UNIWERSYTET GDAŃSKI
WYDZIAŁ MATEMATYKI, FIZYKI I INFORMATYKI

Adam Stobiecki
Numer albumu: 271393

Kierunek studiów: Informatyka
Specjalność: Ogólna

Analiza porównawcza algorytmów Ambient Occlusion

Praca magisterska
napisana pod kierunkiem
dr. Macieja Dziemiańczuka

Gdańsk 2024

Niniejsza pracę dedykuje mojej kochanej mamie, ***Alicji Kowalewskiej****, której zawdzięczam moje osiągnięcia i bez której, nie mógłbym dotrzeć tak daleko. Jej pomoc i wsparcie było dla mnie nieocenione.*

Za wsparcie mentalne i motywację serdecznie dziękuje mojej dziewczynie ***Astuti Ningsih****.*

Za możliwość spełnienia mojego marzenia, żeby móc nauczyć się programować grafikę komputerową, oraz za pomoc przy realizacji tej pracy, serdecznie dziękuje mojemu prowadzącemu, oraz promotorowi ***dr Maciejowi Dziemiańczukowi****.*

Spis treści

Streszczenie

W tej pracy zajmiemy się analizą kilku popularnych technik ambient occlusion (w skrócie *AO*) które można stosować w generowaniu grafiki czasu rzeczywistego. Na początku pracy wprowadzimy czytelnika w świat grafiki komputerowej oraz wyjaśnimy jak grafika komputerowa jest przetwarzana i renderowana na wyświetlaczu. Następnie wyjaśnimy czym jest AO i jak powstaje w poszczególnych implementacjach: SSAO, SSAO+, HBAO. Sama praca skupiać się będzie na analizie poszczególnych algorytmów, czym się różnią, oraz jakie zalety wynikają ze stosowania każdego z algorytmów. Sprawdzimy jak wydajnie dane algorytmy radzą sobie ze skalowalnością (przy różnych rozdzielczości ekranu) i jak ona wpływa na wydajności konkretnych implementacji. Zmierzymy czasy renderowania poszczególnych algorytmów i porównamy je. Przeprowadzimy również podstawową analizę obrazów cyfrowych, w celu ocenienia różnic między okluzjami. Postaramy się empirycznie ocenić, który z nich radzi sobie najlepiej oraz w jakich scenariuszach. Na koniec podamy przykłady w jakich sytuacjach poszczególne algorytmy mogą być bardzo wydajnie wykorzystane.

Do pracy została załączona implementacja algorytmów i wykonane przeze mnie środowisko testowe pozwalające czytelnikowi przetestować dane algorytmy samodzielnie oraz w czasie rzeczywistym. Wszystkie testy potrzebne do wykonania pracy zostały przeprowadzone na moim środowisku. Customizowalność została przeze mnie dostosowana do środowiska w taki sposób, żeby użytkownik mógł z łatwością zreplikować wyniki otrzymane przez nas na potrzebę tej pracy pracy.

Załączony został również mój mały dodatkowy projekt w którym można zreplikować metryki użyte dalej w pracy do porównywania algorytmów AO.

Słowa kluczowe - Generowanie grafiki w czasie rzeczywistym, Ambient Occlusion, Space Screen Ambient Occlusion (SSAO i SSAO+), Horizon-Based Ambient Occlusion (HBAO).

Glosariusz

AO	Ambient Occlusion (pl. okluzja otoczenia)
FBO	Frame Buffer Object (pl. obiekt bufora ramki)
FPS	Frames Per Second (pl. klatki na sekundę)
HBAO	Horizon Based Ambient Occlusion (pl. okluzja otoczenia bazująca na horyzoncie)
MSE	Mean Squared Error (pl. średni błąd kwadratowy)
MVP	Model View Projection (pl. model widok projekcja)
NDC	Normalized Device Space (pl. znormalizowana przestrzeń urządzenia)
PSNR	Peak Signal-to-Noise Ratio (pl. szczytowy stosunek sygnału do szumu)
RTAO	Ray-Traced Ambient Occlusion (pl. okluzja otoczenia oparta na ray tracingu)
SSAO	Space Screen Ambient Occlusion (pl. okluzja otoczenia przestrzeni ekranowej)
SSIM	Structural Similarity Index Measure (pl. indeks miary podobieństwa strukturalnego)
VRAM	Video Random Access Memory (pl. Wideo RAM)

Wstęp

Na przestrzeni ostatnich kilku lat zauważyć można znaczny wzrost realizmu wszelkiego rodzaju wizualizacji grafiki w grach i systemach graficznych 3D. Na samym początku, między innymi gry, były znacznie ograniczone przez możliwości sprzętowe ówczesnych czasów. Grafika była wtedy znacząco uproszczona i minimalistyczna. Z biegiem lat GPU i CPU ewoluowały pozwalając na bardziej wyrafinowaną i realistyczną grafikę. Wraz z tym rozwojem zwiększyły się oczekiwania wobec programów. Rozwój technologii, sprowokował idący z nim w parze rozwój algorytmów, które niskim nakładem zasobów dają efekty podobne do realistycznych[1]. Oświetlenie, cienie, czy okluzja otoczenia to tylko kilka z najbardziej znanych efektów, które można było zaczynać *symulować* przy pomocy obecnie posiadanej technologii.

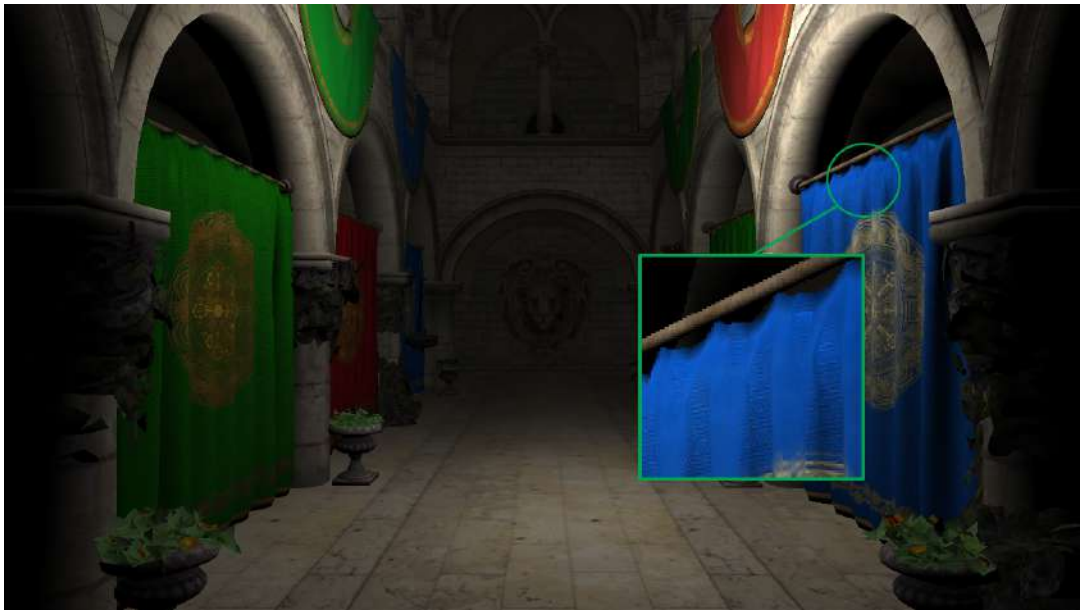
Obecnie znajdujemy się w momencie w którym grafika 3D potrafi symulować rzeczy z prawdziwego świata z ogromną dokładnością[2]. Jesteśmy w stanie stosować algorytmy, które zbliżają nas do realizmu bardziej niż kiedykolwiek wcześniej, osiągając efekty nierozróżnialne od rzeczywistego stanu rzeczy. Osiągane jest to metodą wielu programistycznych i sprzętowych sztuczek, które pozwalają na bardzo efektywne optymalizacje całego procesu powstawiania grafiki.

Stan zaawansowania technicznego obecnych komputerów pozwala nam z łatwością korzystać z najlepszych wizualnie, dostępnych od wielu lat, algorytmów - takich jak Ray Tracing. Niestety, efekty te są może bardzo realistyczne, jak również bardzo zasobożerne. Bez wsparcia sprzętowego (które jest już dostępne) obecne wyniki nie byłyby możliwe do uzyskania. Na domiar złego, na rynku kart graficznych panuje obecnie *wojna* o zunifikowanie standardów technologicznych GPU. Obecnie nie osiągnięto konsensusu między producentami kart graficznych. Każdy z producentów chce ustanowić swój standard, będący obecnie panującym standardem. Powstaje przez to duży chaos i część technologii, które jedno urządzenie obsługuje, inne nie koniecznie musi obsługiwać, a tym bardziej z tak samą wydajnością. Deweloperzy i konsumenci muszą ograniczać wydajność sprzętową tylko dlatego, że mając najwyższej klasy produkt, nie jest on produktem *tej* firmy, która obsługuje daną technologię.

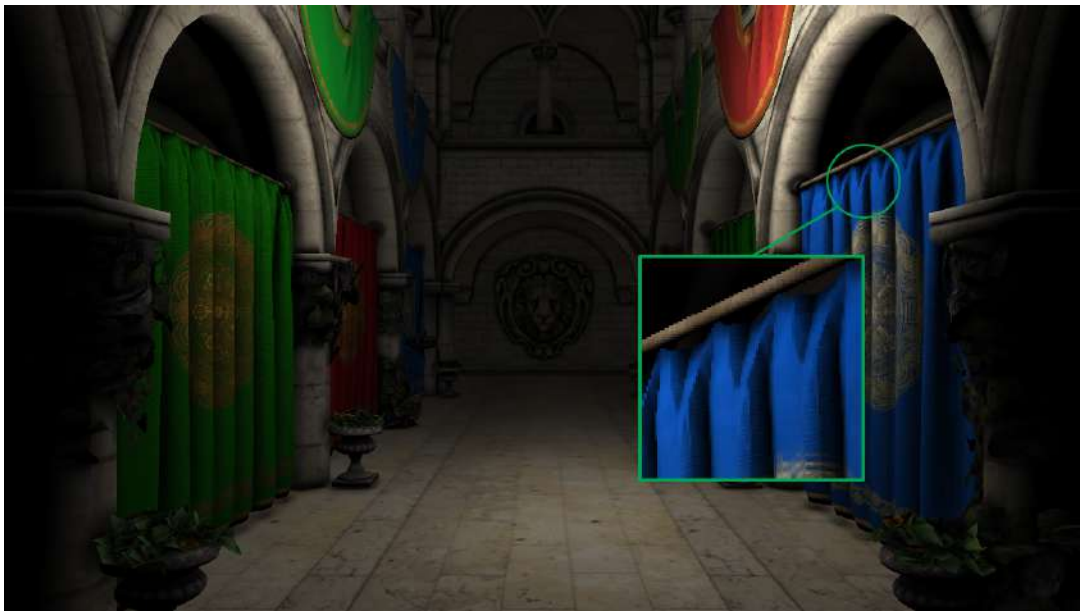
To co zatem można zrobić to implementować algorytmy, które wiadomo, że będą obsługiwane przez wszystkie urządzenia, niezależnie od technologi. Chodzi nam bardziej o implementacje algorytmiczne, aniżeli rozwiązania sprzętowe. Takie podejście pozwala nam odseparować się od potencjalnie daleko idących problemów z kompatybilnościami sprzętowymi i skupić się na samej implementacyjnej części.

Same algorytmy mogą się różnić w zależności od potrzeb i zastosowań[3]. Jedne mogą być bardzo wydajne, ale ich efekt wizualny mało przyjemny dla oka, natomiast inne mogą być wolniejsze, ale dawać pożądany wizualny efekt. Finalnie, najważniejszy jest użytkownik który powinien być zadowolony z rezultatu gdyż to on jest odbiorcą naszego produktu i jego zadowolenie powinno być nadrzędną kwestią.

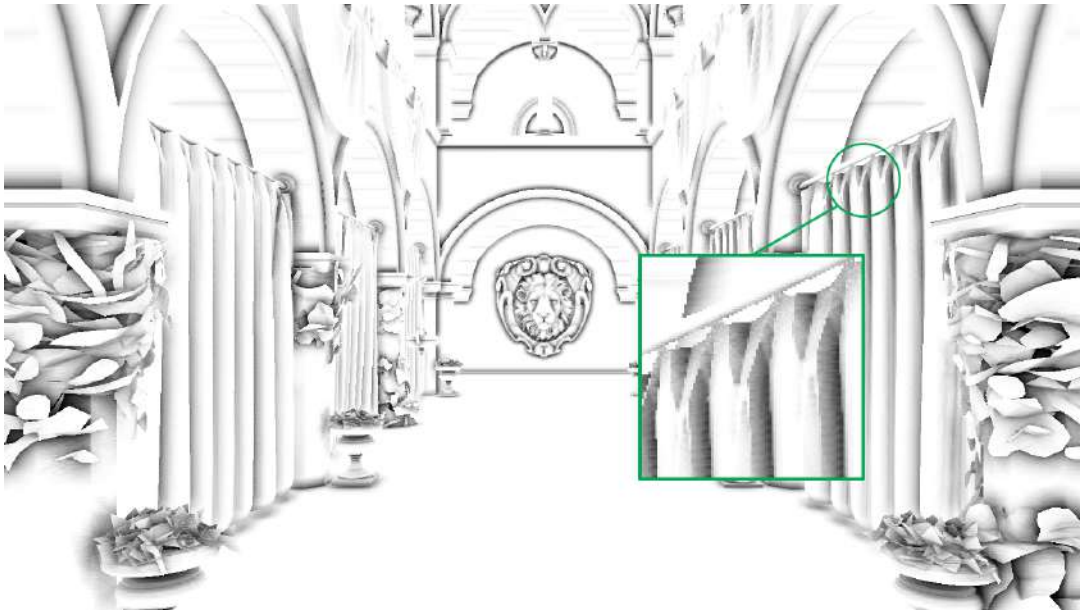
Zatem, gdzie w tym wszystkim znajduje się okluzja otoczenia i czym ona jest? Jest to metoda cieniowania obiektów 3D, którą stosuje się do zwiększenia realizmu danego modelu. Zacieniowywane są miejsca przy ostrych (wypukłych) krawędziach. Taki zabieg dodaje realizmu i zwiększa immersję grafiki 3D.



