

Politechnika Wrocławskiego
Wydział Informatyki i Telekomunikacji

Kierunek: informatyka techniczna (ITE)
Specjalność: grafika i systemy multimedialne (IGM)

PRACA DYPLOMOWA MAGISTERSKA

Porównanie wydajności silników graficznych Unreal 5
i Unity 2021

Michał Nawrot

Opiekun pracy
dr inż. Tomasz Kaplon

Slowa kluczowe: Unreal Engine, Unity, Silnik graficzny, analiza porównawcza, grafika 3D

WROCŁAW 2023

Streszczenie

Celem niniejszej pracy jest zbadanie wpływu wielkości terenów, szczegółowości obiektów 3D i globalnego oświetlenia na wydajność silników Unity 2021 oraz Unreal Engine 5. We wstępie wprowadzono czytelnika w problematykę i zarysowano zakres pracy. Następna część pracy została poświęcona opisowi wykorzystanych narzędzi oraz wyjaśnieniom teoretycznym zagadnień związanych z celem pracy, takich jak: oświetlenie globalne, *ray tracing*, systemy LUMEN oraz NANITE w silniku Unreal Engine 5, HDRP (High Definition Render Pipeline) w silniku Unity. W dalszej części pracy opisano w jaki sposób wykorzystano narzędzia w celu stworzenia w silnikach scen do testów. Kolejna część zawiera informacje o przeprowadzonych badaniach na scenach dynamicznych i statycznych oraz analizę uzyskanych danych. W ostatniej części pracy przedstawione są wnioski pozwalające zdecydować, który z silników jest bardziej wydajny. Tereny poddane analizie zostały wygenerowane przy pomocy programu Gaea, a obiekty 3D przygotowano przy pomocy programu Blender.

Abstract

The aim of the work is to examine the impact of the size of areas, detail of 3D objects and global lighting on the performance of Unity 2021 and Unreal Engine 5. In the introduction, the reader is introduced to the problem and outlined the scope of work. Next part of the thesis was devoted to the description of the tools used and explanations of theoretical issues related to the purpose of the work, such as: global lighting, *ray tracing*, LUMEN and NANITE systems in Unreal Engine 5, HDRP (High Definition Render Pipeline) in Unity. The further part of the work describes how the tools were used to create test scenes in the engines. The next part contains information about the research carried out on dynamic and static scenes and the analysis of the obtained data. In the last part of the work, conclusions are presented to decide which of the engines is more efficient. The terrains were generated using the Gaea program, the 3D objects were prepared using the Blender program.

Spis Treści

1. Wprowadzenie	4
1.1 Cel i zakres pracy	5
2. Przegląd wykorzystanych narzędzi i technologii	6
2.1 Gaea	6
2.2 Blender	7
2.3 Unity	8
2.3.1 High Definition Render Pipeline (HDRP)	8
2.3.2 Globalne oświetlenie	9
2.4 Unreal Engine	11
2.4.1 Nanite	11
2.4.2 Lumen	13
3. Przygotowanie scen	18
3.1 Wygenerowanie terenów	18
3.2 Stworzenie modeli 3D	20
3.3 Import danych do silnika Unity	23
3.3 Import danych do silnika Unreal Engine	26
4. Badanie wydajności	30
5. Podsumowanie i wnioski	35
Bibliografia	36
Wykaz rysunków	38
Wykaz tabel	39
Wykaz listingów	40

1. Wprowadzenie

Obsługa silników graficznych takich jak Unity oraz Unreal Engine (zwany również w dalszej części wprowadzenia "silnikami") staje się coraz bardziej przystępna dla twórców, co jednocześnie przekłada się na zwiększone zainteresowanie samymi silnikami, jak i możliwością wykorzystania ich w coraz to innych dziedzinach. Wspomniana przystępcość polega na znacznym ułatwieniu rozumienia ich działania, jak i łatwiejszego używania w celu uzyskania oczekiwanych efektów. Można postawić tezę, że obecnie przeciętny użytkownik tych silników nie musi posiadać nad wyraz specjalistycznej wiedzy, aby móc tworzyć z nich pomocą własne dzieła i wykorzystywać je do swojej pracy. Należy jednak nadmienić, że nie oznacza to, że ich używanie nie wymaga żadnego doświadczenia czy obycia z narzędziami podobnego typu. O ich przystępcości, względem innych narzędzi, decyduje fakt, że korzystanie z nich jest co do zasad darmowe, a to z kolei sprawia, że cechuje je duże kompendium wiedzy dostępne w Internecie.

Przedmiotowe silniki, dzięki regularnemu wsparciu i oferowanym coraz liczniejszym funkcjom, są coraz częściej wykorzystywane w różnorakich dziedzinach takich jak architektura, edukacja, transport czy w interfejsach człowiek-komputer. Popularnym przykładem sektora, w którym pojawia się rosnące zainteresowanie tego typu narzędziami jest branża filmowa, w której silnik Unreal Engine znajduje szereg zastosowań w produkcji filmów i w zakresie tworzenia wirtualnych scenografii. Silnik umożliwia nie tylko tworzenie kompozycji w czasie rzeczywistym, ale także możliwość niemal natychmiastowej edycji scenografii, co znacznie przyśpiesza pracę na planach filmowych. Z wykorzystaniem technologii tego silnika można oglądać takie popularne seriale telewizyjne jak *Mandalorian* czy *Westworld* [1]. Jednak najpopularniejszą dziedziną, w której są wykorzystywane przedmiotowe silniki graficzne jest produkcja gier wideo. Aktualnie, wspomniana branża warta jest ponad 200 miliardów dolarów, a w czasie samej pandemii zwiększyła swoją wartość o 30% względem roku 2020, kiedy to była warta 155 miliardów dolarów [2][26]. Wzrost ten sprawia, że obecnie gry wideo są jednym z największych i najbardziej lukratywnych rynków w sektorze rozrywkowym.

Można byłoby znaleźć i wskazać wiele przyczyn popularności tych silników, prawdopodobnie jednak jedną z głównych przyczyn jest kwestia finansowa - najczęstszym powodem ich wybierania przez producentów są koszty produkcji, pozwalają one bowiem zaoszczędzić pieniądze z budżetu projektu poprzez likwidację potrzeby projektowania i pisania autorskiego silnika, co jest niezwykle złożonym i pochłaniającym zasoby procesem. Dodatkowo mnogość zastosowań, do jakich można wykorzystać przedmiotowe silniki powoduje, że różne ich elementy są wciąż doskonalone, rozwijane, a to z kolei sprzyja kreowaniu nowych technologii i systemów usprawniających pracę. Kolejnymowąną kwestią, którą warto wziąć pod uwagę jest łatwość nauki obsługi tych silników. Według badań, silnik Unity jest narzędziem, którego obsługa jest łatwiejsza i lepiej dostosowana dla osób rozpoczynających z nim pracę. Jednak dla doświadczonych twórców i rozbudowanych projektów lepiej zdecydować się na pracę w Unreal Engine [27]. Połączenie tych oraz innych benefitów sprawia, że oba silniki odniósły niezaprzeczalny sukces, a silnik Unreal Engine nie bez powodu został nazwany "największym sukcesem wśród silników gier wideo" przez Księgę Rekordów Guinnessa [3]. Nasuwa się jednak kluczowe pytanie, który z tych silników cechuje się lepszą wydajnością?

Jak się okazuje, jednym z kluczowych aspektów, który wpływa na odbiór pracy wykonanej na danym silniku jest wydajność. Może ona bowiem wpływać na jakość grafiki,

płynność animacji i przede wszystkim na ogólne doświadczenia odbiorcy, a to właśnie odbiorca decyduje o powodzeniu danego projektu. Prowadzi to do dążenia przez twórców do osiągnięcia pożądanej wysokiej wydajności, oznaczającej, że aplikacja działa płynnie, bez opóźnień, a grafika wyświetlana jest w wysokiej, możliwie najwyższej jakości. Istnieją badania, pokazujące, że Unity oraz Unreal Engine są zbliżone do siebie pod kątem wydajności, jednak Unity wykazuje się większą oszczędnością pamięci i mniejszym obciążeniem procesora[28][30]. Z drugiej strony wykazano, że silnik Unreal Engine w przypadku obsługi i generowania większej liczby elementów, potrafi przewyższać silnik Unity pod kątem wydajności [29].

1.1 Cel i zakres pracy

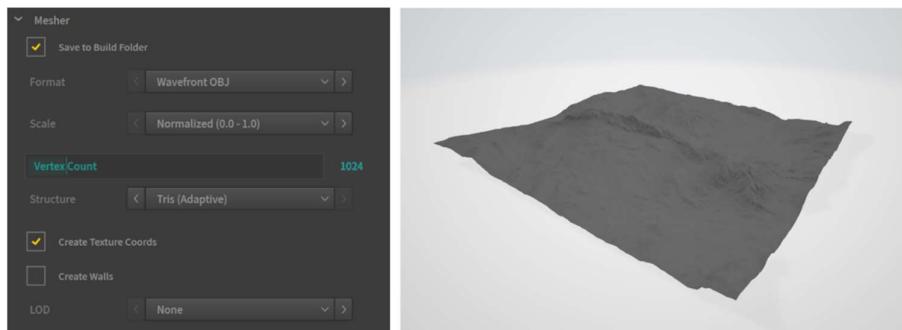
Celem pracy była analiza porównawcza wydajności silników Unreal Engine 5 oraz Unity 2021, w kontekście renderowania scen. Badania obejmowały wpływ wielkości scen i terenów, rodzaju użytego oświetlenia, szczegółowości i ilości modeli 3D oraz rodzaju dynamiki scen, na wydajność obu silników. Stworzenie sceny obejmuje wygenerowanie terenu wraz z obiektami 3D i ich import oraz skonfigurowanie silników w kontekście zastosowania oświetlenia oraz cieniowania i ustawieniem grup LOD obiektów. Badania miały na celu sprawdzenie obciążenia procesora, karty graficznej i pamięci RAM.

2. Przegląd wykorzystanych narzędzi i technologii

W celu przeprowadzenia przedmiotowych badań, wymagane było utworzenie plików (mapy wysokości, maski tekstur), pozwalających na zimportowanie terenów do silników oraz modeli 3D z różnymi ilościami wierzchołków. Później zostały zdefiniowane i opisane kluczowe narzędzia wykorzystane do utworzenia wspomnianych plików oraz przeprowadzenia zasadniczego porównania między dwoma silnikami graficznymi. Dla każdego narzędzia dokumentacje są szczegółowe, jednakże mogą być niewystarczające, ponieważ wiele zagadnień ciężko wyjaśnić za pomocą tekstu. Z tego powodu coraz częściej można spotkać nagrania instruktażowe, tłumaczące dane zagadnienia.

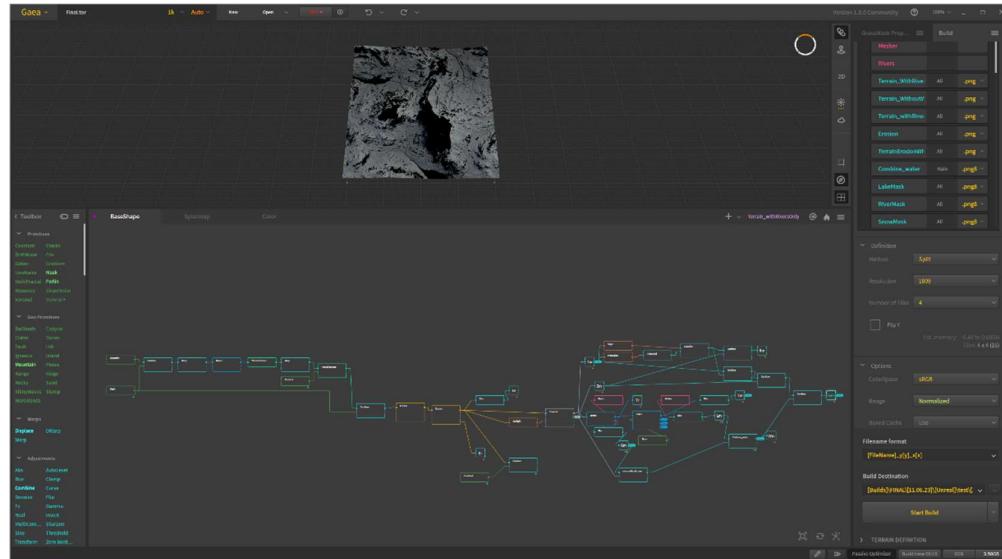
2.1 Gaea

Gaea to program do proceduralnego generowania terenu. Przy pomocy węzłów tworzy się graf procesów i symulacji geograficznych, takich jak erozja terenów i gleby, ukształtowanie i siła nurtu rzeki, istnienie jezior, opadów śniegu czy widocznych roztopów. Z każdego węzła można wyeksportować wygenerowany teren na danym kroku, a rozdzielcość wynikowych plików można łatwo dostosować do tej wymaganej przez silniki Unity i Unreal Engine, poprzez jej wybór w menu kontekstowym. Program umożliwia eksport map wysokości (*heightmap*) z różnymi rozszerzeniami plików oraz przy pomocy węzła “mesher” pozwala zapisać teren jako obiekt siatkowy (*mesh object*) [4].



