Grafika 3D – Etap 3

W trzecim etapie należy użyć **shaderów**, czyli małych programów uruchamianych bezpośrednio na karcie graficznej.
Powstały dla nich trzy języki wysokiego poziomu, które wyparły niewygodny Assembler, tj.: HLSL (DirectX, XNA), GLSL (OpenGL) oraz Cg (nVidia). Wszystkie są bardzo podobne, więc opanowanie któregokolwiek z nich gwarantuje biegłość we wszystkich trzech.

HLSL:
http://msdn.microsoft.com/en-us/library/windows/desktop/bb509635(v=vs.85).aspx

GLSL:
http://www.opengl.org/documentation/glsl/

Cg:
http://http.developer.nvidia.com/Cg/Cg_language.html

Istnieją dwa podstawowe typy shaderów, którymi będziemy się zajmować na laboratorium:

- VertexShader – program cieniowania (transformacji) wierzchołków
- PixelShader (FragmentShader) – program cieniowania pikseli (lub fragmentów)

Pozostałe, takie jak: Geometry Shader, Hull (Control) Shader, Domain (Evaluation) Shader, Compute Shader powstały później z myślą o bardziej wyspecjalizowanych zadaniach i nie każda aplikacja graficzna ich używa. Dodatkowo dwa ostatnie wymagają zarówno nowszych kart graficznych i nowszej wersji API.
Na przestrzeni lat pojawiło się wiele generacji shaderów począwszy od wersji 1.0 (DirectX 8.0) do najnowszych 5.0 (DirectX 11 i OpenGL 4.1 dostępny na kartach nVidia od serii GTX 400 oraz AMD od serii Radeon HD 5xxx). Każda kolejna generacja wprowadza dodatkowe możliwości. Wszystkie zadania można wykonać przy użyciu modelu **3.0**, a niektóre nawet przy pomocy **2.0**.

Tylko w zadaniu nr 6 (Variance Shadow Mapping) wymagana jest wcześniejsza scena z parkiem. W pozostałych zadaniach scena jest w miarę dowolna, aby zaprezentować utworzony efekt – chyba, że treść zadania dokładniej określa wymagania.

W przypadku kamery, sterowanie może być dowolne, byle tylko umożliwiało spojrzenie na dowolny punkt w scenie pod dowolnym kątem w celu zaprezentowania efektu z danego zadania – może być interfejs kamery zgodny ze specyfikacją z pierwszego etapu projektu.

W tym etapie można zdobyć maksymalnie **20 punktów**. Można wybrać dowolny zestaw zadań tak, aby zgromadzić planowaną ilość punktów. Poniżej znajduje się tabela z punktacją za poszczególne zadania. Wybranie dwóch zadań, które przekraczają 20 punktów ma sens, aby zapewnić sobie margines błędu. Przypominam, że **10 punktów** mogło być przeniesione z etapu 2.

Zadanie	Punkty (max.)
1. Rybie oko	**4**
2. Szklany obiekt	**8**
3. Głębia ostrości	**8**
4. Renderowanie niefotorealistyczne	**8**
5. Spływająca tekstura	**10**
6. Variance shadow mapping	**14-20**
7. Relief Mapping	**15-20**
8. SSAO	**20**

Ostateczny termin oddania projektu: 27. stycznia 2016 r.

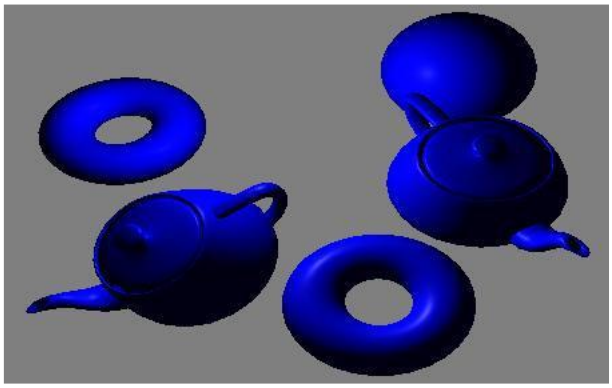
Wszelkie pytania proszę przesyłać na adres:
p.aszklar@mini.pw.edu.pl

Zadanie 1 – Rybie oko

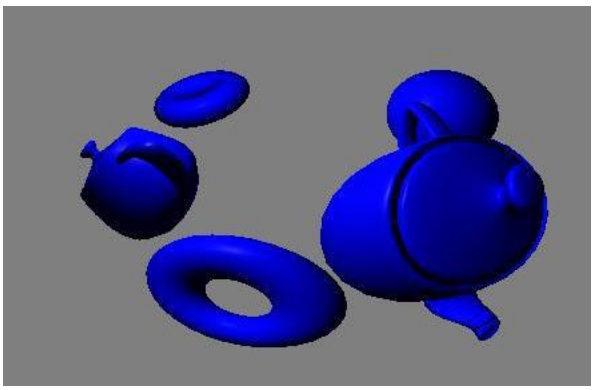
Standardowo w grafice 3D rzutowanie geometrii z trójwymiarowej przestrzeni sceny na dwuwymiarową przestrzeń ekranu odbywa się na jeden z dwóch sposobów: perspektywicznie lub równolegle. Wybór sposobu rzutowania zależy od tego jaka jest macierz projekcji DirectX/OpenGL. W fotografii używa się czasem soczewek typu rybie oko, dla których uzyskuje się projekcje niemające odpowiednika w postaci liniowego przekształcenia współrzędnych punktów przez macierz projekcji. Zadanie polega zatem na napisaniu prostego shadera wierzchołków, który będzie symulował obiektyw typu rybie oko.

Na początek należy zrealizować wyświetlanie sceny przy pomocy shaderów. Shader wierzchołków oblicza w standardowy sposób pozycję wierzchołka w scenie i na ekranie. Shader pikseli na podstawie pozycji cieniowanego punktu w scenie, pozycji kamery i pozycji światła oblicza kolor korzystając z modelu oświetlenia Phonga. Załóżmy dla uproszczenia, że cała scena oświetlona jest jednym światłem punktowym, a siatki mają przypisane tylko materiały (można pominąć nakładanie tekstur).

Shader wierzchołków dla obiektywu typu rybie oko różni się od standardowego shadera tylko jednym elementem. Standardowo pozycja wierzchołka na ekranie (oznaczmy ją przez **p**) zwracana przez shader jest pozycją wierzchołka w układzie kamery (oznaczmy ją przez **v**) pomnożoną przez macierz rzutowania perspektywicznego. Oznaczmy kąt pomiędzy **v** a kierunkiem patrzenia przez θ, a odległość **v** od osi kierunku patrzenia przez r. Zwracana pozycja wierzchołka na ekranie ma cztery współrzędne. Trzy pierwsze dzielone są zawsze przez czwartą, by dało się uzyskać efekt perspektywy. W tym przypadku chcemy, aby czwarta współrzędna była zawsze równa 1. Trzecia współrzędna służy do wykonania testu bufora głębokości i dla obiektów w zasięgu wzroku powinna należeć do przedziału [0;1]. Ustawmy ją na odległość **v** od kamery (czyli długość **v**) podzieloną przez odległość obcinania dalekich obiektów (zasięg wzroku). Dwie pierwsze współrzędne **p** otrzymamy biorąc dwie pierwsze współrzędne v i mnożąc przez $a\frac{\sin\left(\frac{\theta}{2}\right)}{r}$, gdzie a jest dowolną stałą (np. równą 4).