Rysunek 1: Grafika bez okluzji
Źródło: opracowanie własne.



Rysunek 2: Grafika z okluzją
Źródło: opracowanie własne.



Rysunek 3: Okluzja
Źródło: opracowanie własne.

Na rysunku 1 można zaobserwować zwykłą grafikę z oświetleniem. Jest widoczne, że różni się ona od grafiki 2 w zagięciach na tkaninie. Jest to spowodowane przyciemnieniem grafiki 1 w miejscach gdzie grafika 3 jest ciemniejsza. To co widzimy na grafice 3 to sama okluzja na podstawie której tworzy się ten efekt zaciemnienia.

Okluzja otoczenia (potocznie znana jako AO) jest szeroko stosowana we wszystkich systemach wizualizacji 3D. Efekt ten jest nieodłączną częścią oświetlenia i cieniowania, i tak samo jak one, można stosować wiele sztuczek, aby wyświetlany efekt efekt był ładny (realistyczny), niskim kosztem. Tego typu działanie, tak jak powiedzieliśmy wcześniej, pozwala nam poprzez uproszczenie algorytmu osiągnąć zamierzony efekt, bez znaczącego spadku wydajności.

W poniższej pracy zagłębimy się, w kilka algorytmów okluzji otoczenia, celem przeanalizowania ich wydajności. Zwrócimy uwagę na ich aspekty wizualne, wydajność i aplikowalność. Opowiemy o technikach stosowanych w powstawaniu symulowanej okluzji w środowisku 3D. Przyjrzymy się parametrom każdej z implementacji. Ocenimy wady i zalety każdej z nich oraz do jakich celów można by je stosować oraz jakie parametry mogą nas interesować przy dobieraniu algorytmy. Na koniec podejmiemy próbę dobrania scenariuszy do których dane algorytmy będą najbardziej pasować.

Motywacja

Wyróżniamy wiele rodzajów grafiki komputerowej. Grafika, która będzie interesować nas w naszej pracy to *grafika 3D czasu rzeczywistego*. Jest to rodzaj grafiki cechujący się interaktywnością w czasie działania programu. Wszystkie parametry danego modelu *mogą* być edytowane w czasie działania programu.

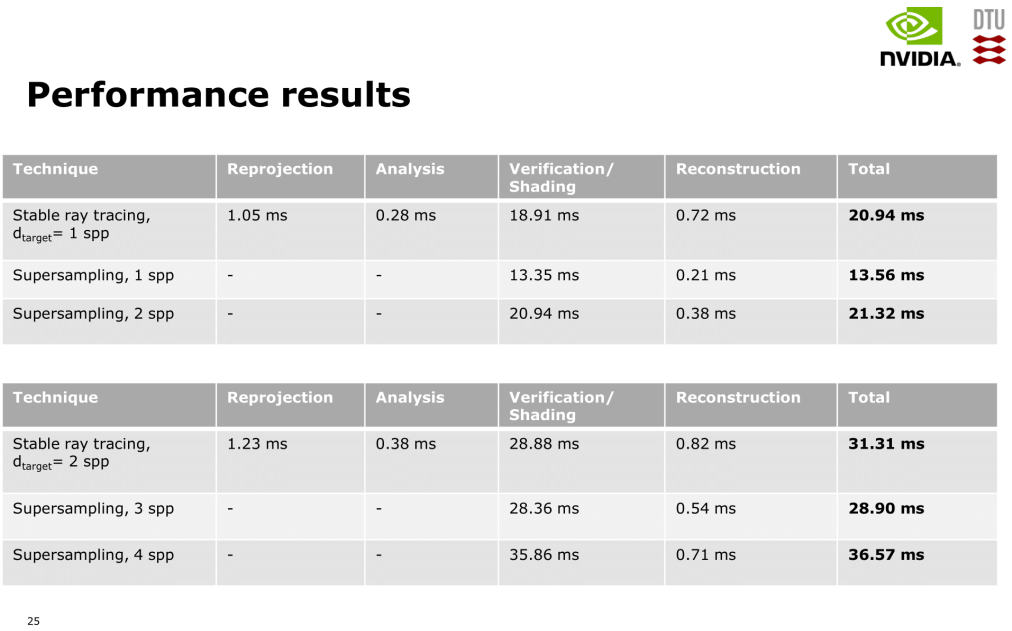
To co różni grafikę czasu rzeczywistego od prerenderowanej grafiki jest to, że pozwala ona na stworzenie interaktywnego środowiska umożliwiającego większą immersję i realizm, w przeciwieństwie do swojego statycznego odpowiednika. Tego rodzaju technika pozwala też na pełną kontrolę nad całym zachowaniem i renderingiem środowiska na bieżąco, w czasie korzystania z danego oprogramowania. Technika ta ma jednak, jedną znaczącą wadę: w przeciwieństwie do renderowania, które wykonywane jest w dowolnym czasie, nasze renderowanie musi być wykonywane bardzo szybko i bardzo optymalnie. Mówimy tutaj o maksymalnym czasie procesu renderingu równemu $16\frac{1}{3}ms$ ($\frac{1000}{60}[\frac{ms}{Hz}]$, który możemy interpretować jako *FPS*). Wszystkie nowoczesne wyświetlacze obsługują częstotliwość odświeżania na poziomie $60Hz$; natomiast próg, przy którym ludzkie oko przestaje postrzegać ciągły obraz a widzieć pokaz slajdów wynosi w przybliżeniu $\sim 24FPS$ ($41\frac{2}{3}ms$). Zatem $41\frac{2}{3}ms$ jest absolutnie nieprzekraczalną granicą częstotliwością odświeżania, a $16\frac{2}{3}ms$ jest minimalna preferowaną częstotliwością odświeżania ekranu. W tym czasie (lub preferowanie krótszym) powinien odbywać się się cały proces produkcji jednej klatki wyświetlanej na ekranie.

Oczywiście, produkowane są obecnie nawet wyświetlacze z częstotliwością dochodzą nawet do $500Hz$ ($2ms$). Oznacza to, że gdybyśmy chcieli wyświetlić obraz najpłynniej jak tylko się da, to cały proces renderingu będzie musiał trwać maksymalnie do $2ms$. Dla większości nowoczesnych gier taki czas (z małymi wyjątkami) jest nieosiągalny.

Celem programistów przy pracy z grafiką czasu rzeczywistego, jest osiągnięcie *jak najkrótszego* czasu renderowania, oraz *jak najmniejszejszej* latencji. Sama latencja jest niepożądana, a moment jej powstawania przeważnie związany jest z samym czasem komunikacji między GPU, a CPU, który jest po prostu dość kosztowny. Tam gdzie to możliwe stosuje się tzw. UBO (ang. Uniform Buffer Object), jest to technika polegająca na "opakowaniu" naszych informacji w jedną dużą strukturę i przesłanie jej całej zawartości na raz do shadera (programu wykonywanego na karcie graficznej), zamiast pojedyncze przesyłanie każdej z danych po kolei (metoda ta zmniejsza ilość komunikacji z GPU).

Technik do osiągnięcia jak najlepszych wyników w obu z ww. celów jest bardzo wiele; zatem czy nie można od razu przejść do najwydajniejszych technik? Obecnie najlepszymi algorytmami naśladującymi rzeczywiste oświetlenie i zacienianie dają *ray tracing* i *path tracing* (oraz ich wszystkie podgatunki). Temat tych dwóch metod wybiega poza temat pracy ale jest idealnym sposobem na pokazanie problemu z jakim się mierzymy.

Poniżej mamy przedstawiony fragment prezentacji HPG (ang. High-Performance Graphics) na temat ray tracingu[4]:



Performance results

Technique	Reprojection	Analysis	Verification/ Shading	Reconstruction	Total
Stable ray tracing, d_{target}= 1 spp	1.05 ms	0.28 ms	18.91 ms	0.72 ms	**20.94 ms**
Supersampling, 1 spp	-	-	13.35 ms	0.21 ms	**13.56 ms**
Supersampling, 2 spp	-	-	20.94 ms	0.38 ms	**21.32 ms**

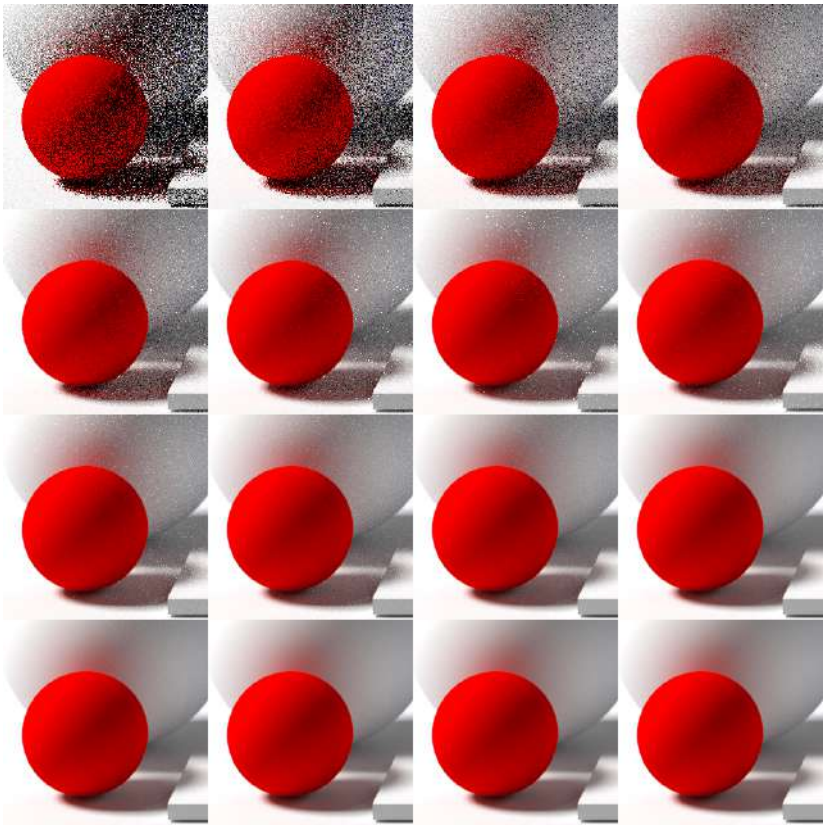
Technique	Reprojection	Analysis	Verification/ Shading	Reconstruction	Total
Stable ray tracing, d_{target}= 2 spp	1.23 ms	0.38 ms	28.88 ms	0.82 ms	**31.31 ms**
Supersampling, 3 spp	-	-	28.36 ms	0.54 ms	**28.90 ms**
Supersampling, 4 spp	-	-	35.86 ms	0.71 ms	**36.57 ms**

25

Rysunek 4: Ray tracing i czas renderingu.
Źródło: highperformancegraphics.org.

Na rysunku 4 widać wyniki Ray tracingu jakie wyszły przy badaniu wydajności różnych wariantów implementacyjnych tego algorytmu. W kolumnie *Technique* (technika), zawarte są nazwy poszczególnych algorytmów oraz ilość *spp* (ang. samples-per-pixel), czyli ilość samplowań na poszczególny piksel grafiki. Każda kolejna kolumna na prawo oznacza czas poświęcony na poszczególny proces w tworzeniu grafiki z ray tracingiem. W ostatniej kolumnie jest podsumowany czas poświęcony na wszystkie etapy.

Zwróćmy uwagę, że jedyny wynik pomiaru akceptowalny ze względu na warunki postawione wcześniej to ten z 1 spp. Wszystkie pozostałe znajdują się poniżej 60FPS, a ich czas rośnie wraz z ilością próbek. Przypomnijmy, że referencyjny czas 60FPS to $16\frac{2}{3}ms$. Każdy czas powyżej tego wiąże się z automatyczną dyskwalifikacją danego algorytmu jako algorytm który można wykorzystać do grafiki czasu rzeczywistego. Ten jeden akceptowalny wynik cechuje się w pracy niską *ostrością* i *szumem*. Przez szum rozumiemy niechciane artefakty pojawiające się na renderowanej grafice.

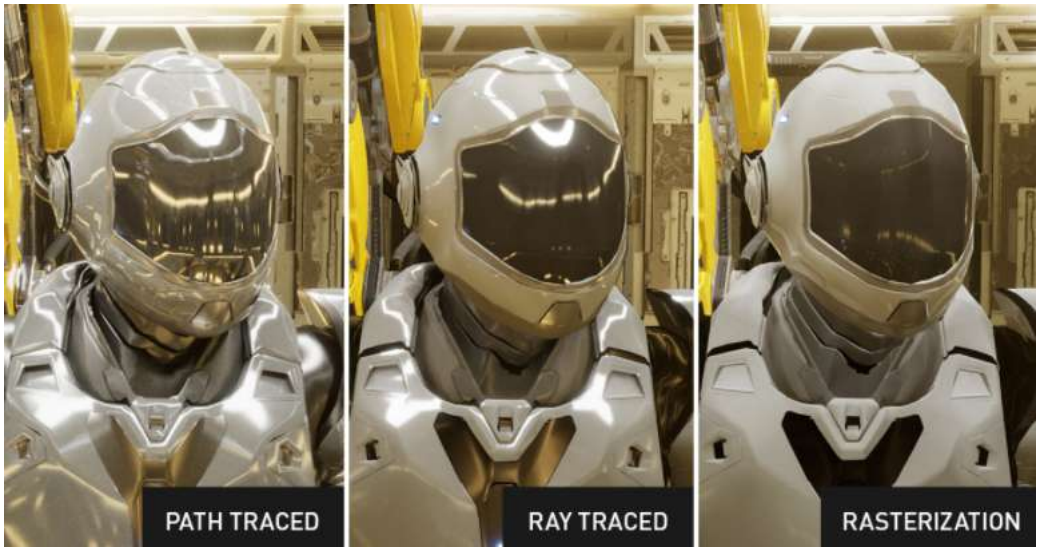


Rysunek 5: Próbkowanie path traceingu (w lewym górnym rogu ilość próbek wynosi 1 i zwiększa się dwukrotnie).
Źródło: wikipedia.org.