Rysunek 1: Eksport terenu za pomocą węzła Mesher

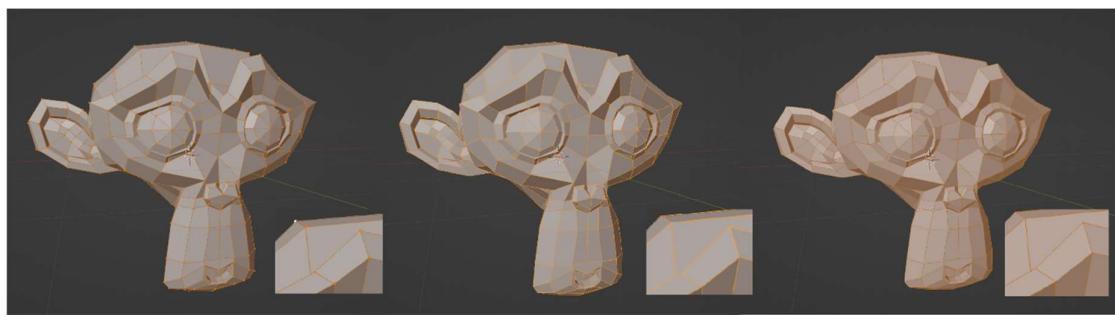
W wykorzystanej wersji *community* maksymalna rozdzielcość terenu wynosi 1024x1024. Ilość szczegółów w wyniku końcowym obliczana jest na podstawie rozmiaru terenu oraz rozdzielcości wynikowego obrazu. Większość aspektów Gaea jest niezależna od rozdzielcości, dlatego całkowity kształt i wynik zastosowanych symulacji nie powinny ulegać zmianie się wraz z próbą jej modyfikacji. Jednakże, twórcy programu zaznaczają, że zawsze istnieje szansa wystąpienia zmian, co jest naturalnym efektem ubocznym terenów opartych na wysokościach.



Rysunek 2: Obszar roboczy Gaea z grafem węzłów

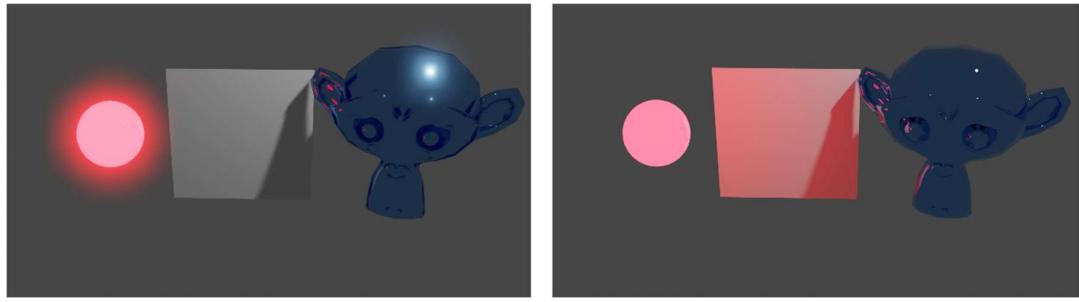
2.2 Blender

Blender to darmowe narzędzie do tworzenia modeli i animacji trójwymiarowych, licencjonowane jako GNU GPL (*GNU General Public License*) czyli wolne i otwarte oprogramowanie. Umożliwia szybki i wydajny proces tworzenia treści 3D, w tym modelowanie, teksturowanie, komponowanie i renderowanie [5]. Podczas pracy z obiektami pozwala na pełną manipulację poprzez edycje wierzchołków, krawędzi i płaszczyzn danego obiektu.



Rysunek 3: Edycja wierzchołków, krawędzi i płaszczyzn

Blender umożliwia renderowanie obrazu wyjściowego przy pomocy silników Eevee oraz Cycles. Eevee jest silnikiem czasu rzeczywistego, co oznacza, że podgląd jest generowany natychmiastowo. Z tego powodu jest domyślnie ustawiony jako aktywny, pomimo tego, że obraz wyjściowy nie jest najlepszej jakości [6]. Cycles w porównaniu do Eevee uwzględnia zaawansowane zachowanie światła i materiałów, dzięki czemu generuje bardziej realistyczne sceny. Niemniej jednak, nie jest on w stanie wykonać działań na tyle szybko, aby obraz był generowany w czasie rzeczywistym, przez co nie jest wykorzystywany podczas pracy w edytorze [7].



Rysunek 4: Wynik renderingu silnika Eevee i Cycles

2.3 Unity

Unity to silnik do tworzenia gier 2D i 3D, zaprezentowany po raz pierwszy na konferencji Worldwide Developers Conference w 2005. Jest to międzyplatformowy silnik pozwalający tworzyć aplikacje na urządzenia mobilne, komputery osobiste oraz konsole. Silnik uważany jest za łatwy w użyciu dla początkujących twórców, a dodatkowo dzięki swojemu modelowi biznesowemu, często jest wybierany do produkcji gier niezależnych. Umożliwia pisanie skryptów przy użyciu 2 języków programowania: JavaScript, C# [aa]. 28 października 2009 roku, silnik przeszedł z licencji płatnej na darmową dla twórców, których dochód nie przekracza 100 tysięcy rocznie. Silnik posiada możliwość zaimportowania bibliotek DLL (*Dynamic-Link Library*), których można używać podczas pisania skryptów. Oprócz tego Unity pozwala także na skorzystanie z *Asset Store*, czyli sklepu internetowego, z płatnymi lub darmowymi pakietami modeli, tekstur oraz skryptów. Unity 2021 wprowadziło dużo nowych funkcji, takich jak nowa biblioteka do obsługi gier wieloosobowych, chmury wolumetryczne i poprawione rzucanie cieni dla potoku renderowania HDRP czy globalne oświetlenie przestrzeni ekranu.



Rysunek 5: Logo Silnika Unity, źródło: [8]

2.3.1 High Definition Render Pipeline (HDRP)

Omawiając to zagadnienie należy wpierw wyjaśnić, że potok renderowania to zbiór kroków, które muszą zostać wykonane, aby wyświetlić utworzoną scenę na ekranie. Składa się to głównie z 3 etapów: wykluczenie z renderowania obiektów zasłoniętych przez inne obiekty (*Culling*), wygenerowanie klatki, przetwarzanie końcowe (*post processing*).

Potok renderowania w wysokiej rozdzielczości (HDRP) jest natomiast potokiem sterowanym przy pomocy skryptów C# (*Scriptable Render Pipeline*), wykorzystującym techniki obliczania fizycznie poprawnego oświetlenia, oświetlenia liniowego (oświetlenie punktowe emitowane wzdłuż linii) oraz oświetlenia HDR (*High Dynamic Range*), przy

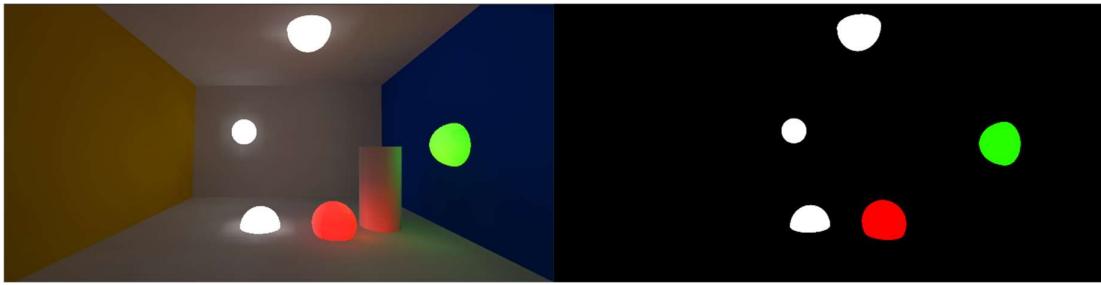
czym informacje o kolorze pikseli w oświetleniu HDR są przechowywane przy użyciu liczb zmiennoprzecinkowych. HDRP jest wspierany przez Unity 2019 LTS i wyższe wersje silnika oraz posiada własny stos post-processingu wysokiej jakości [9]. Dodatkowym wsparciem, które możemy zaobserwować to również wsparcie dla ray tracingu, VFX Graph, Shader Graph i najnowocześniejsze funkcje graficzne. Warto wspomnieć, że aby aktywować ray tracing, projekt Unity trzeba utworzyć w konfiguracji HDRP + DXR.



Rysunek 6: Okno konfiguracyjne HDRP

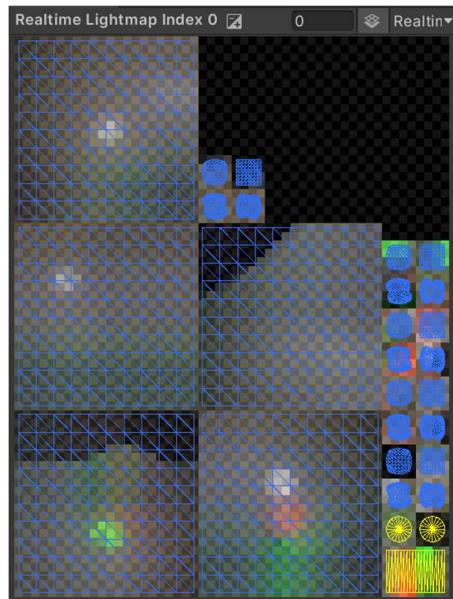
2.3.2 Globalne oświetlenie

Globalne oświetlenie (*global illumination*) to model oświetlenia w grafice trójwymiarowej, w którym każdy obiekt na scenie 3D oświetlany jest zarówno przez światło emitowane bezpośrednio ze źródła światła, jak również przez światło odbite od innych obiektów na scenie (w przeciwnieństwie do oświetlenia lokalnego, gdzie obiekty oświetlane są wyłącznie bezpośrednio przez źródło światła) [10]. Taki model wymaga zastosowania algorytmów, które są bardziej zaawansowane niż algorytmy oprogramowania średniego, jednak dzięki temu możliwe jest zwiększenie stopnia realizmu.



Rysunek 7: Włączone i wyłączone użycie globalnego oświetlenia

Do tej pory tradycyjne aplikacje wykorzystujące techniki oświetlenia w czasie rzeczywistym korzystały wyłącznie z oświetlenia lokalnego. Związane to było z faktem, że oświetlenie pośrednie wymagało dodatkowego czasu na obliczenia, przez co wykorzystywane było tylko w przypadkach, gdzie możliwe było prerenderowanie wyświetlanej grafiki. Dlatego też w grach video wykorzystuje się technikę wypalania światła (obliczanie i przechowywanie informacji o padaniu światła) na obiektach statycznych, czyli takich co do których istnieje pewność, że nie będą się poruszać. W silniku Unity technika ta jest nazywana *Baked Global Illumination* lub *Baked Lightmaps* [11].



Rysunek 8: Wypalone światło na statycznych obiektach

Oprócz obliczania światła pośredniego, *Baked Global Illumination* generuje realistyczne, miękkie cienie na podstawie świateł obszarowych i światła pośredniego. Ponadto, w momencie gdy dany obiekt lub źródło światła zmieni swoje położenie, pojawią się potrzeba ponownego obliczenia informacji o oświetleniu. W Unity od wersji 5.0 udostępniono również obsługę techniki *Precomputed Realtime GI*. Generowane przez nią oświetlenie wciąż dotyczy tylko obiektów statycznych, ale oblicza wszystkie możliwe odbicia światła i zapisuje informację do wykorzystania w czasie rzeczywistym. Pozwala to na modyfikowanie materiałów przypisanych do obiektów oraz samych źródeł światła, ponieważ silnik odczyta z wcześniej obliczonych danych w jaki sposób obiekt ma reagować

ze światłem. Niemniej jednak, jeśli w oświetleniu sceny zostanie wprowadzone dużo zmian, silnik będzie potrzebował więcej czasu na znalezienie i zastosowanie odpowiednich wariantów dla obiektów co prowadzi do konkluzji, że z platformami o mniejszej mocy lepiej poradzi sobie *Baked Global Illumination*.

2.4 Unreal Engine

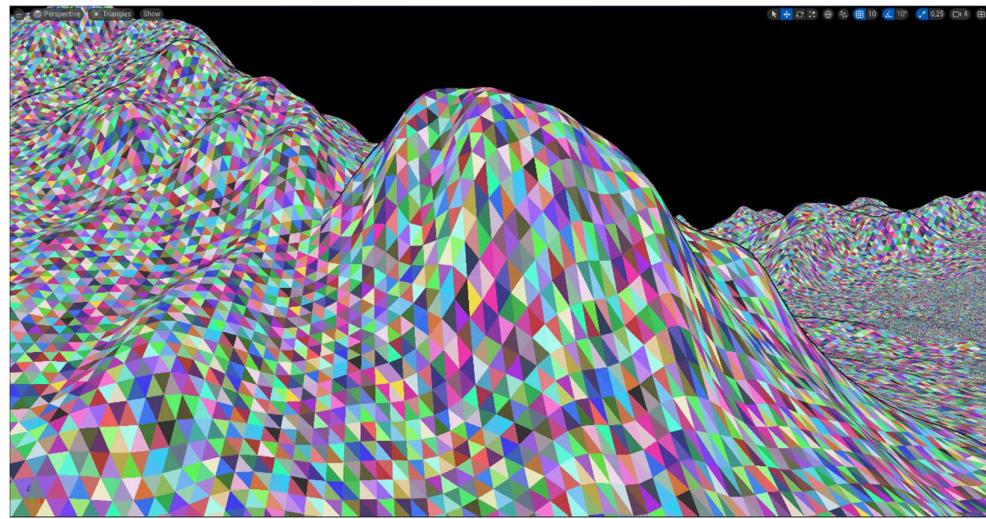
Unreal Engine to silnik stworzony przez firmę Epic Games. Jest to narzędzie do tworzenia gier oraz wysokiej jakości animacji i jednocześnie jeden z najpopularniejszych silników do tworzenia gier na urządzenia mobilne, komputery osobiste oraz konsole. Ze względu na wysoki poziom technologiczny, wykorzystywany jest do tworzenia gier o dużych budżetach, tak zwanych AAA (*triple A*) [aa]. Silnik zadebiutował w 1998 roku w grze Unreal, zdobywając duże uznanie recenzentów ze względu na swoją przełomowość w ówczesnym momencie. Dopiero jednak jego czwarta wersja stała się darmowa dla wszystkich twórców, co zostało ogłoszone rok po premierze tej wersji (w marcu 2015 roku). Piąta, czyli obecnie najnowsza wersja silnika, zadebiutowała 5 kwietnia 2022 i wprowadziła między innymi nową technologię oświetlenia globalnego Lumen, system Nanite pozwalający na budowę scen 3D o dużym poziomie szczegółowości oraz narzędzie MetaHuman Creator do tworzenia realistycznych animacji twarzy [12][13]. Technologie te na tyle istotnie wpłynęły na pracę z samym silnikiem, że zostaną opisane w dalszej części pracy.



