Grafika 3D – Etap 3

W trzecim etapie należy użyć **shaderów**, czyli małych programów uruchamianych bezpośrednio na karcie graficznej.
Powstały dla nich dwa języki wysokiego poziomu, które wyparły niewygodny Assembler, tj.: HLSL (DirectX, XNA) oraz GLSL (OpenGL).

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

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

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

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

Pozostałe, takie jak: Geometry Shader, Hull (Control) Shader, Domain (Evaluation) Shader, Compute Shader powstały niedawno 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. Każda kolejna generacja wprowadza dodatkowe możliwości. Wszystkie zadania można wykonać przy użyciu modelu **3.0**, a niektóre nawet przy pomocy **2.0**.

We wszystkich 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. Snop Światła	**4**
2. Uogólnienie modelu oświetlenia Phonga	**6**
3. Renderowanie niefotorealistyczne	**8**
4. Głębia ostrości	**8**
5. Spływająca tekstura	**10**
6. HDR	**12-20**
7. Variance Shadow Mapping	**14-20**
8. Deferred Shading	**15-20**

Ostateczny termin oddania projektu: 24. stycznia 2019 r.

Zadanie 1 – Snop światła

Efekt snopu światła na renderowanym obrazie można uzyskać licząc punkty przecięcia promienia biegnącego od kamery przez cieniowany punkt z walcem. Walec dany jest za pomocą promienia, punktu na osi walca oraz kierunku osi walca, zapisanych w stałych shaderów. Mając pozycję rysowanego punktu oraz pozycję kamery (obie w układzie światał), w piksel shaderze można znaleźć równanie promienia biegnącego od obserwatora.

Istotne jest jaką odległość pokona ten promień wewnątrz walca, zanim dotrze do cieniowanego piksela. Przypuśćmy, że równanie promienia ma postać $\boldsymbol{p} = \boldsymbol{p}_0 + \boldsymbol{r} \cdot t$, gdzie $\boldsymbol{r}$ jest wektorem jednostkowym, a $\boldsymbol{p}_0$ pozycją kamery. Licząc przecięcia promienia z walcem znajdujemy wartości t_0, t_1 (t_1>t_0) parametru t, odpowiadające punktom wejścia i wyjścia promienia z walca (jeśli nie ma przecięcia – promień biegnie równolegle – można przyjąć obie wartości równe zero). Łatwo możemy też znaleźć wartość t_2, dla której promień dociera do cieniowanego piksela. Odległość jaką promień przebiegnie wewnątrz walca na drodze do niego od kamery wynosi:

$$d = \max(0, \min(t_2, t_1) - \max(t_0, 0))$$

Ostateczna modyfikacja koloru liczonego w piksel shaderze powinna wyglądać:

```
float c = 1.0 - exp(-d);
float3 result = (1.0 - c) * color + c * float3(1.0, 1.0, 1.0);
```

, gdzie `color` to wartość koloru liczona normalnie w podstawowej wersji shadera pikseli.

Zadanie 2 – Uogólniony Model Oświetlenia Phonga

Zadanie dotyczy cieniowania obiektu z nałożoną teksturą dla pojedynczego punktowego źródła światła i nietypowego modelu oświetlenia.

W modelu Phonga kolor cieniowanego piksela liczony jest ze wzoru

$$\boldsymbol{c} = \boldsymbol{\rho_d} \cdot \max\{0; \langle \boldsymbol{n}, \boldsymbol{l} \rangle\} + \boldsymbol{\rho_s} \cdot (\max\{0; \langle \boldsymbol{l}, \boldsymbol{r} \rangle\})^m$$

Oznaczenia:
$\boldsymbol{c}$ – wynikowy kolor (wektor o trzech składowych)
$\boldsymbol{\rho_d}$ – kolor światła rozproszonego dla materiału (ang. diffuse)
$\boldsymbol{\rho_s}$ – kolor światła odbijanego dla materiału (and. specular)
$\langle , \rangle$ - iloczyn skalarny dwóch wektorów
$\boldsymbol{n}$ – jednostkowy wektor normalny w cieniowanym punkcie (zwykle w układzie świata)
$\boldsymbol{l}$ – jednostkowy wektor do światła w cieniowanym punkcie (w tym samym układzie co $\boldsymbol{n}$)
$\boldsymbol{r}$ – odbity jednostkowy wektor od obserwatora $\boldsymbol{v}$: $\boldsymbol{r} = 2\langle \boldsymbol{n}, \boldsymbol{v} \rangle \boldsymbol{n} - \boldsymbol{v}$
m – wykładnik rozbłysku dla materiału