Rzutowanie perspektywiczne



Rybie oko

Zadanie 2 – Szklany obiekt

W tym zadaniu należy dodać jeden nietrywialny obiekt, który renderowany będzie jako bryła szkła. Efekt szkła zostanie uzyskany poprzez policzenie (dla każdego wierzchołka i dla każdego piksela) promieni załamanych i odbitych biegnących od obserwatora do cieniowanego punktu i przybliżenie śledzenia tych promieni oparte na korzystaniuy z mapy sześciennej.

Na początek potrzebna jest mapa sześcienna środowiska otaczającego wybrany obiekt. Trzeba umieścić kamerę w środku obiektu i renderować widok z tego punktu na każdą z sześciu ścian sześcianu otaczającego obiekt. Wynik renderowania należy zapisać w jednej z teksturze sześciennej dbając o prawidłowe umieszczenie renderowanych obrazów na ścianach tekstury (przez odpowiednią orientację kamery podczas renderowania każdej ze ścian). Jeśli wynik jest prawidłowy, efektem nałożenia powstałej tekstury na powierzchnię sześcianu jest ciągły obraz odpowiadający w przybliżeniu odbiciom sceny w jego ścianach. Ten etap jest identyczny jak przy realizacji techniki mapowania środowiska z zadania 22 z poprzedniego etapu projektu.

Jeśli mapowanie środowiska mamy już za sobą, to możemy przejść do napisania shaderów wierzchołków i pikseli dla szkła. Istotne element, które należy dodać do kodu to policzenie promieni załamanego i odbitego w shaderze wierzchołków i wykorzystanie tych promieni w shaderze pikseli do pobrania kolorów z mapy środowiska, policzenie ze wzoru przybliżenia współczynnika Fresnela (proporcji światła załamanego i odbitego) i na jego podstawie zmieszanie pobranych kolorów w odpowiedniej proporcji.

Zacznijmy od promieni odbitych i załamanych w shaderze wierzchołków. Dane wejściowe to wektory jednostkowe (w układzie mapy środowiska, powinien być zorientowany tak samo jak układ świata): $\boldsymbol{n}$ – wektor normalny, $\boldsymbol{v}$ – wektor od wierzchołka do obserwatora. Potrzebne są też współczynniki załamania. Ogólnie współczynnik złamania, który oznaczymy jako r, zależy od długości fali. Tutaj

przybliżymy ten efekt biorąc trzy promienie załamane dla trzech współczynników odpowiadających mniej więcej barwie czerwonej, zielonej i niebieskiej (można na przykład przyjąć współczynniki 1,15 dla kolor czerwonego, 1,14 dla zielonego i 1,13 dla niebieskiego)(. Wektor odbity dany jest wzorem:

$$2\boldsymbol{n}\langle \boldsymbol{n}, \boldsymbol{v}\rangle - \boldsymbol{v}$$

wektor załamany wzorem:

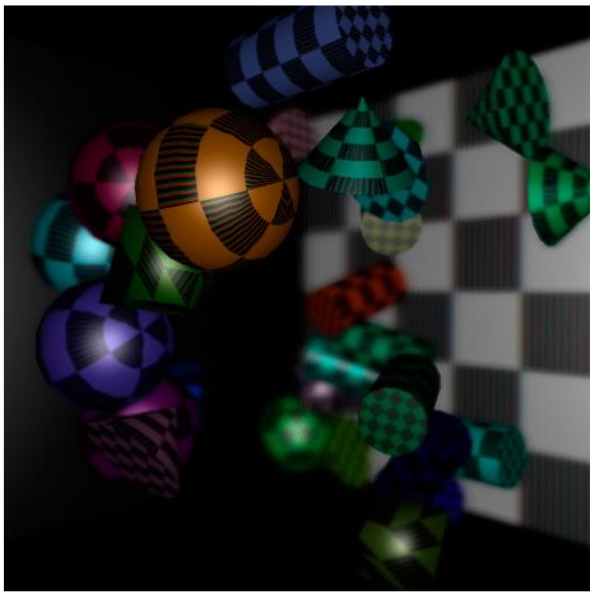
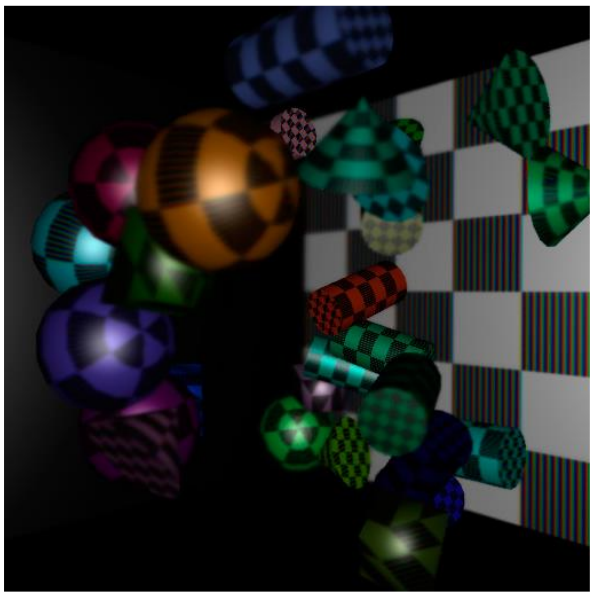
$$\left(rd - \sqrt{1 - r^2(1 - d^2)}\right)\boldsymbol{n} - r\boldsymbol{v}$$

gdzie $d = \langle \boldsymbol{n}, \boldsymbol{v}\rangle$ (iloczyn skalarny)

Przejdźmy do obliczeń koloru w shaderze pikseli. Kolor widziany przez szkło powinien zostać policzony w następujący sposób: wartość każdej ze składowych (czerwonej, zielonej i niebieskie) koloru załamania jest odpowiednią składową koloru pobranego z mapy środowiska dla jednego z trzech wektorów promieni załamanych (dla współczynnika załamania odpowiadającego danej barwie). Kolor odbicia jest wartością z mapy środowiska dla wektora promienia odbitego. Należy oba kolory zmieszać w pewnej proporcji. Zgodnie z prawami fizyki proporcja ta zależy od wartości współczynnika Fresnela. Istnieje wiele sposobów jego aproksymacji. Wystarczy skorzystać z najprostszej z nich: $(1 - d)^2$. Współczynnik wynosi 1, gdy patrzymy wprost na powierzchnię i zbliża się do 9, gdy patrzymy na powierzchnię pod kątem. Udział koloru załamania w wynikowym kolorze powinien zależeć od tego współczynnika.

[1] https://developer.nvidia.com/sites/default/files/akamai/gamedev/docs/ChromaticAberration_CEDEC_E.pdf

Zadanie 3 – Głębia ostrości

Zadanie polega na stworzeniu efektu głębi ostrości (ang. Depth of Field). Można zastosować się do sposobu wykonania opisanego poniżej (**zalecane**) lub (alternatywnie) skorzystać z innego, dostępnego w literaturze bądź Internecie źródła.

Scena (dodatkowe wymagania)

Niech na scenie istnieje miejsce, z którego widać kilka obiektów o różnej odległości od kamery.

Sposób wykonania

Efekt głębi ostrości jest symulacją rozmycia jakie powstaje przy przejściu promieni światła przez soczewkę. Tylko dla jednej wartości odległości od kamery obraz jest ostry. Oznaczmy tą odległość jako d_{focus}. Dla pozostałych wartości odległości przyjmiemy, że wielkość koła rozmycia jest proporcjonalna do wartości bezwzględnej różnicy pomiędzy tą odległością i d_{focus}. Na dwóch rysunkach powyżej widać wpływ wartości d_{focus} na renderowany obraz – po lewej ostrość ustawiona jest na pomarańczowej kuli z przodu, po prawej na czerwonym walcu z tyłu. Zadanie polega na zaimplementowaniu tego efektu tak, aby wartość d_{focus} była wybierana interaktywnie na podstawie odległości kamery w **pikselu wskazywanym przez kursor myszy**.