Rysunek 9: Logo Silnika Unreal Engine, źródło: [14]

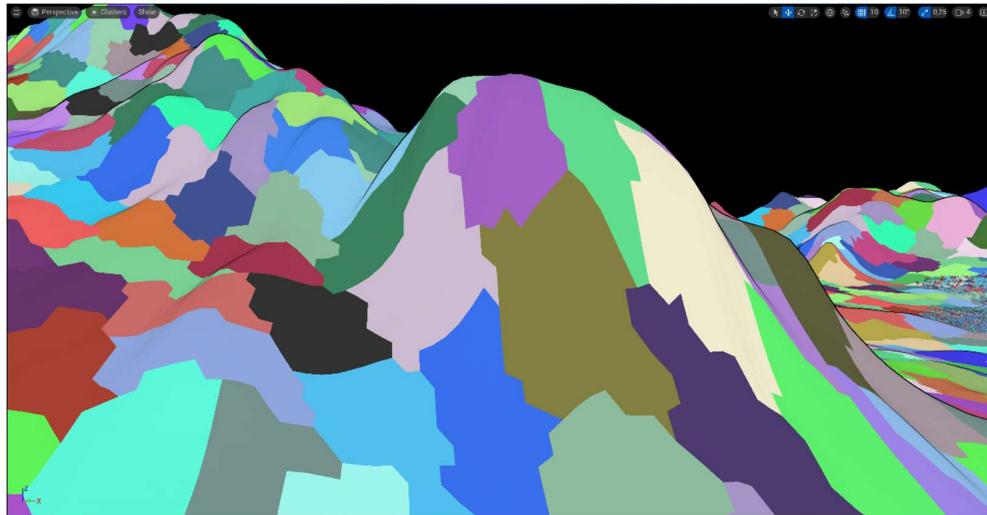
2.4.1 Nanite

System Nanite w silniku Unreal Engine 5 wykorzystuje zwirtualizowaną geometrię mikropolygonów do renderowania szczegółów w skali pikseli i dużej liczby obiektów [15]. Nanite to potok renderowania oparty na procesorze graficznym, co oznacza, że *culling* oraz wybór poziomu szczegółowości odbywa się przy użyciu shaderów obliczeniowych. Całe drzewo klastrów 128-trójkątowych jest obliczane dla każdego obiektu podczas importu lub podczas konwersji na używanie systemu Nanite. Każdy węzeł drzewa posiada co najwyżej 128 trójkątów, a dzieci każdego węzła reprezentują bardziej szczegółowy widok węzła rodzica. Najbardziej szczegółowy widok siatki w Nanite to dokładnie te same trójkąty, które posiada oryginalny obiekt [16].



Rysunek 10: Nanite- podgląd trójkątów

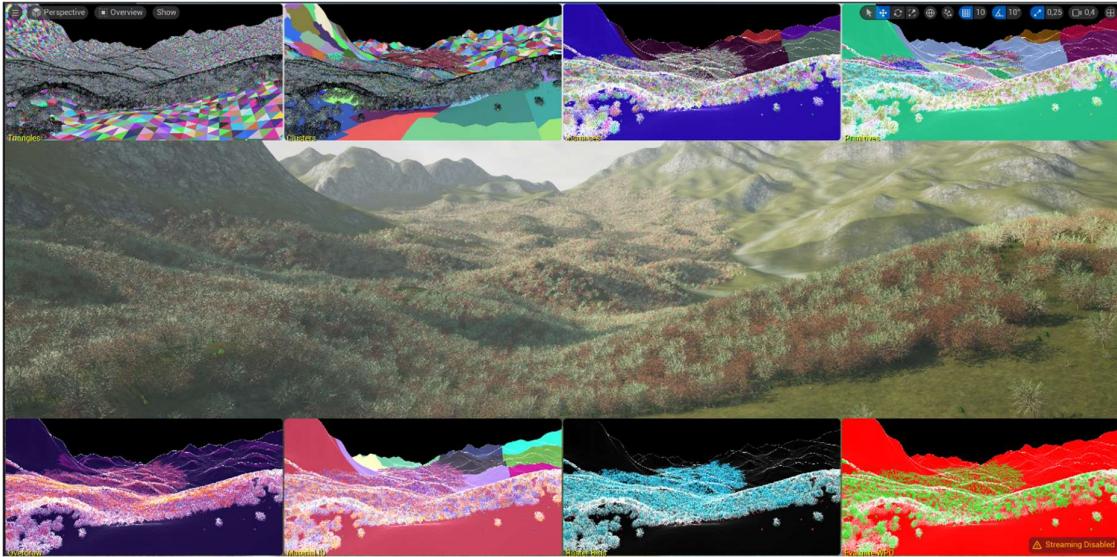
W zależności od odległości obiektu od kamery Nanite samodzielnie wybiera, który poziom klastrów wyświetlić. Aby określić poziom szczegółowości obiektu, obliczana jest różnica między LOD N, a LOD N+1, mająca określić o ile gorszy jest klaster nadzędny od klastrów potomnych.



Rysunek 11: Nanite - podgląd klastrów

Określenie hierarchii klastrów i obliczenie różnic grup LOD pozwala na wybór odpowiednich węzłów szczegółowości danej klatki. *Culling* (czyli wybór, który obiekt nie jest przesłonięty przez inny i który ma zostać wyrenderowany) odbywa się na poziomie klastra. Aby przyspieszyć ten proces, informacja o widoczności obiektów jest wykorzystywana między kolejnymi klatkami. Te dane wyjściowe zapisywane są w tak zwanym “buforze widoczności”, który zawiera dane dotyczące głębi pixela i indeksu klastrów. Warto zaznaczyć, że w celu oszczędzania miejsca i poprawy wydajności, drzewa nie są w całości przechowywane w pamięci RAM. Zdecydowano się natomiast na rozwiązanie, w którym potrzebne fragmenty obiektów są wczytywane z dysku w momencie, kiedy znajdują się w określonym obrębie oraz usuwane, gdy od jakiegoś czasu nie były

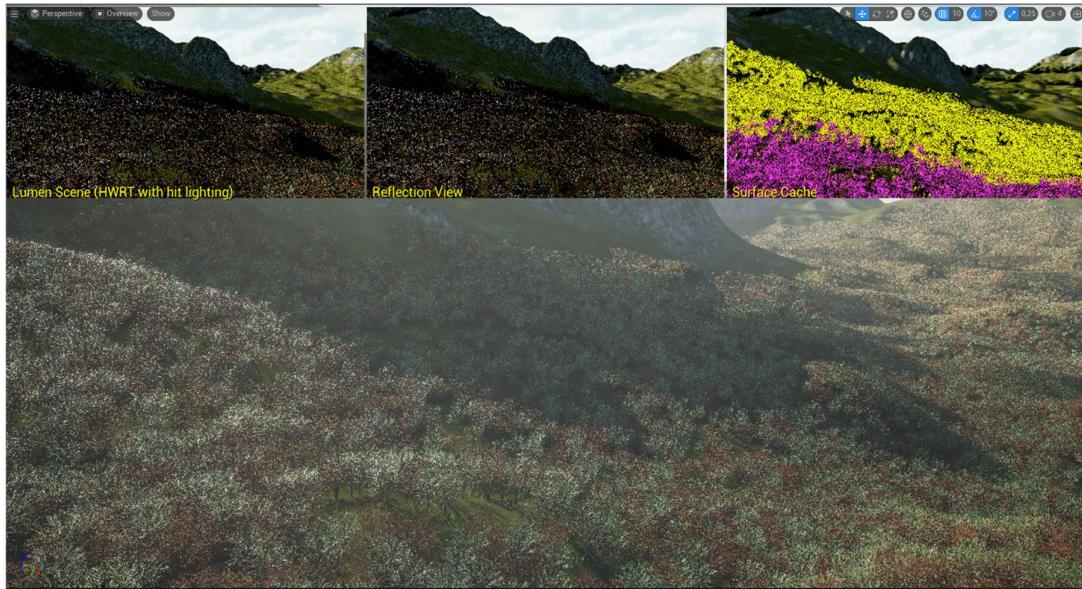
rysowane. Warto w tym miejscu nadmienić, że Nanite i Lumen ściśle współpracują ze sobą, aby zapewnić niezwykłą jakość grafiki w Unreal Engine 5. Nanite zapewnia szczegółową geometrię, a Lumen zapewnia realistyczne oświetlenie i odbicia.



Rysunek 12: Ogólny podgląd Nanite

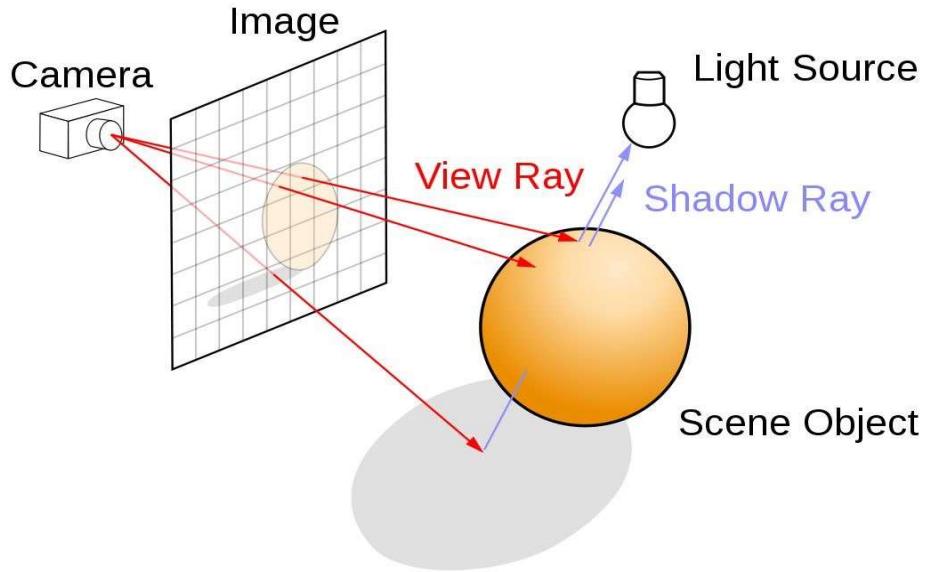
2.4.2 Lumen

Symulowanie zachowania światła w trójwymiarowych światach odbywa się na dwa sposoby: przy użyciu metod oświetlenia w czasie rzeczywistym, które wspierają ruch światła i interakcję światel dynamicznych, lub przy użyciu wstępnie obliczonych informacji o oświetleniu, które są przechowywane w teksturach zastosowanych do geometrycznych powierzchni (*baked*). Unreal Engine zapewnia oba te sposoby oświetlenia scen i można je płynnie łączyć między sobą. Lumen renderuje rozproszone odbicia z nieskończoną liczbą odbić i pośrednie odbicia spekularne w dużych, szczegółowych środowiskach o skalach od milimetrów do kilometrów i w dodatku jest w pełni dynamiczny. Oznacza to, że obiekty na scenie można dowolnie przemieszczać bez ponownego "wypiekania" światła na tekstuze. Światło odbite w sposób rozproszony od powierzchni odbiera kolor tej powierzchni i odbija je na znajdujące się w pobliżu powierzchnie. Lumen generuje automatyczną parametryzację powierzchni pobliskiej sceny o nazwie pamięć podręczna powierzchni (*Surface Cache*). Służy do szybkiego wyszukiwania oświetlenia w punktach trafienia promienia w scenie. Lumen rejestruje właściwości materiału dla każdej siatki pod różnymi kątami. Te pozycje przechwytywania (zwane kartami) są generowane dla każdej siatki [17]. Nanite przyspiesza przechwytywanie siatki używane do synchronizacji pamięci podręcznej powierzchni ze sceną trójkątów. W szczególności siatki wielokątne muszą używać nanitów, aby uzyskać wydajne przechwytywanie. Komponenty liściaste i instancjonowanej siatki statycznej mogą być obsługiwane tylko wtedy, gdy siatka używa nanitów. Po zapełnieniu pamięci podręcznej powierzchni właściwościami materiału, Lumen oblicza bezpośrednie i pośrednie oświetlenie powierzchni. Obliczenia wykonywane są w czasie rzeczywistym i są rozłożone w czasie na wiele klatek, zapewniając wydajną obsługę wielu dynamicznych światel i globalnego oświetlenia z wieloma odbiciami.



