Wysłane: 2016-11-30
Przyjęte: 2016-12-15

Metody optymalizacji wydajności silnika Unity 3D w oparciu o grę z widokiem perspektywy trzeciej osoby

Krzysztof Siarkowski*, Przemysław Sprawka*, Małgorzata Plechawska-Wójcik
Politechnika Lubelska, Instytut Informatyki, Nadbystrzycka 36B, 20-618 Lublin, Polska

Streszczenie. Optymalizacja gier to jeden z najważniejszych aspektów ich tworzenia. Artykuł opisuje metody optymalizacji silnika Unity, a jako przedmiot analizy posłużyła gra z widokiem perspektywy trzeciej osoby. Zbadano jaki wpływ na wydajność gry mają metody, które polegają na odciążeniu karty graficznej, poprzez zwiększenie wykorzystania procesora oraz pamięci.

Słowa kluczowe: optymalizacja; silnik gier; Unity

*Autor do korespondencji.

Adresy e-mail: krzsiarkowski@gmail.com, przemek1992spr@gmail.com

Methods for optimizing the performance of Unity 3D game engine based on third-person perspective game

Krzysztof Siarkowski*, Przemysław Sprawka*, Małgorzata Plechawska-Wójcik
Institute of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract. Game optimization is one of the most important aspects of their creation. The article describes methods to optimize Unity Engine using third person perspective game as an example. Various methods that rely on offloading graphics card, by increasing the use of CPU and memory were used in order to check how the game performance changes.

Keywords: optimization; game engine; Unity

*Corresponding author.

E-mail addresses: krzsiarkowski@gmail.com, przemek1992spr@gmail.com

1. Wstęp

W ostatnich latach pojawiło się na rynku wiele darmowych silników służących do tworzenia gier, co znacznie obniżyło barierę wejścia do środowiska twórców gier osobom, dla których komercyjne silniki były zbyt drogie. Jednakże cechy i jakość produktu finalnego, jaką oczekują od gier użytkownicy rośnie z każdym dniem. Należy oczekiwać, że każdy aspekt gry zostanie szczegółowo przeanalizowany zarówno przez graczy jak i krytyków, dlatego nie można pozwolić sobie na problemy związane z wydajnością.

Postanowiono więc wyjść naprzeciw tym problemom i w niniejszym artykule opisać zastosowanie i analizę działania wybranych metod optymalizacji. Zbadano wpływ tych metod na wykorzystanie zasobów komputera takich jak procesor i pamięć oraz głównych statystyk odpowiedzialnych za pomiar wydajności renderowania, zaś jako przedmiot analizy posłużyła gra utworzona w Unity z perspektywy trzeciej osoby. Postarano się udowodnić, że użycie metod optymalizacji odciążających jeden z zasobów komputera odpowiedzialnych za wydajność, powoduje większe wykorzystanie innego.

Wybór Unity był spowodowany tym, że silnik ten ostatnio zyskuje sporą popularność oraz posiada wbudowane narzędzia do analizowania statystyk podczas renderowania. Dodatkowo z poziomu interfejsu graficznego edytora Unity, można w prosty sposób użyć danej metody optymalizacji i skonfigurować jej parametry.

2. Renderowanie

Renderowaniem nazywamy proces zamiany informacji wejściowych na inną formę reprezentacji tych danych, adekwatną do danego medium. W grafice trójwymiarowej renderowanie to proces konwertowania trójwymiarowego, matematycznego modelu obiektu na wyświetlany obraz dwuwymiarowy w postaci pojedynczej klatki, bądź animacji [1]. W skład procesu renderowania 3D wchodzą różne elementy, takie jak kąt padania światła, fizyczne właściwości materiałów na obiekcie, odbicia oraz cienie. Dodawanie takich cech wizualnych sprawia, że modele 3D stają się bardziej realistyczne.

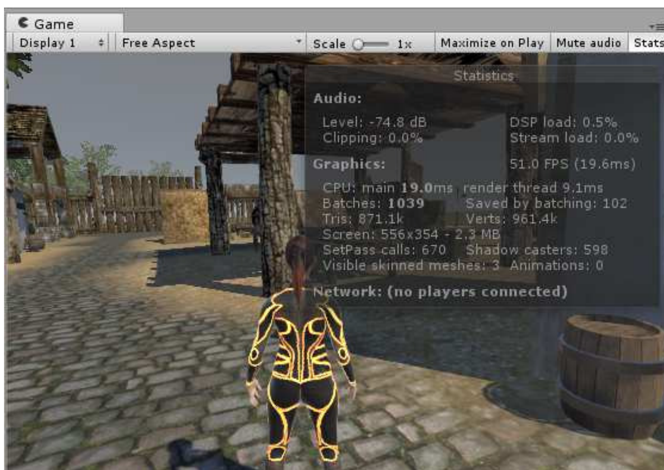
Renderowanie w czasie rzeczywistym polega na generowaniu obrazów wystarczająco szybko, aby stworzyć wrażenie płynnej animacji. Wykorzystywane jest szczególnie w grach, gdzie wszystkie modele 3D i elementy scen muszą być przedstawiane użytkownikom w określonej liczbie klatek na sekundę. Wpływa to na jakość przedstawianych modeli ponieważ, muszą być przygotowane tak, aby jak najwierniej odzwierciedlały obiekty, które mają przedstawiać, ale jednocześnie składały się z jak najmniejszej liczby wielokątów (ang. *polygon*) bez utraty swojej formy. Najczęstszym błędem przy badaniu liczby klatek na sekundę