1. Pierwsza cześć zadania polega na wyrenderowaniu do tekstury zwyczajnego obrazu sceny. Lepszy efekt końcowy uzyska się, jeżeli obiekty będą miały nałożone tekstury.
2. Druga część polega na wyrenderowaniu do tekstury wartości odległości od kamery. Wystarczy, jeśli wynikowa tekstura będzie miała ten sam format, co tekstura z obrazem sceny, czyli RGB lub RGBA z 8 bitami na kanał. To oznacza, że można w jednym kanale zakodować 256 poziomów odległości od kamery, co powinno wystarczyć do uzyskania dobrego rezultatu.
   Należy napisać shadery wierzchołków i pikseli, które zapewnią, że efektem renderowania obiektów będzie zapisanie odległości od kamery dla poszczególnych pikseli. Shader wierzchołków jest bardzo prosty: liczy pozycję wierzchołka po zrzutowaniu na ekran oraz pozycję wierzchołka w układzie kamery. Shader pikseli natomiast liczy odległość od środka

układu współrzędnych pozycji w układzie kamery, otrzymanej z shadera wierzchołków, kodując ją w przedziale [0, 1] i zwraca na kanale czerwonym.
Do kodowania odległości od kamery potrzebne będą nam dwie wartości policzone w programie i przekazane do shadera pikseli w postaci stałych – minimalna i maksymalna odległość punktu sceny od kamery. Należy przyjąć, że scena ograniczona jest pewnym prostopadłościanem. Wtedy można łatwo policzyć odległość od kamery do najbliższego (d_{min}) i najdalszego (d_{max}) punktu takiego prostopadłościanu. Zakodowana odległość d to $\frac{d-d_{min}}{d_{max}-d_{min}}$.
Należy przy wykonywaniu tej części zadania pamiętać o wyczyszczeniu buforów koloru i głębokości, przy czym bufor koloru powinien zostać wypełniony kolorem białym, który odpowiada maksymalnej odległości od kamery.

3. Trzecia część polega na odczytaniu i zdekodowaniu odległości zapisanej w pikselu wskazywanym przez kursor myszy. Oczywiście można tą odległość wyznaczyć w dowolny inny sposób, na przykład znajdując przecięcia promienia biegnącego od kamery z obiektami sceny. Oznaczamy tą odległość przez d_0.
4. Czwarta i ostatnia część polega na zastosowaniu filtru rozmycia do obrazu wyrenderowanego w pierwszej części pierwszej, dobierając lokalną wielkość rozmycia na postawie głębokości wyrenderowanych w części drugiej.
   Na początek trzeba po raz kolejny wyczyścić bufory koloru i ewentualnie głębokości (można też wyłączyć test bufora głębokości w tej części). Należy wyrenderować pojedynczy prostokąt obejmujący cały ekran, korzystając przy tym z odpowiedniego shadera pikseli. Shader pikseli powinien otrzymać od shadera wierzchołków współrzędne danego piksela w układzie ekranu, zakładając, że lewy dolny róg ekranu to (0, 0) a prawy górny to (1, 1). W shaderze pikseli będą próbkowane obie tekstury powstałe jako wynik dwóch poprzednich części zadania. Należy do shadera pikseli przekazać w postaci stałych następujące wartości: d_0, d_{min}, d_{max}, s. Ostatni parametr s to pewna stała, która steruje średnią wielkością rozmycia (np. 0.01).
   Rozmycie polega na uśrednieniu koloru z pewnej grupy pikseli w otoczeniu danego. Pixel Shader otrzymuje (w stałych) 32 elementową tablicę wektorów przesunięć:

   (-0,18739064; 0,32457009), (-0,12072340; -0,20909910), (0,40601942; 0,34069073), (-0,15457620; 0,056261148), (0,077416219; -0,43904927), (0,053590287; 0,30392551), (-0,60999221; -0,22201918), (0,087604932; -0,073509291), (0,39825967; 0,094389275), (-0,18909506; 0,20042911), (-0,16704133; -0,55795676), (0,12249254; 0,16453603), (-0,48221043; 0,056362342), (0,13524430; -0,31353170), (-0,043583177; 0,74829435), (-0,067014404; -0,044076078), (0,35016176; -0,17585780), (0,23117460; 0,11610025), (-0,46366262; 0,30495578), (-0,010788819; -0,18523592), (0,18424729; 0,42713308), (-0,32282057; -0,037732307), (0,41358548; -0,55554169), (-0,040507469; 0,13530438), (-0,29319274; -0,31076604), (0,28428185; -0,067376055), (0,61147833; 0,047527857), (-0,12650430; 0,18444775), (-0,21871497; -0,45740339), (0,25559866; 0,25068942), (-0,80733979; 0,22473846), (0,014091305; -0,054705754)

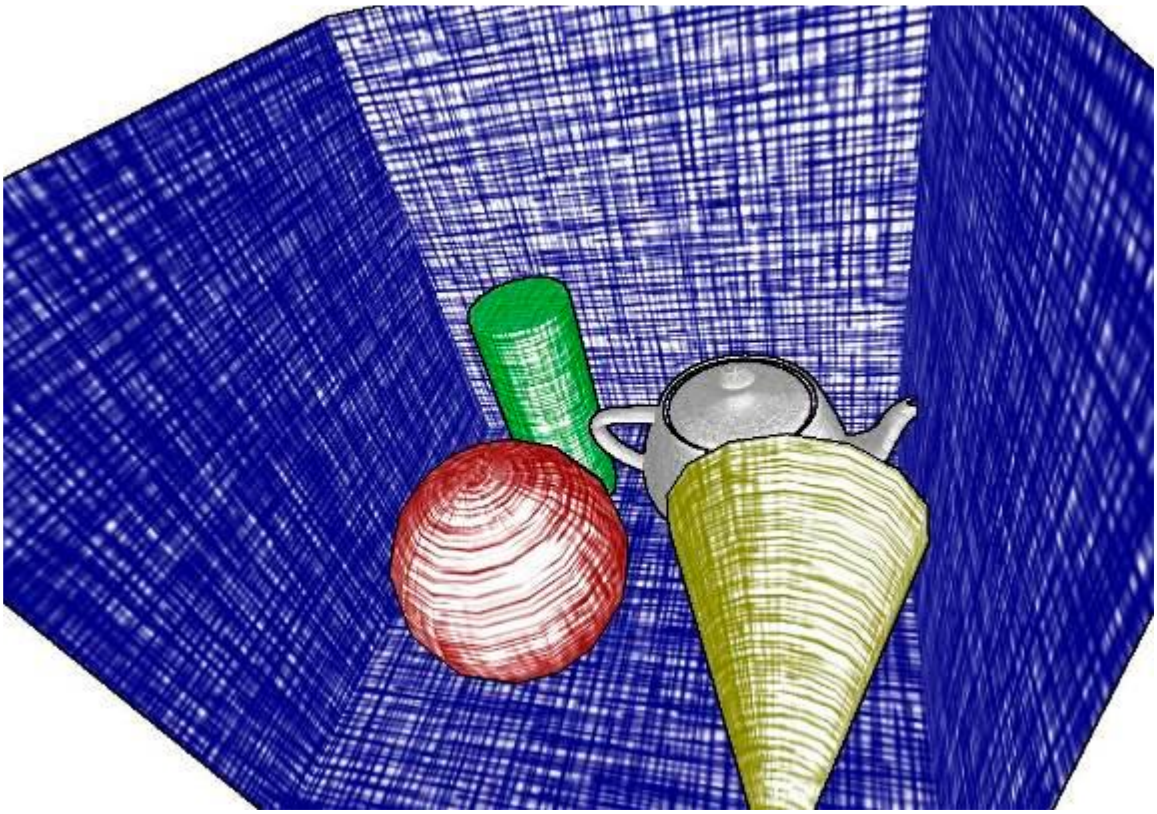
Trzeba napisać dwie funkcje pomocnicze:

- `decode` – dekodowanie odległości od kamery zapisanej w teksturze
- `coc` – liczenie promienia koła rozmycia dla danej odległości kamery, danego wzorem $|d - d_0| \cdot s$.