Oczywiście, istnieją obecnie technologie które radzą sobie z problemem licznego próbkowania i szumu. Jednym z takich rozwiązań są dedykowane core'y na kartach graficznych Nvidia pozwalających znacznie przyspieszyć ray tracing[2]. Rozwiązanie to daje bardzo dobre rezultaty graficzne i realny czas renderowania dla grafiki czasu rzeczywistego. Problem jest jednak to, że sprzęt taki jest dość nową i drogą technologią. Poza tym rozwiązanie to jest ściśle związane z technologiami Nvidia, a nie jest zunifikowaną technologią obsługiwaną przez wszystkie karty graficzne na rynku. Jeśli założylibyśmy, że te wszystkie nie są znaczące to dalej zostajemy z problemem, że wszelkie oprogramowanie oparte o tą technologią automatycznie nie będzie miało, żadnego wsparcia wstecznego, dla starych modeli GPU.

Oczywiście istnieje sposób, by osiągnąć pożądane rezultaty graficzne oraz bardzo dobry czas renderowania. Można skorzystać z metod aproksymujących rozwiązanie, bez niepotrzebnych kosztów związanych z wielokrotnych próbkowaniem danych pikseli. Oczywiście takie rezultaty nie będą bardziej realistyczne od path tracingu, czy ray tracingu, ale będą dawały nam bardzo dobrą płynność obrazu i dobrą grafiką.

To samo tyczy się okluzji otoczenia; ale najpierw czym jest okluzja otoczenia (AO)? Jest to metoda zacieniania, krawędzi/przestrzeni/obiektów znajdującej się w bardzo bliskim środowisku względem siebie. Technika ta określa odsłonięcie danej



Rysunek 6: Porównanie metod oświetlenia.
Źródło: blogs.nvidia.com.

powierzchni na rozproszone światło. Jej celem jest nadanie głębi obrazowi oraz sprawienie, żeby wydawał się bardziej realistyczny.

Rysunek 7: Implementacja okluzji otoczenia - HBAO.
Źródło: opracowanie własne.

Istnieje algorytm dający bardzo realistyczne rezultaty zwany RTAO (ang. Ray-Traced Ambient Occlusion)[5]. Jednak cierpi on na taką samą przypadłość jak ray tracing używany do oświetlenia - jest bardzo wolny.

Rysunek 8: Porównanie RTAO dla 1, 10, 100 i 1000 promieni. Źródło: alexis.breust.fr.

1 promień/piksel	10 promieni/piksel	100 promieni/piksel	1000 promieni/piksel
$1{,}67ms$	$6{,}03ms$	$49{,}38ms$	$485{,}1ms$

Tabela 1: Czasy osiągane dla poszczególnych ilości promieni. Źródło: alexis.breust.fr.

Jak można zauważyć RTAO tak samo jak ray tracing nie jest idealnym rozwiązaniem w grafice czasu rzeczywistego.
To co można zrobić, to zastąpić bardzo realistyczną okluzję, algorytmem aproksymującym okluzję otoczenia. Efekt ten będzie poprawiał głębię obrazu, oraz będzie **znacznie** szybszy od dokładnego rozwiązania.

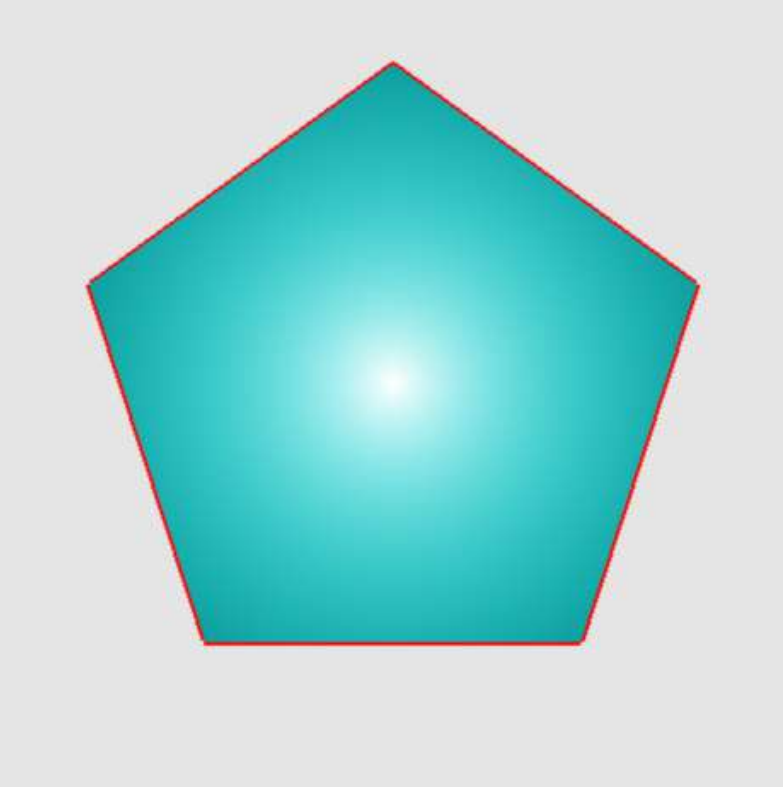
Rozdział 1

Wprwadzenie do renderowania grafiki 3D

1.1. Wstęp do potoku graficznego

Aby móc wyrenderować dowolny model na ekranie musi on przejść przez cały proces potoku graficznego potocznie nazywanego *pipelinem graficznym*. Modele są przechowywane za pomocą wierzchołków (później dokładniej to wyjaśnimy). Każdy model posiada z góry określoną liczbę wierzchołków. Przy ich podstawie tworzone są tak zwane *prymitywy*, czyli figury które wyznaczane są przez te wierzchołki. Mogą to być: punkty, odcinki, łamane, trójkąty i wiele innych (na nasze potrzeby ograniczymy się do trójkątów). Każdy z takich prymitywów następnie przechodzi proces montażowy.

Na tym etapie mamy już teoretycznie figury odpowiadające poszczególnym prymitywom. Następnie dokonywana jest ich rasteryzacja, czyli *przekonwertowanie* danego prymitywu, na poszczególne piksele (zwane również *fragmentami*). Proces ten polega na określeniu w jakim stopniu dany prymityw pokrywa dany fragment. Jeśli dany fragment zawiera wystarczająco dużą powierzchnię piksela, to będzie brał on dalej udział w przetwarzaniu. Dzieje się to w pełni automatycznie. Jedyne co nam teraz zostało to pokolorować dane fragmenty.



Rysunek 9: Przykładowa zrasteryzowana figura.
Źródło: opracowanie własne.

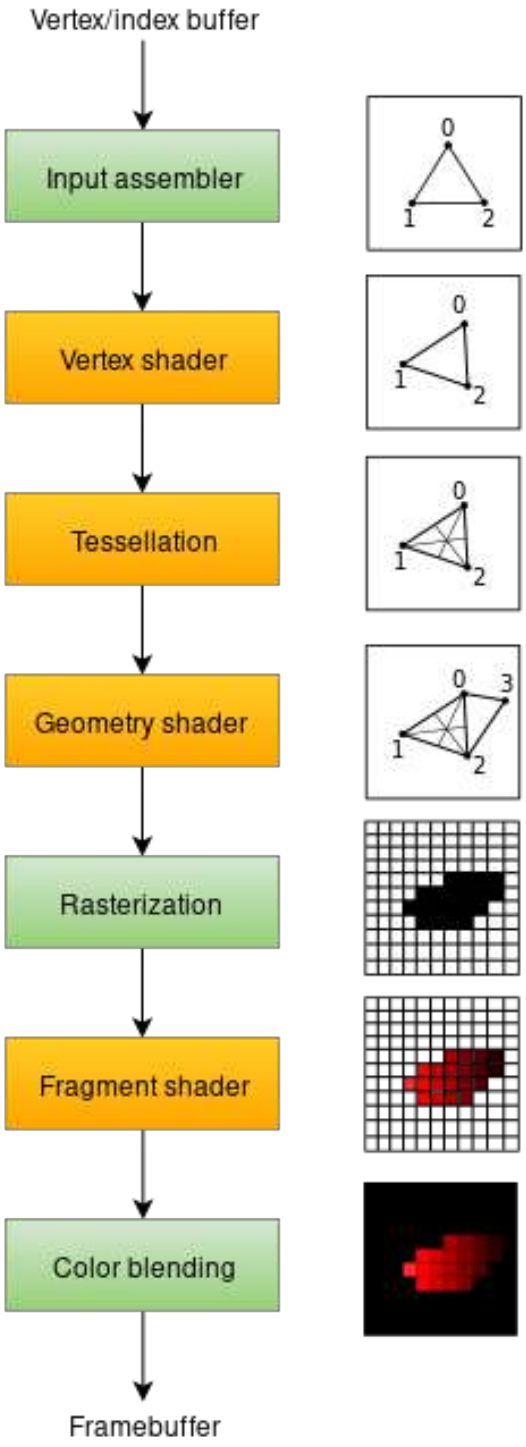
Grafika 9 przedstawia przykładową zrasteryzowaną grafikę. Składa się ona z 6-ciu wierzchołków (5-ciu pokrywających się z wierzchołkami pięciokąta, oraz jednego znajdującego się w samym środku figury). Tak jak wcześniej wspomnieliśmy figury powstają na podstawie tak zwanych prymitywów. Jest wiele sposobów na stworzenie takiej figury, tutaj zastosowane zostały trójkąty, co obliguje nas do dodania dodatkowego wierzchołka w środku figury (każdy trójkąt to 2 kolejne wierzchołki pięciokąta połączone z wierzchołkiem w środku); dla podkreślenia faktu, że można mieszać prymitywy zostały dodane czerwone odcinki nachodzące na krawędzie figury.

Po stworzeniu figury z prymitywów na rysunku 9 następuje rasteryzacja. Pokrycie figury jest zależne od odległości w jakiej dany fragment znajduje się od wierzchołka w środku figury.

Warto wspomnieć, że do tego momentu mogliśmy jedynie ingerować w *wierzchołki* oraz *fragmenty* (teoretycznie istnieją jeszcze 2 etapy *geometria* i *teselacja*, ale je można ominąć i nie będą nam one tutaj potrzebne). Ta *ingerencja* odbywa się przy pomocy tak zwanych *shaderów*, czyli programów wykonywanych na karcie graficznej. Te małe programy pozwalają nam ingerować w kształt i położenie w przypadku shadera geometrii, oraz kolor, teksturowanie i post-processing w przypadku shadera fragmentów.

Teraz pozostają nam tylko tak zwane *testy*. Rozumiemy przez to finalne operacje sprawdzające które fragmenty powinny być wyświetlane, czy blendowane (mieszanie). Musimy pamiętać, że nasze prymitywy mogą się np. pokrywać, więc musimy zdecydować, które z pokrywających się pikseli powinny się wyświetlać. Taki test nazywa się *testem głębokości*. Testów jest kilka i nie będziemy się w nie wszystkie zagłębiać.

Grafika 10 zawiera diagram przedstawiający pipeline graficzny. Dane wejściowe wchodzą na samym początku diagramu, a na wyjściu dostajemy gotową grafikę. Każdy z kolejnych bloków w diagramie reprezentuje jeden z etapów pipelinu. Żółtym kolorem oznaczone są etapy, które można konfigurować, a zielonym kolorem etapy których nie da się konfigurować[6].



Rysunek 10: Pipeline graficzny. Żółtym kolorem oznaczone są etapy na które użytkownik ma wpływ, a zielonym niezależne
Źródło: vulkan-tutorial.com.

1.2. Shader wierzchołków

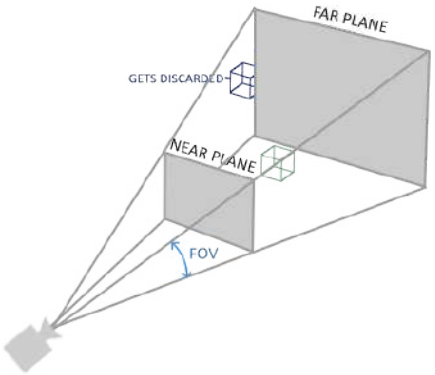
Znany również jako *vertex shader*, jest pierwszą częścią w potoku graficznym na jaką można wpływać. Na wejściu dostaje ona uprzednio przygotowane wierzchołki danych prymitywów. Przy pomocy *shadera* możemy teraz ingerować w położenie wierzchołków. Dla konkretnego modelu, shader taki będzie wykonywać się tyle razy ile jest wierzchołków określonych dla poszczególnych modeli. Wszystkie te programy wykonywane są współbieżnie (tak samo jak we fragment shaderze). Na koniec wykonywania każdego shadera wierzchołków, wartością zwracaną powinna być pozycja w postaci wektora 4D. Tutaj warto podkreślić, że na tym poziomie wszystkie współrzędne wierzchołki muszą znajdować się między $-1{,}0$, a $1{,}0$. Takie znormalizowanie nazywamy normalizacją do postaci NDC (ang. Normalized Device Space).

1.2.1. NDC

Wierzchołki znajdujące się w NDC, nie koniecznie muszą się jednocześnie znajdować w przedziale od $-1{,}0$, do $1{,}0$, natomiast każdy wierzchołek, który wychodzi poza ten zakres, zostanie odrzucony w dalszej części potoku. Gwarantuje nam to tak zwana *przestrzenią obcinania*.

1.2.2. Bryła obcinania

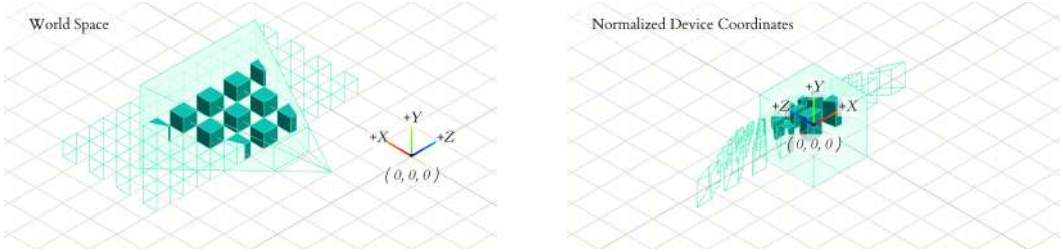
Bryła obcinania jest to ostrosłup ścięty wyznaczający przestrzeń obcinania. Bryła ta jest niezależnym i abstrakcyjny bytem niebędący częścią żadnego systemu graficznego. Jest to część techniki rzutowania elementów 3D na 2D. Przy pomocy tej bryły można łatwo przejść z *widoku świata* do *widoku kamery*, a następnie do NDC. Proces ten polega na wymnożeniu macierzy modelu (macierzy przechowującej informacje o pozycji, rotacji itd.) przez 3 macierze odpowiadające za poszczególne przekształcenia, doprowadzając tym samym do NDC.



