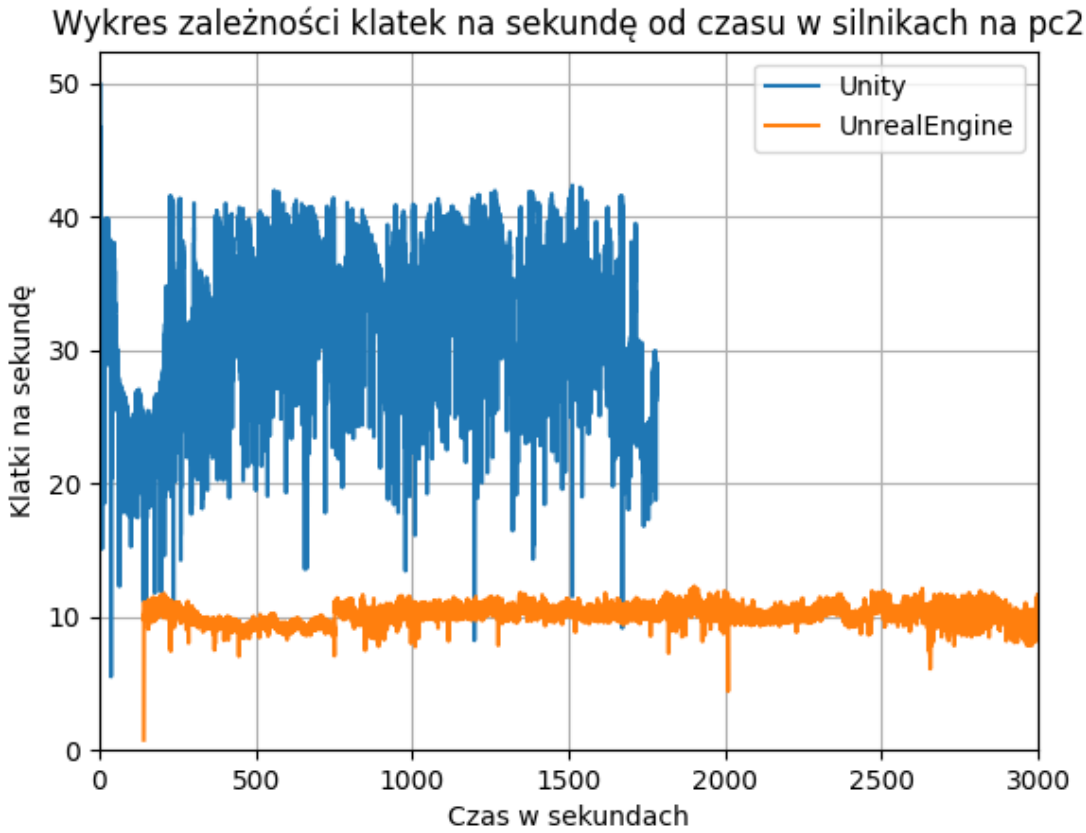
Wszystkie wykresy dla testów silnika Unity posiadają dolinę w podobnym miescu trwania testu, która znajduje się parędziesiąt sekund od zaczęcia testu i trwa do 200 sekund. Pod koniec trwania testu zachodzi podobne zjawisko. Ze względu na zaistnienie zjawiska w testach na różnych komputerach oraz ze względu na fakt, że występują one w tym podobnym miejscu w czasie, to jest to cecha silnika Unity. W pierwszych sekundach wykresy mają wartość 50 lub 60 klatek na sekundę. Na wykresach najmniejsze fluktuacje zachodzą na wykresie 5.9. Dziwne zachowanie można zaobserwować na wykresie 5.10, gdzie wykres ma wartość ok. 30 klatek na sekundę i są to pojedyncze skoki. Największe fluktuacje na wykresie znajdują się na wykresie 5.7. Poza tym na wszystkich wykresach można zaobserwować spadki, które wynoszą ponad 10 klatek, a na niektórych spadki wynoszą nawet 20 klatek na sekundę.

Wszystkie wykresy dla silnika Unreal Engine posiadają dolinę kilkadziesiąt sekund od rozpoczęcia testu, jednak nie jest ona tak głęboka jak w przypadku testów wykonanych na silniku Unity. Dolina ta jest głęboka na ok. 2 lub 3 FPS. Jedynymi wyjątkami są pc6 5.9 i pc3 5.6. Długość doliny jest zależna od sprzętu na którym został wykonany test. Początek wykresu zaczyna się wartością bliską 0. Fluktuacje na wykresach są niewielkie i poza nielicznymi przypadkami nie ma większych odchyleń od średniej. Największe fluktuacje zachodziły dla pc4 5.7 Test silnika Unreal trwał dłużej.