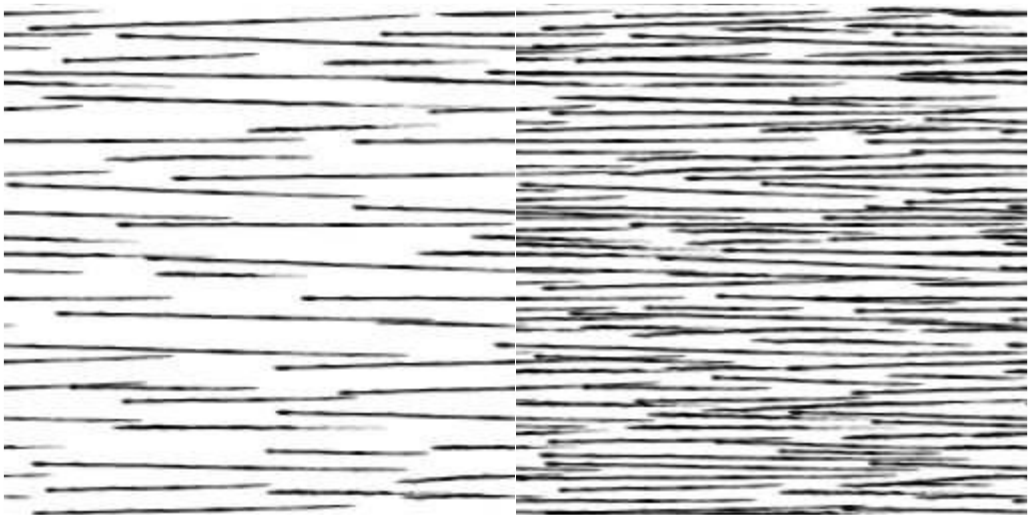
Należy policzyć coc dla danego piksela, przemnożyć przez tą wartość 32 wektory przesunięć, po czym w pętli policzyć średnią kolorów z danego piksela i 32 pikseli przesuniętych o wynikowe wektory. Dla poprawy jakości warto jeszcze policzyć coc w każdej z 32 próbek i nie brać do średniej tych spośród nich, dla których wartość coc jest co najmniej o połowę mniejsza niż dla centralnego piksela. Zapobiega to „rozlewaniu się" koloru obiektów, które powinny być ostre, na nieostre tło.

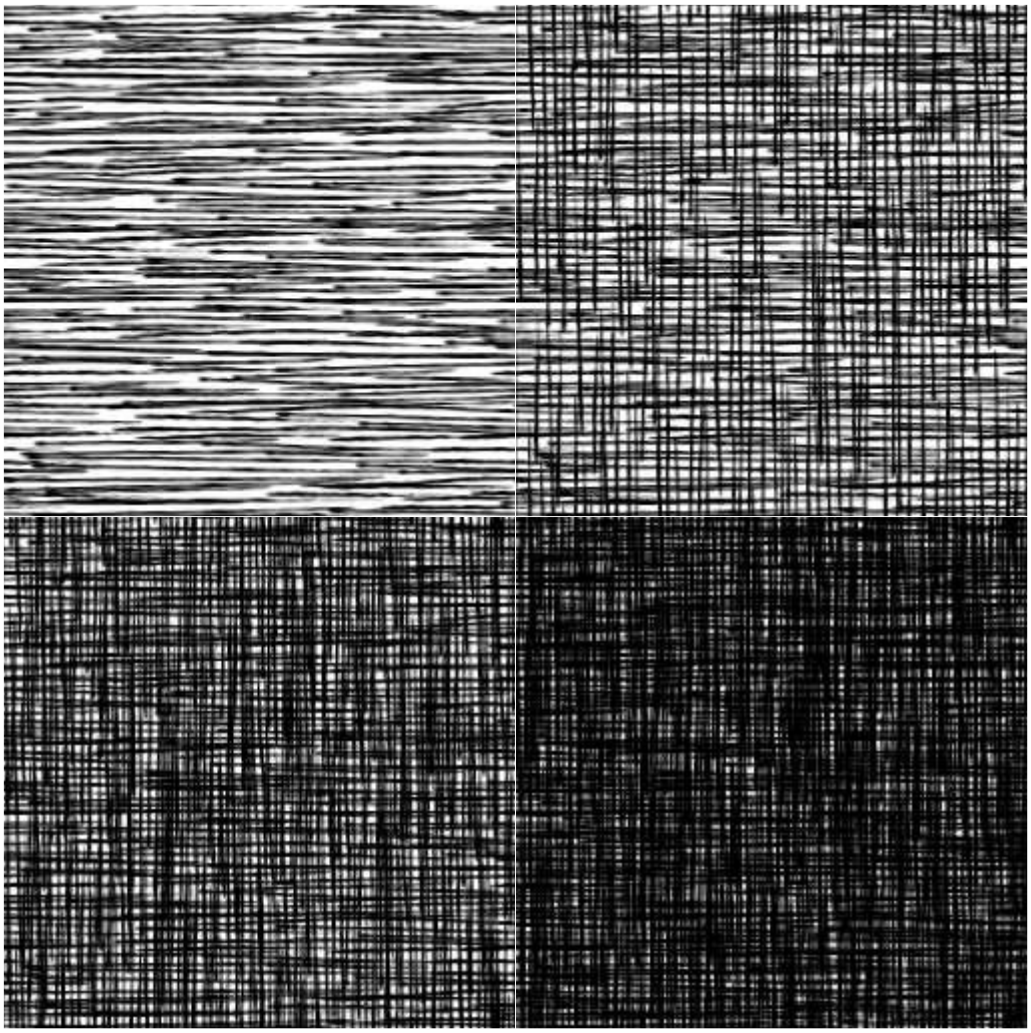
[1] http://http.developer.nvidia.com/GPUGems/gpugems_ch23.html

Zadanie 4 – Renderowanie niefotorealistyczne (ołówkowe)



Rendering niefotorealistyczny opiera się głównie na naśladowaniu sposobu rysowania obiektów przez człowieka. Zajmiemy się symulacją kreskowania. Kreskowanie w grafice czasu rzeczywistego może być naśladowane przy pomocy kilku przygotowanych uprzednio obrazów z coraz gęstszymi wzorami kresek w odcieniach sharości. Oto zestaw 6 przykładowych takich tekstur:





Najlepiej jest po wczytaniu do programu zakodować je w dwóch teksturach RGB tak, żeby każdy kanał koloru każdej tekstury przechowywał jeden ze wzorów kreskowania.