Rysunek 11: Przestrzeń obcinania wyznaczana przez near plane i far plane
Źródło: learnopengl.com.

Doprowadzenie do postaci NDC w praktyce oznacza to, że obiekty znajdujące się w przestrzeni obcinania stają się rozciągnięte i znormalizowane. Wierzchołki w takiej postaci są gotowe do przesłania dalej do potoku.

Musimy jednak zwrócić uwagę, że doprowadzanie do postaci NDC jest czasem niepożądanym rezultatem. W takich sytuacjach najczęściej nie ingeruje się w położenie w ogóle.



(a) Przestrzeń świata. (b) NDC.

Rysunek 12: Porównanie przestrzeni świata z NDC
Źródło: jsantell.com.

1.3. Shader fragmentów

Następnym w pipelinie jest *shader fragmentów*. Tak jak wcześniej wspomnieliśmy, między shaderami wierzchołków i fragmentów znajdują się 2 (pomijalne) shadery: geometrii i tesselacji. Nie będziemy się w nie zagłębiać ale dla spójności i lepszego zrozumienia, jednym zdaniem opiszemy ich rolę żeby lepiej zrozumieć co dzieje się we fragment shaderze.

- Shader geometrii pozwala nam dodać dodatkowe wierzchołki (prymitywy) do istniejących wierzchołków.

- Teselacja pozwala na dzielenie istniejącej geometrii na mniejsze (najczęściej) trójkąty.

Ważne jest, żeby wiedzie, że na moment w którym wchodzimy do shadera fragmentów nie mówimy już o geometri a *pikselach* i *tekselach*. W poprzednich shaderach operowaliśmy na geometrii i figurach, teraz natomiast mamy już tylko do czynienia z rzeczywistym[1] pikselem na ekranie.

Celem shadera fragmentów jest ustawienie koloru danego piksela. Przez *piksel* rozumiemy - najmniejszą niepodzielną jednostkę wyświetlaną na ekranie. Tak samo jak w przypadku shadera wierzchołków, każdy z pikseli będzie miał uruchomioną własną

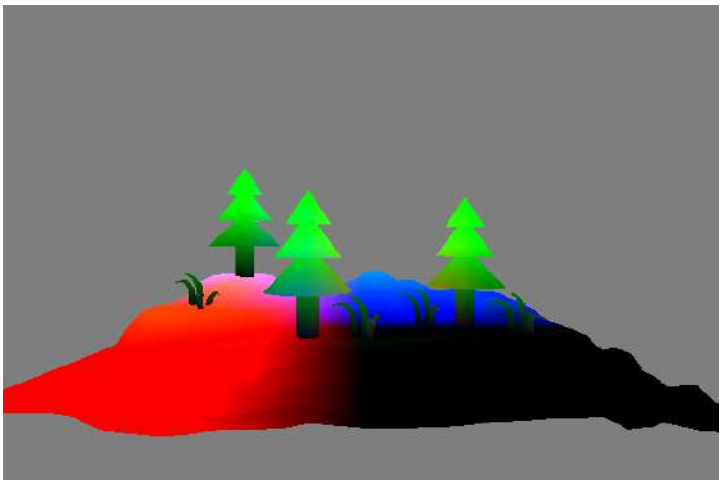
[1]W dalszej części pracy omówimy pojęcie pojęcie FBO, które wyjaśni, że to co będzie rezultatem shadera fragmentów może mieć inne zastosowanie niż wyświetlanie swojej zawartości na ekranie. Będziemy wtedy mówić, że obiekt taki jest *tekselem*, a nie pikselem.

instancję programu na karcie graficznej.

To czego nie wyjaśniliśmy wcześniej to jak dane dostają się do shadera. Są 3 podstawowe metody przesyłania zmiennych do shaderów, gdzie każde z nich ma inne zastosowanie.

- *Layout binding*, czyli technika służąca do przesyłania dużych bolków pamięci.
- *Zmienne jednorodne*, czyli przesyłanie pojedynczej zmiennej na zasadzie powiązania jej, z nazwą zmiennej po stronie shadera.
- *Interpolacja*, czyli przesyłanie danych z jednego shadera do kolejnego shadera w potoku.

Na chwilę pochylimy się nad interpolacją, żeby zrozumieć dlaczego na poziomie shadera fragmentów jest to bardzo ważne. O ile dwie pierwsze techniki wykonywane są z pozycji CPU, to interpolacja odbywa się między shaderami, co oznacza, że dane, które przesyłamy są w pełni odseparowane od programu i dzieją się bezpośrednio na karcie graficznej. Jest to o tyle ważne, że mogą istnieć dane które będą mogły być wygenerowane tylko na poziomie shadera wierzchołków. Przykładem takim może być uzależnienie koloru od pozycji wierzchołka. Takiej operacji nie da się wykonać z pozycji shadera fragmentów, ani z pozycji CPU. Pozostaje więc wygenerowanie danych z pozycji shadera wierzchołków i rozpropagowanie danych dalej[2].



Rysunek 13: przykład użycia pozycji jako koloru
Źródło: opracowanie własne.

[2]Oczywiście, zawsze można na siłę przesyłać dane bezpośrednio do shadera fragmentów i obejść ten problem, ale może być to bardzo mało wydajne, a czasem tak jak w tym przypadku niemożliwe. Pamiętajmy, że takie operacje generowania per-wierzchołek, a per-fragment mogą diametralnie się różnić się co do ilości potrzebnych zasobów.

1.4. FBO

Po wykonaniu shadera fragmentów pozostają już tylko testy i mieszanie kolorów. Nasza grafika przeszła cały pipeline graficzny i jest gotowa do wyświetlenia. Jest gotowa, a nie wyświetlana ponieważ nasza grafika trafia do tak zwanego *FBO* (ang. Frame Buffer Object). Dla wielu grafik renderowanych przy pomocy pipelina graficznego, można abstrahować od używania FBO w ogóle. Nie zmienia to faktu jednak, że grafika trafia do tak zwanego *domyślnego framebuffera*, który może być obsłużony automatycznie. Więc czym jest FBO? Jest to bufor przechowujący dane wyjściowe z naszego pipelina. Ze względu na kosztowność kontaktu między GPU, a CPU, wyrenderowane FBO przechowywane bezpośrednio jest na karcie graficznej w miejscu zwanym VRAM (ang. Video Random Access Memory), żeby mieć do niego szybszy dostęp.

1.5. Tekstura a FBO

W poprzednich rozdziałach wprowadziliśmy pojęcie *teksela*, będzie nam ono teraz bardzo przydatne. Teksel mianowicie oznacza niepodzielną jednostkę tekstury/grafiki. Teksturą nazywamy macierz prostokątną tekseli. Informacje przechowywane w takiej macierzy są ograniczane przez wcześniej ustalony format danych[3], natomiast interpretacja danych w tekslach nie ogranicza nas do traktowania takich danych jako kolor. Samo FBO pozwala na załączanie i przechowywać takich macierz informacji. Oznacza to, że możemy je próbkować, albo prowadzić jego postprocessing. Daje nam to możliwość generowania wielu grafik zanim jakąkolwiek wyświetlimy na ekranie. Mówimy wtedy o *renderowaniu pozaekranowym*. FBO posiada także inne *załączniki* poza *kolorem* (omówimy je dokładniej podczas opisu pojęcia *G-buffer*'a). Mogą tam być mianowicie przechowywane:

- *głębokość*
- informacje do testu *szablonowego*
- dodatkowe załączniki uniwersalnego przeznaczenia zdefiniowanego przy ich tworzeniu[4].

W przypadku generowania okluzji otoczenia, sposobność ta otwiera przed nami bardzo wiele możliwości i jest ona kluczową częścią działania wielu algorytmów.

[3] Przeważnie jest to do 24 bitów na teksel, w przypadku chęci przechowywania danych na 4 kanałach (RGBA).

[4] Przykładem takiego dodatkowego załącznika może być tekstura, która w swoich tekselach przechowuje informacje o wektorach normalnych modeli.

Rozdział 2
Okluzja otoczenia

2.1. Czym jest Okluzja?

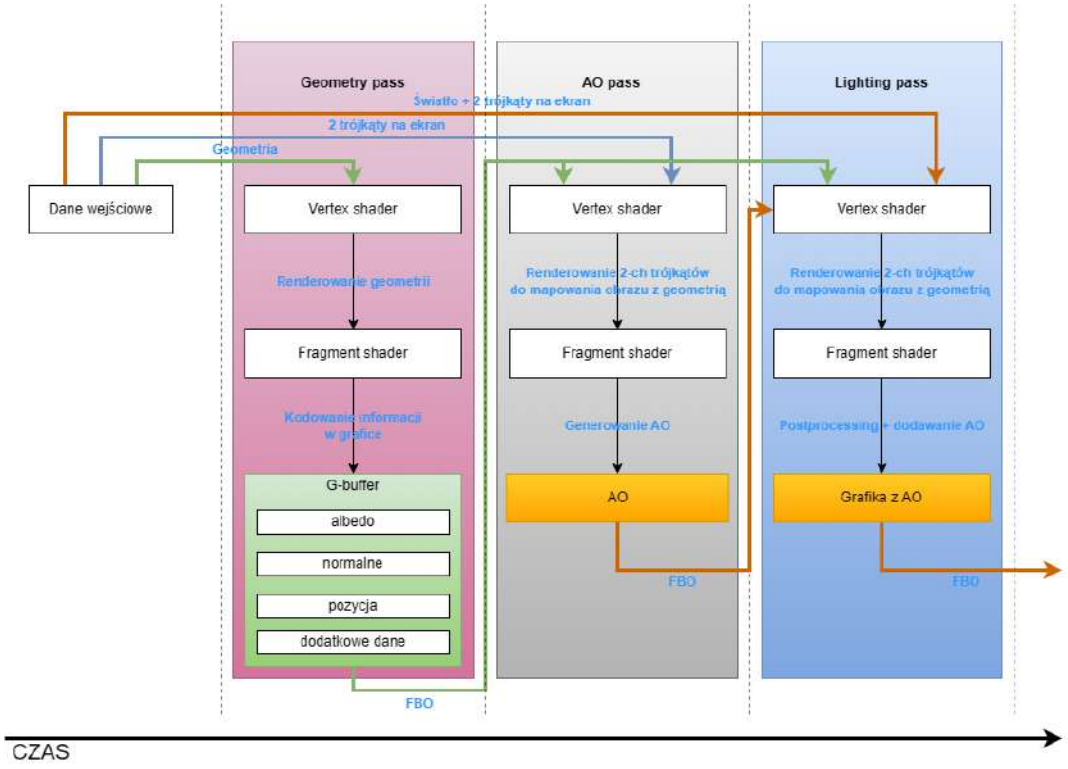
Po wstępnym zrozumieniu potoku graficznego możemy przejść bezpośrednio do okluzji otoczenia. Tak jak wcześniej wspomnieliśmy jest to technika zacieniania przestrzeni przy ostrych krawędziach. Imitować ma ona zachowanie rozproszonego światła, które nie dociera do zasłoniętych miejsc. Widoczne jest to we wszystkich bardzo przysłoniętych miejscach, do których światło powinno mieć bardzo ograniczony dostęp (np. za meblami, pomiędzy wieloma małymi przedmiotami, przy zgięciach na tkaninach itd.).

Efekt ten jest osiągany przez wyrenderowanie osobnego FBO (grafiki) z samą okluzją, a następnie nałożenie jej na grafikę docelową (kolejno grafiki 3 i 1). Wiele z AO stosuje technikę *odroczonego cieniowania* (ang. deffered shading). Polega ona na kompletnym odseparowaniu procesów[1] renderowania *geometrii* od *oświetlania* - stąd nazwy tych faz: *geometry pass* i *lighting pass*. Pozwala to nam w bardzo łatwy sposób dodać okluzję do finalnej grafiki, a przy okazji zoptymalizować czas renderowania[7]. Deffered shading niesie ze sobą koszt pełnego przebiegu pipelina graficznego. Jest to kosztowny proces, ale przy założeniu, że proces ten nie będzie zbyt często wykonywany przed wyświetleniem jednej klatki na ekranie, cena jaką trzeba ponieść jest akceptowalna.

Poza geometry passem i lighting passem które konstytuują deffered shading występuje coś co roboczo nazwiemy *AO pass*. Przebieg ten polegać będzie na generowaniu okluzji w oparciu o dane wychodzące z geometry passa. Dane te będziemy nazywać *G-buffer*. Dane będą przechowywane w postaci wyrenderowanych obrazów, które mają zakodowane informacje w postaci kolorów w poszczególnych tekselach. Oznaczać to będzie, że gdy będziemy odwoływać się konkretnego teksela, ale w różnych *warstwach* G-buffera, będziemy dostawać wszystkie potrzebne nam informacje o modelu, patrząc z pozycji ekranu. Podejście takie są jednym z najbardziej trywialnych i nazywane jest okluzją *przestrzeni ekranowej* (ang. space screen).

[1] Warto tutaj zaznaczyć, że mimo, że proces ten jest odseparowany od siebie (za geometrię odpowiada vertex shader, a za oświetlenie najczęściej fragment shader), to w przypadku stosowania normalnego oświetlenia do wyrenderowania jednego modelu potrzebne są wszystkie *używane* światła, które biorą udział w oświetleniu (oznacza, to że w najgorszym przypadku mogą to być wszystkie światła). Oznacza to, że w pesymistycznym przypadku dla m modeli i n świateł, ilość przesyłanych danych do shadera, może być równa co najmniej $m \cdot n$. Stosowanie deffered shadingu pozwala nam dokonywać obliczenie po wyrenderowaniu wszystkich obiektów po kolei (bez świateł), a na koniec dostarczyć wszystkie światła za jednym razem, redukując tym samym złożoność do $m + n$. Dla bardzo uproszczonych scen nieznacznie pogarsza nam to wydajność, ale, dla bardziej nas interesujących złożonych scen, przyspieszenie jest bardzo znaczące.

Tak opisany proces powstawania AO możemy przedstawić w następujący sposób przedstawiony poniżej.



