

# Warszaty z Shaderów

Paweł Aszklar

P.Aszklar@mini.pw.edu.pl

Warszawa, 8 czerwca 2020

## 1 Mapowanie wypukłości

Mapowanie wypukłości polega na wyznaczeniu normalnej dla każdego fragmentu (piksla) renderowanej powierzchni na podstawie informacji pobranej z tekstuury. Tak wyznaczona normalna jest następnie wykorzystywana do obliczeń oświetlenia w danym punkcie powierzchni.

Przy mapowaniu wypukłości wykorzystywane są wektory określone w przestrzeni stycznej do powierzchni. Przestrzeń ta wyznaczona jest przez trzy wersory: styczną, bistyczną i normalną. Wersor normalny w danym punkcie powierzchni pojawił się już na przykład w opisie modelu oświetlenia Phonga. Dwa pozostałe wersory leżą w płaszczyźnie stycznej do powierzchni. Są one znormalizowanym gradientem współrzędnych tekstuury (gradient liczony jest w płaszczyźnie stycznej do powierzchni). Przy mapowaniu wypukłości dopuszcza się, żeby styczna i bistyczna nie były dokładnie prostopadłe. Wersory wyznaczające przestrzeń styczną do powierzchni są liczone w każdym wierzchołku siatki trójkątów. W języku HLSL istnieją wbudowane funkcje pomagające policzyć styczną  $T$ . Mając ją oraz normalną  $N$ , bistyczną wyznaczyć można ze wzoru:

$$B = \frac{N \times T}{\|N \times T\|} \quad (1)$$

Jeżeli chcemy, aby wynikowy układ był ortonormalny, możemy następnie *poprawić* styczną  $T$ :

$$T = B \times N \quad (2)$$

Mając te trzy wersory możemy wyznaczyć macierz przekształcenia z układu stycznego do powierzchni do układu w którym wyrażone są wersory (np. układ sceny):

$$\begin{bmatrix} T & B & N \end{bmatrix}^{-1} = \begin{bmatrix} T & B & N \end{bmatrix}^T \quad (3)$$

Z reguły mapa wypukłości (mapa normalnych) przechowuje współrzędne nowej, zaburzonej wartości normalnej w układzie tworzonym przez styczną, bistyczną i oryginalną normalną. Współrzędne wersorów mogą być z zakresu [-1,1], wektory

normalne w tekstuurze reprezentującej wartości z przedziału [0,1] zakodowane są stosując odpowiednie przekształcenie liniowe.

W innej odmianie mapowania wypukłości, którą nie będziemy się zajmować — mapowaniu wypukłości Blinna — w mapie wypukłości przechowywane są gradienty mapy wysokości w kierunkach wersora stycznego i bistycznego. Gradienty te są dodawane do oryginalnej normalnej, a wynik jest następnie normalizowany. W odmianie, którą będziemy się zajmowali, zakładamy, że tak obliczony wynik, wyrażony w przestrzeni stycznej, jest już zakodowany w mapie wypukłości. Zdarza się również, chociaż rzadko, że mapa wypukłości zawiera normalne wyrażone już w układzie modelu (wtedy ta sama mapa nie może być wykorzystana z innym modelem).

Wykonanie:

1. Model sfery użyty początkowo w programie nie posiada współrzędnych tekstuury potrzebnych do nałożenia mapy wypukłości oraz wyznaczenia wektora stycznego. W konstruktorze klasy `ShaderDemo` wczytaj więc model Teapot .3ds, którym posłużymy się w kolejnych punktach:

```
auto teapot = addModelFromFile("models/Teapot.3ds");
```

2. Dla nowego modelu dodaj do projektu pliki shadera wierzchołków teapot VS.hlsl (kopią sphereVS.hlsl) i pikseli teapotPS.hlsl (kopią spherePS.hlsl).
3. Zamiast sfery rysować chcemy model czajnika. W tym celu w konstruktorze `ShaderDemo` usuń lub zakomentuj linie:

```
auto passSphere = addPass(L"sphereVS.cso",
                           L"spherePS.cso");
addModelToPass(passSphere, sphere);
```

a zamiast nich dodaj przebieg renderowania czajnika z nowymi shaderami i modelem:

```
auto passTeapot = addPass(L"teapotVS.cso",
                           L"teapotPS.cso");
addModelToPass(passTeapot, teapot);
```

4. Po uruchomieniu programu łatwo zauważyc, że czajnik nie wyświetla się poprawnie, jest on zdecydowanie za duży i dodatkowo położony jest na boku. Aby poprawić jego rysowanie wyznacz macierz modelu złożoną kolejno z: skalowania o  $\frac{1}{60}$ , obrotu wokół osi  $X$  o  $-\frac{\pi}{2}$ , translacji wzdłuż osi Y o 0,5. To ostatnie przekształcenie pozwoli przesunąć spód czajnika na poziom początku układu współrzędnych, co przyda nam się później. Wyznaczoną macierz przypisz do modelu czajnika zaraz po jego wczytaniu:

```

XMFLOAT4X4 modelMtx;
...
model(teapot).applyTransform(modelMtx);

```

5. Wczytaj teksturę mapy wypukłości `normTex` oraz stwórz domyślny sampler `samp`:

```

m_variables.AddSampler(m_device, "samp");
m_variables.AddTexture(m_device, "normTex",
    L"textures/normal.png");

```

6. W shaderze wierzchołków `teapotVS.hls1` do struktur `VSInput` i `VSOutput` dodaj pole `tex` typu `float2` i semantycę `TEXCOORD0`, które posłużą do przechowywania współrzędnych tekstury wierzchołka:

```
float2 tex : TEXCOORD0;
```

7. W funkcji `main` shadera przepisz pole `tex` z parametru wejściowego do rezultatu. W przypadku naszego modelu czajnika warto te współrzędne podzielić przez 4 aby tekatura miała większy rozmiar na powierzchni:

```
o.tex = i.tex / 4.0f;
```

8. W shaderze pikseli `teapotPS.hls1` dodaj teksturę 2D `normTex` oraz sampler `samp`.

9. Analogicznie do zmiany `VSOutput` z shadera wierzchołków, dodaj w strukturze `PSInput` pole `tex`.

10. Stwórz funkcję `normalMapping`, która na postawie normalnej `N`, stycznej `T` (wyrażonych w układzie świata) oraz zaburzonej normalnej `tn` pobranej z tekstuury (wyrażonej w przestrzeni stycznej) wyznaczy nowy wektor normalny zgodnie z równaniami 1–3.

11. W funkcji `main` wyznacz styczną `T` korzystając z funkcji pomocniczych `ddx` oraz `ddy`:

```

float3 N = normalize(i.norm);
float3 dPdx = ddx(i.worldPos);
float3 dPdy = ddy(i.worldPos);
float2 dtex = ddx(i.tex);
float2 dtdy = ddy(i.tex);
float3 T = normalize(-dPdx*dtex.y + dPdy*dtdy.y);

```

12. Pobierz z tekstuury `normTex` zaburzony wektor normalny `tn` w punkcie `i.tex` i przeskaluj jego współrzędne z zakresu  $[0, 1]$  do  $[-1, 1]$ . Wymagane może być też odwrócenie współrzędnej  $y$ .

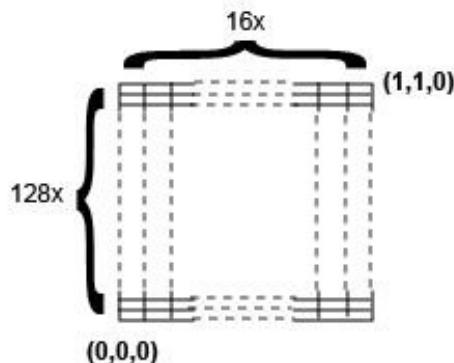
13. Korzystając z funkcji `normalMapping` wyznacz nowy wektor normalny:

```
float3 norm = normalMapping(N, T, tn);
```

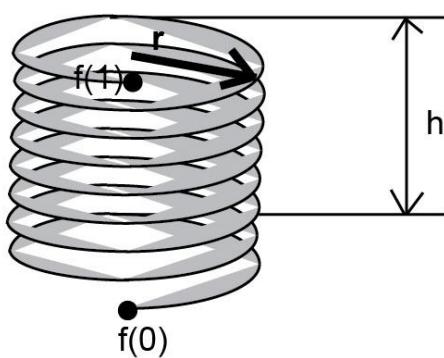
Nie zapomnij użyć go w dalszych obliczeniach oświetlenia.

## 2 Generowanie sprężyny

Do rysowania sprężyny wykorzystamy jako bazę siatkę kwadratu o przeciwnie gęstych wierzchołkach  $[0, 0, 0]^T$  i  $[1, 1, 0]^T$  podzielonego na 16 kolumn i 128 wierszy (łącznie 4096 trójkątów składających się na kwadrat). Siatka ta w shaderze wierzchołków zostanie przekształcona w sprzęzynę.