Podstawą symulacji kreskowania będzie pobieranie wyników (prawie) zwyczajnie liczonej intensywności oświetlenia, a następnie wykorzystanie jej do wybrania dwóch najbardziej odpowiednich wzorów kresek i zmieszanie pobranych z nich kolorów.

Przygotowanie sceny

Ponieważ kolor dla powierzchni obiektów musi zostać pobrany z tekstur kresek, wszystkie siatki w scenie cieniowane przy pomocy tego efektu muszą posiadać ustalone współrzędne tekstury 2D.

Do uzyskania efektu potrzebne będzie napisanie zarówno shadera wierzchołków jak i pikseli.

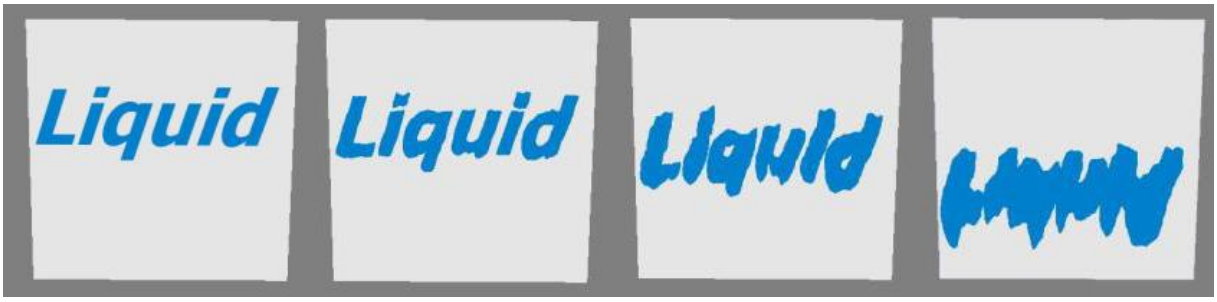
Zajmijmy się najpierw shaderem wierzchołków. Powinien on obliczyć:

1. Pozycję wierzchołka przekształconą do układu kamery i pomnożoną przez macierz rzutowania perspektywicznego.
2. Współrzędne tekstury dla wierzchołka (skopiowane z danych pobranych z siatki).
3. Dwa trzyelementowe wektory, które będą w sumie przechowywać 6 wag. Wagi zostaną użyte do wybrania spośród sześciu kanałów dwóch tekstur kresek pary kanałów z odpowiednimi wzorami, a następnie do interpolacji liniowej pomiędzy pobranymi z nich kolorami.

Najważniejsza część obliczeń to wyliczenia oświetlenia i wektorów wag. Przyjmijmy dla uproszczenia, że jest tylko jedno światło kierunkowe w scenie i jego kierunek pokrywa się z osią kamery. Przyjmijmy $5.0 \cdot \langle N, L \rangle^4$ jako intensywność oświetlenia, gdzie $\langle N, L \rangle$ oznacza iloczyn skalarny normalnej i kierunku światła, zaś współczynnik 5.0 pomaga łatwiej wybrać na dalszym etapie jeden z pięciu przypadków. Każdy z nich odpowiada interpolacji pomiędzy parą kolejnych wzorów kresek. Część całkowita intensywności wyznacza odpowiednią parę, a część ułamkowa współczynnik przy interpolacji liniowej. Mamy dwa 3-wektory z wagami i dwie 3-kanałowe tekstury RGB. Chcemy, żeby suma iloczynów skalarnych wektorów wag przez pobrane z tekstur wektory RGB dawała ostateczną jasność piksela na ekranie. Dlatego wszystkie elementy wektorów wag poza dwoma ustawiamy na zero.

Jeśli wagi są policzone w shaderze wierzchołków, to napisanie shadera pikseli jest proste. Wystarczy pobrać kolory obu tekstur z wzorami kresek oraz policzyć sumę iloczynów skalarnych kolorów i wektorów wag. Tak otrzymana intensywność posłuży wreszcie jako współczynnik przy interpolacji pomiędzy ustalonym kolorem materiału (stała Pixel Shadera) oraz bielą. Kolor materiału nie powinien być zbyt jasny, ponieważ pełni on funkcję koloru nieoświetlonej powierzchni.

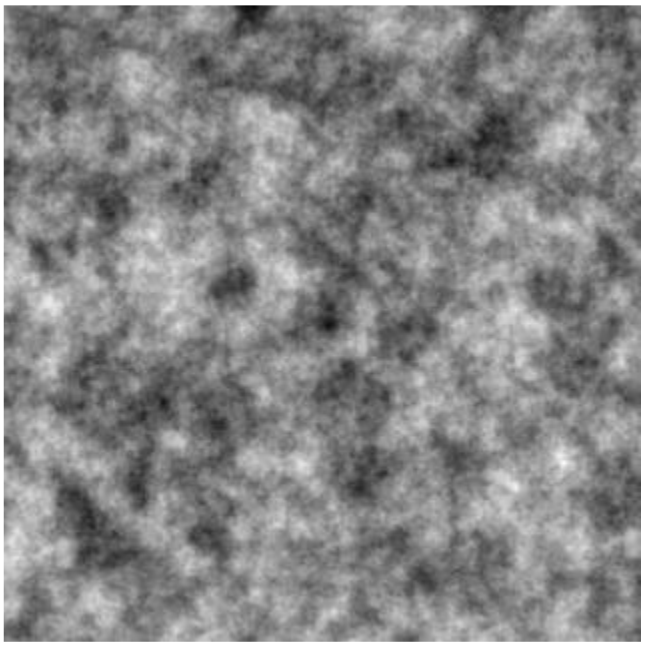
Zadanie 5 – Spływająca tekstura



W tym zadaniu na jeden z obiektów (najlepiej na jedną ścianę prostopadłościanu) należy nałożyć teksturę, której zawartość liczona będzie w kolejnych klatkach animacji w piksel shaderze. Uzyskany efekt symulować będzie farbę ściekającą po ścianie.

Potrzebne będą cztery tekstury, z czego trzy pierwsze muszą mieć te same rozmiary:

- czarno-biały obraz załadowany z pliku (gdzie biały kolor oznacza piksele z farbą, kolor czarny oznacza tło – np. biały napis na czarnym tle)
- dwie tekstury przechowujące dwie ostatnie klatki animacji: tekstura z obrazem z poprzedniej klatki będzie stanowić dane wejściowe dla obliczeń w aktualnej klatce animacji; wynik zapisany będzie w drugiej z tych tekstur; w kolejnej klatce tekstury zamienią się rolami)
- tekstura z sumą kilku oktaw szumu Perlina, np. jak na obrazie poniżej:



Animacja ściekania farby będzie polegała na tym, że w każdej kolejnej klatce nowy kolor piksela będzie dany w postaci pewnej formuły na podstawie kolorów jego sąsiadów z klatki poprzedniej. Należy użyć shadera pikseli w taki sposób, żeby wywołany on został raz dla każdego piksela wynikowej tekstury, mając do dyspozycji współrzędne odpowiadającego mu teksela tekstury źródłowej:

a) tekstura wynikowa ustawiana jest jako rendertarget (renderowanie odbywa się do tekstury)
b) rozmiary okna widoku (viewport) odpowiadać muszą wymiarom tekstury
c) renderowany jest pojedynczy prostokąt, który rozciągnięty jest na całe okno widoku. Współrzędne jego przeciwległych wierzchołków: $[-1 \quad -1 \quad 0{,}5 \quad 1]$ i $[1 \quad 1 \quad 0{,}5 \quad 1]$ (podane współrzędne są już po rzutowani, tak jak powinny być zwrócone z shadera wierzchołków). Wierzchołkom należy przypisać też odpowiadające im współrzędne tekstury, tak aby jej teksele nałożyły się idealnie na piksele tekstury wynikowej (współrzędne z zakresu $[0 \quad 0] - [1 \quad 1]$ – przy czym zajść może potrzeba przesunięcia ich o $0{,}5/size$, gdzie $size$ to rozmiar tekstury w pikselach; uważać trzeba też, by nie obrócić obrazu do góry nogami)
d) Wyłączyć należy filtrowanie liniowe tekstur (czyli ustalić filtrowanie tekstury tak, by zwracany był kolor najbliższego teksela)