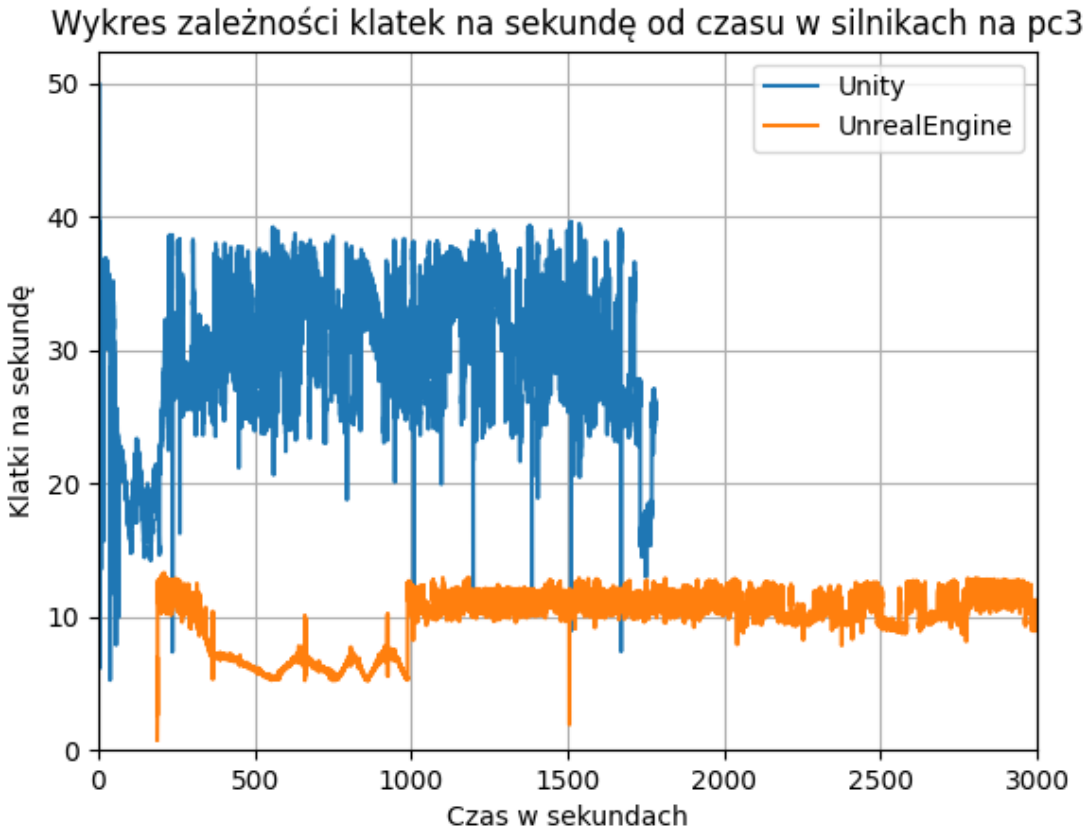
Liczba FPS w teście silnika Unity była znacząco wyższa niż silnika Unreal Enigne, ale też z dużo większymi fluktuacjami. Silnik Unreal Engine był bardzo stabilny z niewielkimi odchyleniami do średniej. W wykresach obu testów ukazały się pewne błędu lub niedociągnięcia silników w początkowych fazach testów.