Przekształcenie oparte będzie o parametryczny opis spiralnej krzywej (funkcja przekształcająca przedział  $[0, 1]$  w oś sprężyny), gdzie współrzędna  $y$  wierzchołka siatki posłuży nam za parametr krzywej. Dzięki niemu uzyskamy punkt środka przekroju sprężyny oraz wektor styczny, normalny i binormalny krzywej w tym punkcie. Kształt krzywej będzie zależeć od następujących parametrów: długość całej krzywej  $l$ , wysokość spirali  $h$ , promień spirali  $R$ .



Odpowiednia krzywa  $f : [0, 1] \rightarrow \mathbb{R}^3$  ma postać:

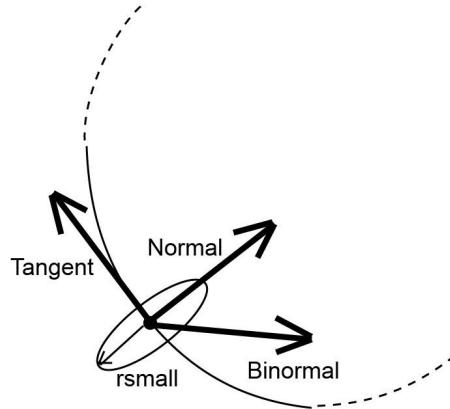
$$f(s) = \begin{vmatrix} R \cos(sa) \\ sh \\ R \sin(sa) \end{vmatrix} \quad (4)$$

, gdzie  $a$  wynosi:

$$a = \frac{\sqrt{l^2 - h^2}}{R} \quad (5)$$

co zapewni, że sprężyna będzie miała długość  $l$ .

Wektory normalny i binormalny definiują płaszczyznę prostopadłą do krzywej. W tej płaszczyźnie na okręgu o ustalonym promieniu (grubość sprężyny  $r$ ) ostatecznie umieszczać będziemy wierzchołki siatki, przy czym współrzędna  $x$ , po przemnożeniu przez  $2\pi$ , posłuży nam za wartość kąta obrotu na tym okręgu.



Wykonanie:

1. W konstruktorze klasy `ShaderDemo` dodaj wspomniane parametry sprężyny `h0`, `l`, `r`, `rsmall` do GUI.

```
auto h0 = 1.5f;
m_variables.AddGuiVariable("h0", h0, 0, 3);
m_variables.AddGuiVariable("l", 15.f, 5, 25);
m_variables.AddGuiVariable("r", 0.5f, 0.01f, 1);
m_variables.AddGuiVariable("rsmall",
    0.1f, 0.01f, 0.5f);
```

2. Wczytaj model płaszczyzny:

```
auto plane = addModelFromFile("models/Plane.obj");
```

3. Stwórz dla niego nowy przebieg rysowania (z nieistniejących jeszcze shaderów sprężyny):

```

auto passSpring = addPass(L"springVS.cso",
    L"springPS.cso");
addModelToPass(passSpring, plane);

```

4. Dodaj do projektu shadery wierzchołków springVS.hlsl (kopią teapotVS.hlsl) i pikseli springPS.hlsl (kopią teapotPS.hlsl) sprężyny.

5. W shaderze wierzchołków springVS.hlsl dodaj globalne zmienne `h0`, `l`, `r`, `rsmall` typu float.

6. Dodaj do `VSOutput` pole `tangent` typu `float3` o semantycie `TANGENT0`:

```
float3 tangent : TANGENT0;
```

7. Dodaj funkcję

```
float dangle_ds(float l, float r, float h);
```

do obliczania parametru  $a$  zgodnie ze wzorem 5 (wartość ta odpowiada pochodnej kąta obrotu punktu krzywej wokół pionowej osi).

8. Dodaj funkcję:

```
float3 get_position(float s, float r,
                     float h, float a);
```

która obliczy pozycję punktu na krzywej zgodnie z równaniem 4. Dodaj analogiczne funkcje `get_tangent` i `get_normal` liczące styczną i normalną do krzywej w tym punkcie (będą one, odpowiednio, pierwszą i drugą pochodną powyższego wzoru względem parametru  $s$ );

9. W funkcji `main` dodaj na samym początku liczenie (na podstawie współrzędnej  $y$  wierzchołka siatki) punktu na krzywej i trzech wersorów tworzących ortogonalny układ współrzędnych stycznych do niej:

```

float h = h0; //przyda sie pozniej
float a = dangle_ds(l, r, h);
float3 CurvePosition =
    get_position(i.pos.y, r, h, a);
float3 CurveTangent =
    normalize(get_tangent(i.pos.y, r, h, a));
float3 CurveNormal =
    normalize(get_normal(i.pos.y, r, h, a));
float3 CurveBinormal =
    cross(CurveNormal, CurveTangent);

```

Wartość `two_pi` można zdefiniować jako globalną stałą:

- ```

static const float two_pi =
    6.283185307179586476925286766559f;

```
10. Policz normalną punktu w układzie sceny obracając wektor jednostkowy w płaszczyźnie wektorów `CurveNormal` i `CurveBinormal` o kąt równy  $2\pi$  razy współrzędna x wierzchołka siatki:
- ```

float3 Normal = CurveNormal*cos(two_pi*i.pos.x) +
    CurveBinormal*sin(two_pi * i.pos.x);

```
11. Policz pozycję punktu na powierzchni sprężyny odsuwając go wzduż `Normal` o `rsmall` od `CurvePosition`:
- ```

float4 Position =
    float4(CurvePosition + rsmall * Normal, 1.0f);

```
12. Popraw odpowiednio obliczenia dla pól struktury `VSOoutput`:
- ```

float4 worldPos = mul(modelMtx, Position);
o.view = normalize(camPos.xyz - worldPos.xyz);
o.norm = normalize(mul(modelInvTMtx,
    float4(Normal, 0.0f)).xyz);
o.worldPos = worldPos.xyz;
o.pos = mul(viewProjMtx, worldPos);
o.tangent = mul(modelInvTMtx,
    float4(CurveTangent, 0.0f)).xyz;
o.tex = float2(i.pos.y * 6.8f, i.pos.x * 0.2f);

```
13. W shaderze pikseli sprężyny `springPS.hlsl` do struktury `PSInput` dodaj pole `tangent` analogicznie jak w kroku 6.
14. W funkcji `main` zamiast wyliczania wektora stycznego `T`, użyj wartości z parametru:
- ```

float3 T = normalize(i.tangent);

```
15. Po uruchomieniu programu zobaczymy, że sprężyna wyświetla się poprawnie, jednak nachodzi ona na model czajnika. Możemy zmodyfikować macierz modelu sprężyny aby przesunąć ją o `h0`. W konstruktorze `ShaderDemo` dodaj do modelu `plane` odpowiednią macierz translacji:
- ```

XMStoreFloat4x4(&modelMtx,
    XMMatrixTranslation(0, -h0, 0));
model(plane).applyTransform(modelMtx);

```

16. W ostatnim kroku chcemy jeszcze upewnić się, że czajnik poruszać będzie się razem ze sprężyną przy modyfikacji parametru `h0` z interfejsu programu. Początkową wartość `h0` zakodujemy w macierzy modelu czajnika. W tym celu w konstruktorze `ShaderDemo` zmodyfikuj składową translacji przy tworzeniu macierzy modelu czajnika, by wyrażała przesunięcie wzdłuż osi `y` o `0.5f - h0`.
17. W shaderze wierzchołków czajnika `teapotVS.hls1` dodaj zmienną globalną `h0`, a w funkcji `main` dodaj aktualną wartość `h0` (która zmieniać będzie się w czasie) do współrzędnej `y` położenia punktu w układzie świata:

```
float4 worldPos = mul(modelMtx,
    float4(i.pos, 1.0f));
worldPos.y += h0;
```

### 3 Animacja sprężyny

Animację sprężyny uzyskamy zmieniając w czasie parametr  $h$  w równaniu krzywej na podstawie analitycznego rozwiązania równania ruchu harmonicznego z tłumieniem:

$$m\ddot{x} + d\dot{x} + kx = 0$$

gdzie  $x$  to wychylenie punktu z położenia równowagi  $\dot{x}$  prędkość,  $\ddot{x}$  przyspieszenie,  $m$  masa,  $d$  współczynnik tłumienia lepkiego,  $k$  stała sprężystości. W zależności od doboru stałych mamy przypadek, gdy punkt po wychyleniu z położenia równowagi wraca do niego i tam się zatrzymuje (ang. *overdamping*) lub gdy punkt wraca do położenia równowagi i przekracza je wielokrotnie, wykonując drgania wokół niego (ang. *underdamping*). Wykorzystamy rozwiązanie analityczne dla drugiego przypadku przy założeniu, że w chwili początkowej  $t = 0$ ,  $x(0) = 0$  i  $\dot{x}(0) = v_{max}$ :

$$h = x_{max} e^{\frac{\ln(0.5)t}{t_{half}}} \sin\left(\frac{v_{max}}{x_{max}} t\right) \quad (6)$$