Rysunek 14: Uproszczony schemat powstawania AO
Źródło: opracowanie własne.

Diagram na rysunku 14 tłumaczy jedną rzecz dotyczącą tworzenia okluzji - mianowicie żeby wyprodukować jeden obraz z okluzją należy przejść przez cały proces pipelina graficznego aż 3 razy. Tworzenie AO natomiast musi/powinien[2] mieć swój własny potok graficzny.

2.2. Generowanie AO

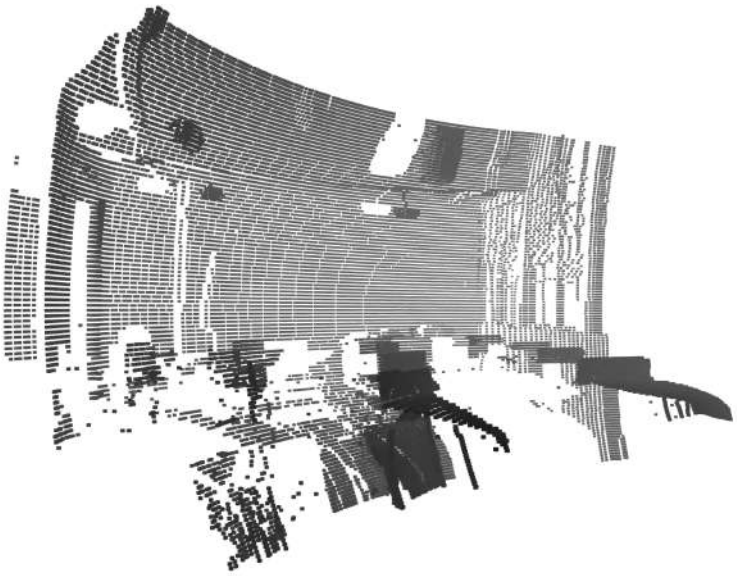
Powiedzieliśmy jak tworzona jest grafika z okluzją, przejdziemy zatem do generowania samej okluzji otoczenia - przebiegu AO (na grafice 14 oznaczony jest jako *AO pass*). Użyjemy algorytmu SSAO (Space Screen Ambient Occlusion) jako przykładu do wytłumaczenia metodyki[8]. Jak już powiedzieliśmy wcześniej, AO pass nie posiada dostępu do geometrii, posiada natomiast dostęp do G-buffera. Zakodowane mamy tam informacje dotyczące: koloru, wektorów normalnych wierzchołków, pozycji, oraz dodatkowych informacji które mogą być potrzebne w postprocessingu. Wszystkie te dane są w odniesieniu do obiektów znajdujących się najbliżej ekranu (dla danego teksela).

[2]Można cały AO pass wykonać w lighting passie, ale wiąże się to z licznymi ograniczeniami (np. *blur* (rozmycie), nie może być zastosowany).

(a) Grafika komputerowa.

(b) Spikselizowana grafika komputerowa.

Rysunek 15: Uproszczona pikselizacja grafiki
Źródło: youtube.com.



Rysunek 16: Uproszczona interpretacja graficzna widoku z rysunku 15
Space Screen widziana z boku (z uwzględnieniem głębokości).
Źródło: youtube.com.

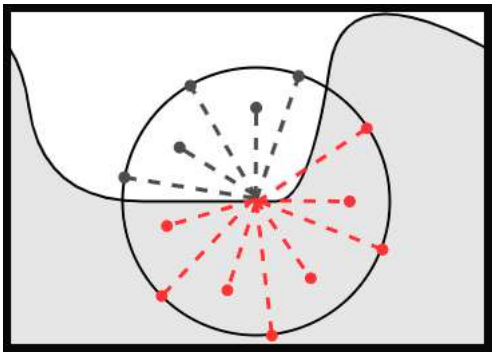
Weźmy scenę która przedstawia szkolną klasę i umieśćmy w niej kamerę (rysunek 15a). Kamera ta będzie przy pomocy odcieni szarości określać głębokość (odległość od kamery). Kolorem czarnym będziemy kolorować obiekty blisko kamery, a białym znajdujące się bardzo daleko. Wykonajmy następnie separację każdego z pikseli tak, żeby każdy z nich był oddzielnie (rysunek 15b)

Na rysunku 16 można zaobserwować to co widzi kamera. Każdy ze znajdujących się tam punktów odpowiada jednemu pikselowi grafiki widzianej z pozycji ekranu, ale z uwzględnieniem odległości od kamery w zależności od nasycenia danego fragmentu.

Należy pamiętać, że wykonując przebieg AO nie mamy dostępu do *modeli*, natomiast mamy dostęp do grafiki *reprezentującej modele*. Okluzja musi mieć dostęp do informacji o modelach znajdujących się w jej otoczeniu. Nie da się tego osiągnąć w prosty sposób. Na przykład działając na geometry passie, informacje o geometrii w

danych pikselach są renderowana równolegle (dany piksel nie ma wglądu w zawartość innego piksela) zatem piksele nie mogą się komunikować. Deffered shading idealnie rozwiązuje nam ten problem dając dostęp do wspólnej, wcześniej wyrenderowanej tekstury (G-buffer) każdemu pikselowi, z której można już łatwo otrzymać informacje, jako że jest ona w pełni wyrenderowaną grafiką.

To co teraz robi SSAO, to dostarcza do shadera fragmentów uprzednio wylosowane punkty[3] znajdujące się w jakimś promieniu od danego piksela. Następnie sprawdza czy wylosowane punkty znajdują się *przed*, czy *za*, pikselami sąsiadującymi z naszym fragmentem względem ich głębokości. Operację tę wykonuje się dla każdego punktu, a następnie określa się stosunek punktów znajdujących się *za* geometrią, do ilości wszystkich punków. Stosunek ten będzie określał nam nasz *stopień okluzji* dla danego fragmentu.



Rysunek 17: Kernel z wylosowanymi punktami w algorytmie SSAO
Źródło: ceur-ws.org.

Przedstawimy teraz pseudokod generowania AO z pozycji fragment shadera.

Dane na wejściu:

- pozycja danego fragmentu (zm: *pozycjaFragmentu*),
- splot (Kernel z pseudo losowymi punktami, wielkością kernela i promieniem) (zm: *kernel*),
- bias (współczynnik stronniczości) (zm: *bias*),
- G-buffer (tekstury z zakodowanymi informacjami) (zm: *G_buffer*).

Dane na wyjściu:

- Pojedynczy teksel ze współczynnikiem okluzji (zm: *AO*).

[3]Punkty te zaczepione są w tak zwanym *splocie* (inaczej *kenelu*). W naszym przypadku jest to środek piksela.

```
for p in pixels pardo: // wywoływanie dla każdego fragmentu
{
    var AO = 0.0;
    for(int i = 0; i < kernel.size; i++)
    {
        var przesunięcie =
            p.pozycjaFragmentu + p.kernel.position[i] * p.kernel.radius;

        var offset =
            rzutowanieNaNDC(przesunięcie);

        var głębokość = p.G_buffer.position[offset].głębokość;

        AO +=
            (głębokość >= przesunięcie + p.bias ? 1.0 : 0.0)
    }
    AO = 1.0 - (AO / kernel.size);
    return AO;
}
```

Kod 1: Pseudokod wyliczania AO dla SSAO

Na wyjściu otrzymujemy FBO zawierający nasze AO (grafika 3 jest przykładową grafiką która może być rezultatem tworzenia AO). Powyższy pseudokod jest bardzo uproszczoną reprezentacją wszystkich algorytmów stosujących *ekranowe* wersje AO. To co często wykonuje się, a nie jest bezpośrednio częścią algorytmu AO, to podniesienie wartości końcowej okluzji w danym pikselu do jakiejś potęgi, aby osiągnąć efekt nieliniowości rozchodzenia się zacienienia na obrazie.

2.3. Postprocessing

Po otrzymaniu FBO z naszym AO, możemy przejść do przetwarzania grafiki w następnym procesie (lighting pass'ie). Na tym etapie mamy pełen dostęp do informacji o geometrii i kolorach (G-buffer), oraz posiadamy teksturę reprezentującą naszą okluzję (FBO z teksturą AO). Wszystkie te informacje będą nam teraz potrzebne do połączenia tych grafik w jedną całość[4].
Deferred shading pozwala nam teraz w bardzo prosty sposób dodać różne *efekty* graficzne takie jak AO, czy tak jak wskazuje nazwa tego przebiegu, dodanie światła do grafiki.

[4]Jeśli uwzględnimy potencjalną sytuację w której pominiemy światło, a nie jest ono wymagane, to całe *mieszanie* G-buffera z okluzją, można we fragment shaderze, można ograniczyć do przemnożenia koloru (albedo) przez współczynnik okluzji, trywializując lighting pass do jednej linii kodu we fragment shaderze.

Przedstawimy teraz pseudokod łączenia AO z oświetleniem i G-bufferem.

Dane na wejściu:

- G-buffer (zm: *G_buffer*),
- textura z AO (zm: *AO*),
- pozycja i natężenie światła (zm: *daneŚwiateł*).

Dane na wyjściu:

- kolor fragmentu finalnej grafiki (z nałożonymi teksturami, AO, oraz oświetleniem) (zm: *FragColor*).

```
for p in pixels pardo: // wywoływanie dla każdego fragmentu
{
    void main()
    {
        vec3 współczynnikŚwiatła =
            oświetlanie(p.G_buffer, p.daneŚwiateł);
        FragColor =
            vec4(współczynnikŚwiatła * p.G_buffer.albedo * p.AO, 0.0f);
        return FragColor;
    }
}
```

Kod 2: Pseudokod tworzenia grafiki na podstawie AO

Funkcja *oświetlanie()*, odpowiada dowolnej funkcji, która na bazie pozycji (pozyskanej z G-buffera), pozycji światła i jego natężenia, oraz informacji o teksturze, jest w stanie oświetlić dany piksel. Następnie dane o oświetleniu są przemnażane przez albedo (kolor) i AO.

Taka grafika otrzymana na wyjściu lighting pass'a kończy cały proces deferred shading'u, a tym samym generuje nam grafikę z AO.

Rozdział 3

Omówienie poszczególnych algorytmów AO

Algorytmy generujące AO na których skupimy się w tej pracy to: *SSAO*, *SSAO+* i *HBAO*. Omówimy teraz charakterystykę każdego z nich.

3.1. SSAO

Algorytm SSAO pierwotnie został stworzony na potrzeby gry komputerowej o nazwie *Crysis* w 2007 roku przez Vladimira Kajalina. Zapoczątkował on wszelkie rodzaje okluzji przestrzeni ekranowej. Omówiliśmy go częściowo w poprzednim rozdziale, więc teraz skupimy się tylko na jego cechach szczególnych.

Rysunek 18: Okluzjia w algorytmie SSAO i rozmycie
Źródło: Opracowanie własne.

SSAO (rysunek 18) cechuje się niskimi (ciemnymi) wartości kolorów nawet na środkach *gładkich* płaszczyzn (ścian, podług, itp.). Jak już powiedzieliśmy w rozdziale o postprocessingu, łączenie FBO w finalnym obrazie odbywa się na zasadzie mnożenia *per-pixel* (AO, światło i kolor teksela). Wartość wyliczona dla np. środka podłogi będzie zatem bardziej zacieniona niż by się tego oczekiwało. Prowadzi to do sytuacji w której finalny obraz jest nieco przyciemniony.

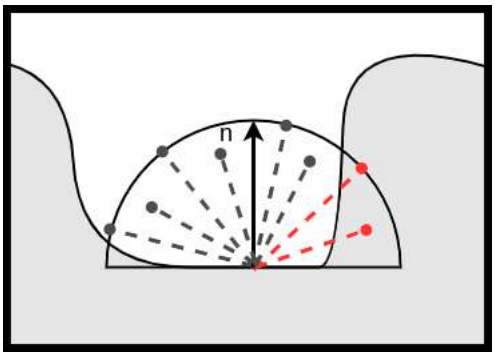
Spowodowane jest to przez sferę z kernelem. Oryginalne SSAO wymaga użycia pełnej sfery, co oznacza, że na płaskiej płaszczyźnie, *statystycznie* połowa punków znajdzie się po jednej stronie, a druga połowa po drugiej stronie. Oznaczać to będzie, że obraz będzie przyciemniony o 50%, niezależnie od położenia i orientacji. Wartość ta będzie się zmieniać dopiero w środowisku krawędzi wklęsłych i wypukłych.

3.2. SSAO+

Algorytm SSAO+ powstał w odpowiedzi na dość widoczne wady oryginalnego algorytmu SSAO, mianowicie zacienianie w miejscach które nie powinny być zacieniane (np. środek ściany).

Rysunek 19: Okluzjia w algorytmie SSAO+ i rozmycie
Źródło: Opracowanie własne.

Na rysunku 19 od razu widać znaczącą poprawę względem SSAO (rysunek 18). Jest to spowodowane zastąpieniem sfery, półsferą skierowaną zgodnie z wektorem normalnym danej płaszczyzny (rysunek 20).



Rysunek 20: Kernel z wylosowanymi punktami w algorytmie SSAO+
Źródło: ceur-ws.org.

Tak zorientowana półsfera sprawia, że jeśli w jej towarzystwie nie znajduje się żadna krawędź, to wartość jej stopnia zacieniowania zawsze będzie równa 0%.

Przedstawimy teraz pseudokod generowania AO dla SSAO+ z pozycji fragment shadera. Dane wejściowe i wyjściowe są podobne do danych z metody SSAO.

Dane na wejściu:

- pozycja danego fragmentu (zm: *pozycjaFragmentu*),
- splot (zm: *kernel*),
- bias (zm: *bias*),
- G-buffer (zm: *G_buffer*).