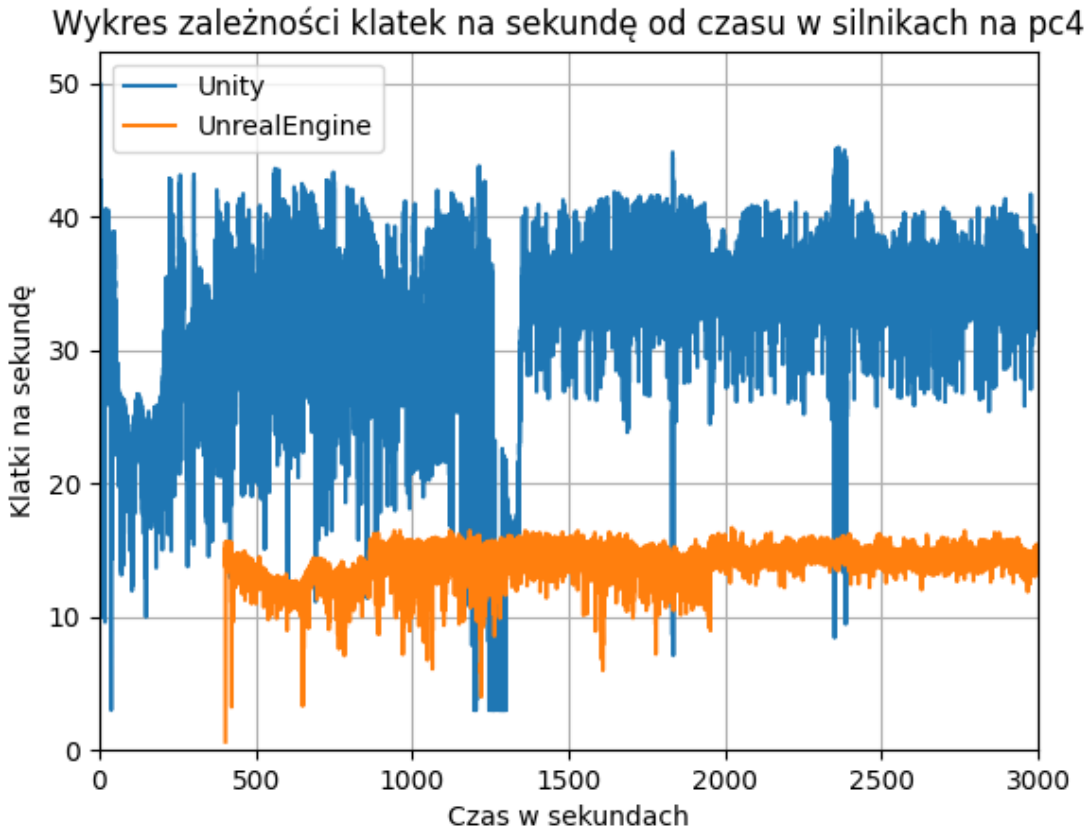
Rysunek 5.4: Liczba klatek na sekundę generowana dla testów przeprowadzonych na PC1



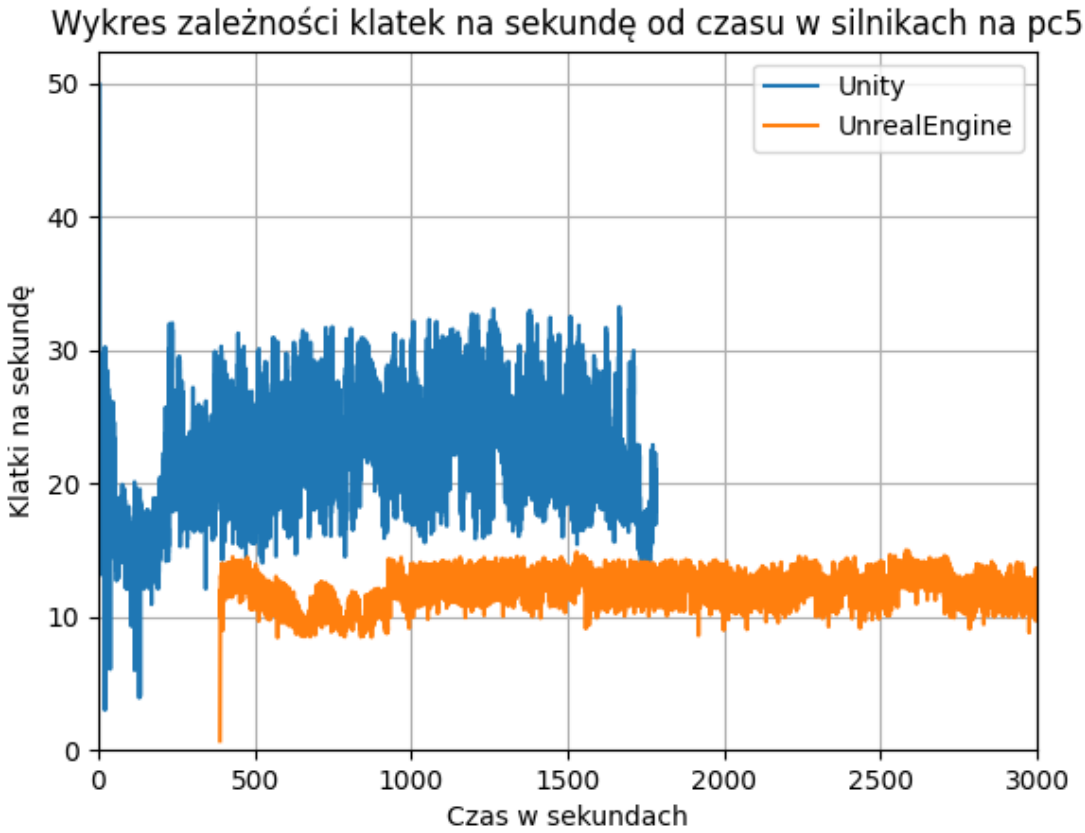
Rysunek 5.5: Liczba klatek na sekundę generowana dla testów przeprowadzonych na PC2