W powyższym wzorze:  $h_0$  to wartość  $h$  w położeniu równowagi,  $t$  to czas w sekundach,  $t_{half}$  to czas malenia amplitudy drgań o połowę,  $v_{max}$  to prędkość początkowa, a  $x_{max}$  to maksymalne wychylenie. Dla uproszczenia można przyjąć, że:

$$\ln(0.5) = -0.693147$$

Wykonanie:

1. Do shadera wierzchołków sprężyny (`springVS.hls1`) dodaj zmienne `time`, `xmax`, `vmax`, `thalf` typu `float`.
2. Zaimplementuj funkcję `springHeight` na podstawie wzoru 6.
3. w funkcji `main` shadera dodaj wynik funkcji `springHeight` do zmiennej `h`.

```
h = h0 + springHeight(time);
```

4. Podobne zmiany wprowadź w shaderze wierzchołków czajnika (`teapotVS.hls1`) — wynik funkcji `springHeight` dodaj do współrzędnej `y` zmiennej `worldPos`.

```
worldPos.y += springHeight(time);
```

5. W konstruktorze `ShaderDemo` zdefiniuj odpowiednie zmienne shaderów:

```
m_variables.AddGuiVariable("thalf", 3.f, 1.f, 5.f);
m_variables.AddGuiVariable("xmax", .5f, .1f, 1.f);
m_variables.AddGuiVariable("vmax", 4.f, .5f, 10.f);
m_variables.AddSemanticVariable("time",
    VariableSemantic::FloatT);
```

6. Opcjonalnie, aby animacja rozpoczęła się ponownie po pewnym czasie, jako wartości parametr  $t$  można użyć obecnego czasu modulo  $t_{max}$ , gdzie  $t_{max}$  to czas trwania animacji:

```
time = modf(time/tmax) * tmax;
```

## 4 Animacja wody z użyciem szumu Perlina

Powierzchnię wody będziemy reprezentować jako kwadrat. Dla każdego piksela policzyć musimy odbity i załamany promień od obserwatora. Dla obu promieni, które zaczepimy w położeniu przeciwnobrazu danego piksela w układzie lokalnym powierzchni wody, policzyć musimy przecięcie z sześcianem otaczającym scenę. To przecięcie posłuży nam za współrzędne tekstury przy pobieraniu koloru z tekstury sześciennnej. Oba kolory (dla pomienia odbitego i załamanego) zmieszane zostaną w proporcji ustalonej na podstawie przybliżenia współczynnika Fresnala  $f$ . To przybliżenie będzie funkcją cosinusa kąta  $\theta$  pomiędzy wektorem od obserwatora  $V$  a wektorem normalnym powierzchni  $N$ :

$$f = F_0 + (1 - F_0)(1 - \cos \theta)^5 \quad (7)$$

$$\cos \theta = \max(\langle N, V \rangle, 0)$$

$$F_0 = \left( \frac{n_2 - n_1}{n_2 + n_1} \right)^2$$

Gdzie  $n_1, n_2$  to współczynniki załamania ośrodka źródłowego (powietrza,  $n_1 = 1$ ) i docelowego (wody,  $n_2 \approx \frac{4}{3}$ ). Wartość  $F_0$  to współczynnik Fresnala przy patrzeniu na powierzchnię pod kątem prostym, który dla powierzchni powietrze/woda wynosi  $\approx 0,14$ .

Warto zauważyć, że powyżej wyprowadzenie zakłada, że kolor wyrażony jest poprzez fizyczną intensywność światła (ang. *luminance*). Dlatego w zadaniu użyjemy tekstury HDR. W przeciwnieństwie do standardowych tekstur, gdzie wartości kanałów w pikseli zapisane jako 8-bitowe wartości całkowite wyrażają postrzeganą jasność (ang. *brightness*), w teksturach HDR zapisane są zmiennopozycyjne wartości intensywności. Uprościć nam to obliczenia w tym i kolejnych zadaniach, jednakże przed wyświetleniem takiego koloru na ekranie, gdzie musimy podać jasność, należy wykonać korekcję gamma.

Jeżeli używalibyśmy kolorów ze standardowej tekstury, wartość współczynnika Fresnela wyznaczona że wzoru 7 byłaby za niska. Aby uzyskać poprawny rezultat w takim przypadku, wartość współczynnika  $F_0$  wynieść powinna:

$$\left( \frac{n_2 - n_1}{n_2 + n_1} \right)^{\frac{2}{2,2}} \approx 0,17$$

(2,2 to wartość współczynnika gamma w przestrzeni kolorów sRGB).

Do symulacji zaburzeń powierzchni wody użyjemy trójwymiarowej tekstury szumu Perlina. Tekstura jest wygenerowana tak, by zachodziła zgodność wartości na przeciwnieństwie do standardowej tekstury, gdzie wartości reprezentują czas symulacji. Pozostałe dwie odpowiadają będą położeniu punktów na kwadracie powierzchni wody na scenie. Wartości z tekstuury użyjemy do zaburzenia wektora normalnego.

Wykonanie:

1. Zacznię od wczytania w konstruktorze `ShaderDemo` potrzebnych tekstur: sześciennnej tekstuury otoczenia `envMap` oraz trójwymiarowej tekstuury szumu perlina `perlin`.

```
m_variables.AddTexture(m_device, "envMap",
    L"textures/cubeMap.dds");
m_variables.AddTexture(m_device, "perlin",
    L"textures/NoiseVolume.dds");
```

Dodaj również dwie zmienne które posłużą nam do ustalenia pozycji powierzchni wody: `mvpMtx` zawierającą iloczyn macierzy modelu, widoku i perspektywy; oraz `waterLevel` którą sterować będziemy z poziomu GUI wysokością powierzchni wody.

```
m_variables.AddSemanticVariable("mvpMtx",
    VariableSemantic::MatMVP);
m_variables.AddGuiVariable("waterLevel",
    -0.05f, -1, 1, 0.001f);
```

2. Stwórz model kwadratu `quad`, który posłuży nam za powierzchnię wody, oraz sześcian `envModel` na który nałożymy tekstudę otoczenia, by zilustrować, że odbicia i załamania na powierzchni wody są poprawne. Zamiast wczytywać

je z pliku możemy je stworzyć za pomocą ciągu znaków w zmodyfikowanym formacie [NFF](#) wspieranym przez bibliotekę AssImp. W celu uzyskania potrzebnego nam modelu musimy podać typ bryły i jej parametry. Kwadrat stworzymy jako wielokąt (pp — z ang. *polygonal patch*) podając liczbę wierzchołków a następnie, w kolejnych liniach tekstu, współrzędne ich położenia i wektorów normalnych. Sześciian (hex - z ang. *hexahedron*) tworzymy podając współrzędne środka i *promień*, czyli połowę długości głównej przekątnej (dla sześciianu o boku 2 jednostki będzie to  $\sqrt{3} \approx 1.73205$ ).

```
auto quad = addModelFromString(
    "pp 4\n1 0 1 0 1 0\n1 0 -1 0 1 0\n"
    "-1 0 -1 0 1 0\n-1 0 1 0 1 0");
auto envModel =
    addModelFromString("hex 0 0 0 1.73205");
```

Oba modele warto jeszcze przeskalać (np. razy 20) aby były dużo większe od czajnika.

```
XMStoreFloat4x4(&modelMtx,
    XMMatrixScaling(20, 20, 20));
model(quad).applyTransform(modelMtx);
model(envModel).applyTransform(modelMtx);
```

3. Stwórz przebieg do rysowania sześciianu `passEnv` i dodaj do niego model `envModel`. Jako że scenę otaczającą będziemy obserwować od środka, przy rysowaniu musimy odwrócić orientację ścian przednich. Zrób to dodając do przebiegu odpowiedni *Rasterizer State*.

```
auto passEnv = addPass(L"envVS.cso", L"envPS.cso");
addModelToPass(passEnv, envModel);
addRasterizerState(passEnv,
    RasterizerDescription(true));
```

Następnie dodaj do projektu pliki shaderów wierzchołków `envVS.hlsl` i pikseli `envPS.hlsl` sceny otaczającej.