Na postawie wektora $\boldsymbol{n}$ w układzie świata można utworzyć lokalny ortonormalny układ współrzędnych $\boldsymbol{t}, \boldsymbol{b}, \boldsymbol{n}$ biorąc $\boldsymbol{t}$ jako iloczyn wektorowy $(0,1,0)$ i $\boldsymbol{n}$ oraz biorąc $\boldsymbol{b}$ jako iloczyn wektorowy $\boldsymbol{n}$ i $\boldsymbol{t}$ oraz na koniec normalizując wszystkie wektory tak, by miały jednostkową długość. W tym lokalnym układzie współrzędnych, wyznaczonym dla każdego cieniowanego piksela, można wyrazić wektory jednostkowe do światła i obserwatora jako:

$$\boldsymbol{l}' = (\langle \boldsymbol{t}, \boldsymbol{l} \rangle, \langle \boldsymbol{b}, \boldsymbol{l} \rangle, \langle \boldsymbol{n}, \boldsymbol{l} \rangle) = \left(\boldsymbol{l}'_x, \boldsymbol{l}'_y, \boldsymbol{l}'_z\right)$$
$$\boldsymbol{v}' = (\langle \boldsymbol{t}, \boldsymbol{v} \rangle, \langle \boldsymbol{b}, \boldsymbol{v} \rangle, \langle \boldsymbol{n}, \boldsymbol{v} \rangle) = \left(\boldsymbol{v}'_x, \boldsymbol{v}'_y, \boldsymbol{v}'_z\right)$$

W tym nowym układzie współrzędnych model Phonga wyraża się wzorem:

$$\boldsymbol{c} = \boldsymbol{\rho_d} \cdot \max\{0; \boldsymbol{l}'_z\} + \boldsymbol{\rho_s} \cdot \left(\max\left\{0; -\boldsymbol{l}'_x \boldsymbol{v}'_x - \boldsymbol{l}'_y \boldsymbol{v}'_y + \boldsymbol{l}'_z \boldsymbol{v}'_z\right\}\right)^m$$

Lafortune zaproponował uogólnienie wzoru Phonga wprowadzając trzy dodatkowe współczynniki c_x, c_y, c_z wyznaczane razem z $\boldsymbol{\rho_d}, \boldsymbol{\rho_s}$ i m metodami optymalizacji na podstawie mierzonych parametrów rozbłysków rzeczywistych materiałów.

$$\boldsymbol{c} = \boldsymbol{\rho_d} \cdot \max\{0; \boldsymbol{l}'_z\} + \boldsymbol{\rho_s} \cdot \left(\max\left\{0; c_x \boldsymbol{l}'_x \boldsymbol{v}'_x + c_y \boldsymbol{l}'_y \boldsymbol{v}'_y + c_z \boldsymbol{l}'_z \boldsymbol{v}'_z\right\}\right)^m$$

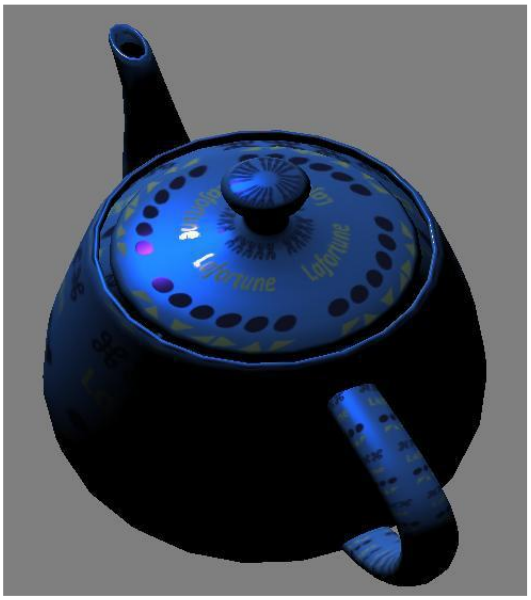
Dodatkowo w modelu Lafortune'a rozbłysków zwierciadlanych może być kilka dla jednego materiału. Wtedy licząc wartość koloru sumuje się kilka wyrazów postaci $\boldsymbol{\rho_s} \cdot \left(\max\left\{0; c_x \boldsymbol{l}'_x \boldsymbol{v}'_x + c_y \boldsymbol{l}'_y \boldsymbol{v}'_y + c_z \boldsymbol{l}'_z \boldsymbol{v}'_z\right\}\right)^m$, z których każdy ma inne wartości $c_x, c_y, c_z, \boldsymbol{\rho_s}$ i m.

Poniżej podane sią parametry trzech materiałów w modelu Lafortune'a:

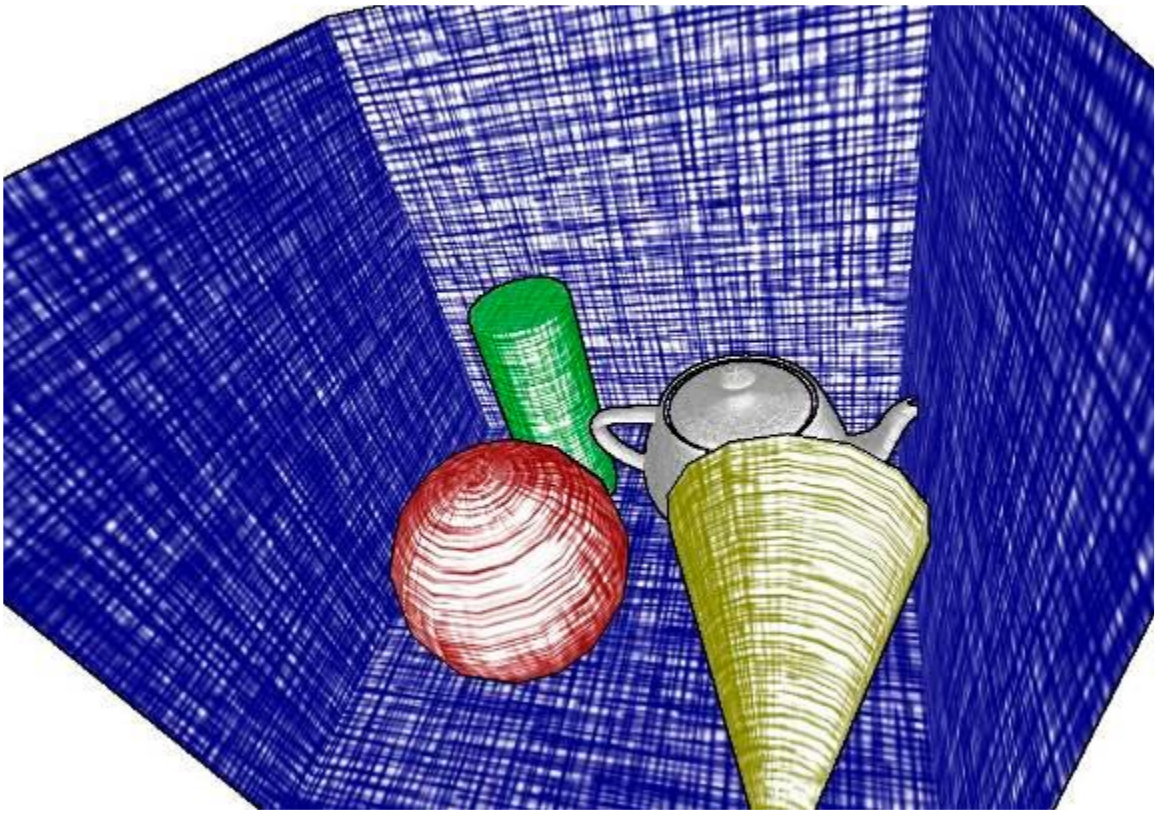
	$\boldsymbol{\rho_d}$	rozbłysk 1			rozbłysk 2		
		$\boldsymbol{\rho_s}$	(c_x, c_y, c_z)	m	$\boldsymbol{\rho_s}$	(c_x, c_y, c_z)	m
1	(0.16,0.19,0.14)	(1.27,0.69,0.75)	(-1.01,-1.01,1.00)	346			
2	(0.03,0.04,0.08)	(0.62,0.19,1.48)	(-1.01,-1.01,1.00)	231			
3	(0.06,0.13,0.24)	(0.00,0.80,3.10)	(-0.95,-0.95,0.97)	27	(0.83,0.02,0.00)	(-0.49,-0.49,0.38)	502

Należy na obiekt nałożyć teksturę ze wzorami w trzech kolorach: czerwonym, zielonym i niebieskim. Napisać shader pikseli, który według wyżej podanych wzorów policzy kolor w modelu Lafortune'a dla każdego z trzech podanych materiałów. Wynikowy kolor w piksel shaderze niech będzie sumą ważoną tych trzech kolorów, gdzie wagami są składowe czerwona, zielona i niebieska koloru pobranego z tekstury obiektu.