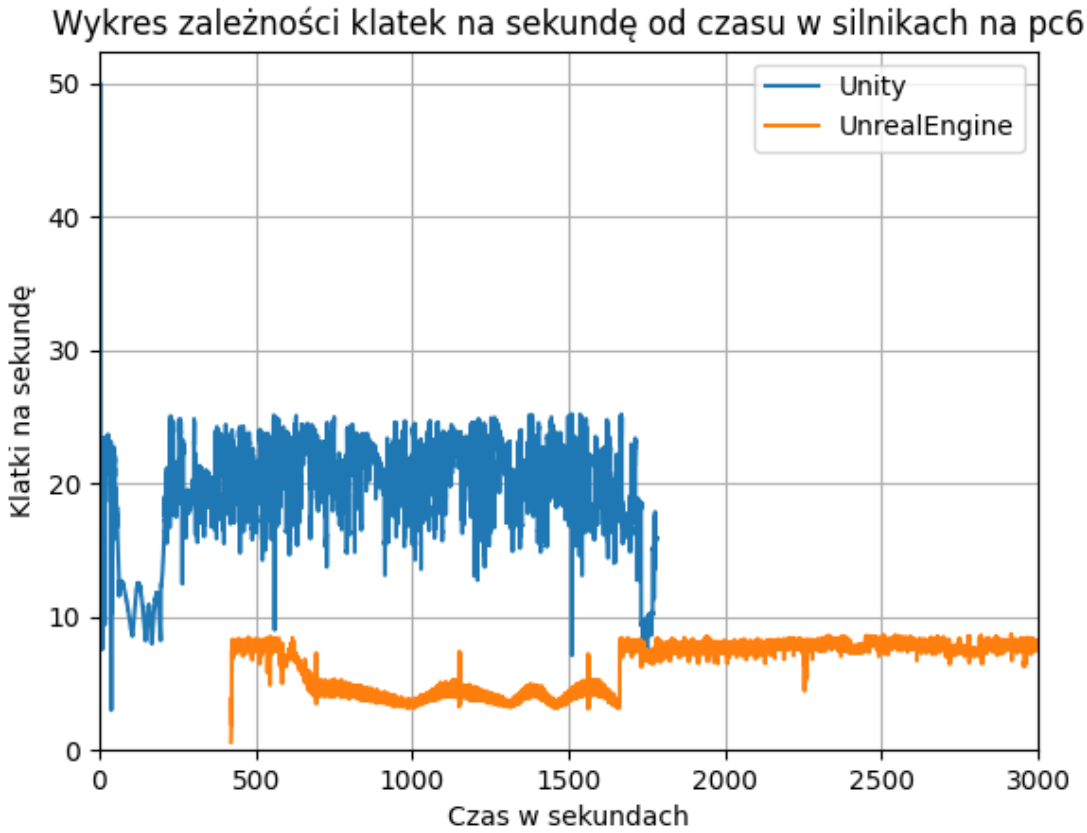
Rysunek 5.6: Liczba klatek na sekundę generowana dla testów przeprowadzonych na PC3