wyświetlanych na ekranie jest przekonanie, że głównym czynnikiem na nie wpływającym jest złożoność geometrii, czyli złożoność siatek (ang. *meshes*) modeli 3D. W rzeczywistości wpływ na czas renderowania jednej klatki ma liczba świateł na scenie, złożoność materiałów, ilość pamięci zajętej przez tekstury i siatki, fizyka, animacje, cząsteczki i inne pomniejsze elementy [2].

3. Narzędzia

Unity posiada zaimplementowane w sobie dwa główne narzędzia do optymalizacji wydajności. Pierwszym z nich jest okno statystyk renderowania, zaś drugim Profiler, z którego można korzystać na licencji darmowej, wraz z wydaniem Unity 5 i udostępnieniem w niej prawie wszystkich funkcji, które wcześniej były dostępne tylko w płatnej wersji Pro.

3.1. Rendering Statistics Window



Rys. 1. Okno statystyk renderowania

Statystyki renderowania widoczne są w oknie (Rys. 1), które ukazuje się po kliknięciu przycisku *Stats* w sekcji *Game View*. Statystyki te pokazywane są w czasie rzeczywistym i są bardzo przydatne do analizy oraz optymalizacji wydajności.

3.2. Rendering Profiler

Rys. 2. Okno Profilera

W górnej części okna Profilera wyświetlane są dane dotyczące wydajności, w miarę upływu czasu (Rys. 2). Po uruchomieniu gry, dane zapisywane są co każdą klatkę, a wyświetlana jest jedynie historia kilkuset ostatnich klatek. Klikając na pojedynczą klatkę, zostaną wyświetlone jej szczegóły w dolnej części okna. Różne informacje zostaną zaprezentowane w zależności, która sekcja osi czasu została wybrana. Oś czasu zawiera kilka obszarów: wykorzystanie procesora, renderowanie i użycie pamięci.

3.3. Statystyki renderowania

Statystyki mające największy wpływ na wydajność renderowania, których wartości można odczytać zarówno z okna statystyk renderowania oraz Profilera to:

Time per frame and FPS – liczba klatek na sekundę (ang. *frames per second*) oraz czas w milisekundach potrzebny w Unity do przetworzenia i renderowania jednej klatki. Klatką nazywamy finalny render złożony z pikseli przekazany z karty graficznej i wyświetlony na ekranie. Podczas przygotowywania gotowej klatki, zarówno procesor jak i karta graficzna muszą wykonać wiele obliczeń. To wszystko odzwierciedla złożoność renderowania i czas potrzebny do jego wykonania. Dłuższy czas renderowania może skutkować spadkiem FPS, ponieważ mniej klatek zostanie przetworzonych w ciągu sekundy. Ogólnie rzecz biorąc, im wyższa ich liczba, tym lepiej. Liczba klatek nie powinna być mniejsza niż 15-30 na sekundę, ponieważ spadek poniżej tej wartości powodują zauważalne dla ludzkiego oka zburzenia płynności oraz zawieszanie się ekranu.

SetPass calls – liczba odwołań procesora do karty graficznej (ang. *draw calls*). Odwołania te odnoszą się do liczby wysyłanych żądań do karty graficznej, w celu pobrania danych z pośredniczących buforów, a następnie dane te utworzą ostateczny render. Dzięki temu obiekty są widoczne na ekranie, w momencie, w którym powinny. Zwykle wykonuje się jedno odwołanie na każdy renderowany obiekt. W przeciwieństwie do liczby klatek na sekundę, im mniej *draw calls* tym lepiej, ponieważ każde odwołanie pociąga za sobą znaczne zapotrzebowanie i obciążenie procesora. Oznacza to, że odwołania do karty graficznej są bezpośrednio związane z wydajnością, którą można poprawić poprzez redukcję ich liczby. W osiągnięciu tego celu, Unity posiada wbudowany system pakietowania (ang. *batching*), pozwalający grupować odwołania i przetwarzać kilka obiektów razem w jednym odwołaniu, zamiast w wielu [3].