4. Zmień kod szadera wierzchołków `envVS.hlsl`. Na wejściu powinien otrzymać pozycję punktu w układzie lokalnym sześciianu. Policzyć i zwrócić powinien on pozycję punktu rzutowanego na ekran, co uzyskujemy po przemnożeniu przez macierze modelu, kamery i perspektywy — ich iloczyn mamy w zmiennej `mvpMtx`. Dodatkowo musi on wyznaczyć współrzędne tekstury sześcienną — będzie to początkowa pozycja punktu w układzie lokalnym sześciianu, ewentualnie podzielona przez połowę rozmiaru sześciianu.

```
matrix mvpMtx;

struct VSOutput
```

```

{
    float4 pos : SV_POSITION;
    float3 tex : TEXCOORD0;
};

VSOutput main(float3 pos : POSITION0)
{
    VSOutput o;
    o.tex = normalize(pos);
    o.pos = mul(mvpMtx, float4(pos, 1.0f));
    return o;
}

```

5. Zmień kod shadera pikseli. Na wejściu otrzymać powinien współrzędne tekstury z shadera wierzchołków i zwrócić odpowiedni kolor z tekstuury otoczenia envMap.

```

sampler samp;
textureCUBE envMap;

struct PSInput
{
    float4 pos : SV_POSITION;
    float3 tex : TEXCOORD0;
};

float4 main(PSInput i) : SV_TARGET
{
    float3 color = envMap.Sample(samp, i.tex).rgb;
    return float4(color, 1.0f);
}

```

6. Po uruchomieniu programu zauważać można, że tekstura otoczenia wygląda na bardzo ciemną. Dzieje się tak dlatego, że jest to tekstura HDR, a bufor koloru oczekuje postrzeganej jasności. Aby dokonać konwersji dodaj do shadera korekcję gamma.

```
color = pow(color, 0.4545f);
```

7. Stwórz przebieg rysowania powierzchni wody `passWater` i dodaj do niego model kwadratu. Chcemy, aby kwadrat był widoczny z obu stron, dlatego do przebiegu dodać trzeba *Render State* z wyłączeniem obcinania ścian tylnych.

```
auto passWater =
```

```

        addPass(L"waterVS.cso", L"waterPS.cso");
addModelToPass(passWater, quad);
RasterizerDescription rs;
rs.CullMode = D3D11_CULL_NONE;
addRasterizerState(passWater, rs);

```

Następnie dodaj do projektu shadery wierzchołków waterVS.hlsl i pikseli waterPS.hlsl powierzchni wody.

- Zmień kod shadera wierzchołków waterVS.hlsl.

```

matrix modelMtx, viewProjMtx;
float waterLevel;

struct VSOutput
{
    float4 pos : SV_POSITION;
    float3 localPos : POSITION0;
    float3 worldPos : POSITION1;
};

VSOutput main( float3 pos : POSITION0 )
{
    ...
}

```

W funkcji `main` wyznacz pozycję punktu w układzie lokalnym (`localPos` — zmień wartość współrzędnej  $y$  na `waterLevel`), układzie sceny (`worldPos`) oraz po zrzutowaniu perspektywicznym (`pos`).

- Zmień kod shadera pikseli powierzchni wody (waterPS.hlsl).

```

float4 camPos;
sampler samp;

struct PSInput
{
    float4 pos : SV_POSITION;
    float3 localPos : POSITION0;
    float3 worldPos : POSITION1;
};

float4 main(PSInput i) : SV_TARGET
{
    float3 viewVec =
        normalize(camPos.xyz - i.worldPos);
}

```

```

    float3 norm = float3(0.0f, 1.0f, 0.0f);
    ...
}

```

W funkcji `main` policz wektory odbity (przydatna funkcja `reflect`) oraz załamany (funkcja `refract` — trzecim parametrem jest współczynnik załamania przy przejściu pomiędzy ośrodkami, równy  $\frac{n_1}{n_2}$ ).

10. Zaimplementuj pomocniczą funkcję `intersectRay`, która mając dany punkt wewnętrz sześcianu  $[-1, 1] \times [-1, 1] \times [-1, 1]$  oraz wektor kierunkowy policz przecięcie tego promienia z najbliższą ścianą sześcianu. Celem tej funkcji jest policzenie współrzędnych tekstury sześciennnej na podstawie pozycji renderowanego piksela w układzie lokalnym powierzchni wody i kierunku promienia odbitego lub załamanego.
11. Dodaj do shadera zmienną `envMap` (typ `textureCUBE`) tekstury sześciennej otoczenia.
12. Korzystając z funkcji `intersectRay` pobierz z tej tekstury kolor dla promienia odbitego i załamanego. Przed przejściem do kolejnego kroku można przetestować działanie funkcji zwracając z shadera jeden z tych kolorów.
13. Zaimplementuj funkcję `fresnel` która wyznaczy współczynnik Fresnala dla danego piksela zgodnie z równaniem 7.
14. Zmieszaj ze sobą dwa pobrane kolory zgodnie z wyznaczonym współczynnikiem i wykonaj korekcję gamma (jak w kroku 6) w celu uzyskania ostatecznego koloru piksela. Uruchom program i sprawdź rezultat.  
*Uwaga!* Spojrzenie na powierzchnię wody od spodu da nam póki co niepoprawne rezultaty. Chcemy poprawnie rozpatrzyć również taki przypadek.
15. Zmodyfikuj shader pikseli tak, że jeśli kamera znajduje się pod powierzchnią wody (tnz.  $\langle N, V \rangle < 0$ ) musimy odwrócić wektor normalny, a przy liczeniu promienia załamanego użyć odwrotności współczynnika załamania (zauważ jednak, że współczynnik  $F_0$  pozostaje bez zmian niezależnie od tego z której strony patrzymy na powierzchnię).
16. Ze względu na to, że w tym przypadku współczynnik załamania jest większy od 1, funkcja `refract` może zwrócić pusty wektor (można to sprawdzić z użyciem funkcji `any`). Mamy w takim przypadku do czynienia z całkowitym wewnętrznym odbiciem, więc kolor piksela zależy będzie tylko od promienia odbitego. Uruchom program i sprawdź rezultaty zmian.
17. Zadeklaruj w shaderze piksli zmienną `perlin`, która zawierać będzie trójwymiarową teksturę (typ `texture3D`) szumu Perlina, którą wczytaliśmy w kroku 1.

18. Dodaj również do shadera zmienną `time` (tą samą, którą używaliśmy przy animacji sprężyny).

19. Wyznacz współrzędne tekstury. Dwie pierwsze będą równe współrzędnym  $x$  i  $z$  piksela w układzie lokalnym przemnożone przez 10. Trzecią współrzędną będzie wartość zmiennej `time`.

```
float3 tex = float3(i.localPos.xz*10.0f, time);
```

20. Z tekstuury `perlin` pobierz dwie wartości: pierwszą `ex` w punkcie `tex`, drugą `ez` w punkcie przesuniętym o 0.5 wzdłuż każdej współrzędnej. Tekstura zawiera wartości w skali szarości, należy więc wartości pobierać z kanału `r`. Następnie `ex` i `ez` należy przeskalać z zakresu  $[0, 1]$  do  $[-1, 1]$ .

```
ex = 2*ex - 1;
```

(Analogicznie dla `ez`)

21. Stwórz nowy wektor normalny, którego współrzędna  $x$  wynosi `ex`, współrzędna  $z$  wynosi `ez`, a współrzędna  $y$  wynosi 20. Po znormalizowaniu otrzymamy zaburzony wektor normalny, którego należy użyć do dalszych obliczeń.

```
float3 N = normalize(float3(ex, 20.0f, ez));
```

## 5 Model cieniowania PBR

Model cieniowania PBR (ang. *physically based rendering*), w odróżnieniu od empirycznego modelu cieniowania Phonga, jest pewnym przybliżeniem fizycznego zachowania światła. Kosztem większej złożoności obliczeniowej — która dla dzisiejszych kart graficznych nie stanowi jednak większego problemu — oferuje on bardziej realistyczne rezultaty. Ponadto parametry modelu są prostsze i bardziej intuicyjne, co ułatwia odwzorowanie rzeczywistych materiałów.