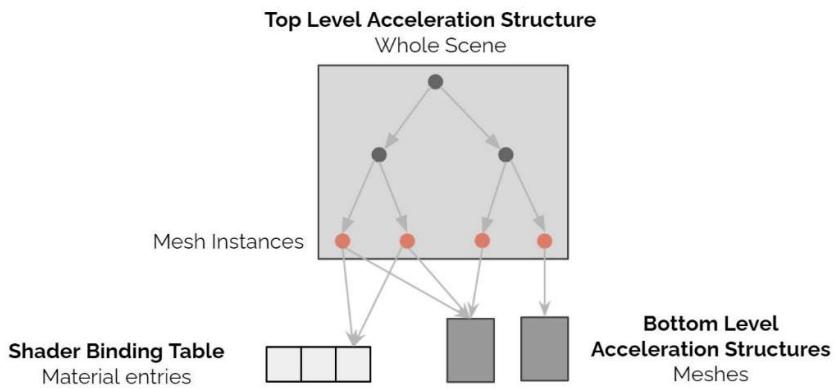
Rysunek 13: Ogólny podgląd Lumen

Śledzenie promieni (*ray tracing*) to technika renderowania, która może realistycznie symulować oświetlenie sceny i jej obiektów poprzez renderowanie fizycznie dokładnych odbić, załamań, cieni i oświetlenia pośredniego. Ray tracing generuje obrazy grafiki komputerowej, śledząc ścieżkę promieni światła z kamery podglądu (która określa widok sceny), przez płaszczyznę pikseli podglądu, na scenę 3D i z powrotem do źródeł światła. Jeśli promień przechodzący przez piksel i wychodzący na scenę 3D uderza w trójkąt, wówczas określana jest odległość wzduż promienia od kamery do trójkąta, a kolor trójkąta ma wpływ na ostateczny kolor wyświetlanego piksela [18]. Promień może również odbijać się i uderzać w inne obiekty, odbierając od nich informacje o kolorze i oświetleniu. Światło może powodować odbicia, cienie lub powodować załamania w przezroczystych i półprzezroczystych obiektach.



Rysunek 14: Schemat podstawowego działania techniki *ray tracing*, źródło: [18]

Takie testowanie każdego promienia jest nieefektywne i kosztowne obliczeniowo, dlatego Lumen wykorzystuje Organizacyjną Hierarchię Woluminów (*Bounding Volume Hierarchy*). BVH ma strukturę drzewiastą, a każdy promień musi być tylko przetestowany przy użyciu procesu przechodzenia drzewa w pierwszej kolejności zamiast każdego prymitywu w scenie.



Rysunek 15: Organizacyjna Hierarchia Woluminów (BVH) w przypadku przechodzenia promienia, źródło [19]

Struktury przyspieszania poziomu górnego zwane *Top Level Acceleration Structure* (TLAS), zawierają wszystkie instancje siatki dla całej sceny, natomiast siatki, do których odwołują się te instancje, to struktury przyspieszania poziomu dolnego - *Bottom Level Acceleration Structures* (BLAS). Struktury przyspieszania poziomu dolnego dla siatek statycznych budowane są raz przy załadunku. Dynamiczne siatki muszą zostać przebudowane z każdą klatką, gdy następuje zmiana na scenie, aby struktura BVH była dopasowana do zmian. Struktury przyspieszenia poziomu górnego przebudowywane są przy

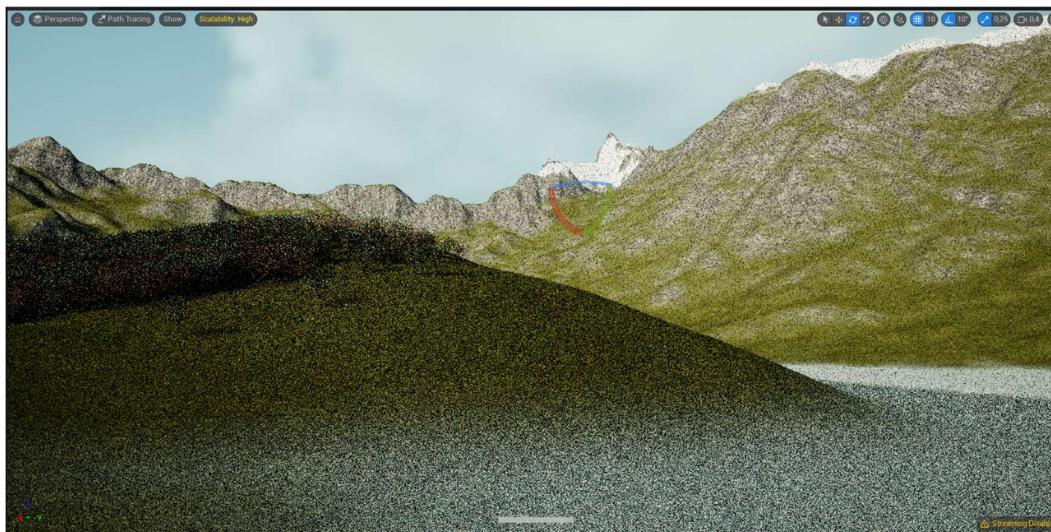
każdej klatce, aby Lumen mógł poradzić sobie z dynamicznym oświetleniem i odbiciami w złożonych scenach, na wypadek wystąpienia animacji, deformacji siatki lub proceduralnego generowania obiektu na scenie [19].

Typy geometrii, przebudowujące się co klatkę obrazu dla BVH to:

- siatki używające GPUSkinCache,
- krajobrazy,
- kolekcje dla systemu destrukcji Chaos,
- włosy,
- siatki proceduralne,
- system cząstek Niagara;

Należy wspomnieć, że Lumen udostępnia dwie metody śledzenia promieni. Pierwszą z nich jest programowe śledzenie promieni, które wykorzystuje pola odległości siatki do działania na najszerzym zakresie sprzętu i platform, ale jest ograniczone pod względem rodzajów geometrii, materiałów i przepływów pracy, z których może skutecznie korzystać. Drugą metodą jest sprzętowe śledzenie promieni, które obsługuje większy zakres typów geometrii w celu uzyskania wysokiej jakości, śledząc trójkąty i oceniacąc oświetlenie w momencie trafienia promienia, zamiast pamiętać podręcznej powierzchni o niższej jakości. Do działania wymaga obsługiwanych kart graficznych i systemów.

Kiedy Lumen korzysta z programowego śledzenia promieni, obejmuje tylko 200 m mierzonych od pozycji kamery, ale można tę wartość zwiększyć do 800 m za pomocą ustawienia *Lumen Scene View Distance* w opcji Post Process Volume. Powyżej największej odległości, dla globalnego oświetlenia aktywne są tylko *Screen Tracing*, czyli pierwsze bezpośrednio wystrzelone promienie z ekranu, nieuwzględniające wpływu obiektów znajdujących się poza kadrem [20].



Rysunek 16: Podgląd Path Tracing

Należy również nadmienić, że istnieje jeszcze bardziej intensywna technika śledzenia o nazwie Śledzenie Ścieżki (*Path Tracing*), która śledzi tysiące promieni przechodzących przez każdy piksel i podąża za promieniami przez liczne odbicia i obiekty przed dojściem do źródła światła w celu zebrania informacji o kolorze i oświetleniu. Unreal Engine udostępnia progresywny tryb renderowania z akceleracją sprzętową o nazwie *Path*

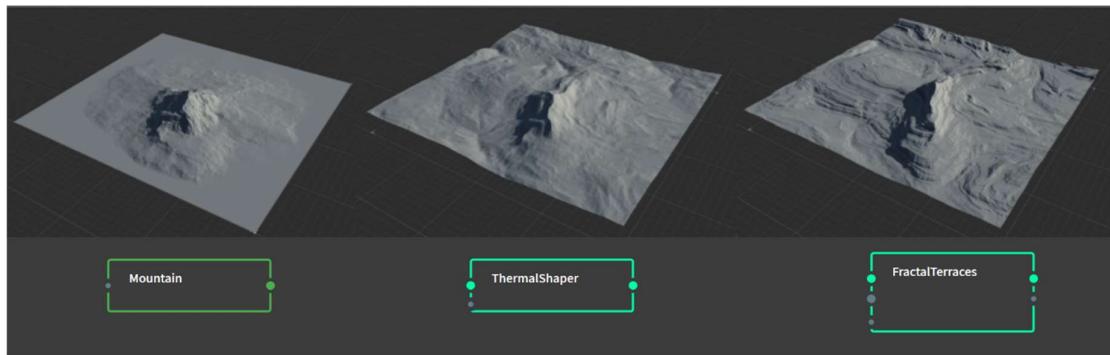
Tracer, który nie działa w czasie rzeczywistym, przez co oferuje fizycznie poprawne i bezkompromisowe globalne oświetlenie [21]. Technika ta pozwala uzyskać czyste i fotorealistyczne rendery oraz korzysta z tej samej architektury śledzenia promieni co wcześniej omawiany *ray tracing*, jednakże wykorzystuje tylko geometrię i materiały obecne w scenie i nie korzysta z tego samego kodu, który został opracowany dla renderowania w czasie rzeczywistym. Oznacza to, że przy każdym ruchu kamery lub zmiany na scenie, wykonywane są na nowo obliczenia. Objawia się to pojawiением się ziarna w pierwszej fazie obliczeń i renderingu. W podobny sposób działa silnik *cycles* w Blenderze.

3. Przygotowanie scen

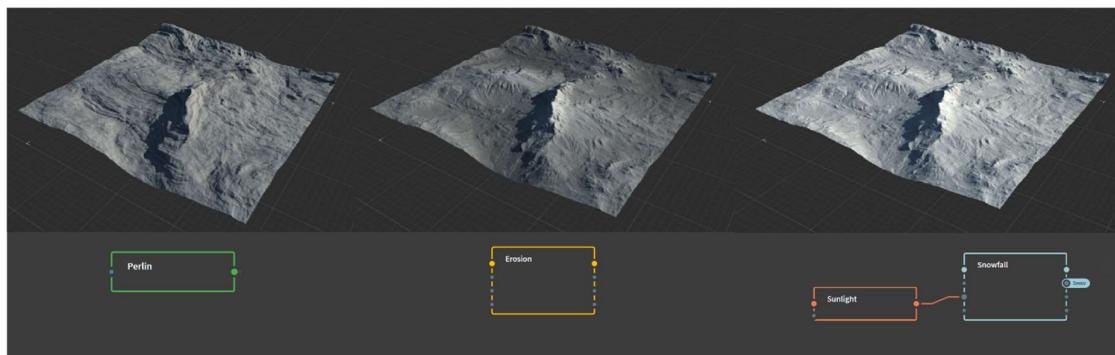
Kluczowym krokiem dla porównania wydajności silników było przygotowanie scen. W tym celu utworzono po 3 sceny w każdym z silników, korzystając z utworzonych wcześniej obiektów 3D oraz map wysokości i masek dla teksturowania wygenerowanych terenów. Następnie skonfigurowano oświetlenie, wolumetryczne chmury i mgłę, zimportowano modele LOD i utworzono odpowiednie materiały i skrypty obracania źródła światła. Rozdział ten będzie się zatem koncentrował na technicznym opisie całego procesu przygotowywania poszczególnych scen i ich konfiguracji w obu silnikach.

3.1 Wygenerowanie terenów

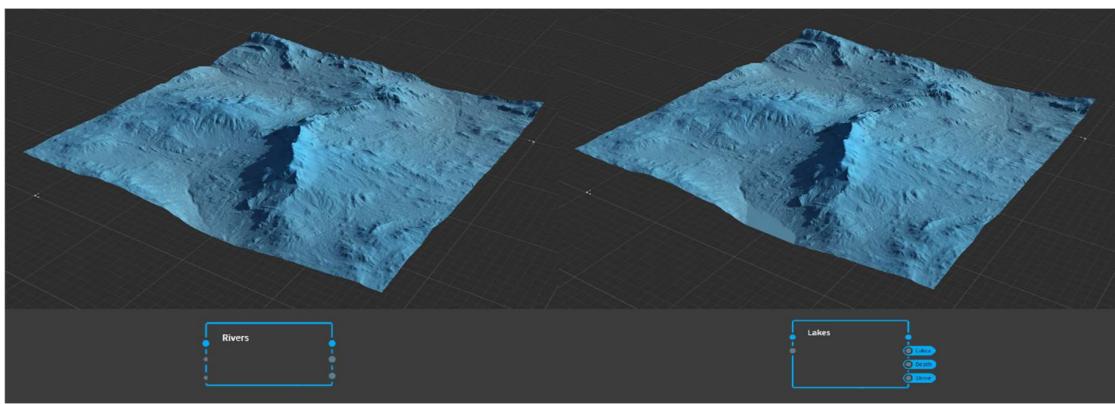
Aby wygenerować naturalnie wyglądający teren przy użyciu programu Gaea, trzeba zasymulować wpływ na teren zjawisk atmosferycznych, przebieg i moc rzek, erozję gleby, wpływ temperatury czy roztopy. W pierwszym kroku wybrano węzeł *Mountain* generujący górę, następnie przy użyciu węzła *ThermalShaper* symulowany jest wpływ termicznej erozji na teren. W kolejnym kroku użyto węzła *FractalTerrace*, które tworzy rozwarstwienie terenu, dodatkowo pozwalając na ustawienie twardości skał, kształtu krawędzi i ogólny charakter utworzonych tarasów. Następnie dodano szum wygenerowany przez węzeł *Perlin* i otrzymany wynik poddano erozji przy użyciu węzła *Erosion*. Kolejnym krokiem było utworzeni śnieżnych szczytów, dla których wcześniej trzeba było ustawić kierunek padania i mocy promieni słonecznych, w celu uwzględnienia zjawiska roztopów w symulacji. Następnie utworzono koryta rzek oraz jeziora.