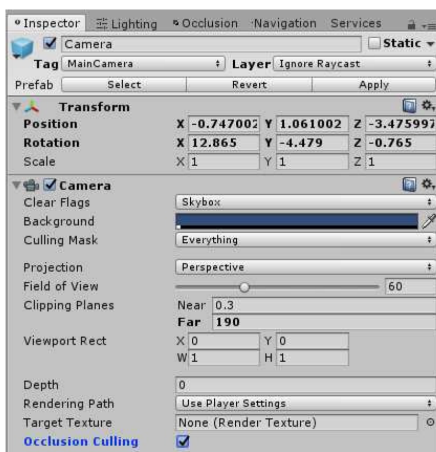
Tris and Verts – określa łączną liczbę trójkątów i wierzchołków obiektów aktualnie renderowanych, a nie wszystkich znajdujących się na scenie. W momencie przesuwania się kamery i obiektów, ta liczba zmienia się, ponieważ obiekty wychodzą poza obszar widzenia kamery, a niektóre w tym samym momencie wkraczają. Współczesny sprzęt komputerowy i systemy renderowania są w stanie bez problemu poradzić sobie z milionami trójkątów i wierzchołków, jednak nadal obowiązuje zasada, że utrzymanie jak najmniejszej ich liczby wpływa na wydajność, ponieważ mniej danych należy przetworzyć.

4. Metody optymalizacji

W rozdziale tym szczegółowo opisano działanie metod optymalizacyjnych, które zostały użyte w grze wykonanej w Unity, w celu przeanalizowania, na jakie parametry renderowania wybrane metody mają największy wpływ.

4.1. Occlusion Culling

Usuwanie niewidocznych powierzchni (ang. *occlusion culling*) pozwala uniknąć renderowania obiektów, które są zasłaniane przez inne. Nie jest to domyślna funkcja w grafice trójwymiarowej, ponieważ zazwyczaj najpierw renderowane są obiekty, które znajdują się najdalej od kamery, a dopiero na nich te bliżej.

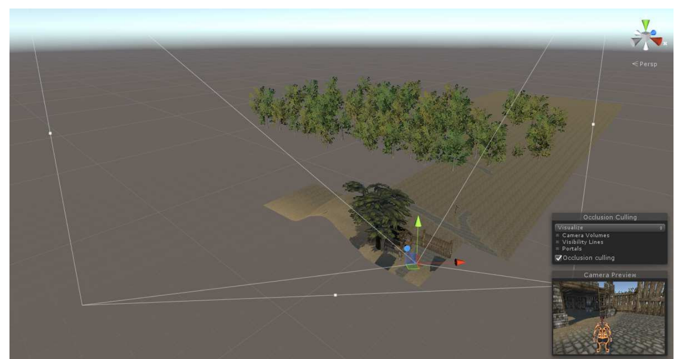


Rys. 3. Okno Inspektora obiektu kamery z zaznaczoną opcją Occlusion Culling

Usuwanie obiektów znajdujących się za innymi obiektami, nie jest tym samym co usuwanie obiektów, które wychodzą poza obszar widzenia kamery (ang. *frustum culling*). *Frustrum culling* jest wykonywane automatycznie, zaś usuwanie zasłoniętych obiektów musi zostać włączone w edytorze Unity w oknie Inspektora obiektu kamery (Rys. 3) i poprawnie skonfigurowane.

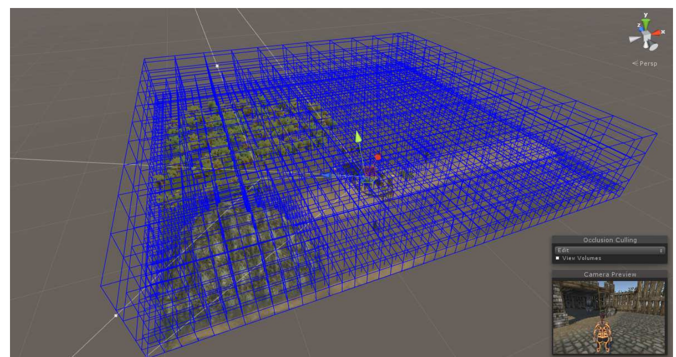
Rys. 4. Widok sceny z wyłączoną opcją usuwania zasłoniętych obiektów

Usuwanie obiektów wychodzących poza obszar kamery przedstawia rysunek 4, zaś połączenie tych dwóch technik można zaobserwować na rysunku 5, gdzie doskonale widać różnice pomiędzy nimi.



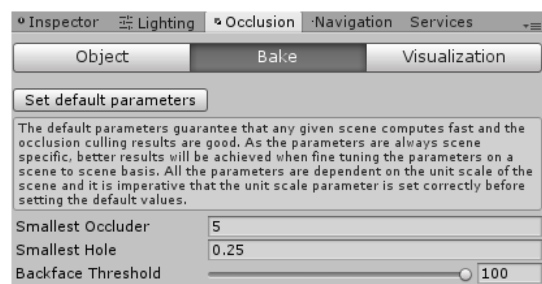
Rys. 5. Widok sceny z włączoną opcją usuwania zasłoniętych obiektów

Przy użyciu wirtualnej kamery, budowana jest hierarchia potencjalnie widzianych obiektów. Dane te następnie są wykorzystywane przez wszystkie kamery na scenie do sprawdzenia, które obiekty powinny zostać wyrenderowane, a które potraktowane jako zasłonięte przez inne obiekty [4].