Dane na wyjściu:

- Pojedynczy teksel (część tekstury AO).

```
for p in pixels pardo: // wywoływanie dla każdego fragmentu
{
    var AO = 0.0;
    kernel = ustawienieKernelaProstopadleDoPłaszczyzny(kernel);
    for(int i = 0; i < kernel.size; i++)
    {
        var przesunięcie =
            p.pozycjaFragmentu + p.kernel.position[i] * p.kernel.radius;

        var offset =
            rzutowanieNaNDC(przesunięcie);

        var głębokość = p.G_buffer.position[offset].głębokość;

        AO +=
            (głębokość >= przesunięcie + bias ? 1.0 : 0.0)
    }
    AO = 1.0 - (AO / p.kernel.size);
    return AO;
}
```

Kod 3: Pseudokod wyliczania AO dla SSAO+

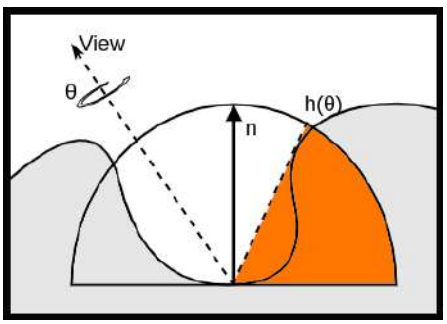
Jedyne znaczące różnice względem algorytmu SSAO to fakt, że pseudolosowe punkty znajdują się w półsferze, sama półsfera jest zorientowana prostopadle do płaszczyzny.

3.3. HBAO

HBAO jest obecnie jednym z najbardziej popularnych algorytmów AO. Został on zaprojektowany przez Nvidia'e pod kontem wysokiej wydajności, oraz większego realizmu niż SSAO+[9][10].

Rysunek 21: Okluzja w algorytmie HBAO i rozmycie
Źródło: Opracowanie własne.

Swój poziom zaawansowania gwarantuje sobie przez uwzględnianie horyzontu przy pomocy kąta azymutalnego θ (kąt między konkretnym wyznaczonym punktem, a wysokością (obiektu na półsferze) spadającą na podstawę półsfery. Im bardziej kąt azymutalny zmierza w kierunku w którym przepływ światła jest zablokowany przez inne obiekty, tym większe zaciemnienie jest uwzględniane w obliczeniach. Przeszukuje się określone sąsiedztwo danego piksela i wybiera się największy z pośród ostrych kątów azymutalnych, a następnie na podstawie tego kąta określa się stopień zacienienia danego piksela (rysunek 22).



Rysunek 22: Horyzont w algorytmie HBAO
Źródło: ceur-ws.org.

Przedstawimy teraz pseudokod generowania AO za pomocą HBAO z pozycji fragment shadera. Dane wejściowe i wyjściowe są podobne do danych z metody SSAO.

Dane na wejściu:

- pozycja danego fragmentu (zm: *pozycjaFragmentu*),
- wektor normalnych (zm: *vecNormalny*),
- bufor głębokości (wylicznany w *FetchViewPos()*),
- współrzędne UV tekstury (wylicznany w *texCoord*).

Dane na wyjściu:

- Pojedynczy teksel (część tekstury AO).

```
for p in pixels pardo: // wywoływanie dla każdego fragmentu
{
    vec2 widok = FetchViewPos(texCoord);
    float AO = 0.0; // okluzja
    float promieńWPikselach = stosunekPromieniaDoEkranu / widok.z;
    for(var kierunekID = 0; kierunekID < ILOŚĆ_KIERUNKÓW; kierunekID++){
        var kąt =
            aktualnyKierunekPróbkowania(kierunekID, ILOŚĆ_KIERUNKÓW);

        var kierunek =
            obróćWektorZDrżeniem(kąt, getJitter().xy);

        var promieńWPikselach =
            (getJitter().z * promieńWPikselach + 1.0);

        for(float i = 0; i < ILOŚĆ_KROKÓW; i++){
            vec2 shiftUV = przesunięcieTekstury(promieńWPikselach,
                kierunek);

            vec3 pozycjaPróbki = PozycjaPróbkiWWidoku(shiftUV);

            AO += policzAO(
                p.pozycjaFragmentu, p.vecNormalny, pozycjaPróbki);

            promieńWPikselach +=
                (promieńWPikselach / (ILOŚĆ_KROKÓW + 1));
        }
    }
    // uśrednienie okluzji dla wszystkich kierunków i próbek
    AO /= (ILOŚĆ_KIERUNKÓW * ILOŚĆ_KROKÓW);
    // korekcja/ograniczenie wartości do przedziału <0, 1>
    return clamp(1.0 - AO * 2.0,0,1);
}
```

Kod 4: Pseudokod wyliczania AO dla HBAO

Opis najważniejszych fragmentów pseudokodu:

- *getJuitter()* - funkcja zwraca aktualny kierunek próbkowania jitteringu (niewielkie odchylenia od oryginalnej wartości - porównywalne z szumem),
- *policzAO()* - funkcja obliczająca okluzję dla danej próbki w przestrzeni widoku z uwzględnieniem fragmentu, wektora normalnego i pozycji próbki,
- *promieńWPikselach* - parametr kontrolujący odległość próbkowania we fragmentach od punktu widoku, w danym kierunku, pozwalając nam uniknąć efekt artefaktów poprzez zwiększenie losowości.

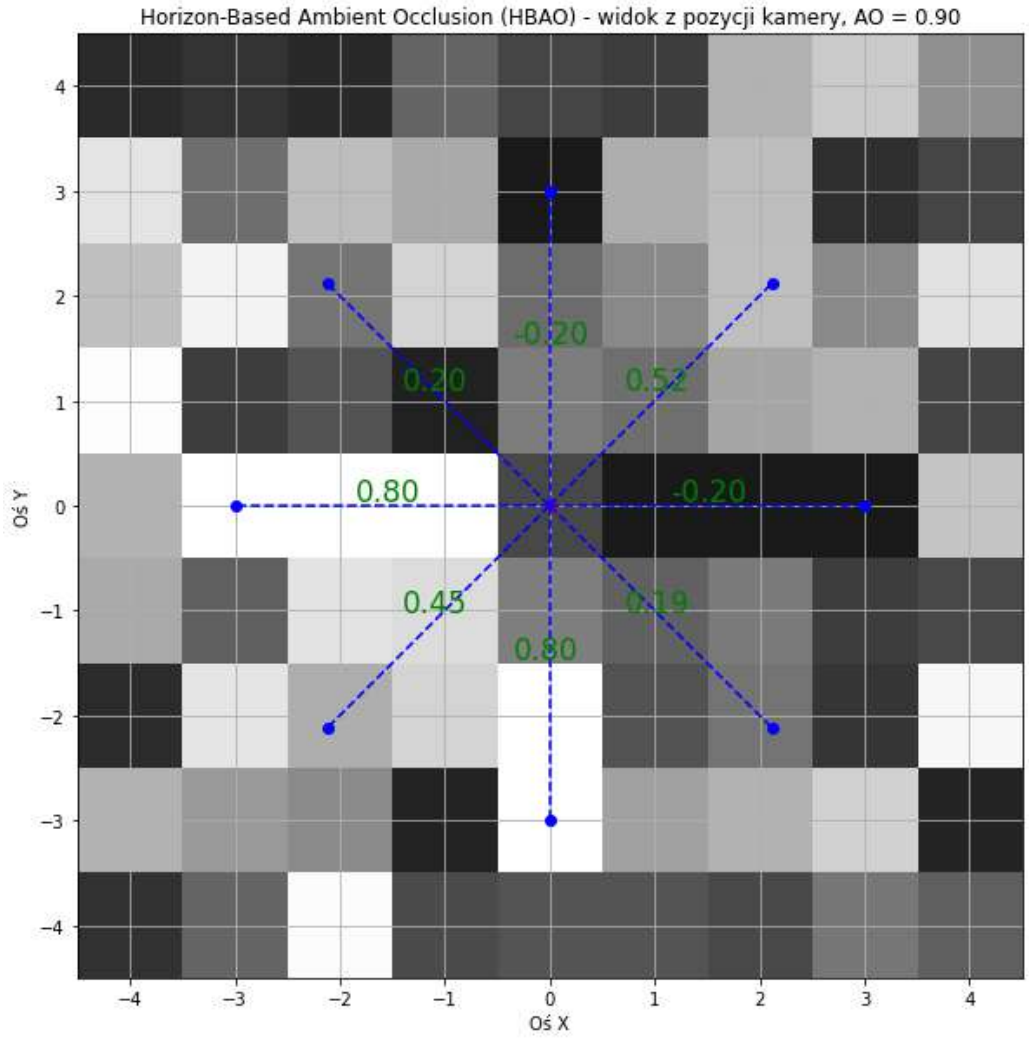
To co wykonuje algorytm HBAO, to uwzględnienie okluzji w różnych kierunkach. Osiąga to poprzez liczenie pola powierzchni wyznaczanej między powierzchnią, a odcinkiem wyznaczanym przez kąt azymutalny θ. Pole to to możemy opisać wzorem[9]:

$$AO \approx 1 - \frac{1}{2\pi} \int_{\theta=-\pi}^{\pi} W(\theta) \sin(h(\theta))\, d\theta.$$

Funkcja $W(\theta)$ jest funkcją tłumienia (odległość od najbliższego obiektu w kierunku θ), natomiast $\sin(h(\theta))$ reprezentuje wpływ kąta horyzontu dla danego kierunku, gdzie $h(\theta)$ to maksymalny kąt elewacji pod którym widzimy najbliższy obiekt, który blokuje światło w danym kierunku θ. Operacja to jest wykonywana jest w kilku wyznaczonych kierunkach (ograniczonych przez najbliżej znajdujące się fragmenty). Wynikiem jest poziom okluzji otoczenia w danym punkcie.

Sam wzór jest przybliżeniem Monte Carlo dla losowo wybranych kątów azymutalnych. Jako, że metoda Monte Carlo polega na próbkowaniu losowym, w przypadku okluzji też mówimy o przybliżeniu.

Możemy też spojrzeć na to inaczej - z pozycji widoku kamery (widok z góry na teksturę głębokości). Poniższa grafika przedstawia taką sytuację (rysunek 23). Grafika ta w uproszczony sposób symuluje zachowanie HBAO. Ze środka wypuszczane są *promienie* które sprawdzają różnicę *głębokości* obu tekstur, a wynik dostarczany jest jako wartość wejściowa do obliczeń. Sama grafika przedstawia sytuację w której piksel centralny ma głębokość równą 0.20, natomiast górny i dolny odpowiednio 0 i 1. W poziomie (na lewo i prawo od środka) są 3 piksele które są tego samego samej głębokości (lewy 1, prawy 0). Jak można zauważyć górny i prawy, oraz lewy i dolny promień są sobie równe. Jest tak dla tego, że algorytm nie sprawdza pikseli pomiędzy sprawdzanym pikselem, a pikselem centralnym.



Rysunek 23: Widok tekstury głębokości z góry w algorytmie HBAO.
Źródło: opracowanie własne.

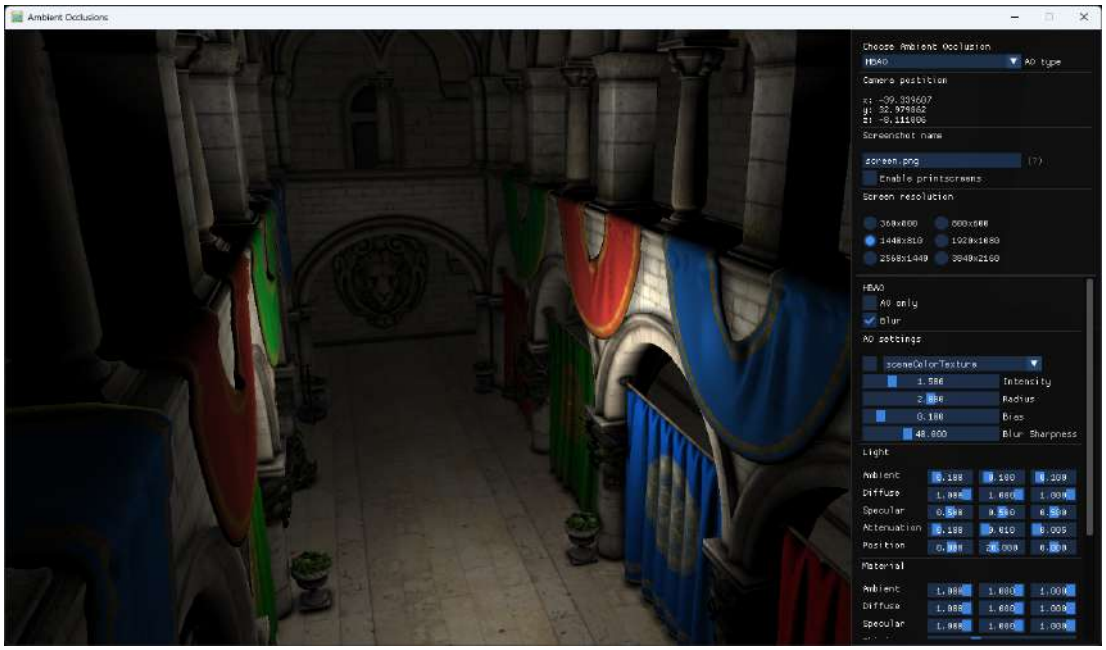
Sama grafika nie reprezentuje *poprawnego* działania algorytmu, ma ona jedynie na celu ułatwienie interpretacji algorytmu HBAO.

Rozdział 4

Omówienie środowiska testowego

Środowisko testowe napisane na potrzeby mojej pracy, pozwala przetestować wszystkie 3 algorytmy. Istnieje możliwość dostosowania parametrów każdego z algorytmów w dowolny sposób. Następnie program udostępnia możliwość przeprowadzenia testów na potrzeby porównania algorytmów. Udostępnione są również pomniejsze funkcjonalności takie jak zapisywanie obrazów bezpośrednio z FBO w celu empirycznego porównania wizualnych efektów każdego z algorytmów, czy też przełączanie pomiędzy poszczególnymi rozdzielczościami.

Sam program został napisany w języku C++ przy pomocy API graficznego *OpenGL*. Shadery zostały napisane w języki GLSL. Interfejs graficzny został zaprojektowany przy pomocy biblioteki *ImGui*. Do ładowania modeli i scen użyta została biblioteka *ASSIMP*. Projekt posiłkuje się fragmentami kodu ze stron *Learn OpenGL* oraz *nvpro-samples*.



Rysunek 24: Wygląd programu do kontroli AO.
Źródło: opracowanie własne.