W piksel shaderze symulacji współrzędne tekstury sąsiadów można uzyskać przesuwając współrzędne tekstury uzyskane na wejściu o $1{,}0/size$ (, gdzie $size$ to rozmiar tekstury w pikselach). Fragment kodu piksel shadera odpowiadający za symulację w języku HLSL wygląda następująco:

```
float d = tex2D(PerlinSampler, texcoord+float2(8.0*time,time)).r*2.0-1.0;
float e = tex2D(PerlinSampler, texcoord).r;
float r;
float a = 100.0*dt;
float w = 0.4;
if (d < 0.0)
{
     d = -d;
     r = a*e*((1.0-d)*h.w+d*h.y)+(1.0)*h.x-w*a*e;
}
else
     r = a*e*((1.0-d)*h.w+d*h.z)+(1.0)*h.x-w*a*e;
```

W tym przykładzie `PerlinSampler` jest obiektem próbkowania tekstury związanym z teksturą szumu Perlina, `time` oznacza czas animacji w sekundach, `dt` to czas jaki upłynął od ostatniej klatki animacji w sekundach, `h.x` to wartość odpowiadająca aktualnemu pikselowi pobrana z tekstury źródłowej, `h.y`, `h.z`, `h.w` to wartości sąsiadujących pikseli odpowiednio: na lewo i do góry od aktualnego piksela, do góry i na prawo oraz bezpośrednio do góry.

W programie powinna istnieć możliwość zresetowania animacji po naciśnięciu pewnego klawisza. Restart polega na przekopiowaniu oryginalnej czarno-białej tekstury ze wzorem do jednej z dwóch tekstur używanych w symulacji.

Niech podaczas rysowania ściany z nałożoną teksturą z aktualnym efektem animacji, kolor piksela będzie wynikiem działania prostego piksel shadera z interpolacją liniową pomiędzy dwoma dowolnymi ustalonymi kolorami na podstawie jasności teksela pobranej z tekstury aktualnej klatki animacji. W przeciwieństwie do symulacji, podczas rysowania ściany należy próbkować teksturę z włączonym filtrowaniem trójliniowym.

Zadanie 6 – Variance Shadow Mapping

Zadanie polega na zaimplementowaniu metody mapy cieni (Shadow Map) na scenie z poprzednich części projektu. Jest to jedyne zadanie w tym etapie wymagające sceny stworzonej w poprzednich etapach. Można założyć, że wyznaczany cień pochodzi wyłącznie od któregoś światła punktowego lub reflektorowego (jedno do wyboru).

Teoria związana z techniką map cienia jest szczegółowo omówiona na wykładzie. Większość algorytmów związanych z wyświetlaniem cieni cierpi na liczne artefakty, stąd powstała duża liczba przeróżnych usprawnień i modyfikacji podstawowego algorytmu Shadow Mapping (SM). Implementacja cieni w komercyjnych silnikach wiąże się z mozolnym dopracowywaniem parametrów – bo pomimo, że algorytm jest uniwersalny i w teorii niezależny od sceny, to konkretne właściwości świateł i geometrii ukrywają lub eksponują pewne niedociągnięcia.

Do podstawowej metody dochodzi zastosowane usprawnienie (*Variance*):

1. **Podczas tworzenia mapy cienia**
   Zapisujemy nie tylko wartości głębokości Z, ale również jej kwadrat Z^2 (ważna jest tutaj odpowiednia **32-bitowa** precyzja).
   Definiujemy dwa momenty liniowe (funkcji/rozkładu/sygnału):
   Niech głębokość Z oznacz pierwszy moment liniowy:
   $$M_1 = Z$$
   Drugi moment to
   $$M_2 = Z^2 + bias$$
   $$bias = 0.25(dFdx(Z)^2 + dFdy(Z)^2)$$
   Funkcje *dFdx* oraz *dFdy* to funkcje w shaderze pikseli obliczające pochodną cząstkową po x oraz y.
2. **Podczas wykorzystania mapy cienia**
   Odtwarzamy średnią oraz wariancję rozkładu głębokości na pewnym wybranym regionie wokół piksela, dla którego rozpatrujemy czy jest w cieniu czy nie (można zacząć od regionu 1×1).
   $$\mu = E(x) = M_1$$
   $$\sigma^2 = E(x^2) - E(x)^2 = M_2 - M_1^2$$
   Następnie obliczamy prawdopodobieństwo, że piksel jest w cieniu korzystając z nierówności Czebyszewa:
   $$P(x \geq t) \leq p_{max}(t) = \frac{\sigma^2}{\sigma^2 + (t - \mu)^2}$$
   Otrzymane prawdopodobieństwo to stopień zacienienia – dzięki temu cień staje się miękki.

Uwaga

- Za zadanie wykonane zgodnie z powyższym opisem można uzyskać **14 punktów**.
- Brak ulepszenia VMS (zaimplementowanie tylko zwykłego SM) oznacza obniżoną punktację.
- Można uzyskać **20 punktów** za zadanie poprzez dodatkowe zaprogramowanie innego znanego usprawnienia algorytmu Shadow Mapping. Aby uzyskać 20 punktów oba usprawnienia muszą działać w jednym programie, na tej samej scenie – nie ma jednak wymogu, by były włączone jednocześnie
- Sugerowane ulepszenie to CMS (Cascaded Shadow Maps) lub PCF (Percentage Closer Filtering).

[1] http://msdn.microsoft.com/en-us/library/windows/desktop/ee416307(v=vs.85).aspx

Zadanie 7 – Relief mapping

Relief Mapping to jedna z wielu technik mapowania wypukłości. Wspólną ideą tych metod jest zwiększenie szczegółowości obiektów bez zwiększania liczby trójkątów siatki (bez ingerencji w faktyczną geometrię). Efekt uzyskiwany jest za pomocą różnych optycznych właściwości światła i postrzegania obrazu przez człowieka.

Do podstawowych algorytmów mapowania nierówności należą:

- Normal Mapping, Bump Mapping (mapowanie normalnych, wybojów).
- Parallax Mapping

Mapowanie normalnych (wybojów) wymaga przygotowania dodatkowej tekstury przechowującej wektory normalne dla każdego piksela tekstury z kolorem w pewnym spoczynkowym układzie odniesienia. Metoda polega na zastąpieniu wektorów normalnych z wierzchołków tymi z tekstury (mapy normalnych) i obliczaniu oświetlenia dla każdego piksela. Dzięki temu można odwzorować nawet niewielkie nieregularności powierzchni – kształty, które zanikłyby w przypadku interpolacji normalnych z wierzchołków. Wadą metody jest fakt, że jakość efektu zależy od kąta patrzenia – im większy kąt patrzenia, tym efekt jest gorszy i widać płaski charakter powierzchni. Podobnie, aby efekt był zadowalający, wypukłości nie powinny być zbyt duże.