Rys. 6. Podział sceny na komórki

Dane wykorzystywane do usuwania obiektów składają się z komórek, które zostają wydzielone z głównego obszaru obejmującego całą scenę. Komórki te tworzą tzw. siatkę ukrywania obiektów (Rys. 6). Wszelkie obiekty mniejsze niż wielkość komórki nie spowodują zasłonięcia innych obiektów, zaś obiekty większe niż komórka będą zasłaniać obiekty za sobą.



Rys. 7. Okno parametryzownia metody Occlusion Culling

Wielkość najmniejszego obiektu (ang. *smallest occluder*), odpowiada najmniejszej wielkości komórki, na jaki może zostać podzielona scena. Parametr ten jak i drugi z parametrów *occlusion culling*, jakim jest najmniejszy dopuszczalny prześwit pomiędzy siatkami obiektów, przez który obiekty powinny być renderowane (ang. *smallest hole*) można ustawić w oknie Occlusion widocznym na rysunku 7.

Rys. 8. Ustawienie kamery dla zbadania działania occlusion culling

Przesuwając po scenie kamerę z włączoną wizualizacją usuwania niewidocznych powierzchni, wybrano najbardziej odpowiednie miejsce do przeprowadzenia badania wpływu na renderowanie tej metody optymalizacji. Duża część sceny została zasłonięta przez wzgórze, a dodatkowo obszar widzenia kamery obejmował prześwit pomiędzy ogrodzeniem wioski (Rys. 8).

Parametr *smallest hole* ustawiony na zbyt wysoką wartość spowodował, że włączenie opcji usuwania niewidocznych obiektów, potraktowało teren za ogrodzeniem oraz zbyt dużo drzew za zasłonięte, co zostało ukazane na rysunku 9.

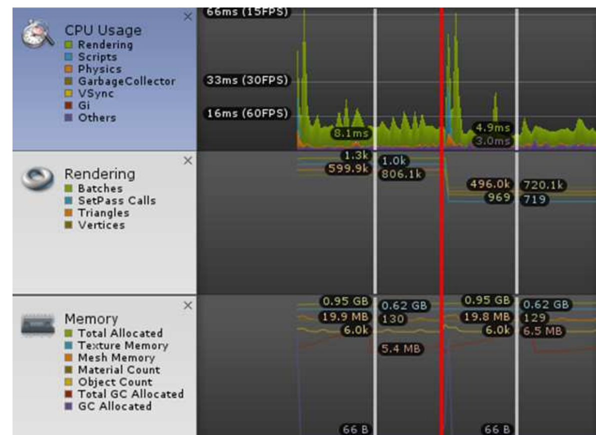
Rys. 9. Efekt zbyt dużego dopuszczalnego prześwitu między siatkami obiektów, przez który obiekty traktowane są jako zasłonięte

Wyniki

Occlusion culling wyłączone	Occlusion culling włączone
Statistics	Statistics
Audio:	Audio:
Level: -74.8 dB DSP load: 0.9%	Level: -74.8 dB DSP load: 0.7%
Clipping: 0.0% Stream load: 0.0%	Clipping: 0.0% Stream load: 0.0%
Graphics: 58.5 FPS (17.1ms)	Graphics: 78.2 FPS (12.8ms)
CPU: main 15.6ms render thread **16.6**ms	CPU: main 10.5ms render thread **12.4**ms
Batches: **1324** Saved by batching: 165	Batches: **992** Saved by batching: 56
Tris: 599.8k Verts: 805.9k	Tris: 508.1k Verts: 726.8k
Screen: 1284x688 - 10.1 MB	Screen: 1284x688 - 10.1 MB
SetPass calls: 1017 Shadow casters: 444	SetPass calls: 743 Shadow casters: 494
Visible skinned meshes: 3 Animations: 0	Visible skinned meshes: 3 Animations: 0
Network: (no players connected)	Network: (no players connected)

Rys. 10. Okno statystyk renderowania – Occlusion Culling

Włączenie usuwania niewidocznych powierzchni, zmniejszyło liczbę trójkątów i wierzchołków, ponieważ zasłonięte obiekty nie zostały wyrenderowane. Pozwoliło to zaoszczędzić dużą liczbę odwołań do karty graficznej, co pozytywnie odbiło się na wzroście liczby wyświetlanych klatek na sekundę (Rys. 10).



Rys. 11. Okno Profilera – Occlusion Culling

Na rysunku 11 widać linię czasu z Profilera podczas renderowania sceny z wyłączonym i włączonym *occlusion culling* – czerwona linia oznacza moment włączenia usuwania niewidocznych powierzchni. W wyraźniejszy sposób został przedstawiony spadek trójkątów i wierzchołków oraz liczby odwołań, co nie wpłynęło negatywnie na zużycie pamięci lub procesora.