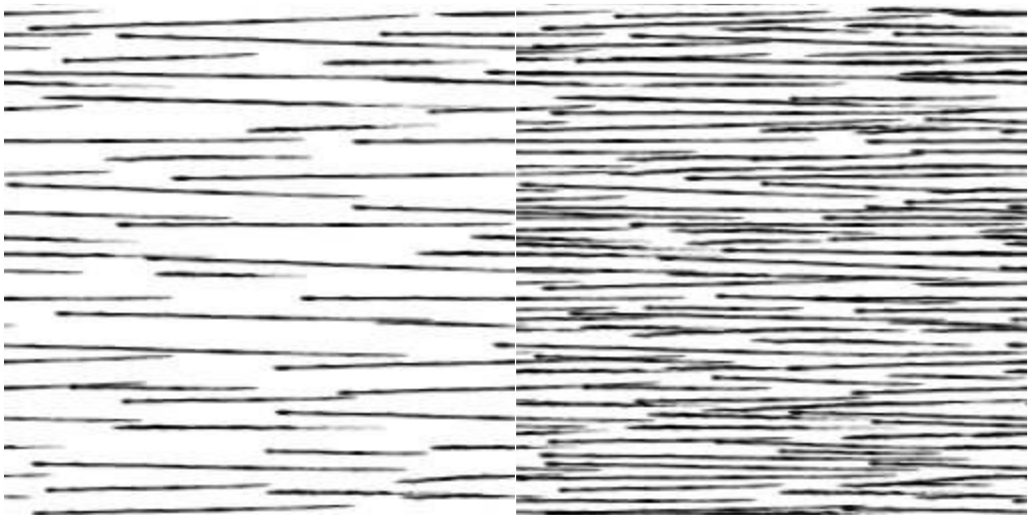
Przykładowa tekstura

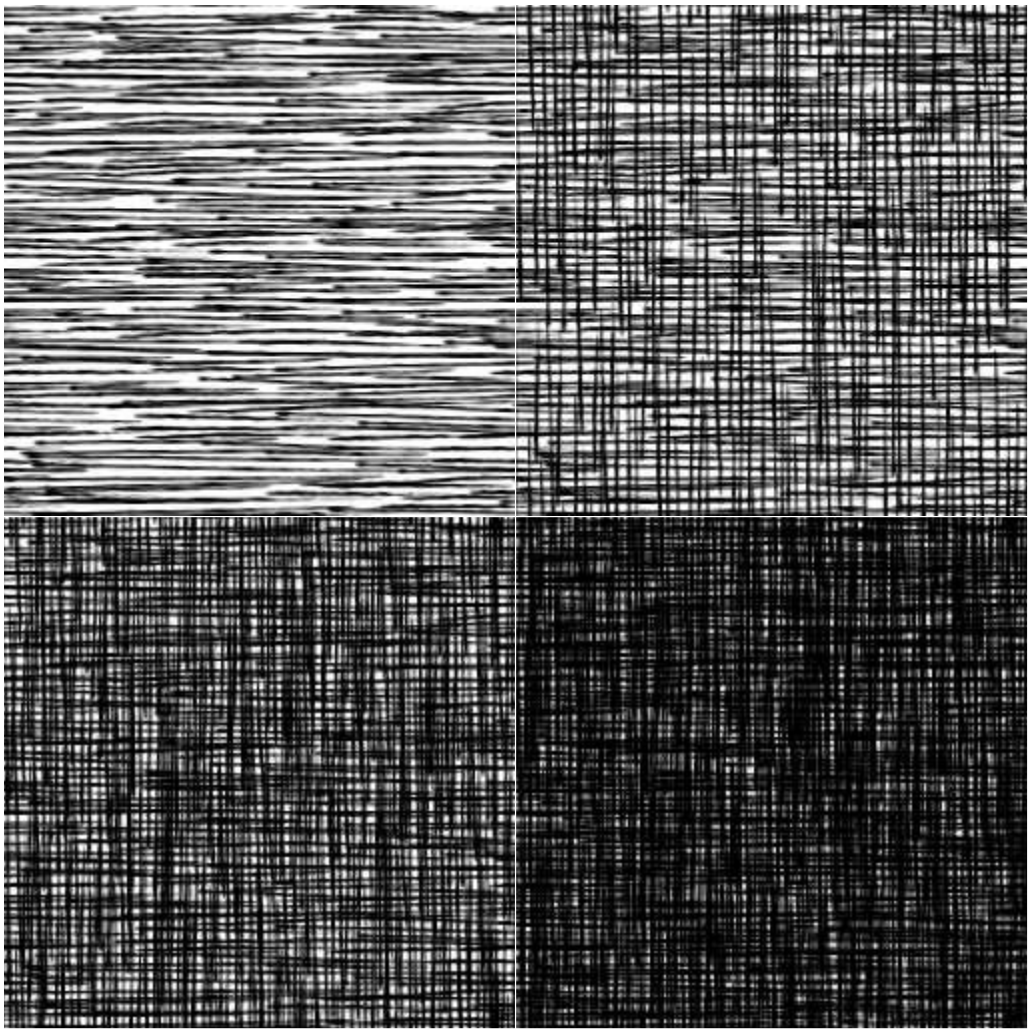

Wynik

Zadanie 3 – 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.