Parallax Mapping jest pewnym rozszerzeniem metody mapowania normalnych i uznawany jest za krok naprzód. Wykorzystuje przesunięcia współrzędnych tekstury w zależności od kąta patrzenia symulując efekt „paralaksy" – tj. niezgodności różnych obrazów tego samego obiektu obserwowanego pod różnym kątem. Metoda nie dla każdego obiektu daje dobre rezultaty – najlepiej sprawdza się dla kamiennych ścian.

Relief Mapping jest uznawany za kolejny krok naprzód w porównaniu do poprzednio omówionych technik.

Opis zadania

Założenia sceny:

- Wystarczy jeden obiekt do zaprezentowania metody Relief Mapping np. fragment kamiennej ściany.
- Można założyć całkowity brak przezroczystości na scenie, aby ułatwić implementację.

Należy przygotować trzy współgrające ze sobą mapy:

- koloru (a więc zwykłą teksturę)
- wysokości (jak wysoki punkt dany piksel opisuje)
- normalnych (do wyznaczenia przestrzeni stycznej)

Wysokości wygodnie jest zapisać w kanale Alpha tekstury RGBA, ponieważ obiekt jest nieprzezroczysty.

Należy przygotować mapę normalnych oraz pasującą do tekstury mapę głębokości (wysokości) tak, aby wartości zawierały się w przedziale od 0.0 (od zewnętrznej strony) do 1.0 (po stronie wewnętrznej).

Podpowiedź – mapę można stworzyć na różne sposoby:

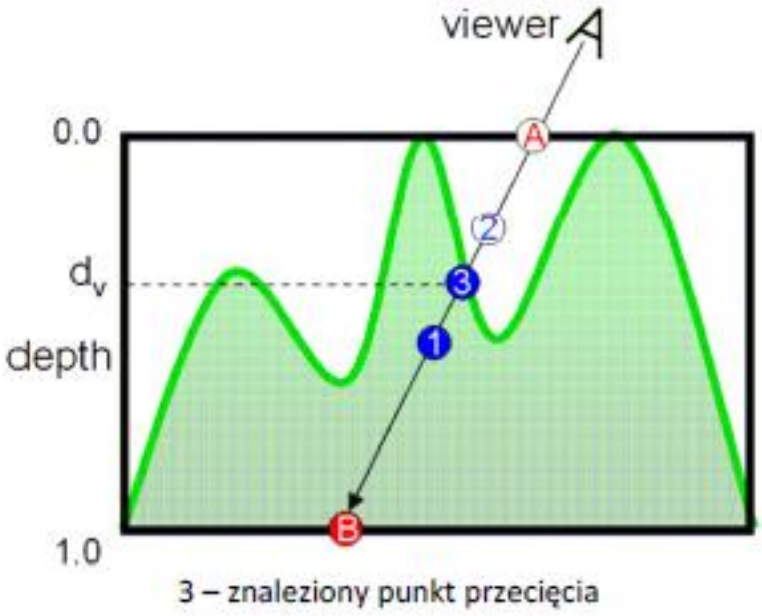
- wygenerować bezpośrednio z tekstury, polecanym programem do tego jest CrazyBump (dostępny trial)
- przygotować teksturę z prostymi kształtami takimi jak koła, trójkąty czy kwadraty, dla których łatwo jest wyliczyć wysokość ze wzorów
- użyć bardziej szczegółowej siatki hi-poly jako bazy do pobrania wysokości i użycia ich w modelu low-poly
- pobrać z Internetu już gotowe zestawy tekstur

Schemat działania: (szczegółowy opis po angielsku znajduje się w linku źródłowym)

- Oblicz kierunek patrzenia VD jako wektor od obserwatora do pozycji 3D punktu siatki
- Przekształć VD do układu związanego z zapisanymi wektorami normalnymi (a więc do przestrzeni stycznej zdefiniowanej przez wektor normalny, binormalny i styczny). Tak przekształcony wektor nazwijmy VD'
- Niech:
    - A – punkt, w którym VD' przecina płaszczyznę o głębokości 0.0 = bliżej obserwatora (patrz Rysunek)
    - B – punkt, w którym VD' przecina płaszczyznę o głębokości 1.0 = dalej obserwatora (patrz Rysunek)

  Należy wyznaczyć punkty A i B oraz obliczyć odpowiadające im współrzędne tekstury (s,t) i (u,v). Następnie przy pomocy przeszukiwania binarnego, liniowego (z pewnym krokiem) na linii AB lub połączenia obu metod wyznaczyć z pewnym przybliżeniem (możliwie dobrym) punkt przecięcia z mapą wysokości.
- Na koniec należy dokonać teksturowania i oświetlenia używając współrzędnych tekstury wyznaczonego punktu przecięcia. Należy zamodelować przynajmniej jedno światło, aby efekt był widoczny.

Rysunek:



3 – znaleziony punkt przecięcia

Za powyższe zadanie można uzyskać maksymalnie **15 punktów**. Wynik można zwiększyć poprzez zaprogramowanie wybranych ulepszeń:

- dopracowanie metody, aby:
  - działała dla obiektów półprzezroczystych
  - na scenie znajdowały się 3 obiekty z Relief Mapping, każdy z inną teksturą i niech podczas rysowania każdego obiektu włączone będzie inne światło – odpowiednio kierunkowe, punktowe i reflektorowe

  +3 punkty
- Uwzględnienie współczynnika odbicia zwierciadlanego (specular) **+1 punkt**
- Zaimplementowanie prostego bump-mappingu na scenie dla tego samego obiektu w celu porównania jakości obu metod. **+1 punkt**

[1] http://www.inf.ufrgs.br/~oliveira/pubs_files/Policarpo_Oliveira_Comba_RTRM_I3D_2005.pdf
[2] http://www.crazybump.com/

Zadanie 8 – Screen Space Ambient Occlusion

Technika zwana *Ambient Occlusion* (*AO*) została opracowana, aby symulować efekt globalnego oświetlenia poprzez obliczenie stopnia zaciemnienia powierzchni przez obiekty, które znajdują się w sąsiedztwie i ją przysłaniają.

W przypadku oświetlenia lokalnego (np. standardowe oświetlenie dla świateł punktowych, kierunkowych czy reflektorowych w OpenGL/DirectX) bierzemy pod uwagę jedynie interakcję światło-powierzchnia.

Dzięki stosowaniu Ambient Occlusion otrzymujemy dodatkowe informację o sile pokolorowania – jasności danego miejsca – która jest wykorzystywana w połączeniu z normalną procedurą dla danego światła lub samodzielnie. W drugim przypadku otrzymujemy obraz w kolorach szarości, który prezentuje efekt wpływu AO:

Są dwie główne wary podejścia Ambient Occlusion:

- Należy przechowywać całą scenę w takiej postaci, aby móc obliczać przecięcia z rzucanym (z każdego punktu powierzchni) promieniami. Stąd powstają problemy:
    - z dynamicznymi scenami
    - wyborem sposobu reprezentacji
    - złożoności pamięciowej
- Ogromna złożoność obliczeniowa, która dodatkowo zależy od złożoności sceny. Prawdziwe Ambient Occlusion to wciąż efekt stosowany głównie w programach do modelowania 3D dla statycznych scen – obliczane jest w preprocessingu. Dla skomplikowanych scen AO nie osiąga interaktywnych ilości klatek na sekundę.

SSAO

W SSAO (ang. *Screen Space Ambient Occlusion*) obliczenia przeniesione są do przestrzeni ekranu. Jest to modyfikacja, która kosztem jakości, niweluje wady podejścia AO.