Materiał powierzchni opisują następujące współczynniki:

- $r \in [0, 1]$  - szorstkość (ang. *roughness*)
- $m \in [0, 1]$  - metaliczność (ang. *metalness*)
- $C_A \in [0, 1]^3$  - albedo (*bazowy kolor* powierzchni)

Albedo często definiowane jest jako kolor zapisany w przestrzeni sRGB, gdzie wartości kanałów odpowiadają postrzeganej jasności, co nie odpowiada liniowo fizycznej luminancji. Dlatego musimy poddać wartość  $C_A$  korekcji gamma (*gamma decoding*):

$$A = C_A^{2.2} \quad (8)$$

Dla uproszczenia założymy, że scena oświetlona jest przez  $k$  światel punktowych.  $i$ -te światło opisane jest przez:

- $\mathbf{p}_l^i$  - położenie
- $I_l^i$  - intensywność, tj. ilość światła emitowanego w jednostce czasu przez jednostkowy kąt bryłowy (nie operujemy tu postrzeganą jasnością światła, czy też poszczególnych kanałów RGB, lecz na fizycznej mierze strumienia promieniowania elektromagnetycznego)

Mając dany pewien punkt  $\mathbf{p}$  na powierzchni obiektu, w którym wektor normalny wynosi  $\mathbf{n}$ , chcemy wyznaczyć intensywność światła odbitego przez pewien jednostkowy obszar wokół  $\mathbf{p}$  (odpowiadający jednemu pikselowi) w kierunku obserwatora  $\mathbf{v}$ . W naszym przypadku będzie to suma ilości światła odbitego w kierunku  $\mathbf{v}$  z każdego źródła, co wyrazić ogólnie można wzorem:

$$I_o(\mathbf{p}, \mathbf{n}, \mathbf{v}) = \sum_{i=1}^k f_{BRDF}(\mathbf{n}, \mathbf{v}, \mathbf{l}_i) L_i(\mathbf{p}, \mathbf{n}) \quad (9)$$

gdzie  $\mathbf{l}_i$  to wektor z punktu  $\mathbf{p}$  w kierunku  $i$ -tego światła:

$$\mathbf{l}_i = \frac{\mathbf{p}_l^i - \mathbf{p}}{\|\mathbf{p}_l^i - \mathbf{p}\|} \quad (10)$$

$L_i$  to tzw. *radiancja*, czyli ilość światła padającego na jednostkowy obszar wokół  $\mathbf{p}$  z  $i$ -tego źródła. Zależy ona od intensywności źródła, kąta pomiędzy wektorem normalnym i wektorem do światła i jest odwrotnie proporcjonalna do kwadratu odległości:

$$L_i(\mathbf{p}, \mathbf{n}) = I_l^i \frac{\max(\langle \mathbf{n}, \mathbf{l}_i \rangle, 0)}{\|\mathbf{p}_l^i - \mathbf{p}\|^2} \quad (11)$$

Zgodnie z prawami fizyki, ilość światła odbijanego we wszystkich kierunkach nie może przekroczyć iradiancji (tj. sumy radiancji ze wszystkich źródeł światła). Ilość światła odbitego w kierunku obserwatora musimy więc odpowiednio przeskalać. Służy do tego funkcja  $f_{BRDF}$  (dwukierunkowa funkcja rozkładu odbicia, ang. *Bi-directional Reflectance Distribution Function*).

Skorzystamy z modelu Cooka-Torrence'a, który szorstkie powierzchnie aproksymuje za pomocą modelu mikro-luster. Dla powierzchni idealnie gładkiej (tudzież jednego mikro-lustra) padające światło dzielone jest na światło załamane, które wnika wgłąb powierzchni, oraz odbicie zwierciadlane. Część światła załamanej opuszcza powierzchnię w postaci odbicia rozproszonego, które ma jednakową intensywność we wszystkich kierunkach. Światło odbite (odbicie zwierciadlane) wędruje w kierunku promienia odbitego od źródła, tzn. jest widoczne tylko wtedy gdy wektor ten pokrywa się z wektorem do kamery (lub też wektor połówkowy  $\mathbf{h}$  – równanie 12 — jest równy wektorowi normalnemu do powierzchni).

$$\mathbf{h}(\mathbf{v}, \mathbf{l}) = \frac{\mathbf{v} + \mathbf{l}}{\|\mathbf{v} + \mathbf{l}\|} \quad (12)$$

Jeżeli powierzchnia nie jest idealna, odbicie rozproszone się nie zmienia (orientacja mikro-luster nie ma wpływu na ilość światła rozproszonego emitowanego w danym kierunku). Natomiast ilość światła odbitego w kierunku obserwatora zależy od tego jaka część mikro-luster ma wektory normalne zgodne z wektorem połkowym  $\mathbf{h}$ . Ostatecznie funkcja  $f_{BRDF}$  ma postać:

$$f_{BRDF}(\mathbf{n}, \mathbf{v}, \mathbf{l}) = k_d(\mathbf{v}, \mathbf{l}) f_L + f_{CT}(\mathbf{n}, \mathbf{v}, \mathbf{l}) \quad (13)$$

Jaka część światła podlega odbiciu rozproszonemu zależy od wartości współczynnika Fresnala. Można go policzyć, podobnie jak to robiliśmy przy renderowaniu wody, korzystając z aproksymacji Schlicka:

$$F(\mathbf{n}, \mathbf{l}) = F_0 + (1 - F_0) (1 - \langle \mathbf{n}, \mathbf{l} \rangle)^5 \quad (14)$$

W porównaniu do równania 7 z poprzedniej części zadania współczynnik  $F_0$  (jak i wartość funkcji  $F$  oraz współczynnik  $k_d$ ) ma trzy elementy (po jednym dla składowej R, G i B). Jest nam to potrzebne, gdyż powierzchnie metaliczne mają różnych współczynników Fresnala dla różnych długości fal. Wartość współczynnika  $F_0$  dana jest wzorem:

$$F_0 = \begin{bmatrix} 0.04 \\ 0.04 \\ 0.04 \end{bmatrix} (1 - m) + Am \quad (15)$$

Jak łatwo zauważyc dla powierzchni niemetalicznych, wszystkie elementy  $F_0$  wynoszą 0.04. Jest to dobre przybliżenie dla materiałów dielektrycznych, dla których wartości zazwyczaj oscylują w granicach  $0.03 \sim 0.05$  (istnieją wyjątki np. diament, dla którego  $F_0$  wynosi 0.17). Powierzchni metalicznej, współczynniki Fresnala będą równe albedo. Tak więc w powyższym wzorze robimy interpolację liniową pomiędzy wartością 0.04 a albedo materiału ( $A$ , równanie 8) na podstawie stopnia metaliczności powierzchni  $m$ .

Ilość światła załamanego  $(1 - F(\mathbf{n}, \mathbf{l}))$ , która opuszcza powierzchnię w postaci odbicia rozproszonego również zależy od typu powierzchni. Metale pochłaniają takie światło w całości (nie mają odbicia rozproszonego), dielektryki rozpraszą je w całości. Stąd wzór na współczynnik odbicia rozproszonego  $k_d$  ma postać:

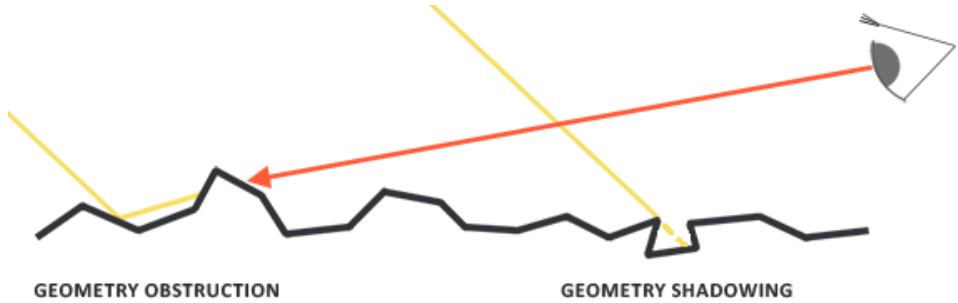
$$k_d(\mathbf{n}, \mathbf{l}) = (1 - F(\mathbf{n}, \mathbf{l})) (1 - m) \quad (16)$$

Tą wartość musimy przemnożyć jeszcze przez współczynnik skalujący, który powie nam ile światła rozproszonego przechodzi przez jednostkowy kąt bryłowy (wg. modelu Lambertego):

$$f_L = \frac{A}{\pi} \quad (17)$$

Pozostaje nam do policzenia ile światła odbitego skierowane jest do obserwatora. Ilość ta zależy od tego, ile mikro-luster ma wektor zgodny z wektorem połkowym  $\mathbf{h}$ .

kowym. Ponadto część tego światła może być zasłonięta przez inne mikro-lustra (tzw. efekt samo-zasłaniania geometrii powierzchni), co ilustruje rysunek 1:



Rysunek 1: Samo-zasłanianie geometrii powierzchni

Zgodnie z modelem Cooka-Torrence'a funkcja licząca ilość światła zwierciadlanego odbitego w kierunku obserwatora ma postać:

$$f_{CT}(\mathbf{n}, \mathbf{v}, \mathbf{l}) = \frac{F(\mathbf{h}(\mathbf{v}, \mathbf{l}), \mathbf{l}) D(\mathbf{n}, \mathbf{v}, \mathbf{l}) G(\mathbf{n}, \mathbf{v}, \mathbf{l})}{4 \max(\langle \mathbf{n}, \mathbf{v} \rangle, 0) \max(\langle \mathbf{n}, \mathbf{l} \rangle, 0)} \quad (18)$$

W porównaniu do równania 16, liczymy współczynnik Fresnela tylko da tych mikro-luster, dla których wektor normalny jest zgodny z wektorem połówkowym  $\mathbf{h}$  (równanie 12). Stąd występuje on we wzorze zamiast makroskopowego wektora normalnego powierzchni.