4.2. Clipping Plane

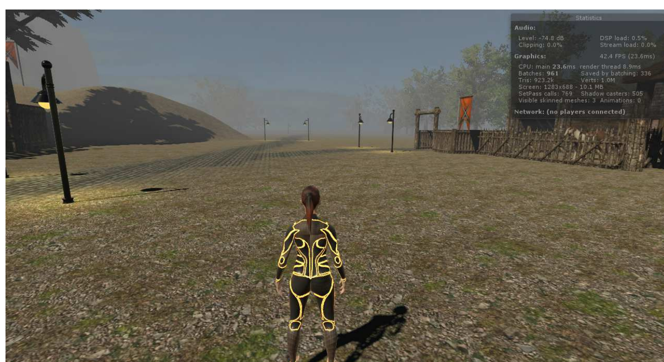
Płaszczyzny odcinające (ang. *clip plane*) pozwalają zmniejszyć zasięg renderowania obiektów. Działają w ten sposób, że obiekty, które znajdują się w większej odległości od kamery niż określono, nie będą wyświetlane [5].

Rys. 12. Widok z gry z domyślnym zasięgiem renderowania

Za zmniejszenie wielkości obszaru, który powinien być renderowany odpowiada parametr *Far Clip Plane* (ustawienie tego parametru jest możliwe w oknie Inspektora kamery widocznego na rysunku 3). Widok z gry, z domyślną wartością tego parametry został przedstawiony na rysunku 12. Zmniejszenie wartości parametru spowoduje, że większa liczba obiektów nie będzie renderowana, ponieważ ograniczono obszar widoczności kamery. Powstała wolna przestrzeń (Rys. 13), która w miarę zbliżania się postaci, będzie wypełniana odciętymi wcześniej obiektami.

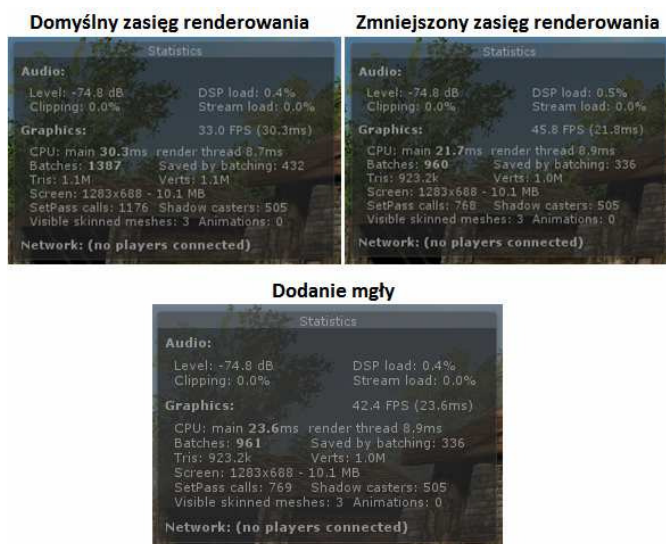
Rys. 13. Efekt działania płaszczyzny odcinającej

Ten niepożądany efekt można za to w łatwy sposób ukryć, używając mgły. Nie dość, że odcięte obiekty zostaną zasłonięte przez mgłę, to równocześnie zwiększą się wrażenia towarzyszące graczowi zwiedzającemu wirtualny świat (Rys. 14).



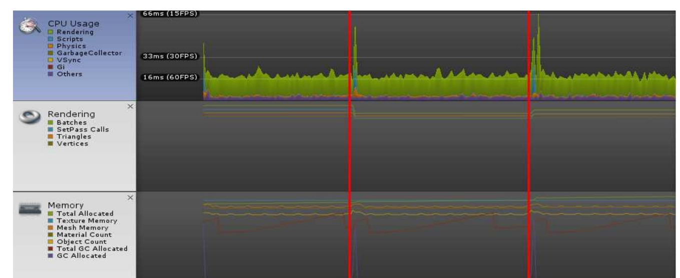
Rys. 14. Widok z gry z użyciem płaszczyzny odcinającej i mgły