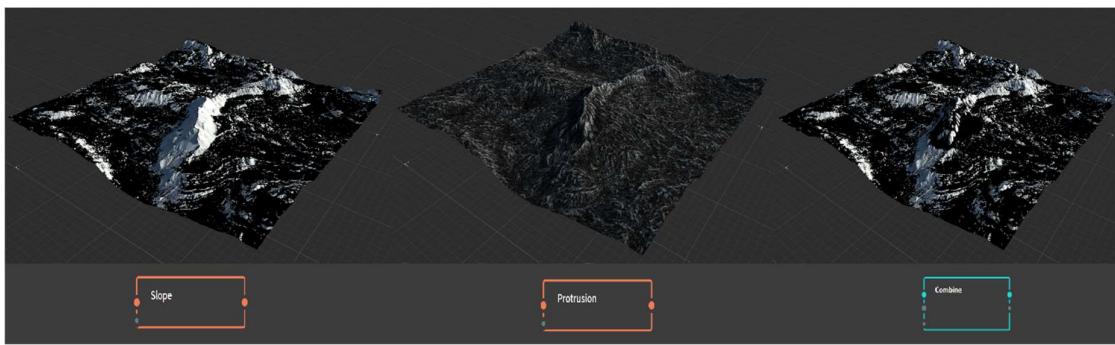
Rysunek 17: Wyniki działania węzłów Mountain, ThermalShaper i FractalTerraces



Rysunek 18: Wyniki działania węzłów Perlin, Erosion, Snowfall

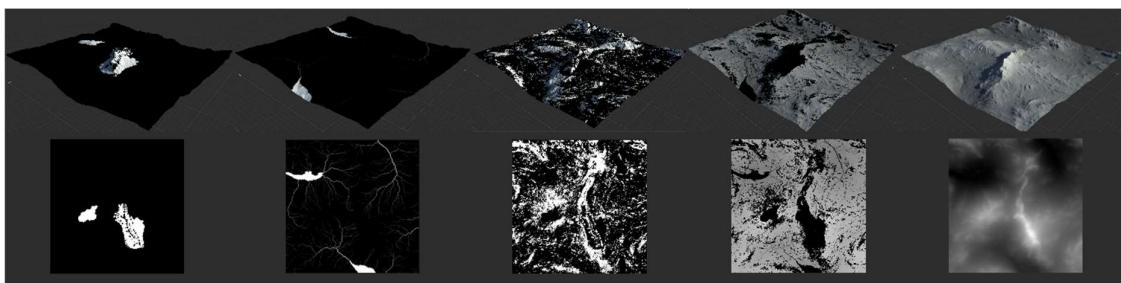


Rysunek 19: Wyniki działania węzłów Rivers i Lakes



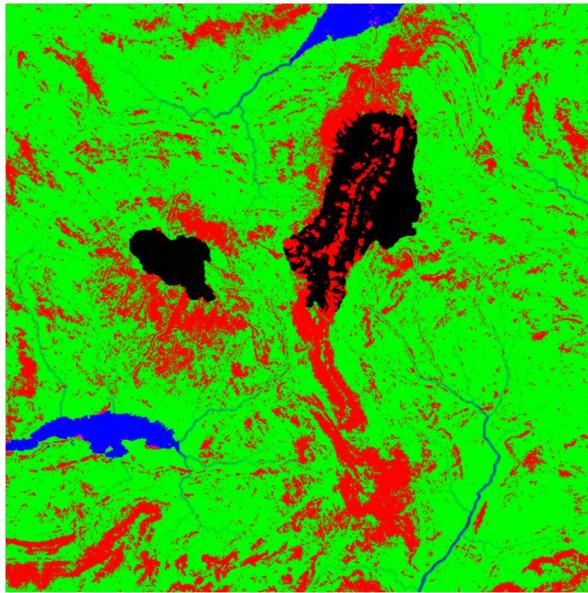
Rysunek 20: Wyniki działania węzłów Slope, Petrusion i ich kombinacji z odjęciem śniegu

Na tym etapie finalny kształt terenu został już uformowany, ale dla oteksturowania go w silnikach potrzebne było wygenerowanie masek dla warstw tekstur w Unreal Engine oraz splatmapy dla silnika Unity. Maski są używane jako źródło informacji, gdzie na terenie narysować dany materiał. Dla Unreal Engine wszystkie maski oraz mapy wysokości (*heightmap*) eksportuje się w formacie png oraz w skali szarości, gdzie dla masek istnieje zależność, że im większa wartość piksela tym większa waga, a dla map wysokości: im większa wartość piksela, tym wyższy punkt terenu.



Rysunek 21: Wyeksportowane maski śniegu, wody, kamieni, trawy oraz mapa wysokości dla Unreal Engine.

Tak jak zostało wspomniane wyżej, dla silnika Unity została wygenerowana splatmapa terenu. Splatmapa to mapa posiadająca najczęściej 4 kolory, dającą również informacje o użyciu tekstu i ich ilości na terenie. Na przedmiotowej mapie kolor niebieski oznacza wodę, kolor zielony trawę, kolor czerwony kamienie, a kolor czarny śnieg. Dla Unity splatmapa została wyeksportowana jako plik w formacie png, a mapa wysokości jako plik w formacie raw.



Rysunek 22: Splatmapa terenu

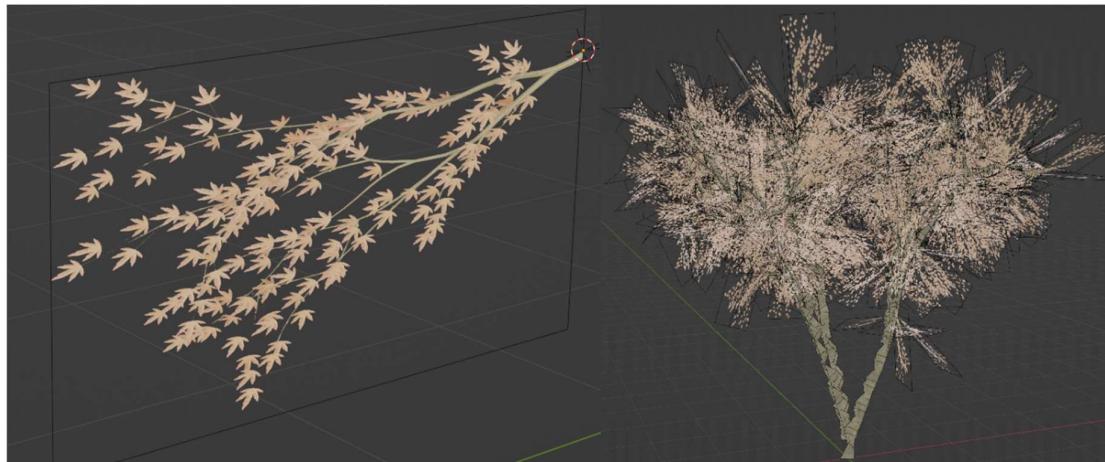
Dodatkową różnicą pomiędzy oboma silnikami jest akceptowana przez nie rozdzielcość plików. Należy zatem nadmienić, iż dla Unreal Engine rozdzielcość plików wyniosła 1009x1009 px, a dla Unity 1025x1025 px. Teren wyeksportowano w 3 wariantach: 10x10x2.5 km, 20x20x5 km oraz 40x40x10 km.

3.2 Stworzenie modeli 3D

Łącznie utworzono 12 modeli drzew oraz 4 modele trawy. Drzewa podzielono na 3 zbiory po 4 modele, które posłużyły do utworzenia grup LOD. Dla trawy wszystkie 4 modele będą zgrupowane w jeden obiekt. Na rysunku 22 znajdują się dwa przykładowe modele drzew, przy czym po lewej stronie mamy obiekty o najmniejszej ilości wierzchołków, a po prawej z największą ilością wierzchołków. Obiekty o najmniejszej ilości wierzchołków zbudowane są z 8 płaszczyzn wyświetlających teksturę, przygotowaną z kolei z wcześniej stworzonej tekstyury wyrenderowanego drzewa. Kolejny poziom szczegółowości wykorzystuje tę technikę do renderowania gałęzi z liśćmi, a następne obiekty są już w pełni wymodelowane. Nie ulega wątpliwości, że jest to często wykorzystywany sposób optymalizacji drzew w grach wideo. Modele zostały wyeksportowane jako pliki fbx.



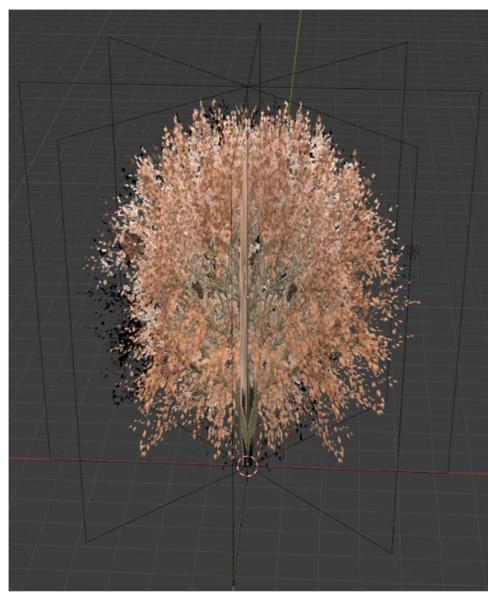
Rysunek 23: Modele drzew zbudowanych przy pomocy płaszczyzn z gry *Dark Souls*, opracowano na podstawie [22]



Rysunek 24: Gałęzie jako oteksturowana płaszczyzna



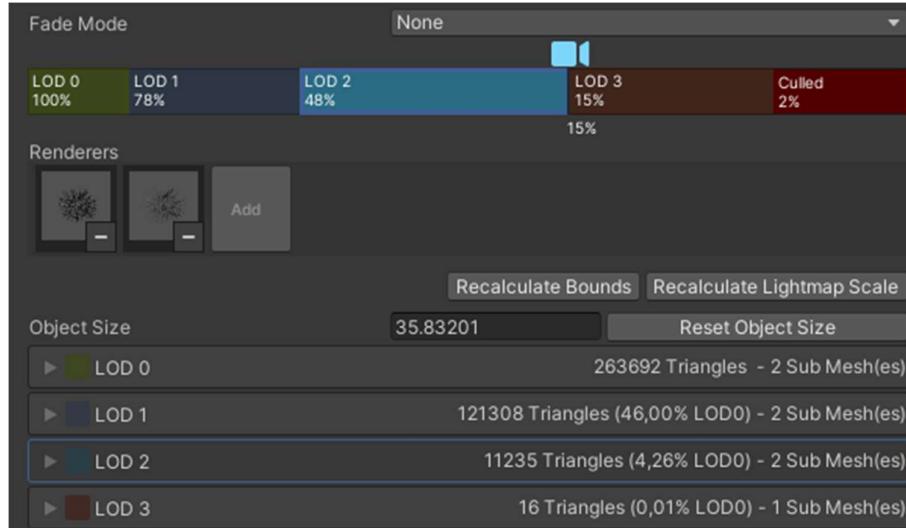
Rysunek 25: Utworzone modele 3D



Rysunek 26: Najmniej szczegółowy obiekt grup poziomu detali zbudowany z 8 płaszczyzn

3.3 Import danych do silnika Unity

Pierwszym krokiem było zainportowanie obiektów 3D do silnika. Dla każdej grupy obiektów 3D utworzono oddzielne foldery, do których je zainportowano. Dla obiektów posiadających materiały z informacjami o przezroczystości należało zmodyfikować materiały, ustawiając w opcjach materiału typ płaszczyzny na przezroczysty, z trybem mieszania ustalonym na kanał *alpha*. Następnie należało utworzyć *prefab*, czyli szablon właściwości i komponentów danego modelu. Wewnątrz takich szablonów zdefiniowano grupy szczegółowości obiektów.



Rysunek 27: Tworzenie obiektu z grupami LOD w silniku Unity

Następnym etapem było przygotowanie oświetlenia. Dla sceny HDRP należało utworzyć *Volume Profile*, w którym uruchomiono tworzenie cieni, okluzję otoczenia, volumetryczne chmury, niebo HDRI i jego typ. Po dodaniu kierunkowego źródła światła, należało zmienić tryb na czas rzeczywisty, włączyć wpływ na obiekty volumetryczne oraz renderowanie map cieni z użyciem cieni obliczanych przy pomocy *ray tracing*. Dodatkowo, w projektowych ustawieniach światła należało uruchomić globalne oświetlenie w czasie rzeczywistym. Dla dynamicznych scen dodano ruch źródła światła przy pomocy skryptu C#.