4.1. Możliwości środowiska

Środowisko testowe pozwala przełączać się między 3 algorytmami AO i dostosowywać je do naszych potrzeb. Udostępniona jest również możliwość poruszania się po samym środowisku. Okno środowisko posiada przełączalny pasek GUI, dzięki któremu można sprawdzić i dostosować wszystkie parametry. W ramach samego środowiska możemy również kontrolować jedno źródło światła i jak ono wpływa na otoczenie (parametry światła i materiałów).

4.2. Funkcjonalności

Środowisko zawiera 2 regulowane, połączone ze sobą okna, pierwsze z parametrami/funkcjonalnościami dotyczącymi samego środowiska i drugie które pozwala nam manipulować wybranymi parametrami: okluzji, światła, materiałów oraz przeprowadzania testów. Okno z opcjami można włączyć i wyłączyć przyciskiem *m* na klawiaturze (blokuje to poruszanie się po środowisku).

Pierwsze okno

Poniżej jest wykaz funkcjonalności pierwszego okna:

- wybór okluzji,
- położenie kamery (bez orientacji),
- wybór nazwy pliku pod jakim będzie zapisany screenshot (screenshot wykonuje się przyciskiem p na klawiaturze),
- wybór rozdzielczości ekranu.

Drugie okno

- przełącznik dla samego AO,
- przełącznik dla rozmycia tekstury,
- zestaw customowych parametrów dla każdego algorytmu AO,
- zestaw customowych parametrów dla dostrajania światła,
- zestaw customowych parametrów dla dostrajania materiałów,
- sekcja analityczna do logowania danych dot. wydajności (do konsoli i plików).

Rozdział 5
Problem badawczy

5.1. Omówienie problemu

Celem tej pracy jest ocenienie, które algorytmy działają najszybciej, oraz które dają najlepsze empirycznie efekty wizualne. Ocenimy jak każdy z nich radzi sobie z: czasem renderowania oraz różnymi rozdzielczościami. Postaramy się również wyznaczyć jakie różnice można zaobserwować pomiędzy każdym z algorytmów. Finalnie wyznaczymy zastosowania w których poszczególne algorytmy będą najlepiej się sprawować.

5.2. Rodzaje testów

Dla każdego algorytmu badana będzie jego:

- wydajność,
- skalowalność,
- jakość obrazu,

a następnie oceniana będzie użyteczność każdego algorytmu w przykładowych sytuacjach i scenach.

Wydajność

Zmierzymy czas renderowania poszczególnych klatek. Interesować nas będzie czas tworzenia samej okluzji, jak i łączny czas tworzenia całej klatki.

Skalowalność

Sprawdzimy jak poszczególne algorytmy radzą sobie z różnymi rodzajami rozdzielczości. Wyrenderujemy AO w różnej rozdzielczości i sprawdzimy jak każdy algorytm radzi sobie ze skalowalnością.

Jakość obrazu

Empirycznie ocenimy które z algorytmów imitują okluzję najlepiej i ocenimy czym jest to spowodowane.

5.3. Metodologia

Podczas wykonywania wszystkich pomiarów skupimy się na tych technikach, żeby móc określić efektywność i przydatność naszych algorytmów:

- **pomiary czasu renderowania** - będziemy mierzyć czas renderowania każdej z klatek, lub pewnego procesu, żeby móc porównywać złożoność czasową poszczególnych algorytmów,
- **testy skalowalności i rozdzielczości** - wykorzystamy kilka spreparowanych ustawień scen na nasze potrzeby i porównamy ich wydajność dla kilku rozdzielczości (interesować nas będzie jak algorytmy sobie radzą z obciążeniem),
- **cyfrowe przetwarzanie obrazów** - wyliczenie wskaźników: korelacji, template matching, SSIM, MSE, PSNR oraz ostrości. Wszystkie z nich opiszemy dokładniej podczas przeprowadzania samych badań.

5.4. Narzędzie pomiarowe

Badania przeprowadziłem na swoim laptopie. Poniżej przedstawiam najważniejsze specyfikacje:

- CPU - 12th Gen Intel(R) Core(TM) i7-12700H 2.70 GHz,
- GPU - NVIDIA GeForce RTX 370 Ti Laptop GPU,
- zintegrowane GPU - Intel(R) Iris(R) Xe Graphics,
- RAM - 2x 32GB DDR4.

Rozdział 6

Badania

Poniżej przedstawiamy badania dla predefiniowanych parametrów do każdego algorytmów. Ma to o tyle znaczenie, że nawet najdrobniejsza zmiana w dowolnym z algorytmów wpływa znacząco na testy jakości obrazów, więc, żeby ujednolicić wszystkie testy, badania przeprowadzone zostały na niemodyfikowanych wariantach algorytmów i dla jednego widoku (rysunek 25).

Rysunek 25: Widok dla którego będą wykonywane badania.
Źródło: opracowanie własne.

- **SSAO**:

  kernel = 64 (ilość punktów używana w kernelu)

  promień = 0.5

  bias = 0.025 (współczynnik stronniczości)

  intensywność = 1 (jest to wykładnik finalnego zacienienia fragmentu)

- **SSAO+**:

  kernel = 64

  promień = 0.5

  bias = 0.025

  intensywność = 1

- **HBAO**:

  promień = 2

  bias = 0.1

  intensywność = 1.5

  ostrość rozmycia = 40

6.1. Badanie wydajności

Wyjaśnienie technologii

OpenGL dostarcza mechanizm *time query* umożliwiający dokładny pomiar czasu wykonywania operacji na karcie graficznej. Uproszczony model działania *time query*:

```
// Generowania identyfikatora który będzie przechowywał przedział czasu
GLuint query;
glGenQueries(1, &query);

// Pomiary przed wywołaniem procesu generowania obrazu
glBeginQuery(GL_TIME_ELAPSED, query);

// Tutaj wykonywane są testy

// Wykonywanie mierzonych operacji
glEndQuery(GL_TIME_ELAPSED);

// Odczyt wyniku pomiaru
GLuint64 timeElapsed = 0;
glGetQueryObjectui64v(query, GL_QUERY_RESULT, &timeElapsed);

// Czas w nanosekundach
double timeInSeconds = timeElapsed / 1e9;

// Wyświetlenie czasu wykonania
std::cout << timeInSeconds << std::endl;

// Usunięcie/Zrestartowanie identyfikatora
glDeleteQueries(1, &query);
```

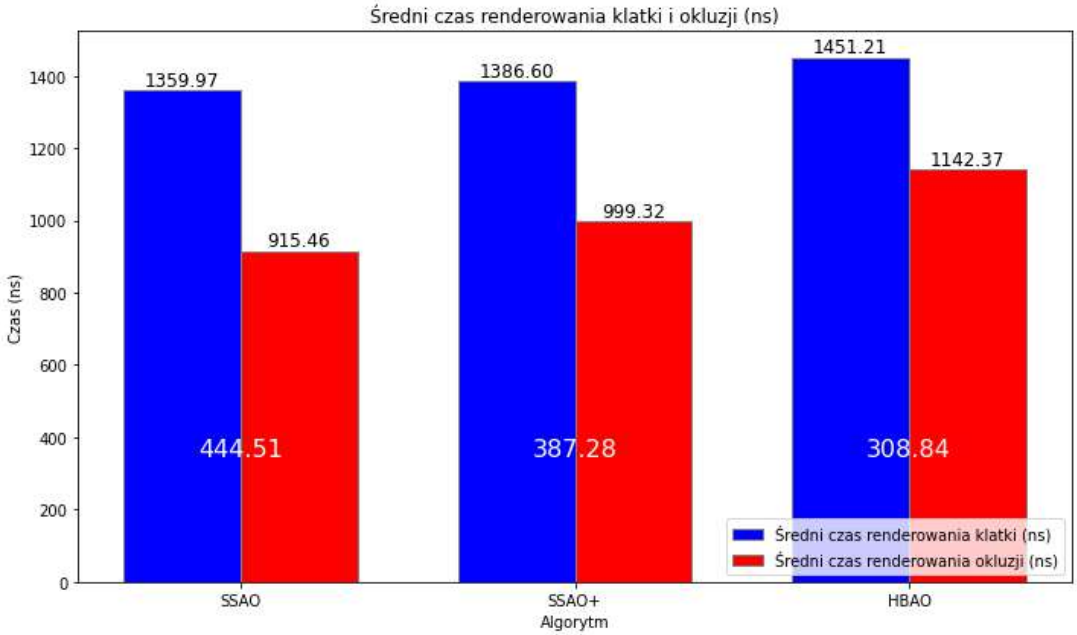
Planowany przebieg badania

Program przeprowadzi serię n operacji sprawdzania czasu renderowania dla każdego pomiaru. Timery zostają włączone przed pierwszą operacją z danego ustawienia, oraz od razu po wykonaniu wszystkich przewidzianych operacji. Testy zostały wykonane na karcie GPU oraz dla n równego 10^4. Lista przewidzianych pomiarów (dla każdego rodzaju okluzji):

- renderowanie całej klatki,
- renderowanie samej okluzji.

Otrzymane pomiary czasów

Algorytm / 10^4 pomiarów	SSAO	SSAO+	HBAO
renderowanie klatki (średni czas)	1359.97	1386.6	1451.21
renderowanie okluzji (średni czas)	915.46	999.32	1142.37



Rysunek 26: Wykresy średnich czasów renderowania.
Źródło: opracowanie własne.

Interpretacja wyników

Wraz ze wzrostem zaawansowania algorytmów rośnie czas renderowania obrazu z poszczególnymi implementacjami AO. Różnice pomiędzy implementacjami SSAO i SSAO+ były niewielkie, co można zaobserwować w różnicach renderowania ($84ns$). Algorytm HBAO cechował się znacznie bardziej zaawansowanym sposobem wyliczania AO, co można zaobserwować porównując jego czas rendorwowania z SSAO+ ($143ns$).

6.2. Skalowalność

Wprowadzenie

Zanim przejdziemy do samych testów skalowalności, wyjaśnimy dlaczego nie będziemy testować skalowalności algorytmów pod kontem ilości renderowanych obiektów, ani ilości renderowanych świateł.

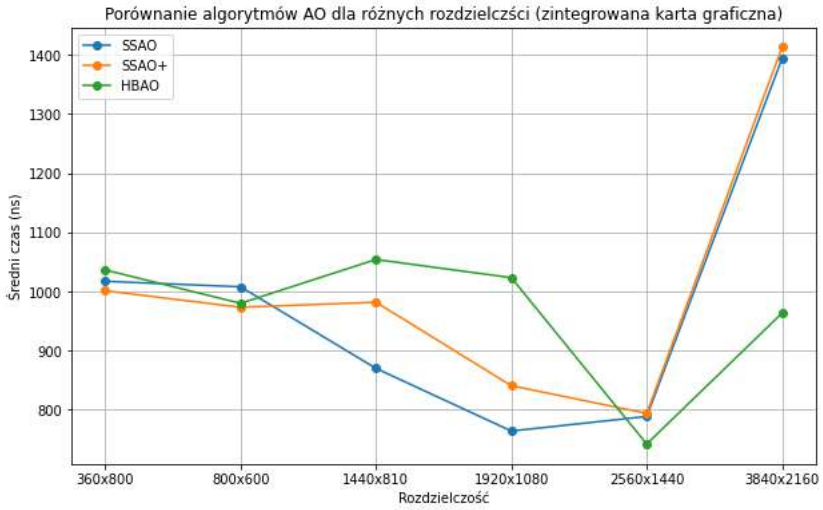
Deffered shading, stosowany w każdym z algorytmów, separuje procesy renderowania obiektów od ich kolorowania. To znaczy, że możemy wyrenderować miliony obiektów i nie wpłynie to w żaden sposób na przebieg AO, ani lighting pass ponieważ na tym poziomie mamy już tylko do czynienia z teksturami (G-bufferem) przy pomocy których wykonywany jest postprocessing. To samo tyczy się dużej ilości świateł. Integrowanie oświetlenia odbywa się dopiero *po* przebiegu AO, zatem oświetlenie samo w sobie nie ingeruje w proces tworzenia okluzji.

Interesuje nas zatem jak bardzo przebieg AO może wpłynąć na przebieg całego renderingu w przypadku gdy np. jakość całego obrazu spadnie, albo się zwiększy. Innymi słowy jak rozdzielczość okna/tekstury wpływa na czas renderingu poszczególnych algorytmów.

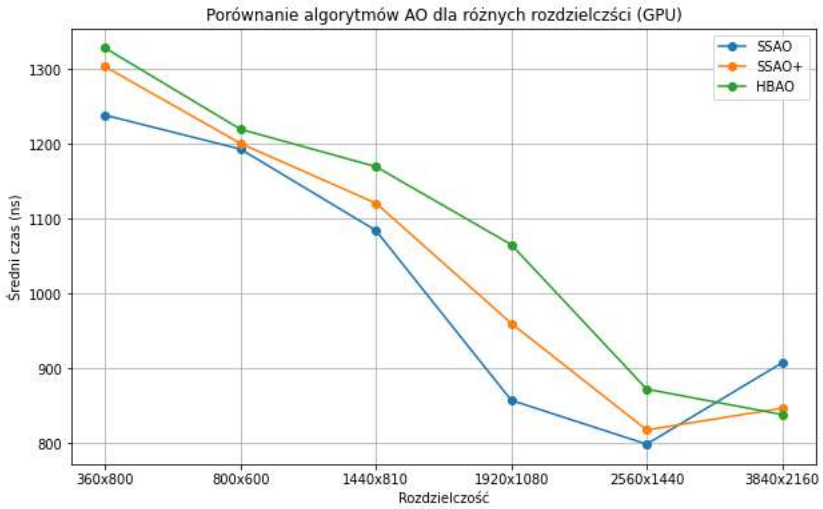
Planowany przebieg badania

Dla każdej rozdzielczości zostaną wykonane pomiary czasowe renderowania samego przebiegu AO dla poszczególnych rozdzielczości i algorytmów. Wyniki zostaną uśrednione, a następnie porównane.

Poniżej przedstawione są dwa testy, jeden dla zintegrowanej karty graficznej, a drugi dla normalnej karty graficznej. Testy są uśrednionym wynikiem dla 10^4 pomiarów w poszczególnych konfiguracjach. Wykorzystane zostały popularne rozdzielczości: *wyświetlacz urządzenia mobilnego* (360x800), *SVGA* (800x600), *1.56M3* (1440x810), *Full HD* (1920x1080), *WQHD* (2560x1440), *4K UHD-1* (3840x2160).