Wyniki



Rys. 15. Okno statystyk renderowania – Far Clip Plane

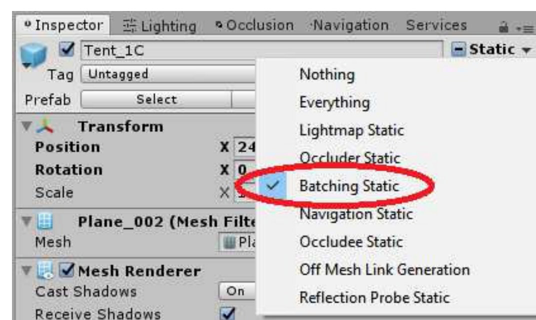
Na rysunku 15 przedstawiono statystyki renderowania w poszczególnych etapach konfigurowania na scenie płaszczyzny odcinającej oraz po dodaniu na scenę mgły. W oknie Profilera (Rys. 16), który również w tym czasie rejestrował dane, przejście do kolejnego etapu zostało zaznaczone na osi czasu czerwoną linią. Zmniejszenie zasięgu usunęło z renderowania kilka obiektów, zatem zmalała liczba trójkątów i wierzchołków, odwołań procesora do karty graficznej renderowania, co przyczyniło się do podniesienia liczby klatek wyświetlanych na sekundę. Analizując oś czasu przedstawiającą użycie procesora, można zauważyć, że kolejne etapy w momencie uruchomienia gry, chwilowo ale coraz bardziej i bardziej obciążają CPU. Może to być spowodowane tym, że procesor musi nie tylko obliczyć odległość obiektów od kamery, ale również je usunąć, żeby nie zostały wyrenderowane, a w ostatnim etapie dodatkowo wykonać skrypt odpowiedzialny za wyświetlenie mgły.



Rys. 16. Okno Profilera – Far Clip Plane

4.3. Batching

Wyróżniamy dwa rodzaje pakietowania: statyczne (ang. *static batching*) oraz dynamiczne (ang. *dynamic batching*) i obie metody mają wpływ na grupowanie obiektów w celu zmniejszenia liczby odwołań procesora do karty graficznej. Pakietowanie przede wszystkim pozwala Unity grupować wiele obiektów razem, traktując je jakby były jedną siatką lub zasobem, więc mogą być przetworzone w pojedynczym odwołaniu, zamiast w wielu [6]. *Batching* jest w głównej mierze procesem wewnętrznym i automatycznym w Unity, czyli to edytor podejmuje decyzję na własną rękę, jak zgrupować obiekty i kiedy. Decyzje te podejmowane są na podstawie ustawień obiektów w oknie Inspektora. Wymusza to pewne wymagania dotyczące przygotowanie przez nas obiektów do pakietowania. Dzięki temu mamy pewien poziom kontroli nad tym jak zadziała pakietowanie.



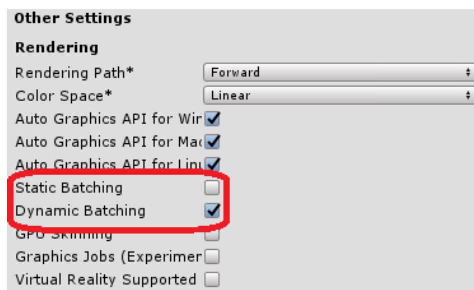
Rys. 17. Oznaczenie obiektu jako statyczny

Dla siatek modeli obiektów takich jak ściany, krzesła, wzgórze, góry, latarnie i innych nieporuszających się w czasie działania gry, należy zawsze oznaczać obiekty jako statyczne (Rys. 17). Jest to sygnał dla Unity, że siatki te mogą zostać

zgrupowane w jedną wielką siatkę, co edytor zrobi już automatycznie. Grupowanie obiektów pomiędzy pakietami zależy od różnych ustawień obiektów. Wszystkie obiekty dzielące ten sam materiał zostaną wysłane w jednym pakiecie, dlatego najczęściej wiele obiektów znajduje się w tylu grupach, z ilu korzystają materiałów [7].

Pakietowanie dynamiczne jest procesem grupowania, który wykonywany jest przez Unity automatycznie, na obiektach niebędących statycznymi, żeby jeszcze bardziej zmniejszyć liczbę *draw calls*. Wszystkie siatki o małej złożoności, które dzielą ten sam materiał i skalę zostaną połączone w pakiet, z wyjątkiem obiektów na które oddziałuje światło w czasie rzeczywistym [8].

Domyślnie w Unity włączona jest opcja pakietowania dynamicznego, lecz zarówno *static batching* jak jak i *dynamic batching* mogą być włączone lub wyłączone jednocześnie. Opcje te dostępne są w oknie Player Settings, widocznym na rysunku 18.



Rys. 18. Okno Player Settings, gdzie możliwe jest włączenie/wyłączenie pakietowania

Wyniki

Rys. 19. Widok z gry użyty do zbadania działania batchingu

Do przetestowania działania poszczególnych mechanizmów pakietowania wykorzystano widok z gry, który został przedstawiony na rysunku 19.

Rys. 20. Statystyki renderowania, używając różnych metod pakietowania

Jak można zauważyć na zestawionych ze sobą oknach renderowania statystyk (Rys. 20), w zależności, który mechanizm pakietowania został włączony, znacząco różni się liczba grup. Włączając kolejno metody pakietowania, inne siatki są ze sobą grupowane, co powoduje że liczba tych grup wzrasta. Nie miało to jednak zbyt dużego wpływu na liczbę wierzchołków, trójkątów, czy klatek na sekundę, ponieważ wahały się one w jednakowych granicach przez cały proces. Wydawałoby się w takim razie, że pakietowanie nie wpłynęło na wydajność naszej gry, lecz zanim to stwierdzimy, spójrzmy co możemy zaobserwować w Profilerze.