Rysunek 28: Scena z zainportowaną mapą i skonfigurowanym globalnym oświetleniem

```
using UnityEngine;

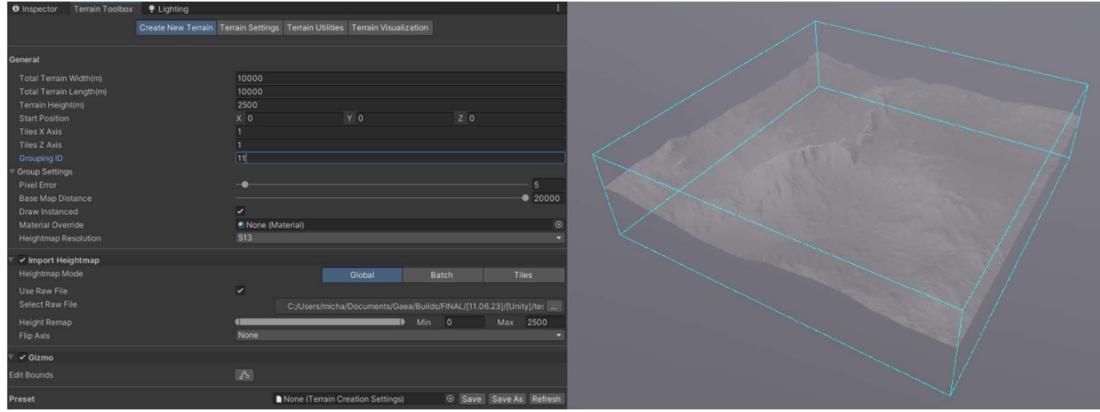
[ExecuteAlways]
public class SunRotation : MonoBehaviour
{
    [SerializeField] private Light directionalLight;
    private float temp = 180;

    private void Update()
    {
        if (Application.isPlaying) {
            temp += Time.deltaTime / 2f;
            temp %= 24;
            directionalLight.transform.localRotation = Quaternion.Euler(new Vector3((temp * 360f / 24f) - 180, 170f, 0));
        }
    }
}
```

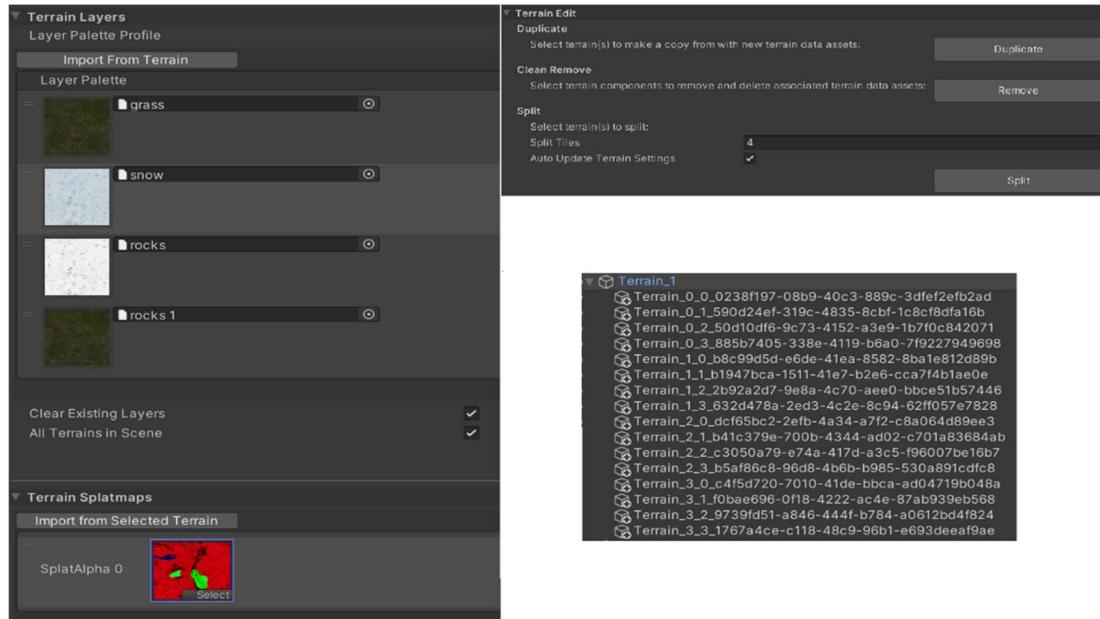
Listing 1: Kod obrotu źródła światła w silniku Unity

Kolejnym krokiem było stworzenie terenu na podstawie wygenerowanej wcześniej mapy wysokości. Unity udostępnia pakiet narzędzi do edycji terenów o nazwie *Terrain Tools*, który nie jest domyślnie zainstalowany, należy go zatem zainstalować za pomocą menedżera pakietów. Aby zainportowany teren miał poprawną skalę, w informacjach generalnych należy podać jego szerokość, długość, wysokość oraz rozdzielcość zgodną z podanymi przy eksportie mapy wysokości w programie Gaea. Podczas wybierania pliku mapy wysokości trzeba zaznaczyć, że używany będzie plik z rozszerzeniem raw. Po utworzeniu terenu, w zakładce *Terrain Settings*, można dostosować wartości związane z drzewami oraz trawą. W *Terrain Utilities* można zarządzać warstwami tekstur, wskazać splatmapę, która określi w jaki sposób mają zostać wykorzystane warstwy tekstur na terenie oraz pozwalającą na podzielenie terenu na mniejsze fragmenty, co jest ważnym elementem przy wypalaniu

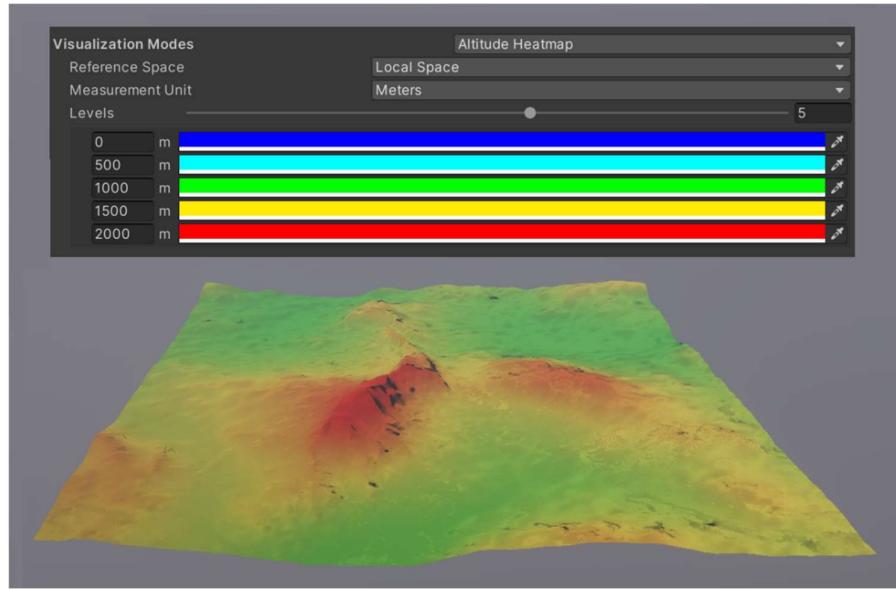
globalnego oświetlenia. Ostatnia zakładka to *Terrain Visualisation*, która pozwala nam sprawdzić, czy teren został prawidłowo wyskalowany. Można wybrać ilość poziomów, ich wysokość początkową oraz kolor. Teren jest tymczasowo zakolorowywany według ustawionych parametrów, co pozwala na zwizualizowanie i weryfikację wysokości terenu.



Rysunek 29: Tworzenie nowego terenu w silniku Unity



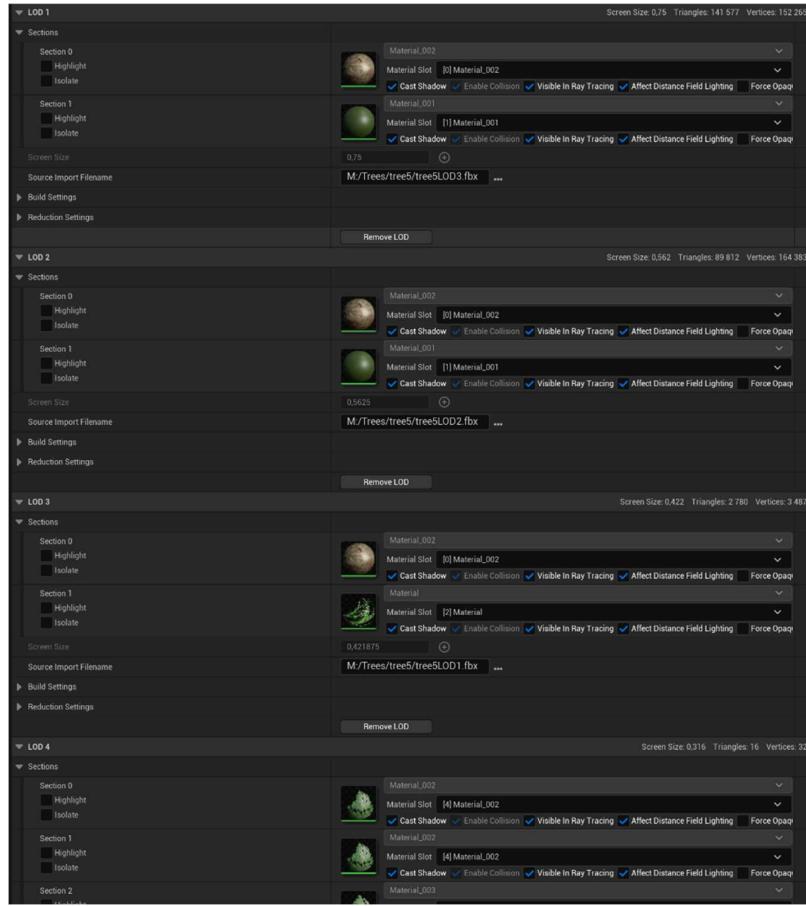
Rysunek 30: Warstwy tekstur, import splatmapy oraz podział terenu



Rysunek 31: Wizualizacja poziomów wysokości terenów

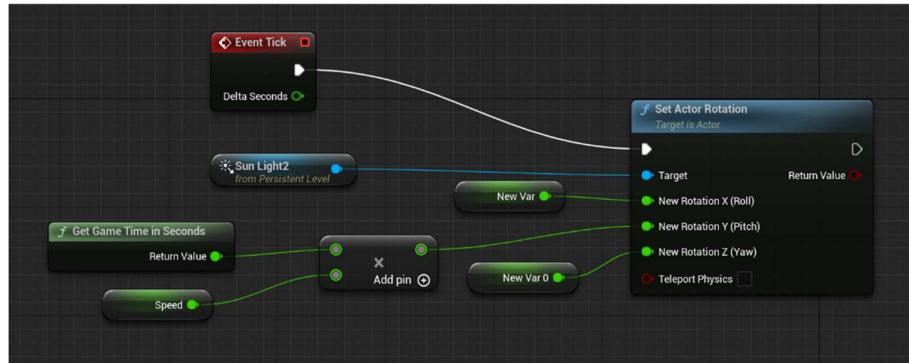
3.3 Import danych do silnika Unreal Engine

Podobnie jak w silniku Unity, zaczęto od zaimportowania do projektu obiektów drzew oraz trawy do oddzielnych folderów. Podczas importowania plików w oknie *FBX Import Options* należało zaznaczyć opcję *Build Nanite* jako aktywną. Zaimportowane materiały trzeba było poprawić, przypisując odpowiednie tekstury do właściwości *Base Color*, z poprawionymi wartościami *UTiling* i *VTiling* w komponencie *TexCoord*. W celu dodania falowania tekstur np. liści, do właściwości *World Position Offset* należało dodać wynik węzła *SimpleGrassWind*. Dla modeli, które mają posiadać przezroczyste tła, *Blend Mode* należało ustawić na *Masked*, a do właściwości *Opacity Mask* wskazać kanał *alpha* tekstury.



Rysunek 32: Tworzenie obiektu z grupami LOD w silniku Unreal Engine

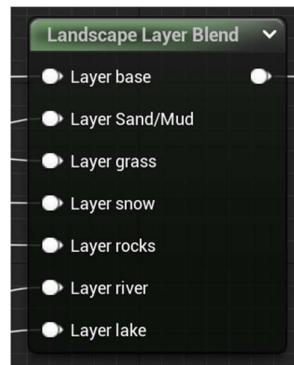
Oświetlenie dodano do sceny przy pomocy miksera oświetlenia środowiska. Zostało dodane słoneczne światło kierunkowe, atmosfera, wolumetryczne chmury i mgła. Uruchomione zostało renderowanie cieni utworzonych przez światło kierunkowe oraz włączono tworzenie cieni przez chmury. W opcjach projektu w sekcji *Platforms-Windows*, wymagane było aktywowanie docelowego formatowania shaderów SM5 i SM6 dla biblioteki DirectX 12 oraz ustawienie DirectX 12 jako docelowego Renderującego Interfejsu Sprzętowego (*RHI*). Następnie w sekcji *Engine-Rendering* należało uruchomić opcje związane z *ray tracingiem* (*Path Tracing*, *Ray Tracing Shadows*). W tej samej sekcji w ustawieniach systemu Lumen została zaznaczona opcja korzystania z *ray tracingu* sprzętowego [20][23]. Następnie należało zaznaczyć, że Lumen był odpowiedzialny za dynamiczne globalne oświetlenie oraz aktywować system Nanite. Dla dynamicznych scen utworzono *blueprint* obracający źródło światła.



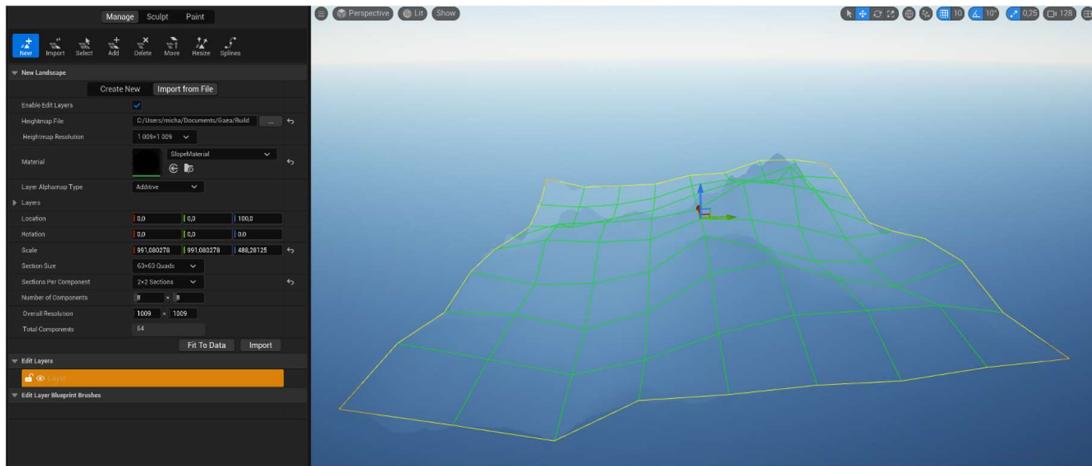
Rysunek 33: Schemat dla obracania źródła światła w Unreal Engine

Przed zimportowaniem terenu z mapy wysokości utworzono materiał, mający za zadanie mieszać tekstury i nanosić je w zależności od kąta nachylenia terenu. Dla każdej tekstury, która ma być nałożona na teren, należało dodać warstwę w materiale. Podczas importu terenu z pliku należało zaznaczyć opcję *Enable Edit Layers* oraz przypisać wcześniej utworzony materiał. Po wyborze pliku z mapą wysokości należało odpowiednio ustawić skalę. Wartości X i Y trzeba było pomnożyć o rozmiar terenu w danej osi i podzielić przez rozdzielcość mapy wysokości, czyli przez 1009. Co więcej, wartość Z należało pomnożyć o wysokość terenu i podzielić przez 512. Jest to związane z tym, że skala jednostki wysokości w silniku Unreal Engine ma wartości z zakresu od -256 do 256 i wartość tę trzeba dostosować do tego zakresu [24].