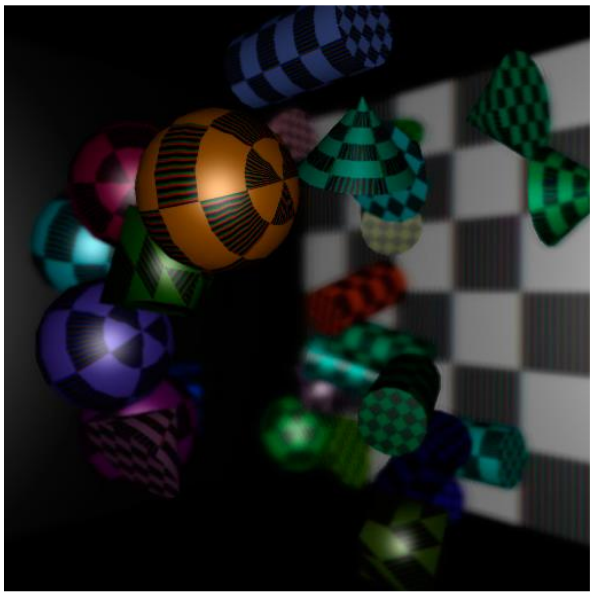
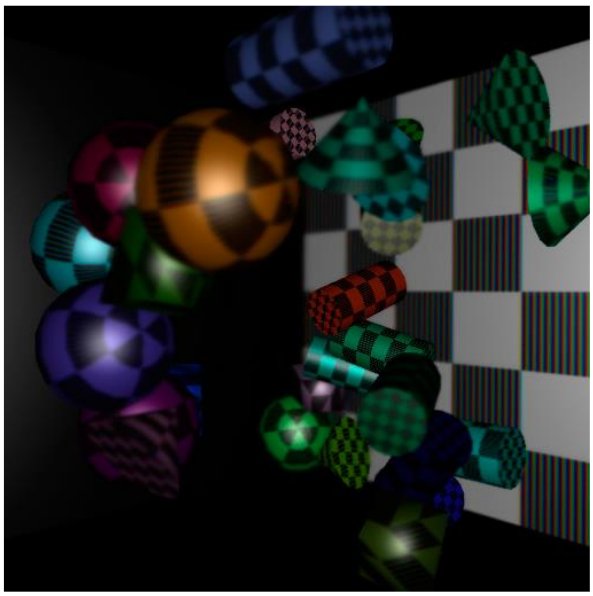
Zajmijmy się najpierw shaderem wierzchołków. Powinien on obliczyć:

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 4 – 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)

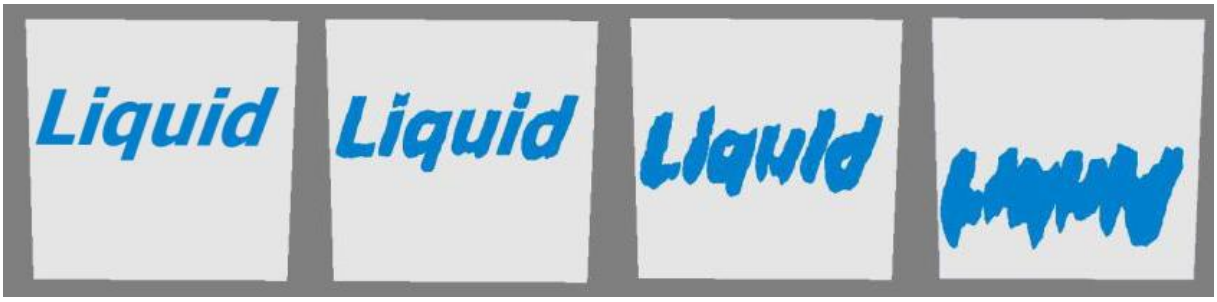
Trzeba napisać dwie funkcje pomocnicze:

- `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

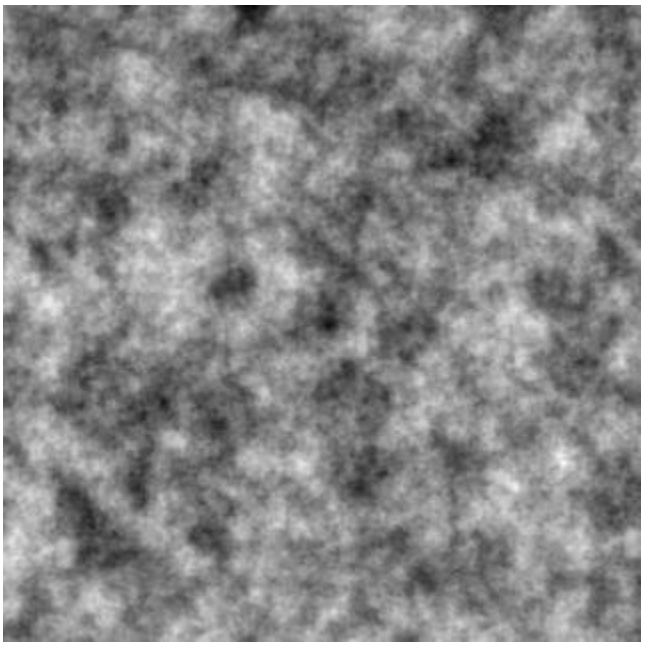
Zadanie 5 – Spływająca tekstura



W tym zadaniu na jedną ze ścian hali (lub dowolnego innego 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 – Hidh Dynamic Range

Zadanie ma **zwiększoną punktacje** (w porównaniu do relatywnie niewielkiej trudności) ze względu na to, że wymaga więcej samodzielnego wkładu w nauczenie się metody.

Pozostałe metody są dość dokładnie opisane w tym dokumencie lub na wykładzie (algorytm generowania cieni).

HDR (ang. High Dynamic Range) to technika znana głównie z fotografii, która pozwala na odwzorowanie większej dynamiki tonalnej niż tradycyjna metoda. Istnieje format kolorów HDR, który pozwala na zapisanie dużo szerszego spektrum barw (w szczególności dużo większych jasności) niż model RGB. Istnieją również bardzo kosztowne monitory przystosowane do wyświetlania obrazu HDR. Jest to jednak technika przyszłości, a na razie stosuje się pewne uproszczenia i sztuczki.