Rys. 21. Okno Profilera – pakietowanie

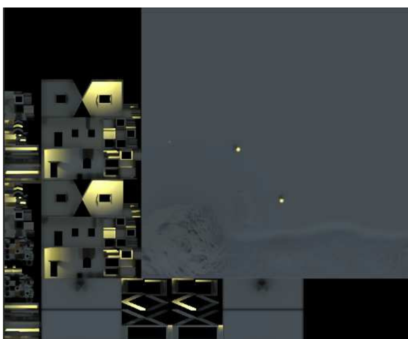
Na rysunku 21 zaznaczony został moment włączenia statycznego pakietowania. Pakietowaniu statycznemu poddanych zostało dużo więcej siatek obiektów niż pakietowaniu dynamicznemu, co doprowadziło do dwukrotnego wzrostu zużycia pamięci. Również doprowadziło to do tego, że podczas uruchomienia gry ze statycznym pakietowaniem chwilowy skok zużycia procesora, większy niż w przypadku samego pakietowania dynamicznego.

4.4. Lighting

Oświetlenie w Unity może być renderowane w czasie rzeczywistym (ang. *real-time lighting*) lub wypalone na mapach światła (ang. *baked lightmaps*). Przy dodawaniu światła na scenę, domyślnie jest ono renderowane w czasie rzeczywistym, czyli aktualizowane co klatkę. W ten sposób

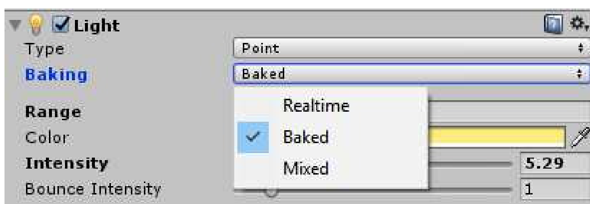
możemy oświetlać modele w ruchu oraz przemieszczać źródła światła. Niestety takie światła nie odbijają się od powierzchni obiektów, a więc nie można uzyskać bardziej realistycznych efektów, takich jak oświetlenie globalne (ang. *global illumination*), a cienie, które generują są czarne. Powinniśmy jednak unikać tego rodzaju renderowania oświetlenia, ponieważ negatywnie wpływa to na wydajność [9].

Oświetlenie wypalone (ang. *baked lighting*) sprawdza się świetnie, kiedy na scenie znajdują się nieruchome obiekty oraz gdy zależy nam na optymalizacji oświetlenia. Światła są obliczane bezpośrednio na scenie i wypalane na teksturach modeli w taki sposób, aby stworzyć efekt padającego światła. Wypalanie mapy światła (ang. *lightmap*) często jest procesem długotrwałym i głównie zależy od prędkości komputera.



Rys. 22. Tekstura lightmapy wygenerowana podczas wypalania oświetlenia

Mapa światła zawiera dane na temat jasności punktów z procesu wypalenia oświetlenia. Generowana jest jeden raz, a później tylko nakładana na teren. *Lightmapy* mogą posiadać w sobie światło bezpośrednie lub pośrednie, czyli takie które odbija się od innej powierzchni i pada na obiekt. Podczas gdy aplikacja jest uruchomiona, światła te nie mogą być zmienione, natomiast mogą na nie wpływać światła w czasie rzeczywistym i dodatkowo dodawać do nich cienie [10]. Mapa światła wygenerowana w edytorze Unity dla sceny w grze została przedstawiona na rysunku 22.



Rys. 23. Ustawianie sposobu renderowania światła

Sposób renderowania światła może zostać zmieniony w oknie Inspektora obiektu światła (Rys. 23). Odpowiada za to parametr Baking, w którym mamy możliwość ustawienia, czy wybrane oświetlenie ma być renderowane w czasie rzeczywistym czy wypalone. Sam proces wypalania lightmap można uruchomić z poziomu okna Lighting, w którym znajdują się opcje konfiguracyjne oświetlenia na całej scenie.

Wyniki

Renderowanie światła w czasie rzeczywistym i wypalanie oświetlenia, zbadano wykorzystując stworzony w programie graficznym Blender model domku oraz latarni, ze światłem punktowym.

Rys. 24. Real-time lighting

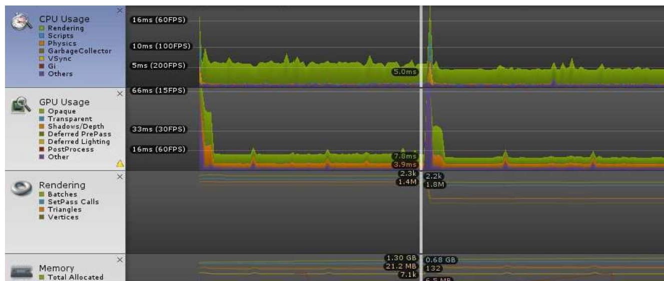
Ściana budynku w przykładzie z *real-time* ma wyraźnie podkreślone detale, a kostki brukowe posiadają zacieniowane miejsca (Rys. 24). Również światło renderowane w czasie rzeczywistym, przy tych samych ustawieniach natężenia i obszaru jest znacznie ostrzejsze.