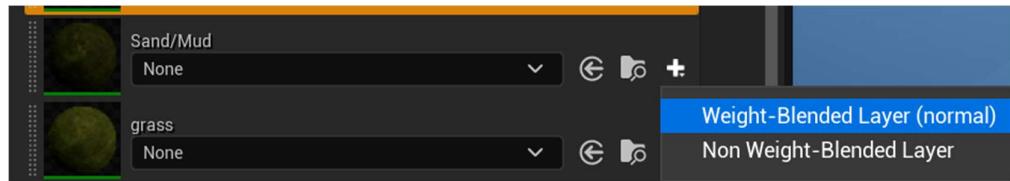
Po zimportowaniu terenu należało przejść do zakładki kolorowania terenu i dla każdej warstwy tekstu określić, że jest mieszana wagowo z innymi. Warstwę *base* wypełniono automatycznie (uzycie tekstu w zależności od nachylenia terenu), a pozostałe warstwy uzupełniono w menu *Manage* w zakładce *Import* przy pomocy wygenerowanych w programie Gaea masek.



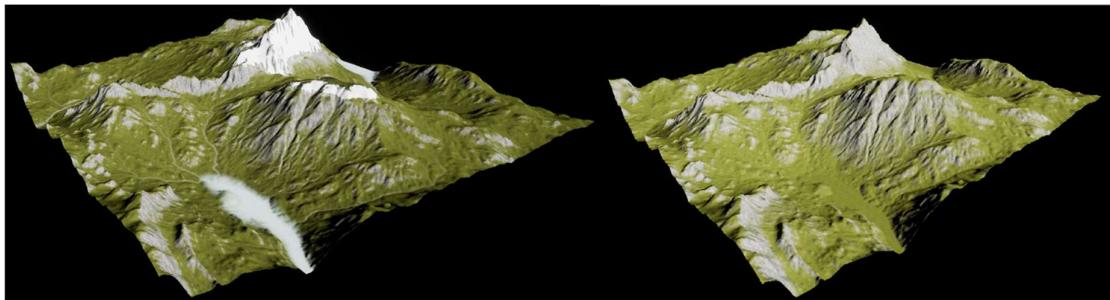
Rysunek 34: Warstwy tekstu w materiale terenu Unreal Engine



Rysunek 35: Tworzenie terenu w silniku Unreal Engine



Rysunek 36: Określenie typu warstw tekstur



Rysunek 37: Wypełnienie wszystkich warstw i samej warstwy base

4. Badanie wydajności

Jednym z głównych wskaźników wydajności jest liczba generowanych klatek na sekundę, a ona z kolei jest ściśle powiązana z obciążeniem nie tylko procesora, ale również karty graficznej i pamięci RAM. Obciążenie procesora CPU (Central Processing Unit) wpływa na wolniejsze symulowanie fizyki, wykonywanie kodu, obciążenie karty graficznej na prędkość generowania grafiki, gdzie z kolei obciążenie pamięci RAM wpływa na spowolnienie działania procesora CPU z powodu braku miejsca w pamięci na przechowywanie danych wykorzystywanych przez procesor. Z tego względu przedstawione parametry zostały użyte do dokonania porównania obu silników. Wartości obciążenia procesora i karty graficznej zmierzono w procentach, a pamięci RAM w gigabajtach. Do porównania użyto silników w wersjach Unreal Engine 5.2 i Unity 2021.3.9.

Platformę sprzętową na której wykonywano badania zaprezentowano w tabeli 1.

Tabela 1: Konfiguracja sprzętowa

Procesor (CPU)	Intel Core i5-12400F 2.5 GHz	6 rdzeni, 12 wątków
Karta graficzna (GPU)	NVIDIA GeForce RTX 3060 Ti	8 GB
Pamięć RAM	Corsair Vengeance LPX	DDR4 64 GB 3200 MHz

Dla takiej konfiguracji komponentów *bottleneck*, czyli wąskie gardło, praktycznie nie istnieje, co pozwala na pominięcie tego zagadnienia w analizie. Według obliczeń dokonanych przy pomocy kalkulatora pc-builds.com, dla rozdzielczości 1920 x 1080, przy możliwym stu procentowym wykorzystaniu procesora, karta graficzna może być wykorzystana w 96,8%, co daje 3,2% wartości wąskiego gardła. Taka wartość uznawana jest za niewpływającą na wydajność[31][32]. Dla zbierania danych w silniku Unity wykorzystano skrypt pobierający i zapisujący dane do pliku (Listing 2) [34]. Dla silnika Unreal Engine posłużono się wbudowaną możliwością sprawdzania ilości klatek na sekundę oraz pamięci. Dla sprawdzenia obciążenia karty i procesora wykorzystano monitor wydajności systemu Windows 10.

```
using System.Text;
using UnityEngine.Profiling;
using UnityEngine;
using System.IO;

public class RenderStatsScript : MonoBehaviour
{
    string statsText;
    ProfilerRecorder verticesRecorder;
    ProfilerRecorder mainThreadTimeRecorder;
    ProfilerRecorder systemMemoryRecorder;
    ProfilerRecorder gcMemoryRecorder;
    System.Diagnostics.PerformanceCounter cpuCounter;
    string path = "Assets/Resources/stats.txt";
    StreamWriter writer;

    static double GetFrameAverage(ProfilerRecorder recorder)
    {
```

```

        var counts = recorder.Capacity;
        if (counts == 0)
            return 0;
        double r = 0;
        unsafe
        {
            var samples = stackalloc ProfilerRecorderSample[counts];
            recorder.CopyTo(samples, counts);
            for (var i = 0; i < counts; ++i)
                r = r + samples[i].Value;
            r = r / counts;
        }

        return r;
    }
    void OnEnable()
    {
        this.cpuCounter = new
System.Diagnostics.PerformanceCounter("Processor", "% Processor Time",
System.Diagnostics.Process.GetCurrentProcess().ProcessName);
        verticesRecorder =
ProfilerRecorder.StartNew(ProfilerCategory.Render, "Vertices Count");
        systemMemoryRecorder =
ProfilerRecorder.StartNew(ProfilerCategory.Memory, "System Used Memory");
        mainThreadTimeRecorder =
ProfilerRecorder.StartNew(ProfilerCategory.Internal, "Main Thread", 15);
        gcMemoryRecorder = new
ProfilerRecorder(ProfilerCategory.Memory, "GC Reserved Memory", 1,
ProfilerRecorderOptions.Default |
ProfilerRecorderOptions.StartImmediately);
    }
    void OnDisable()
    {
        verticesRecorder.Dispose();
        systemMemoryRecorder.Dispose();
        mainThreadTimeRecorder.Dispose();
        gcMemoryRecorder.Dispose();
        this.writer.Close();
    }
    void Update()
    {
        if (this.writer == null)
            this.writer = new StreamWriter(path, false);
        var sb = new StringBuilder(1000);
            sb.AppendLine($"Frame Time:
{GetFrameAverage(mainThreadTimeRecorder) * (1e-6f):F1} ms");
        sb.AppendLine($"Frame Time: {1000 /
(GetFrameAverage(mainThreadTimeRecorder) * (1e-6f)):F1} fps");
        if (systemMemoryRecorder.Valid)

```

```

        sb.AppendLine($"RAM Memory:
{systemMemoryRecorder.LastValue / (1024 * 1024)} MB");
        if (gcMemoryRecorder.Valid)
            sb.AppendLine($"Graphic Card Memory:
{gcMemoryRecorder.LastValue / (1024 * 1024)} MB");
        sb.AppendLine($"GPU Usage: {((gcMemoryRecorder.LastValue /
(1024 * 1024)) / 8192f) * 100} %");
        sb.AppendLine($"CPU Usage: {this.cpuCounter.NextValue()} %");

        statsText = sb.ToString();
        this.writer.WriteLine(statsText);
    }
}

```

Listing 2: Zapisywanie danych o zużyciu CPU, GPU, RAM dla silnika Unity do pliku

Każdy wariant testu trwał minutę i był powtarzony pięć razy, a w tabelach od drugiej do siódmej przedstawiono uśrednione wyniki.

Tabela 2: Wyniki badania wpływu wielkości terenu (oswietlenie statyczne)

Rozmiar terenu	Unity				Unreal			
	CPU	GPU	RAM	FPS	CPU	GPU	RAM	FPS
10x10 x2.5km	17,53%	12,06%	1,87 GB	126	24,4%	33,04%	3,2 GB	120
20x20 x5km	17,75%	12,06%	2,05 GB	123	27,24%	38,31%	4,73 GB	120
40x40 x10km	20,3%	12,25%	2,13 GB	115	26,76%	53,22%	5,05 GB	110

Tabela 3: Wyniki badania wpływu wielkości terenu (oswietlenie dynamiczne)

Rozmiar terenu	Unity				Unreal			
	CPU	GPU	RAM	FPS	CPU	GPU	RAM	FPS
10x10 x2.5km	17,82%	12,24%	1,95 GB	122	27,05%	34,74%	3,4 GB	120
20x20 x5km	18,17%	12,23%	2,12 GB	120	31,1%	40,22%	4,71 GB	120
40x40 x10km	26,5%	12,26%	2,14 GB	113	29,23%	54,33%	5,25 GB	112

Jak wykazały badania wielkość generowanego terenu nieznaczny stopniu wpływa na wydajność silników. Dla silnika Unity rośnie zużycie procesora, a dla Unreal Engine zużycie karty graficznej. Dodatkowo w przypadku Unreal Engine znaczco rośnie wykorzystana pamięć RAM. Dla obu silników ilość klatek na sekundę maleje wraz ze wzrostem wielkości terenu do wyrenderowania.

Tabela 4: Wyniki badania wpływu refleksji świetlnych

Rozmiar terenu	Unity				Unreal			
	CPU	GPU	RAM	FPS	CPU	GPU	RAM	FPS
10x10 x2.5km	18,58%	12,03%	1,89 GB	83	27.6%	34,42%	4.56 GB	80
20x20 x5km	19,53%	16,26%	2,11 GB	81	32,31%	36,21%	4,72 GB	79
40x40 x10km	28,2%	17,15%	2,13 GB	74	32,23%	36,53%	5,28 GB	76

Refleksje i refrakcja światła na tafli wody dla silnika Unity spowodowały wzrost zużycia procesora i karty graficznej, spadek ilości klatek na sekundę oraz w małym stopniu wzrost zużycia pamięci RAM. Dla silnika Unreal Engine wzrosło zużycie procesora oraz pamięci RAM, a ilość klatek na sekundę zmniejszyła się.

Badania wpływu szczegółowości i ilości obiektów wykonano na mapie 20x20x5km, ponieważ do tej pory wyniki badań wykazują, że dla większego rozmiaru mapy wydajność znacząco spada.

Tabela 5: Wyniki badań wpływu ilości obiektów na scenie

Scena 20x20 x5km	Unity				Unreal			
	CPU	GPU	RAM	FPS	CPU	GPU	RAM	FPS
1000 obiektów (62 800 trójkątów)	33,6%	24,13%	2,54 GB	133	36,52%	46,67%	5,27 GB	100
3000 obiektów (188400 trójkątów)	43,2%	27,04 %	2,76 GB	90	40,5%	50,12%	5,88 GB	100
5000 obiektów (314000 trójkątów)	48,3%	25.8%	3.25 GB	74	45,45%	56,2%	6,23 GB	95
10000 obiektów (728000 trójkątów)	54,6%	27.21%	7.36 GB	33	48,34%	62,74%	6,95 GB	61
20000 obiektów (1456000 trójkątów)	75,3%	23.42%	10.45 GB	5	53,86%	72,11%	7,47 GB	40

Dla renderowania dużej ilości tych samych modeli, w silniku Unity zastosowano metodę *gpu instancing*[34][35]. Pozwala ona na renderowanie kopii siatki w jednym wywołaniu funkcji rysowania. Jednakże związane ze zwiększającą się ilością siatek obiektów i ilością wierzchołków, obciążenie procesora mogło wpływać na hamowanie generowania klatek przez kartę graficzną [33]. Dla największej testowanej ilości można zauważyc, że zużycie karty graficznej spadło, a wraz z nią też drastycznie spadła ilość klatek na sekundę. Wraz ze wzrostem ilości obiektów znacząco wzrasta ilość zajętej pamięci RAM. Dla silnika Unreal Engine ilość generowanych klatek na sekundę spada, niemniej jednak nie w takim stopniu jak dla silnika Unity. Unreal od początku zużywa więcej pamięci RAM, jednakże wraz ze wzrostem ilości obiektów zużycie nie wzrasta w takim stopniu jak w silniku Unity. Dodatkowo można zauważyc, że silnik Unreal bardziej obciąża kartę graficzną, a w mniejszym stopniu procesor.