Rysunek 27: Histogramy średnich czasów renderowania.
Źródło: opracowanie własne.



Rysunek 28: Histogramy średnich czasów renderowania.
Źródło: opracowanie własne.

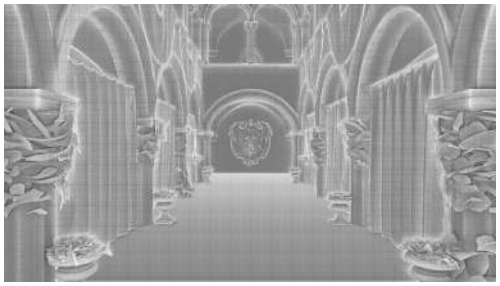
Interpretacja wyników

Wyniki wskazują na korelację między rozdzielczością, a czasem renderowania. Wraz ze wzrostem rozdzielczości, w przypadku używania GPU, można było zaobserwować zysk na czasie renderowania okluzji dla wszystkich algorytmów. Dla zintegrowanej karty graficznej spadek ten nie był taki oczywisty. O ile, Algorytmy SSAO i SSAO+ zyskały na czasie renderowania, o tyle w przypadku HBAO wartości wahają się w okolicach $1000ns$.

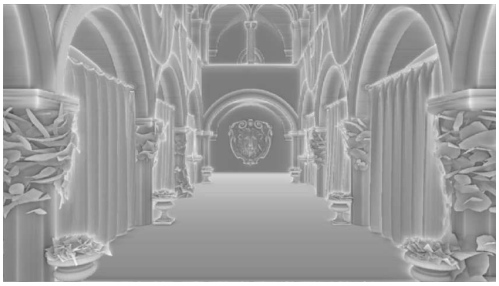
W przypadku wszystkich (poza HBAO dla GPU) wartości, wzrosły, albo drastycznie wzrosły, dla rozdzielczości *4K*. Wielokrotnie powtarzane testy, mimo, że nieznacznie różniły się od siebie, wciąż sugerują podobne wyniki.

6.3. Jakość obrazu

Wszystkie poniższe metryki zostały przetestowane dla poniższych wyrenderowanych obrazów testowych. Dla każdego algorytmu również podany jego wariant z rozmyciem/wygładzeniem (Jak mówiliśmy wcześniej rozmycie jest często stosowanym zabiegiem, żeby poprawiać)



(a) SSAO.



(b) SSAO + rozmycie.

Rysunek 29: Algorytm SSAO.
Źródło: Opracowanie własne.

(a) SSAO+.

(b) SSAO+ + rozmycie.

Rysunek 30: Algorytm SSAO+.
Źródło: Opracowanie własne.

(a) HBAO.

(b) HBAO + rozmycie.

Rysunek 31: Algorytm HBAO.
Źródło: Opracowanie własne.

Lista metryk

1. **Korelacja** (porównanie histogramów) - wyliczane są histogramy reprezentujące częstotliwość występowania wartości intensywności pikseli. Wartości znajdują się w przedziale $\langle -1, 1 \rangle$, gdzie 1 oznacza identyczność histogramów, 0 brak korelacji, -1 idealną odwrotną korelację.

2. **Template matching** (korelacja wzajemna) - jeden obraz jest nakładany na drugi aby znaleźć region najlepszego dopasowania. Wartości znajdują się w przedziale $\langle -1, 1 \rangle$, gdzie 1 oznacza dokładne dopasowanie, 0 brak dopasowania, -1 odwrotne dopasowanie.

3. **SSIM** (Structural Similarity Index Measure) - porównuje obrazy na podstawie: *jasności*, *kontrastu* i *struktury*. Wartości znajdują się w przedziale $\langle -1, 1 \rangle$, gdzie 1 oznacza idealną strukturalne podobieństwo, wartości bliskie 1 sugerują, że obrazy są strukturalnie podobne, 0 brak strukturalnego podobieństwa, -1 odwrotne strukturalne podobieństwo.

4. **MSE** (Mean Squared Error) - wyznacza średnią kwadratową różnicę pomiędzy pikselami dwóch obrazów. Najpierw wyliczana jest różnica między wartościami pikseli a następnie jest ona uśredniana. Wartości znajdują się w przedziale $\langle 0, \infty)$, gdzie 0 oznacza identyczne obrazy, a im większe wartości tym większa różnica między obrazami.

5. **PSNR** (Peak Signal-to-Noise Ratio) - wylicza jakość obrazu porównując go z oryginalnym, nieskompresowanym obrazem. Jest wyznaczany na podstawie MSE, a wyrażany w decybelach. Wartości znajdują się w przedziale $\langle 0, \infty)$, gdzie im większa wartość tym lepsza jakość. Również większa wartość sugeruje większe podobieństwo do oryginału, oraz z mniejszym zaszumieniem.

6. **Ostrość** (Laplacian Variance) - Ostrość wyliczana jest na podstawie laplasjanu, który podkreśla obszary szybkich zmian (tutaj krawędzi). Wartości znajdują się w przedziale $\langle 0, \infty)$, gdzie większe wartości oznaczają bardziej wyraziste krawędzie, a mniejsze wartości bardziej rozmyte krawędzie.

Poniższa tabela przedstawia wyliczone metryki dla obrazów testowych. Testy zostały podzielony na 4 podgrupy:

1. **Porównanie algorytmów między sobą (bez rozmycia)** - porównanie algorytmów AO bez rozmycia między sobą
2. **Porównanie algorytmów między sobą (wraz z rozmyciem)** - porównanie algorytmów AO w wariantach z rozmyciem między sobą
3. **Porównanie standardowych wariantów z ich odpowiednikami z rozmyciem** - badanie przeprowadzone na tych samych obrazach z tym samym algorytmem AO, ale w wariantach z, oraz bez rozmycia.
4. **Ostrość** - badanie zastosowane na każdym z 6-ciu wariantów.

Porównanie	**Korelacja**	**Max Value**	**SSIM**	**MSE**	**PSNR (dB)**
Porównanie algorytmów między sobą (bez rozmycia)					
SSAO vs. SSAO+	-0.154193	0.0262963	0.680231	8398.28	-
SSAO vs. HBAO	-0.0472104	0.225194	0.518602	7341.75	-
SSAO+ vs. HBAO	0.666087	0.608657	0.678986	1450.35	-
Porównanie algorytmów między sobą (wraz z rozmyciem)					
SSAO + blur vs. SSAO+ + blur	0.0393407	0.0632419	0.111966	9426.11	-
SSAO + blur vs. HBAO + blur	0.0367739	0.16769	0.129988	8230.89	-
SSAO+ + blur vs. HBAO + blur	0.855248	0.61033	0.622322	1452.19	-
Porównanie standardowych wariantów z ich odpowiednikami z rozmyciem					
SSAO vs. SSAO + blur	0.604449	0.578513	0.170252	938.644	18.4058
SSAO+ vs. SSAO+ + blur	0.68424	0.878438	0.79139	100.974	28.0887
HBAO vs. HBAO + blur	0.999638	0.992275	0.961814	30.2379	33.3253

Tabela 2: Wartości metryk dla poszczególnych wariantów AO

Algorytm	**Ostrość**
SSAO i rozmycie	36.9594
SSAO+ i rozmycie	25.2731
HBAO i rozmycie	1592.49
SSAO	19406.8
SSAO+	1319.68
HBAO	2396.86

Tabela 3: Współczynnik ostrości dla poszczególnych implementacji AO

Interpretacja wyników

- **SSAO** - Ze względu na swój charakterystyczny wygląd naturalnym jest, że korelacja i Max Value (Template matching) algorytmu SSAO ze wszystkimi pozostałymi algorytmami będzie ujemna. Pomimo faktu, że algorytm w ogóle nie przypomina algorytmu SSAO+, ma jeden z największych współczynników podobieństwa strukturalnego. Wysoki współczynnik mówi nam o bardzo dużej różnicy obrazów. SSAO cechuje się dużym szumem nawet po nałożeniu rozmycia (PSNR). SSAO posiada też najniższy współczynnik ostrości ze wszystkich algorytmów.

- **SSAO+** - W standardowym przypadku korelacja między SSAO+, a HBAO jest wysokie, ale po zastosowaniu rozmycia, algorytmy mają bardzo wysoki współczynnik korelacji. Metryki dopasowań (Max Value i SSIM) w przypadku HBAO wskazują na znaczące dopasowanie obrazów, w przeciwieństwie do SSAO (z wykluczeniem oryginalnych obrazów dla metryki SSIM). MSE wskazuje na średnie różnice między SSAO+, a HBAO. Algorytm ten cechuje się też niskim współczynnikiem zaszumienia (PSNR), ale posiada najgorsze statystyki jeśli chodzi o ostrość.

- **HBAO** - Warto zauważyć, że pomimo zastosowania rozmycia HBAO, zachowuje niemalże idealny współczynnik korelacji ze swoim oryginalnym obrazem. W większości metryk posiada najlepsze oceny w porównaniu do swojego rozmytego odpowiednika. Pomimo zastosowania rozmycia, współczynnik rozmycia pozostaje nadal wysoki, co w obu przypadkach oznacza łagodne przejścia w miejscach okluzji (realistyczny efekt).

Rozdział 7
Podsumowanie

7.1. Podsumowanie badań

W tej pracy zostały zebrane 3 algorytmy AO w celu ich analizy. Jesteśmy teraz w stanie dokonać analizy porównawczej otrzymanych wyników.

- **SSAO** - jest to najszybszy algorytm ze wszystkich 3, jednak w samym wyglądzie znacząco różni się od SSAO+ i HBAO. Chcąc wygładzić jego krawędzie wpływamy znacząco na ilość szumu jaki się pojawia na obrazie (PSNR). Algorytm ten też cechuje się AO, które jest bardzo w porównaniu do pozostałych algorytmów, co oznacza, że że rozmycie jest niemalże obligatoryjne. Jeśli chodzi o stosowanie tego algorytmu w różnych rozdzielczościach, algorytm cechował najlepszą adaptowalnością do rozdzielczości (z wyjątkiem rozdzielczości 4K).

- **SSAO+** - algorytm bardzo podobny do SSAO jeśli chodzi implementację, różni się natomiast znacząco jeśli chodzi o wygląd. Jest nieznacznie wolniejszy od SSAO, ze względu na uwzględnienie znormalizowanej półsfery. Wizualnie rezultaty bardziej przypominają okluzję otoczenia, aniżeli SSAO. Algorytm w obu kategoriach ostrości wypadł jako najmniej ostry algorytm. Posiada też relatywnie mało szumu po zastosowaniu rozmycia. W przypadku skalowalności, wypada trochę gorzej niż SSAO, i tak samo jak ten algorytm, nie słabo sobie radzi z rozdzielczością 4K.

- **HBAO** - najlepszy algorytm jeśli chodzi o metryki. HBAO pierwotnie został stworzony, żeby lepiej odwzorowywał prawdziwą okluzję. Można to chociażby zauważyć, porównując wersję z rozmyciem oraz bez - wszystkie metryki wskazują, że ma bardzo duży współczynnik podobieństwa między obydwoma wersjami (HBAO nie wymaga rozmycia do zwiększenia realizmu AO). Niestety jednak HBAO, cechuje się dłuższym czasem renderowania w porównaniu do pozostałych algorytmów, oraz bardziej skomplikowaną implementacją.

7.2. Wnioski

Analiza porównawcza wszystkich 3 algorytmów miała na celu wskazać wady i zalety każdego z algorytmów. Podsumujmy każdy z algorytmów i powiedzmy do jakich zastosowań można wykorzystać każdy z nich.

- **SSAO** - cechuje się wyjątkowym efektem bardzo mocnego zacieniana. Pomimo tego SSAO, świetnie może odnaleźć się w grach czy animacjach typu horror, w których zamierzonym celem będzie nadmierne zacienianie płaszczyzn.

- **SSAO+** - jeden z najprostszych w implementacji i dość wydajny algorytm. Cechuje się też szybkim czasem renderowania. Algorytm ten może świetnie pasować w sytuacjach w których wysoka jakość AO nie jest wymagana (np. grafika na której wykonywana jest dużo ilość postprocessingu), a proces renderowania trzeba musi być jak najbardziej zoptymalizowany.

- **HBAO** - najwydajniejszy i najbardziej realistyczny ze wszystkich analizowanych algorytmów. Nadaje się świetnie do wszystkich implementacji, tak długo jak optymalizacja renderowania nie jest problemem.

Spis rysunków

Spis tabel

Spis fragmentów kodu

Bibliography

[1] Georgios Papaioannou, Maria Lida Menexi, and Charilaos Papadopoulos. ‘‘Real-time volume-based ambient occlusion’’. In: *IEEE Transactions on Visualization and Computer Graphics* 16.5 (2010), pp. 752–762.

[2] Steven G Parker et al. ‘‘GPU ray tracing’’. In: *Communications of the ACM* 56.5 (2013), pp. 93–101.

[3] Lei Ren and Ying Song. ‘‘AOGAN: A generative adversarial network for screen space ambient occlusion’’. In: *Computational Visual Media* 8.3 (2022), pp. 483–494.

[4] Alessandro Dal Corso et al. ‘‘Interactive stable ray tracing’’. In: *Proceedings of High Performance Graphics*. 2017, pp. 1–10.

[5] Joey de Vries. *Ray-tracing in 10ms: Ambient occlusion*. URL: `http://www.alexis.breust.fr/2022-ray-tracing-in-10ms-ambient-occlusion.html`.

[6] Parminder Singh. *Learning Vulkan*. Packt Publishing Ltd, 2016.

[7] Joey de Vries. *Learn OpenGL*. URL: `https://learnopengl.com/Advanced-Lighting/SSAO`.

[8] Muhammad Mobeen Movania. *OpenGL Development Cookbook*. Packt Publishing Ltd, 2013.

[9] Andrew Astapov, Vladimir Frolov, and Vladimir Galaktionov. ‘‘Pyramid HBAO—a Scalable Horizon-based Ambient Occlusion Method’’. In: -. Vol. 31. 2021, pp. 48–63.

[10] Louis Bavoil, Miguel Sainz, and Rouslan Dimitrov. ‘‘Image-space horizon-based ambient occlusion’’. In: *ACM SIGGRAPH 2008 talks*. 2008, pp. 1–1.