Rys. 25. Baked lighting

Jednym z mankamentów wypalania świateł jest to, że nie oświetlają obiektów dynamicznych takich jak postacie (Rys. 25). Jednak wydajnościowo znacznie lepiej one wypadają. Przy oświetleniu w czasie rzeczywistym, każdy obiekt na który pada światło, wymaga wyrenderowania go tyle razy, ile świateł na niego pada. Powoduje to mnożenie wyświetlanej geometrii (liczba trójkątów i wierzchołków), co wyraźnie widać na zrzutach ekranu z okna statystyk renderowania (Rys. 26).

Rys. 26. Okna statystyk renderowania. Po lewej: oświetlenie w czasie rzeczywistym. Po prawej: oświetlenie wypalone.

Wypalane mapy traktują lightmapy jak tekstury i nie przeliczają świateł padających na poszczególne piksele. Również znacznie zmniejsza się liczba odwołań procesora, a dodatkowo nie przeliczane są cienie, które w czasie rzeczywistym znacząco obciążają grę.



Rys. 27. Okno Profilera w real-time i baked lighting

Oświetlenie renderowane co klatkę obciąża procesor dużo bardziej, niż oświetlenie wypalone na teksturach, co potwierdza odczyt z osi czasu (Rys. 27), gdzie pionowa linia wskazuje moment, w którym zatrzymano grę i Profiler przestał zbierać dane dotyczące renderowania światła w czasie rzeczywistym. W edytorze została wypalona *lightmapa* i ponownie uruchomiono grę, co wznowiło rejestrowanie danych przez Profiler, wykorzystując do renderowania drugą z metod odwzorowania światła.

5. Wnioski

Przeanalizowanie działania wybranych metod optymalizacji uświadomiło nas, że proces optymalizacji wydajności silnika gry, wymaga dużej wiedzy na temat działania najważniejszych funkcji silnika, zanim ktokolwiek zacznie z niego korzystać. Funkcje te niewłaściwie użyte, zamiast zwiększyć wydajność aplikacji, mogą przynieść nieoczekiwane konsekwencje.

Zbadanie wpływu wybranych metod optymalizacji na wydajność wykazało, że w przypadku analizowanej gry, uzyskiwany duży spadek liczby trójkątów i wierzchołków, odwołań procesora do karty graficznej, nie wiązał się z oczekiwanym dużym skokiem wydajności pod względem liczby klatek na sekundę, ponieważ prawie zawsze odbywało się to kosztem zwiększenia zużycia pamięci i mocy obliczeniowej procesora. Wynikało to z próby przeniesienia części obliczeń wykonywanych przez cały czas trwania gry, na moment jej uruchomienia lub dokonanie wyliczeń już na scenie i późniejsze odwoływanie się do nich, tak jak to działo się w przypadku wypalonego oświetlenia,.

Należy więc zachować proporcje pomiędzy optymalizacją liczby klatek na sekundę, a zużyciem pamięci i procesora, żeby odciążając kartę graficzną nie doprowadzić do przeciążenia innych podzespołów komputera, które również składają się na wydajność aplikacji.

Zatem nie powinno się błędnie zakładać, że proces optymalizacji głównie skupia się na zmniejszeniu geometrii sceny, ponieważ w przeprowadzonych metodach optymalizacji udowodniono, że bardzo ważny wpływ na wydajność, oprócz karty graficznej mają również pamięć oraz procesor i nie uzyskamy poprawy liczby klatek na sekundę, co można było zauważyć przy pakietowaniu statycznym.

Literatura

[1] M. F. Shiratuddin, W. Thabet, Utilizing a 3D game engine to develop a virtual design review system, Journal of Information Technology in Construction – ITcon, 16, 2011, 39-68.

[2] S. Blackman, Beginning 3D Development with Unity 4. All-in-One, Multi-Platform Game Development, New York City, Apress, 2013.

[3] A. Thron, Unity 4 Fundamentals: Get Started at Making Games with Unity, Burlington, Focal Press, 2014.

[4] https://docs.unity3d.com/Manual/OcclusionCulling.html [12.11.2016].

[5] W. Goldstone, Unity 3.x Game Development Essentials, Birmingham, Packt Publishing, 2009.

[6] C. Dickinson, Unity 5 Game Optimization, Birmingham, Packt Publishing, 2015.

[7] A. Thorn, How to Cheat in Unity 5: Tips and Tricks for Game Development, Burlington, Focal Press, 2016.

[8] https://docs.unity3d.com/Manual/DrawCallBatching.html [14.11.2016].

[9] A. Thorn, Practical Game Development with Unity and Blender, Boston, Cengage Learning, 2014.

[10] https://unity3d.com/learn/tutorials/topics/graphics/introduction-lighting-and-rendering [16.11.2016].