Tabela 6: Wpływ szczegółowości obiektów (render 1000 obiektów)

Scena 20x20x5km	Unity				Unreal			
	CPU	GPU	RAM	FPS	CPU	GPU	RAM	FPS
LOD3	17,3%	17,42%	2,18 GB	130	27,15%	37,87%	4,21 GB	122
LOD2	17,5%	22,45%	2.53 GB	127	28,11%	39,78%	4,65 GB	120
LOD1	18,15%	27,71%	3.04 GB	117	34,13%	43,96%	4,95 GB	118
LOD0	18,65%	32,15%	3.38 GB	100	45,15%	47,13%	5,21 GB	115

Sprawdzono również wpływ użycia grup poziomu detali (*level of details*). Dla obu silników zużycie procesora, karty graficznej oraz zużycie pamięci RAM wzrasta wraz z użyciem bardziej szczegółowych modeli z grup LOD. Ilość klatek dla silnika Unity w widoczny sposób spada, gdy dla silnika Unreal jest niemal stała.

Tabela 7: Wyniki badania wpływu metody *ray tracing* w oświetleniu

Scena 20x20x5km	Unity				Unreal			
	CPU	GPU	RAM	FPS	CPU	GPU	RAM	FPS
Ray tracing włączony	12.5%	47.42%	2.93 GB	105	34,92%	47,2%	4,05 GB	118
Ray tracing wyłączony	15,6%	31.2%	3.03 GB	120	38,5%	40,62%	3,15 GB	120

Zastosowanie metody *ray tracing* wpływa na spadek ilości klatek na sekundę i wzrost zużycia karty graficznej w obu silnikach, za to zużycie procesora zmniejsza się. Dla silnika Unreal Engine ilość zużytej pamięci RAM znacząco wzrasta, a dla Unity minimalnie zmniejsza się.

5. Podsumowanie i wnioski

Na podstawie średniej ze wszystkich wyników (Tabela 8) można założyć, że silnik Unity jest bardziej wydajny niż silnik Unreal Engine.

Tabela 8: Średnie wszystkich wyników

	CPU	GPU	RAM	FPS
Unity	26,99%	21,23%	3,09 GB	116,65
Unreal	34,53%	45,27%	4,95 GB	107,25

Silnik Unity średnio uzyskał lepsze wyniki w każdej sprawdzanej wartości. Obciążenie procesora w testach było średnio 21,83% mniejsze, obciążenie karty graficznej 53,1% mniejsze, obciążenie pamięci 37,57% mniejsze, a wyrenderowanych klatek na sekundę było o 8,05% więcej. Jednak trzeba zaznaczyć, że w przypadku gdzie silniki musiały wyrenderować bardziej złożone obiekty lub ilość obiektów była ogromna, zdecydowanie lepiej z takim zadaniem radził sobie silnik Unreal Engine, gdzie ilość klatek na sekundę była zadowalająca, a w przypadku silnika Unity, średnia liczba klatek nie pozwalała na płynny rendering. Przyczynę takiego stanu rzeczy tłumaczy zastosowanie systemu NANITE w silniku Unreal Engine, który tak jak mówili twórcy, pozwala na płynny rendering zaawansowanych geometrycznie obiektów na scenie. Dodatkowo silnik Unreal Engine pozwala na łatwiejsze uzyskanie realistycznie wyglądających renderów bez wymagania dodatkowych działań przez użytkownika. Biorąc pod uwagę zużycie zasobów, można stwierdzić, że Unreal Engine będzie dobrym wyborem jeżeli celem jest stworzenie dużej gry na konsole najnowszej generacji i komputery osobiste, bądź do tworzenia teł i obiektów do wykorzystania w produkcji filmowej. Unreal Engine pozwala na pisanie kodu w niższym poziomie niż Unity, dzięki czemu wprawny programista mógłby zoptymalizować działanie kodu pod własne cele, a co za tym idzie można stwierdzić, że silnik Unreal Engine będzie dobrym wyborem dla zaawansowanych technicznie użytkowników. Silnik Unity z kolei będzie dobrym wyborem dla deweloperów gier niezależnych, w których jakość grafiki nie jest tak ważnym aspektem. Silnik ten jest łatwiejszy w obsłudze i nauce, oraz jak wykazały testy, sam w sobie jest bardziej wydajny niż silnik Unreal Engine. Oczywiście tak samo jak w przypadku silnika Unreal Engine, zaawansowany technicznie użytkownik, który zna silnik Unity, będzie w stanie uzyskać zbliżoną jakość graficzną obrazu do generowanego przez silnik Unreal Engine. Podsumowując, silnik Unity uzyskał lepsze wyniki w testach, przez co można założyć, że jest on bardziej wydajnym silnikiem w porównaniu z silnikiem Unreal Engine.

Bibliografia

- [1] C.Giardina, "Too Much Volume? The Tech Behind 'Mandalorian' and 'House of the Dragon' Faces Growing Pains", The Hollywood Reporter, 2022, dostęp 08.2023
- [2] Video Games Market Value to Grow to Over \$200 billion by 2023, Juniper Research, 2020, dostęp 08.2023
- [3] "Most successful videogame engine", Guinness World Records, 2015, dostęp 08.2023
- [4] <https://docs.quadspinner.com/Reference/>, QuadSpinner, 2023, dostęp 08.2023
- [5] <https://www.blender.org/about/>, Blender, 2023, dostęp 08.2023
- [6] https://wiki.blender.org/wiki/Reference/Release_Notes/2.80/EEVEE, Blender, 2023, dostępny 08.2023
- [7] https://wiki.blender.org/wiki/Reference/Release_Notes/2.80/Cycles, Blender, 2023, dostępny 08.2023
- [8] <https://brandguide.brandfolder.com/unity/unitylogo>, Unity Technologies, 2023, dostępny 08.2023
- [9] <https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@16.0/manual/HDRP-Features.html#screen-space-global-illumination>, Unity Technologies, 2023, dostępny 08.2023
- [10] <https://docs.unity3d.com/560/Documentation/Manual/GIIntro.html>, Unity Technologies, 2023, dostępny 08.2023
- [11] <https://docs.unity3d.com/Manual/Lightmappers.html>, Unity Technologies, 2023, dostępny 08.2023
- [12] K. Thor Jensen, "25 years Later: The History of Unreal and an Epic Dynasty", PCMag, 2023, dostępny 08.2023
- [13] Unreal Engine 5.2 Release Notes | Unreal Engine 5.2 Documentation, 2023, Epic Games, dostępny 08.2023
- [14] <https://www.unrealengine.com/en-US/branding>, Epic Games, 2023, dostępny 08.2023
- [15] <https://docs.unrealengine.com/5.0/en-US/nanite-virtualized-geometry-in-unreal-engine/>, Epic Games, 2023, dostępny 08.2023
- [16] B.Karis, R.Stubbe, G.Wihlidal "Advances in Real-Time Rendering In Games", SIGGRAPH, 2021, slajdy 14-16, 28-39, dostępny 08.2023
- [17] <https://docs.unrealengine.com/5.0/en-US/lumen-technical-details-in-unreal-engine/>, 2023, Epic Games, dostępny 08.2023
- [18] <https://developer.nvidia.com/discover/ray-tracing>, Nvidia Corporation, 2019, dostępny 08.2023
- [19] <https://docs.unrealengine.com/5.0/en-US/ray-tracing-performance-guide-in-unreal-engine/>, Epic Games, 2022, dostępny 08.2023
- [20] <https://docs.unrealengine.com/5.0/en-US/lumen-technical-details-in-unreal-engine/>, Epic Games, 2022, dostępny 08.2023
- [21] <https://docs.unrealengine.com/5.1/en-US/path-tracer-in-unreal-engine/>, Epic Games, 2023, dostępny 08.2023
- [22] https://darksouls.fandom.com/wiki/Royal_Wood, Fandom, 2012, dostępny 08.2023
- [23] <https://docs.unrealengine.com/5.2/en-US/lumen-global-illumination-and-reflections-in-unreal-engine/>, Epic Games, 2023, dostępny 08.2023
- [24] <https://docs.unrealengine.com/4.26/en-US/BuildingWorlds/Landscape/TechnicalGuide/>, Epic Games, 2019, dostępny 08.2023

- [25] Andrade, A., “Game engines: a survey, EAI Endorsed Transactions on Serious Games”, 2015, strony 2-3, dostęp 08.2023
- [26] “2021 Gaming Raport- Unity Insights from 2020 and predicted trends for 2021”, Unity Technologies, 2021, dostęp 08.2023
- [27] I.Pachoulakis, G.Pontikakis, “Combining features of the Unreal and Unity Game Engines to hone development skills”. 2011, dostęp 08.2023
- [28] P. Skop, “Porównanie wydajności silników gier na wybranych platformach”, 2018, strony 116118, dostęp 08.2023
- [29] A. Šmíd, “Comparison of Unity and Unreal Engine”, Czech Technical University in Prague, 2017, stony 11-13, 29-30, dostęp 08.2023
- [30] A. Ciekanowska, A. Kiszcza - Gliński, K. Dziedzic, „Comparative analysis of Unity and Unreal Engine efficiency in creating virtual exhibitions of 3D scanned models.”, Journal of Computer Sciences Institute, 2021, strony 247-252, dostęp 08.2023.
- [31] <https://pc-builds.com/pl/bottleneck-calculator/result/1dv17E/1/general-tasks/1920x1080/>, PCBuilds, dostęp 08.2023
- [32] <https://pc-builds.com/pl/bottleneck-calculator/result/1dv17E/1/general-tasks/2560x1440/>, PCBuilds, dostęp 08/2023
- [33] <https://docs.unity3d.com/2020.1/Documentation/Manual/OptimizingGraphicsPerformance.html>, Unity Technologies, 2020, dostęp 08.2023
- [34] D.Aversa, C. Dickinson, “Unity Game Optimization”, Packt Publishing, 2019, strony 16-20, 36-41, 213-214, 274-275, dostęp 08.2023
- [35] <https://docs.unity3d.com/Manual/GPUInstancing.html>, Unity Technologies, 2023, dostęp 08.2023

Wykaz rysunków

- Rysunek 1: Ekspert terenu za pomocą węzła Mesher
Rysunek 2: Obszar roboczy Gaea z grafem węzłów
Rysunek 3: Edycja wierzchołków, krawędzi i płaszczyzn
Rysunek 4: Wynik renderingu silnika Eevee i Cycles
Rysunek 5: Logo Silnika Unity, źródło: [8]
Rysunek 6: Okno konfiguracyjne HDRP
Rysunek 7: Włączone i wyłączone użycie globalnego oświetlenia
Rysunek 8: Wypalone światło na statycznych obiektach
Rysunek 9: Logo Silnika Unreal Engine, źródło: [14]
Rysunek 10: Nanite- podgląd trójkątów
Rysunek 11: Nanite - podgląd klastrów
Rysunek 12: Ogólny podgląd Nanite
Rysunek 13: Ogólny podgląd Lumen
Rysunek 14: Schemat podstawowego działania techniki ray tracing, źródło: [18]
Rysunek 15: Organizacyjna Hierarchia Woluminów (BVH) w przypadku przechodzenia promienia, źródło [19]
Rysunek 16: Podgląd Path Tracing
Rysunek 17: Wyniki działania węzłów Mountain, ThermalShaper i FractalTerraces
Rysunek 18: Wyniki działania węzłów Perlin, Erosion, Snowfall
Rysunek 19: Wyniki działania węzłów Rivers i Lakes
Rysunek 20: Wyniki działania węzłów Slope, Petrusion i ich kombinacji z odjęciem śniegu
Rysunek 21: Wyeksportowane maski śniegu, wody, kamieni, trawy oraz mapa wysokości dla Unreal Engine.
Rysunek 22: Splatmapa terenu
Rysunek 23: Modele drzew zbudowanych przy pomocy płaszczyzn z gry Dark Souls, opracowano na podstawie [22]
Rysunek 24: Gałęzie jako oteksturowana płaszczyzna
Rysunek 25: Utworzone modele 3D
Rysunek 26: Najmniej szczegółowy obiekt grup poziomu detali zbudowany z 8 płaszczyzn
Rysunek 27: Tworzenie obiektu z grupami LOD w silniku Unity
Rysunek 28: Scena z zainportowaną mapą i skonfigurowanym globalnym oświetleniem
Rysunek 29: Tworzenie nowego terenu w silniku Unity
Rysunek 30: Warstwy tekstur, import splatmapy oraz podział terenu
Rysunek 31: Wizualizacja poziomów wysokości terenów
Rysunek 32: Tworzenie obiektu z grupami LOD w silniku Unreal Engine
Rysunek 33: Schemat dla obracania źródła światła w Unreal Engine
Rysunek 34: Warstwy tekstur w materiale terenu Unreal Engine
Rysunek 35: Tworzenie terenu w silniku Unreal Engine
Rysunek 36: Określenie typu warstw tekstur
Rysunek 37: Wypełnienie wszystkich warstw i samej warstwy base

Wykaz tabel

Tabela 1: Konfiguracja sprzętowa

Tabela 2: Wyniki badania wpływu wielkości terenu (oświetlenie statyczne)

Tabela 3: Wyniki badania wpływu wielkości terenu (oświetlenie dynamiczne)

Tabela 4: Wyniki badania wpływu refleksji świetlnych

Tabela 5: Wyniki badań wpływu ilości obiektów na scenie

Tabela 6: Wpływ szczegółowości obiektów (render 1000 obiektów)

Tabela 7: Wyniki badania wpływu metody ray tracing na oświetlenie

Tabela 8: Średnie wszystkich wyników

Wykaz listingów

Listing 1: Kod obrotu źródła światła w silniku Unity

Listing 2: Zapisywanie danych o zużyciu CPU, GPU, RAM dla silnika Unity do pliku