Ponieważ nie dysponujemy monitorem HDR możemy jedynie zastosować taki sposób działania:

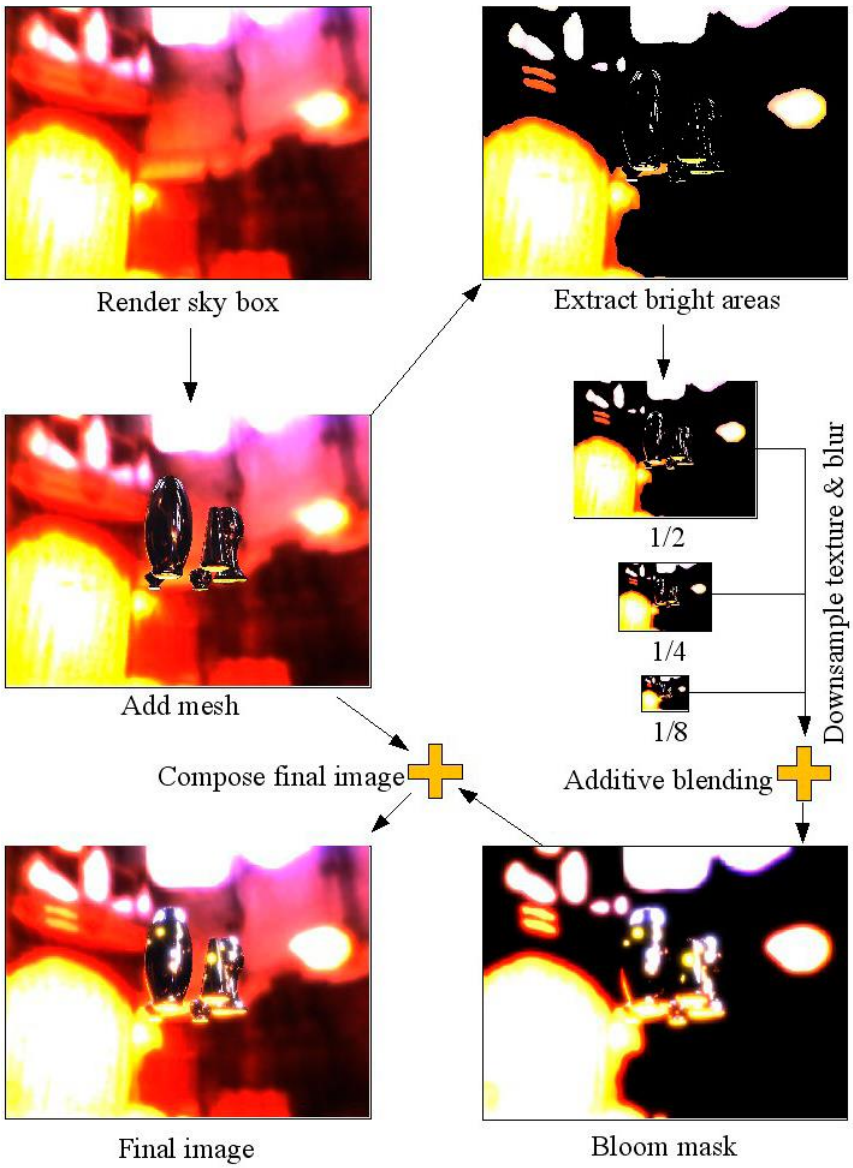
1. Obliczyć jasność piksela na scenie w rozszerzonej przestrzeni HDR
2. Zastosować operator, który na wejściu pobierze kolor o większej dynamice i zwróci obraz zgodny ze standardowym formatem RGB, ale tak przekształcając, żeby możliwie wiernie oddać dynamikę kolorów. Powyższa operacja nazywana jest **tone mapping**.

Sposób wykonania

- Wyrenderowanie sceny do zmiennoprzecinkowej tekstury o wysokiej precyzji T_0.
- Zastosowanie operatora *tone mapping* i zapisanie jego wyniku w teksturze T_1.
- Przeskalowanie T_0 do ¼ oryginalnej wielkości i zastosowanie na niej filtru bloom (bloom wykrywa jaśniejsze fragmenty, powyżej pewnego ustalonego progru kierunkowo je rozmywa wzdłuż X i Y).
- Wyświetlenie pełnoekranowego prostokąta przy pomocy mieszania addytywnego tekstury T_1 i T_2. Ponieważ druga tekstura jest mniejsza, nastąpi dodatkowe rozmycie T_1.

Oświetlenie

Do oświetlenia sugeruję użyć mapy sześciennej, która będzie świecić z każdej strony – może być to skybox. Może też być to kilka świateł punktowych.



(Sposób działania w przypadku oświetlenia ze skyboxa. Obraz pochodzi z [1])

Uwaga! Za zadanie można ostać maksymalnie 12, 15, 17, 18 bądź 20 punktów

[12 punktów]
Jedno światło punktowe + efekt HDR. Tutaj sugeruję wspomóc się przykładem z DirectX SDK.