$D$  jest funkcją rozkładu mikro-luster, tzn. określa ona jaka część powierzchni pokryta jest lustrami o wektorze normalnym zgodnym z wektorem połówkowym. Zależy ona od szorstkości powierzchni  $r$  i możemy ją wyznaczyć np. z modelu Trowbridge'a-Reitz'a (znanego też pod nazwą GGX):

$$D(\mathbf{n}, \mathbf{v}, \mathbf{l}) = \frac{r^2}{\pi \left( \max(\langle \mathbf{n}, \mathbf{h}(\mathbf{v}, \mathbf{l}) \rangle, 0)^2 (r^2 - 1) + 1 \right)^2} \quad (19)$$

Współczynnik samo-zasłaniania wyznaczamy z modelu Smitha. Jak widać z rysunku 1 mamy do czynienia z dwoma jego rodzajami. W pierwszym przypadku promień już odbity może trafić w inny fragment powierzchni (obserwator jest przezeń zasłonięty). Prawdopodobieństwo jest tym większe im większy jest kąt pomiędzy wektorem normalnym powierzchni (w ujęciu makroskopowym) a promieniem do obserwatora. Drugi przypadek występuje, gdy promień natrafi na inny fragment geometrii powierzchni zanim odbije się w danym mikro-lustrze (tzn. znajduje się ono w cieniu innego fragmentu powierzchni). Prawdopodobieństwo jest tym większe im większy jest kąt pomiędzy wektorem normalnym powierzchni a promieniem do źródła światła.

Wartość współczynnika wyznaczamy więc jako iloczyn dwóch rozkładów:

$$G(\mathbf{n}, \mathbf{v}, \mathbf{l}) = G_s(\mathbf{n}, \mathbf{v}) G_s(\mathbf{n}, \mathbf{l}) \quad (20)$$

Natomiast funkcję pomocniczą  $G_s$  wyznaczyć możemy np. z połączenia modeli Schlicka-Beckmanna oraz GGX:

$$G_s(\mathbf{n}, \mathbf{w}) = \frac{\max(\langle \mathbf{n}, \mathbf{w} \rangle, 0)}{\max(\langle \mathbf{n}, \mathbf{w} \rangle, 0)(1 - q) + q} \quad (21)$$

Współczynnik  $q$  wprowadzony jest w celu uproszczenia wzoru i zależy od szorstkości powierzchni  $r$ :

$$q = \frac{(r + 1)^2}{8}$$

Mamy już wszystkie potrzebne elementy do wyznaczenia intensywności dla piksela (ze wzoru 9). Przed zapisaniem wyniku w buforze koloru musimy przekształcić intensywność na jasność piksela, wykonując korekcję gamma (*gamma encoding*):

$$C_o = \sqrt[2.2]{I_o} \approx I_o^{0.4545} \quad (22)$$

Wykonanie:

1. W konstruktorze `ShaderDemo` zmień definicję zmiennych `lightColor` oraz `lightPos` i dodaj parametry: `albedo`, `metalness`, `roughness`:

```
XMFLOAT4 lightPos[2] = {
    { -1.f, 0.0f, -3.5f, 1.f },
    { 0.f, 3.5f, 0.0f, 1.f } };
XMFLOAT3 lightColor[2] = {
    { 12.f, 9.f, 10.f },
    { 1.f, 0.f, 30.f } };
m_variables.AddGuiVariable("lightPos",
    lightPos, -10, 10);
m_variables.AddGuiVariable("lightColor",
    lightColor, 0, 100, 1);
m_variables.AddGuiColorVariable("albedo",
    XMFLOAT3{ 1.f, 1.f, 1.f });
m_variables.AddGuiVariable("metallic", 1.0f);
m_variables.AddGuiVariable("roughness", .3f, .1f);
```

Możesz też usunąć zmienne `ks`, `kd`, `ka` i `surfaceColor`.

2. Do shadera pikseli czajnika (`teapotPS.hls1`) dodaj odpowiadające zmienne `lightColor`, `lightPos`, `albedo`, `metalness` i `roughness`. Usuń definicję i wywołanie funkcji `phong` oraz zmienne `ks`, `kd`, `ka` i `surfaceColor`.
3. W funkcji `main` shadera pikseli wykonaj korekcję gamma wartości albedo (równanie 8) i wyznacz wartość współczynnika `F0` (równanie 15).
4. Następnie dla każdego światła wyznacz:
  - wektor do światła  $\mathbf{l}_i$  (równanie 10),



Rysunek 2: Rozkład wektorów normalnych mikro-luster dla  $r = 0, 3$

- radiancje  $L_l^i$  (równanie 11),
- wektor połówkowy (równanie 12).

5. Stwórz funkcję `normalDistributionGGX` wyznaczającą rozkład mikroluster o wektorach normalnych zgodnych z wektorem połówkowym (wg. modelu GGX, równanie 19).

Możesz ją przetestować zwracając w funkcji `main` shadera:

```
float NDF =
    normalDistributionGGX(N, H, roughness);
return float4(NDF, NDF, NDF, 1.0f);
```

, gdzie `H` jest wektorem połówkowym (równanie 12) dla pierwszego źródła światła, a `roughness` współczynnikiem szorstkości. Uruchom program i porównaj rezultat z rysunkiem 2.

6. Stwórz funkcję pomocniczą `geometrySchlickGGX` wyznaczania współczynnika samozasłaniania geometrii mikroluster (kombinacja rozkładu GGX oraz modelu Schlicka-Beckmanna, równanie 21).

Możesz ją przetestować zwracając w funkcji `main` shadera (wektor `L` to kierunek do pierwszego źródła światła  $l_i$ ):

```
float GGX = geometrySchlickGGX(
    max(dot(N, L), 0.0f), roughness);
return float4(GGX, GGX, GGX, 1.0f);
```

Uruchom program i porównaj rezultat z rysunkiem 3.



Rysunek 3: Funkcja samo-zacienienia geometrii powierzchni dla  $r = 0,3$

7. Stwórz funkcję **geometrySmith** wyznaczania współczynnika samozasłaniania geometrii (uwzględniając samo-zacienianie i okluzję geometrii mikroluster), korzystając z funkcji **geometrySchlickGGX** (równanie 20).

Możesz ją przetestować zwracając w funkcji **main** shadera (ponownie użyj wektora do pierwszego światła **L**):

```
float G = geometrySmith(N, V, L, roughness);
return float4(G, G, G, 1.0f);
```

Uruchom program i porównaj rezultat z rysunkiem 4.



Rysunek 4: Funkcja samo-zasłaniania geometrii powierzchni dla  $r = 0,3$

8. Stwórz funkcję **fresnel** wyznaczającą z równania 14 współczynnik odbicia

zwierciadlanego. Porównaj z podobną funkcją którą stworzyliśmy przy animacji powierzchni wody. Zauważ, że współczynnik  $F_0$  i wynik są wektorami trójelementowymi (różny współczynnik załamania dla różnych składowych koloru).

9. W funkcji `main` dla każdego światła wyznacz:

- Współczynnik odbicia rozproszonego  $k_d$  (równanie 16, korzystając z funkcji `fresnel`).
- Współczynnik odbicia zwierciadlanego z dwukierunkowego rozkładu odbicia wg modelu Cooka-Torrence'a (równanie 18, funkcje `fresnel`, `normalDistributionGGX` oraz `geometrySmith`).
- Ostateczną wartość funkcji BRDF (równanie 13).
- Irradiancję przemnożoną przez wartość funkcji BRDF dodaj do wynikowej intensywności piksela (równanie 9).

10. Do wynikowej intensywności dodaj składową ambient:

```
float3 ambient = 0.03f * albedo;
```

11. Aby uzyskać wynikowy kolor piksela, dokonaj korekcji gamma wynikowej intensywności (równanie 22). Porównaj wynik z rysunkiem 5.



Rysunek 5: Cieniowanie PBR dla  $r = 0, 3; m = 1$

12. Analogiczne zmiany wprowadź w shaderze sprężyny (`springPS.hls1`).

## 6 Mapy PBR

Zamiast definiować wartości albedo, szorstkości i metaliczności dla całego modelu, możemy je pobrać z tekstur.

Wykonanie:

1. w konstruktorze `ShaderDemo` wczytaj tekstury albedo, szorstkości i metaliczności, np.:

```
m_variables.AddTexture(m_device, "albedoTex",
    "textures/rustediron2/albedo.png");
```

(Analogicznie dla `roughnessTex` i `metallicTex`).