Zalety SSAO:

- Niezależność od zawartości sceny.
  Raz zaimplementowane w silniku graficznym automatycznie działa dla dowolnej sceny – nie ma problemów z dostosowaniem odpowiedniej reprezentacji pod konkretne obiekty. Ta technika działa również dla w pełni dynamicznych scen.
- Niezależność od złożoności sceny. Wydajność SSAO jest stała dla ustalonej rozdzielczości ekranu.
- Znaczne przyspieszenie obliczeń.
- Może być zaimplementowane w całości na GPU.

Wady SSAO:

- Uproszczona jakość obrazu. SSAO to mocno aproksymacyjny efekt, zależny nie tylko od geometrii sceny, ale również od bieżącego widoku.
- Często widoczne zaszumienia obrazu – często wymaga stosowania filtrów rozmycia, wygładzania i odszumiania.

Sposób wykonania

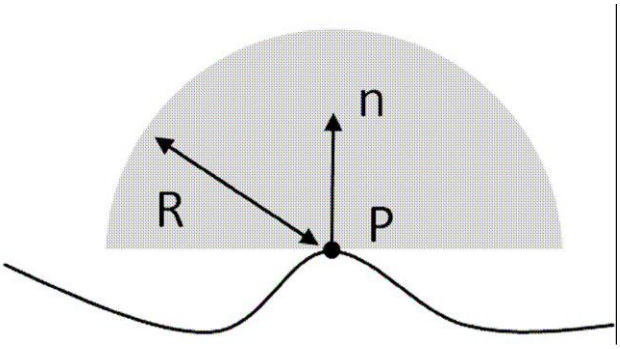
W zadaniu wystarczy wykonać bardzo podstawową wersję algorytmu (w komercyjnych zastosowaniach, w celu uzyskania wysokiej jakości efektu SSAO, stosuje się wiele złożonych usprawnień lub algorytmów pochodnych).

Scenę renderujemy wieloprzebiegowo – na nowszych kartach wystarczą dwa przebiegi. Praktycznie wszystkie czynności związane z algorytmem należy wykonać w shaderze pikseli. Shader wierzchołków wykonuje jedynie przekształcenia widoku i projekcji i przekazuje informacje do shadera pikseli.

1. Renderowanie do tekstury
   W pierwszym przebiegu zapisujemy do tekstury pewne właściwości powierzchni dla tych punktów, które zostały zrzutowane na piksele ekranu. Dzięki temu nie będziemy obliczać stopnia zasłaniania w miejscach, które i tak nie są widoczne. Zapisujemy dane sceny, do których będziemy mieli dostęp na poziomie pikseli:
   - głębia – czyli odległość od kamery (wysokość na mapie wysokości)
   - kolor rozproszony (diffuse) powierzchni
   - wektor normalny i styczny

   Na nielicznych już, starych kartach graficznych nie da się skorzystać z tzw. MTR (Multiple Render Targets), dzięki którym można zapisywać do kilku tekstur w trakcie pojedynczego rysowania sceny. W takim przypadku należy wypełnić każdą z tekstur w oddzielnych przebiegach.
2. Algorytm obliczania stopnia zasłonięcia.
   Dzięki zapisanym teksturom mamy obraz do wyświetlenia (tekstura diffuse) wraz z pewnymi dodatkowymi informacjami o scenie [głębokość, wektor normalny, styczny], które normalnie nie są dostępne już w przestrzeni ekranu. Adresowanie wszystkich tekstur jest zgodne, więc dla każdego piksela mamy dostęp do tych informacji pod tym samym adresem (u, v).

Obliczając AO będziemy traktowali obszar jako siatkę prostokątów o bokach dX oraz dY. Prostokąty definiują nam precyzję algorytmu – każdy prostokąt ma przypisany ten sam współczynnik AO. W szczególnym przypadku dla dX = 1 oraz dY = 1 prostokąty pokrywają się z pikselami. Czyli innymi słowy prostokąty to piksele zgrupowane w większe regiony w celu redukcji obliczeń.
Niech wartości dX, dY będą zmiennymi łatwymi do modyfikacji w kodzie (oczywiści mogą być również modyfikowalne z poziomu UI). Mogą przyjmować wartości {1, 2, 4, 8}.
Mapę z zapisaną głębokością traktujemy jako standardową mapę wysokości. **Uwaga! Tutaj należy poeksperymentować z normalizacją tych głębokości do ustalonego przedziału.**
Iterujemy po dX oraz dY i dla każdego piksela, który staje się reprezentantem swojego regionu, obliczamy współczynnik AO. Przykładamy półsferę, zorientowaną zgodnie z zapisanym w punkcie wektorem normalnym, której podstawa zawiera wektor styczny:



R oznacza promień półsfery wyrażony w liczbie pikseli – jest to względny, dyskretny sposób ustalania jej wymiaru. Niech wartość R będzie łatwo modyfikowalna w kodzie.
Teraz rzucamy N promieni w losowych kierunkach wewnątrz sfery zgodnie z rozkładem normalnym lub Poissona. Niech N będzie możliwie maksymalną ilością, na jaką pozwala wydajność – można zacząć od małej ilości próbek i stopniowo ją zwiększać.
Śledzenie promieni jest dyskretne – w przestrzeni obrazu 2D. Algorytmy dyskretnego śledzenia promieni nazywają się **ray-matching** zamiast **ray-casting**. Przecięcie promienia jest liczone bardzo prosto – dla wszystkich sąsiednich pikseli mamy wysokość w odpowiadającym punkcie, więc wystarczy jedynie porównać „wysokość" promienia z wysokością w punkcie, aby odpowiedzieć na pytanie, czy weszliśmy do geometrii czy nie.
Niech obliczona wartość AO będzie **ułamkiem liczby promieni, które trafiły w powierzchnię do ilości wszystkich promieni rzuconych**:

```
AoFactor = hitCount/raysTotal
```

Na potrzeby zadania wystarczy powyższa metoda, jednak można śmiało poeksperymentować z bardziej zaawansowanym modelem, np.:

- uwzględniającym odległość pierwszego trafienia promienia.
- obliczającym objętość półsfery, która jest zasłonięta przez geometrię – np. przy pomocy metody całkowania Monte Carlo (losowe punkty).
- wprowadzenie heurystyki pod konkretną scenę.

3. Wygładzanie, odszumianie i wyświetlanie
   Wygenerowaną teksturę z zapisanymi wartościami AO należy wygładzić – w tym celu należy wykorzystać filtr graficzny dla obrazu 2D rozmycia lub odszumiania. Można zastosować złożenie zwykłego rozmycia Gaussowskiego oddzielnie dla kierunków X oraz Y.
   Polecam również metodę opisaną w poniższej prezentacji [4].

Na sam koniec należy wyrenderować scenę jako pełnoekranowy prostokąt. Wartość diffuse dla każdego piksela, która normalni służyłaby pokolorowaniu sceny jest zaciemniana proporcjonalnie do współczynnika AO.

Uwaga! Rozdzielczość ekranu może być niewielka, ale co najmniej 800×600.

[1] http://developer.download.nvidia.com/presentations/2008/GDC/GDC08_Ambient_Occlusion.pdf
[2] http://developer.download.nvidia.com/SDK/10.5/direct3d/Source/ScreenSpaceAO/doc/ScreenSpaceAO.pdf
[3] http://www.drobot.org/pub/GCDC_SSAO_RP_29_08.pdf
[4] http://people.csail.mit.edu/sparis/bf_course/slides/03_definition_bf.pdf