[15 punktów]
Oświetlenie przy pomocy mapy środowiskowej + efekt HDR. Tutaj proponuję wspomóc się linkiem [2]

[17 punktów]
Efekt HDR od mapy środowiskowej + działający antyaliasing.

[18 punktów]
Efekt HDR zarówno do mapy środowiskowej jak i światła punktowego.

[20 punktów]
Efekt HDR zarówno od mapy środowiskowej jak i światła punktowego + działający antyaliasing.

[1] http://www.gamedev.net/page/resources/_/technical/graphics-programming-and-theory/high-dynamic-range-rendering-r2108
[2] http://www.smetz.fr/wp-content/uploads/2007/01/hdr_opengl.pdf
http://www.smetz.fr/?page_id=83
[3] http://msdn.microsoft.com/en-us/library/windows/desktop/bb173486(v=vs.85).aspx
[4] http://msdn.microsoft.com/en-us/library/windows/desktop/bb153293(v=vs.85).aspx
[5] http://en.wikipedia.org/wiki/High_dynamic_range_rendering

Zadanie 7 – 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 8 – Deferred Shading

Deferred Shading (Cieniowanie Odłożone) to technika opracowana już w latach 80-tych, która dopiero niedawno zyskała uznanie i obecnie jest stosowana w bardzo wielu przypadkach.
Jednym z podstawowych założeń metody jest zredukowanie obliczeń związanych z oświetleniem i cieniowaniem przez przeniesienie ich od przestrzeni pikseli ekranu – obliczenia są wykonywane tylko dla pikseli widocznych na ostatecznym obrazie.
W tradycyjnym przypadku renderowania obliczenia związane z oświetleniem wykonywane są w przestrzeni kamery i ich czas zależy od ilości oraz stopnia skomplikowania siatek.
Niech N – liczba obiektów, L – liczba świateł.
Pesymistyczna złożoność oświetlenia w tradycyjnym renderingu:

$$O(N, L) = N * L$$

Pesymistyczna złożoność oświetlenia w cieniowaniu odłożonym:

$$O(N, L) = N + L$$

W pierwszym przebiegu przekształcana jest geometria na scenie, a wynik zapisywany jest w tak zwanym G-buforze (a więc w tym przebiegu nie ma wyświetlania). G-bufor zbiera wszelkie potrzebne atrybuty obiektów dostępne w układzie świata/kamery i udostępnia je w przestrzeni 2D (już po rzutowaniu) na poziomie pikseli. Innymi słowy dla danego piksela możliwe jest pobranie atrybutów modelu i materiału z punktu, który na ten piksel został zrzutowany.
Na tym etapie zadania shaderów przedstawiają się następująco:

- **Vertex Shader**: dokonanie przekształcenia przy użyciu macierzy świata, widoku oraz projekcji. Przekazanie wszelkich potrzebnych atrybutów do Pixel Shadera.
- **Pixel Shader**: pobranie koloru ze zwykłej tekstury (z kolorami) i zapisanie wraz z resztą atrybutów (otrzymanych i zinterpolowanych z Vertex Shadera) do G-bufora.

Do realizacji G-bufora należy użyć techniki zwanej MRT (Multiple Render Targets). Każdy RenderTarget to dwuwymiarowa tekstura o rozmiarach równych rozmiarowi ekranu, która w przeciwieństwie do tradycyjnych tekstur nie jest przygotowana wcześniej, lecz powstaje w wyniku pierwszego przebiegu i przechowuje potrzebne atrybuty, o których mowa była w poprzednim akapicie.

Fizycznie G-bufor składa się z kilku takich tekstur (zwykle od 2 do 4 w zależności od ilości potrzebnych danych, stopnia kompresji oraz formatu tekstury – im większa precyzja na składową, tym więcej danych można pomieścić w mniejszej ilości tekstur).

Minimalna ilość potrzebnych atrybutów, które wystarczy przechować:

- **Pozycja** [X, Y, Z] w układzie kamery lub jedynie głębokość [D] (mając parametry macierzy projekcji, głębokość oraz pozycję [x, y] piksela na ekranie można dokonać projekcji wstecznej aby odtworzyć pozycję w układzie kamery).
  W przypadku zapisania pozycji jawnie jako trzy składowe można założyć, że scena zawiera się w wąskim zakresie któregoś z kierunków (np. Z) i przeznaczyć na nią jedynie 16 bitów zamiast 32.
- **Wektor normalny** [Xn, Yn]. Wystarczy przeznaczyć 12-16 bitów na składową wektora normalnego. Wektor ma tylko dwa stopnie swobody – trzecia wartość wynika z warunku

normalizacji (można 1 bit przeznaczyć jedynie na zwrot). Cały wektor normalny nie może zajmować więcej niż 32 bitów w teksturze.

- **Kolor rozproszony (diffuse)** [R, G, B] – 8 bitów na składową.