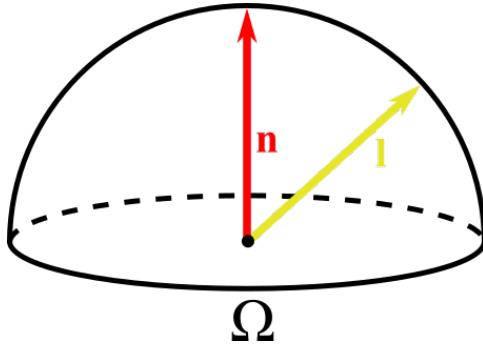
2. W shaderze pikseli czajnika `teapotPS.hls1` dodaj odpowiadające im tekstury `albedoTex`, `roughnessTex` i `metallicTex`.
3. Usuń zmienne `albedo`, `roughness` i `metallic`. Zamiast tego pobierz je z tekstur (nie zapomnij o korekcji gamma wartości albedo).
4. Dalsze obliczenia pozostają bez zmian. Analogiczne zmiany wprowadź w shaderze pikseli sprężyny `springPS.hls1`.
5. Z konstruktora `ShaderDemo` usunąć możesz parametry `albedo`, `roughness` i `metallic`, które nie są już używane.

## 7 Mapy oświetlenia PBR

Bardziej naturalnie wyglądający obraz uzyskamy, jeżeli zamiast danego zbioru pojedynczych światel na kolor powierzchni obiektu wpływ będzie miało jego całe otoczenie. W tym celu zastosować możemy podejście podobne do mapy środowiska. Założymy więc ponownie, iż otoczenie jest dość odległe od obiektu. Dzięki temu możemy przyjąć, że wektor w kierunku danego punktu otoczenia nie zależy od położenia na powierzchni obiektu.

Obliczenia wykonywać będziemy na podstawie sześciennej mapy środowiska przechowującej informacje o intensywności światła dochodzącego z danego kierunku. Zapisane w niej natężenie przychodzące zawsze będzie w sobie tłumienie wynikające z odległości do źródła światła (por. 11). Mapę taką stworzyć można np. przez renderowanie do tekstuury, lub skorzystać ze zdjęć rzeczywistej sceny, tak jak to zrobimy w tym zadaniu.

Wyznaczenie z modelu PBR intensywności światła odbitego w kierunku kamery wymagałoby policzenia całki po wszystkich kierunkach, których kąt do wektora normalnego jest mniejszy niż  $\frac{\pi}{2}$



danej wzorem:

$$I_o(\mathbf{n}, \mathbf{v}) = \int_{\Omega} (k_d(\mathbf{n}, \mathbf{v}) f_L + f_{CT}(\mathbf{n}, \mathbf{v}, \mathbf{l})) I_i(\mathbf{l}) \langle \mathbf{n}, \mathbf{l} \rangle d\mathbf{l}$$

Dla przypomnienia:

- $k_d$  – współczynnik odbicia rozproszonego — dla uproszczenia przyjmiemy, że liczony jest dla kierunku wektora odbitego, przekazując  $v$  jako drugi parametr (por. równanie 16),
- $f_L$  – funkcja rozkładu odbicia rozproszonego modelu Lambert'a, równanie 17,
- $f_{CT}$  – funkcja rozkładu odbicia zwierciadlanego modelu Cooka-Torrence'a, równanie 18,
- $\mathbf{n}$  – wektor normalny do powierzchni obiektu,
- $\mathbf{v}$  – wektor w kierunku kamery,
- $I_i$  – intensywność światła padającego z danego kierunku (pobrała z mapy środowiska).
- $A$  – albedo powierzchni, równanie 8

Takie obliczenia są oczywiście zbyt kosztowne, aby wykonywać je wewnątrz shadera pikseli. Zauważmy jednak, że powyższy wzór możemy rozbić na sumę odbicia rozproszonego i zwierciadlanego

$$I_o(\mathbf{n}, \mathbf{v}) = I_d(\mathbf{n}, \mathbf{v}) + I_s(\mathbf{n}, \mathbf{v}) \quad (23)$$

Wzór na pierwszy składnik przekształcić można następująco:

$$I_d(\mathbf{n}, \mathbf{v}) = \int_{\Omega} k_d(\mathbf{n}, \mathbf{v}) \frac{A}{\pi} I_i(\mathbf{l}) \langle \mathbf{n}, \mathbf{l} \rangle d\mathbf{l} = k_d(\mathbf{n}, \mathbf{v}) A \int_{\Omega} I_i(\mathbf{l}) \frac{\langle \mathbf{n}, \mathbf{l} \rangle}{\pi} d\mathbf{l}$$

Wartość powyższej całki jest funkcją wyłącznie wektora normalnego. Możemy więc przygotować teksturę sześcienną, której teksele zawierają wartości całki

policzone dla odpowiadających im kierunków. Tak przygotowana tekstura zwana jest *mapą irradiancji* i powstaje poprzez odpowiednie rozmycie mapy środowiska. Dzięki temu powyższy wzór sprowadza się do:

$$I_d = k_d(\mathbf{n}, \mathbf{v}) A I_{ir}(\mathbf{n}) \quad (24)$$

, gdzie  $I_{ir}$  jest mapą irradiancji.

Drugi składnik we wzorze 23 odpowiadający intensywności odbicia zwierciadlanego jest dużo bardziej skomplikowany:

$$I_s(\mathbf{n}, \mathbf{v}) = \int_{\Omega} f_{CT}(\mathbf{n}, \mathbf{v}, \mathbf{l}) I_i(\mathbf{l}) \langle \mathbf{n}, \mathbf{l} \rangle d\mathbf{l} \quad (25)$$

Wartość całki zależy tutaj od dwóch wektorów, więc nie możemy zapisać wyników w tekście. Zamiast tego przybliżymy sobie wartość całki poprzez iloczyn dwóch prostszych:

$$\begin{aligned} & \int_{\Omega} f_{CT}(\mathbf{n}, \mathbf{v}, \mathbf{l}) I_i(\mathbf{l}) \langle \mathbf{n}, \mathbf{l} \rangle d\mathbf{l} \approx \\ & \approx \int_{\Omega} f_{CT}(\mathbf{n}, \mathbf{n}, \mathbf{l}) I_i(\mathbf{l}) \langle \mathbf{n}, \mathbf{l} \rangle d\mathbf{l} \int_{\Omega} f_{CT}(\mathbf{n}, \mathbf{v}, \mathbf{l}) I_0 \langle \mathbf{n}, \mathbf{l} \rangle d\mathbf{l} \approx \quad (26) \\ & \approx \int_{\Omega} D(\mathbf{n}, \mathbf{n}, \mathbf{l}) I_i(\mathbf{l}) \langle \mathbf{n}, \mathbf{l} \rangle d\mathbf{l} \int_{\Omega} \frac{FDG}{4 \langle \mathbf{n}, \mathbf{v} \rangle} d\mathbf{l} \end{aligned}$$

Pierwsza z całek reprezentuje rozmycie mapy środowiska zgodnie z rozkładem mikro-luster (równanie 19). Dla uproszczenia na potrzeby policzenia tej całki zakładamy, że wektory normalny oraz do kamery (a więc i wektor odbity) są równe. Możemy stworzyć więc teksturę sześcienną, której teksele zawierają wartości całki policzone dla odpowiadających im kierunków, na kolejnych poziomach mip-mapy zaapisując wyniki dla coraz większych wartości współczynnika szorstkości powierzchni.

Zauważ, że dla szorstkości równej 0, taka mapa będzie identyczna z mapą środowiska, tak więc wystarczy ją rozszerzyć o mip-mapy przefiltrowane dla większych wartości szorstkości.

Druga z całek w iloczynie we wzorze 26, to tak naprawdę wzór 25 przy założeniu, że intensywność światła przychodząca z każdego kierunku jest jednakowa i wynosi  $I_0 = 1$ . Dzięki temu założeniu, wynik nie zależy konkretnych kierunków wektorów normalnego i do kamery, lecz tylko od ich iloczynu skalarnego<sup>1</sup>.