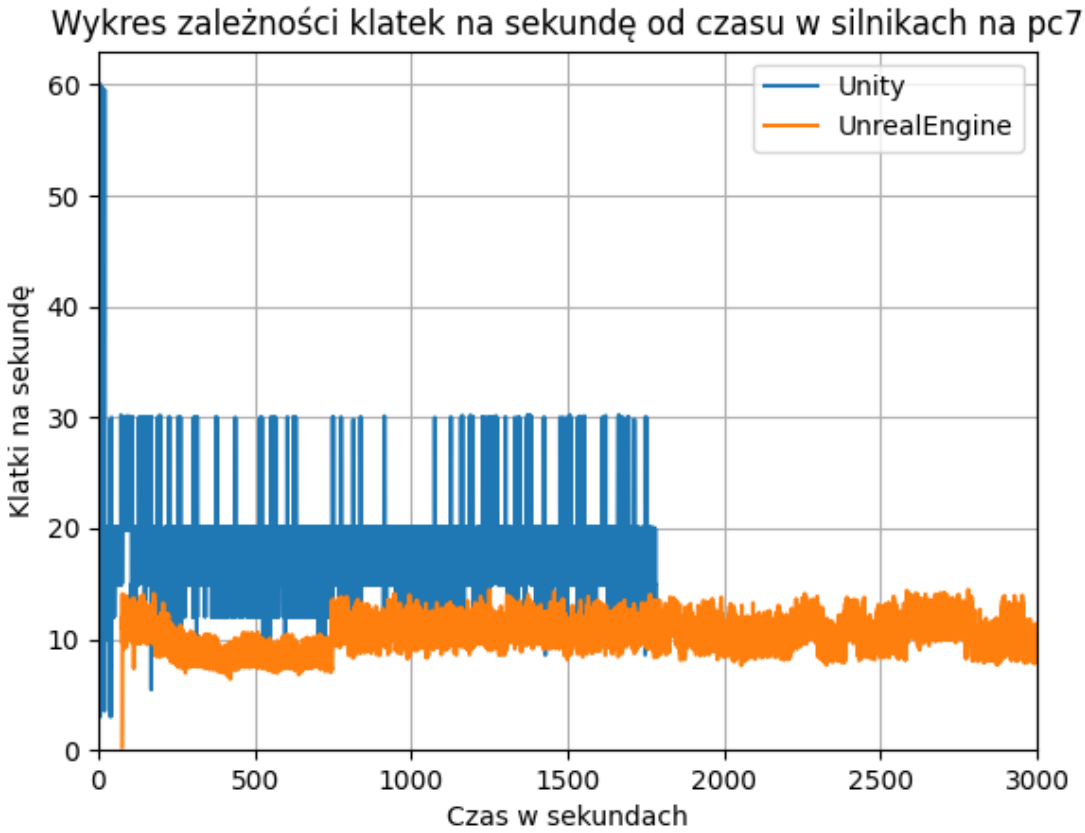
Rysunek 5.7: Liczba klatek na sekundę generowana dla testów przeprowadzonych na PC4



Rysunek 5.8: Liczba klatek na sekundę generowana dla testów przeprowadzonych na PC5



Rysunek 5.9: Liczba klatek na sekundę generowana dla testów przeprowadzonych na PC6



Rysunek 5.10: Liczba klatek na sekundę generowana dla testów przeprowadzonych na PC7

6. Podsumowanie i wnioski

Załączniki

Literatura

[1] Jason Gregory: Game Engine Architecture, CRC Press 2019

[2] David M. Bourg, Bryan Bywalec: Physics for Game Development, OŔeilly 2013

[3] https://docs.unity3d.com/Manual/CreatingGameplay.html Dostęp 20 czerwca 2024

[4] https://docs.unity3d.com/Manual/ExecutionOrder.html Dostęp 20 czerwca 2024

[5] https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/ProgrammingWithCPP/Unre
Dostęp: 20 czerwca 2024

[6] https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/Unre
Dostęp: 20 czerwca 2024

[7] https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/Unre
Dostęp 20 czerwca 2024

[8] https://store.steampowered.com/hwsurvey/Steam-Hardware-Software-Survey-Welcome-to-Steam Dostęp: 20 czerwca 2024

POLITECHNIKA RZESZOWSKA im. I. Łukasiewicza Rzeszów, 2024
Wydział Elektrotechniki i Informatyki

STRESZCZENIE PRACY DYPLOMOWEJ MAGISTERSKIEJ

PORÓWNANIE SILNIKÓW UNREAL ENGINE I UNITY POD KĄTEM TWORZENIA GIER

Autor: Kacper Rudź, nr albumu: EF-16045
Opiekun: dr inż. Sławomir Samolej prof. PRz
Słowa kluczowe: (max. 5 słów kluczowych w 2 wierszach, oddzielanych przecinkami)

Treść streszczenia po polsku

RZESZOW UNIVERSITY OF TECHNOLOGY Rzeszow, 2024
Faculty of Electrical and Computer Engineering

MSC THESIS ABSTRACT

COMPARISON OF UNREAL ENGINE AND UNITY SOFTWARE FOR GAME DEVELOPMENT

Author: Kacper Rudź, nr albumu: EF-16045
Supervisor: PhD Eng Sławomir Samolej
Key words: (max. 5 słów kluczowych w 2 wierszach, oddzielanych przecinkami)

Treść streszczenia po angielsku