Opcjonalnie można zapisać dowolne wybrane przez siebie atrybuty (np. współczynnik odbicia soczewkowego, indeks materiału itp.).
Niech ograniczeniem górnym ilości pamięci potrzebnej do zapisania wszystkich atrybutów piksela będą **192 bity**. Odpowiada to na przykład trzem teksturom o czterech składowych (RGBA, XRGBA) z 16 bitową precyzją na składową (3*4*16=192)
Scena w zadaniu może być prosta (nie ma potrzeby użycia sceny używanej w poprzednich etapach projektu). Może być to pojedyncza płaszczyzna + jeden obiekt.

Jak realizować oświetlenie

Przypomnijmy, że wynikiem pierwszego przebiegu są pełnoekranowe tekstury z pewnymi wartościami zapisanymi dla każdego piksela, dzięki którym można wykonać obliczenia związane z oświetleniem (w ogólności na tym etapie można realizować różnorodne efekty takie jak: cienie, ambient occlusion itp.)
Istnieją przynajmniej dwa podejścia jak oświetlić piksele:

- Pierwsze podejście bazuje na fakcie, że wpływ światła punktowego lub reflektorowego zajmuje pewną określoną przestrzeń – odpowiednio kulę lub stożek. Można założyć stałą intensywność oświetlenia w przypadku światła reflektorowego (w stożku 1.0, na zewnątrz 0.0) a w przypadku światła punktowego liniowy bądź kwadratowy zanik wraz z odległością. Zadaniem jest napisanie sprytnego shadera, który dla każdego światła wyznaczy obszar wspólny z pikselami na ekranie i zastosuje oświetlenie addytywne (każde kolejne światło doda swój udział).
  Przykładowy scenariusz postępowania dla światła punktowego wygląda tak:
  - wyczyszczenie bufora koloru i pozostawienie bufora głębokości (jeśli go nie ma, należy wyrenderować całą scenę jeszcze raz bez zapisu koloru, aby uzyskać wypełniony bufor głębokości).
  - ustawienie bufora głębokości na „większe równe”
  - ustawienie wartości w buforze szablonu na 1 dla pikseli, które przechodzą test głębokości
  - narysowanie kuli (#1)
  - wyłączenie zapisu do bufora koloru i bufora głębokości
  - włączenie obcinania trójkątów zwróconych tyłem do kamery
  - narysowanie kuli (#2)
  - ustawienie testu bufora głębokości na „większe równe”
  - włączenie modyfikacji bufora szablonu i ustawienie testu bufora szablonu na „równe zero”
  - wyłączenie zapisu do bufora koloru
  - wyłączenie zapisu do bufora głębokości
  - włączenie obcinania trójkątów zwróconych przodem do kamery
  - narysowanie kuli (#3) z wykorzystaniem Pixel Shadera dla oświetlenia. Należy pobrać odpowiednie piksele z tekstur utworzonych w 1 przebiegu (MRT). Do shadera jako zmienną globalną należy przekazać środek światła punktowego aby obliczyć

intensywność (wszelkie dane są dostępne – pozycja światła w układzie kamery i punkt sceny w układzie kamery).
    - wyczyszczenie bufora szablonu (na 0).

  Nie ma obowiązku trzymania się powyższych punktów, ważne, aby spełnić założenie – wykonać przebieg z geometrią świateł tak aby wyznaczyć obszar oświetlony, obliczyć intensywność światła i wyświetlić oświetlone piksele używając mieszania addytywnego lub pośredniego bufora akumulującego kolor.
- Drugie podejście to jeden shader mający dostęp do parametrów wszystkich 50 świateł (tzw. ubershader). W tym przypadku wykonywany jest tylko jeden przebieg (od razu rysujący) dla pełnoekranowego prostokąta.
  Przy wyborze tej metody złożoność jest ściśle zależna od rozdzielczości ekranu, ponieważ obliczenia wykonywane są dla każdego widocznego, aczkolwiek niekoniecznie oświetlonego piksela.
  Można opcjonalnie wprowadzić uproszczenie, aby dla każdego piksela uznawać jedynie wpływ 4 najbliższych świateł.

Za zadanie można uzyskać maksymalnie **20 punktów**. Za wykonanie zadaniu używając drugiej metody można otrzymać **15 punktów**. Wynik można zwiększyć poprzez zaprogramowanie wybranych ulepszeń:

- Za użycie pierwszego podejścia przy implementacji oświetlenia lub własnej, równie wydajnej metody **+3 punkty**.
- Za antyaliasing obrazu, który nie jest wykonywany automatycznie w przypadku stosowania RenderTargetów **+5 punktów**.
- Za dodanie obsługi obiektów półprzezroczystych. Można założyć, że z dowolnego punktu na scenie promień wzdłuż kierunku patrzenia kamery przecina nie więcej niż dwa obiekty półprzezroczyste **+2 punkty**.

[1]http://http.download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_Deferred_Shading.pdf
[2] http://www.shawnhargreaves.com/DeferredShading.pdf
[3] http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter09.html
[4] http://http.developer.nvidia.com/GPUGems3/gpugems3_ch19.html