Wzór zawiera niestety jeszcze dwie zmienne:  $F_0$  ukryte w funkcji  $F$  oraz szorstkość  $r$  użytą w funkcjach  $D$  i  $G$ . Pierwszą z nich możemy jednak wyeliminować korzystając z równania 14:

$$\begin{aligned} \int_{\Omega} \frac{FDG}{4 \langle \mathbf{n}, \mathbf{v} \rangle} d\mathbf{l} &= \int_{\Omega} (F_0(1 - \alpha) + \alpha) \frac{DG}{4 \langle \mathbf{n}, \mathbf{v} \rangle} d\mathbf{l} = \\ &= F_0 \int_{\Omega} (1 - \alpha) \frac{DG}{4 \langle \mathbf{n}, \mathbf{v} \rangle} d\mathbf{l} + \int_{\Omega} \alpha \frac{DG}{4 \langle \mathbf{n}, \mathbf{v} \rangle} d\mathbf{l} \end{aligned}$$

---

<sup>1</sup>Możemy przyjąć, że obliczenia wykonywane są w układzie, gdzie wektor normalny wynosi  $\mathbf{n} = [0, 0, 1]^T$ , a wektor do kamery  $\mathbf{v} = [\sqrt{1 - \cos^2 \theta}, 0, \cos \theta]^T$ , gdzie  $\cos \theta$  to wspomniany iloczyn skalarny wektorów  $\mathbf{n}$  i  $\mathbf{v}$ . Dokładna orientacja takiego układu nie ma znaczenia, gdyż oświetlenie otoczenia jest jednorodne

$$\alpha = (1 - \langle \mathbf{h}, \mathbf{l} \rangle)^5$$

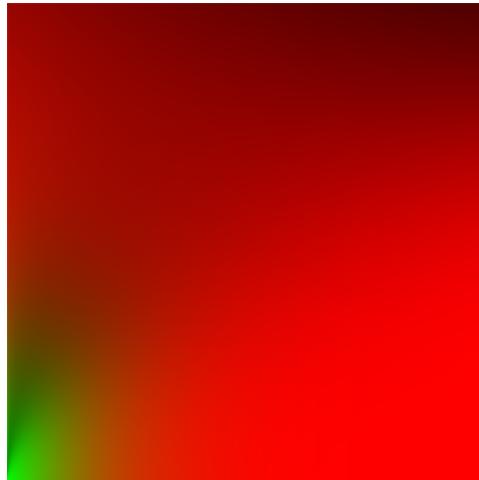
Możemy więc stworzyć teksturę adresowaną wartością iloczynu skalarnego  $\langle \mathbf{n}, \mathbf{v} \rangle$  oraz współczynnika szorstkości  $r$ , która stablicuje powyższy wzór w postaci czynnika skali:

$$f_m (\langle \mathbf{n}, \mathbf{v} \rangle, r) = \int_{\Omega} (1 - \alpha) \frac{DG}{4 \langle \mathbf{n}, \mathbf{v} \rangle} d\mathbf{l}$$

oraz offsetu:

$$f_o (\langle \mathbf{n}, \mathbf{v} \rangle, r) = \int_{\Omega} \alpha \frac{DG}{4 \langle \mathbf{n}, \mathbf{v} \rangle} d\mathbf{l}$$

zapisanych w dwóch kanałach koloru. Wyniki nie zależy ani od otoczenia ani od cienionanego obiektu, a jej zawartość demonstruje rysunek 6.



Rysunek 6: Stabilowana funkcja BRDF odbicia zwierciadlanego

Podsumowując, wzór 26 sprowadza się do:

$$I_s (\mathbf{n}, \mathbf{v}) = I_i (\mathbf{n}, r) (F_0 f_m (\langle \mathbf{n}, \mathbf{v} \rangle, r) + f_o (\langle \mathbf{n}, \mathbf{v} \rangle, r)) \quad (27)$$

, gdzie:

- $I_i$  – intensywność światła pochodząca z przefiltrowanej mapy środowiska (współrzędne tekstury dane są wektorem  $n$ , a poziom mip-mapy wyznaczony przez współczynnik  $r$ )
- $F_0$  – stała materiału (równanie 15).
- $f_m$  – współczynnik skalowania pobrany z pierwszego kanału tekstuury BRDF
- $f_o$  – offset pobrany z drugiego kanału tekstuury BRDF

Przefiltrowaną mapę środowiska (jak i odpowiadającą jej mapę iradiancji potrzebną do wyznaczenia odbicia rozproszonego) oraz teksturę funkcji BRDF odbicia zwierciadlanego znaleźć można w katalogu projektu.

Więcej o modelu oświetlenia PBR i generowaniu prefiltrowanych map środowiska poczytać można [tutaj](#), [tutaj](#) i [tutaj](#). Do generowania tekstur na postawie szerszych map otoczenia można posłużyć się też gotowymi narzędziami takimi jak [cmftStudio](#) lub [IBLBaker](#).

Wykonanie:

1. W konstruktorze klasy `ShaderDemo` wczytaj teksturę iradiancji `irMap`, przefiltrowaną mapę środowiska `pfEnvMap` oraz teksturę funkcji BRDF odbicia zwierciadlanego `brdfTex`

```
m_variables.AddTexture(m_device, "irMap",
    L"textures/cubeMapIrradiance.dds");
m_variables.AddTexture(m_device, "pfEnvMap",
    L"textures/cubeMapRadiance.dds");
m_variables.AddTexture(m_device, "brdfTex",
    L"textures/brdf_lut.png");
```

2. W shaderze pikseli czajnika `teapotPS.hls1` dodaj odpowiadające tym teksturom zmienne `irMap` i `pfEnvMap` typu `textureCUBE` oraz `brdfTex` typu `texture2D`.
3. W funkcji `main` pobierz wartość iradiancji `Iir` podając jako współrzędne tekstury wektor normalny

```
float3 Iir = irMap.Sample(samp, N).rgb;
```

Wyznacz intensywność odbicia rozproszonego ze wzoru 24. Dla przypomnienia albedo  $A$  wyznaczyliśmy wcześniej w kroku 3 części 5. Natomiast współczynnik odbicia zwierciadlanego  $k_d$  liczymy ze wzoru 16 dla wektora odbitego. Ponieważ jednak iloczyny skalarne  $\langle \mathbf{n}, \mathbf{v} \rangle$  i  $\langle \mathbf{n}, \mathbf{r} \rangle$  są równe, możemy zamiast niego jako wektor do światła przekazać wektor do kamery  $\mathbf{v}$ .

4. Wyznacz wektor odbity  $\mathbf{R}$

```
float3 R = reflect(-V, N);
```

5. Pobierz wartość radiancji z przefiltrowanej mapy środowiska, podając wektor odbity jako współrzędne tekstury, oraz współczynnik szorstkości  $r$  pomnożony przez indeks najniższego poziomu mip-map (który dla naszej tekstury wynosi 6), jako poziom szczegółowości:

```
float3 Ii = pfEnvMap.SampleLevel(samp, R,
    roughness * 6.0f).rgb;
```

6. Pobierz wartości współczynników skali  $f_m$  oraz offsetu  $f_o$  z tekstuury `brdfTex` podając jako współrzędne iloczyn skalarny  $\langle \mathbf{n}, \mathbf{v} \rangle$  i współczynnik szorstkości  $r$ :

```
float2 brdf = brdfTex.Sample(samp,
    float2(max(dot(N, V), 0.0f), roughness)).rg;
```

$f_m$  znajduje się w kanale `r`,  $f_o$  w kanale `g` pobranego koloru.

7. Wyznacz intensywność odbicia zwierciadlanego ze wzoru 27. Stałą `F0` policzliśmy wcześniej w kroku 3 części 5.
8. Wynikową intensywność odbicia otoczenia (zgodnie ze wzorem 23) dodaj do zmiennej `ambient`, którą zdefiniowaliśmy w kroku 10 części 5.
9. Analogiczne zmiany wprowadź w shaderze pikseli sprężyny `springPS.hls1`.

## 8 Post-processing

### 8.1 Rozmycie gaussowskie

Do tej pory obiekty renderowane były od razu do okna programu. Zmienimy trochę ten schemat tworzenia obrazu. Wszystkie przebiegi, które zostały utworzone do tej pory przekierujemy tak, żeby zapisywały obraz do tekstuury o wymiarach identycznych z oknem podglądu. Oprócz tej tekstuury wykorzystamy dwie dodatkowe tekstuury, których każdy wymiar będzie o połowę mniejszy od wymiaru okna programu. Te tekstuury posłużą nam jako pamięć pomocnicza przy filtrowaniu obrazu zapisanego w dużej tekstuurze.

Filtr gaussowski jest przykładem filtru macierzowego: wynikowy kolor piksela obrazu powstaje przez pomnożenie wartości koloru pikseli w pewnym kwadracie naokoło niego przez macierz współczynników rzeczywistych.

Zaletą rozmycia gaussowskiego jest to, że jest separowalne. Identyczny efekt jak dla macierzy kwadratowej można uzyskać filtrując najpierw obraz w poziomie z odpowiednią macierzą o jednym wierszu, a następnie wynik tej operacji filtrując w pionie z użyciem transpozycji tego wiersza (separowalność polega na tym, że macierz kwadratowa filtru gaussowskiego powstaje z pomnożenia wektora kolumnowego przez identyczny wektor wierszowy).

Postprocessing obrazów przy pomocy shaderów odbywa się z reguły według następującego schematu. Renderowany jest pojedynczy kwadrat (dwa trójkąty), który wypełnia cały obszar okna widoku (tekstyury, do której zapisujemy wynik). Obraz, który filtrujemy jest tekstuurą nałożoną na ten kwadrat. W shaderze pikseli mamy współrzędne tekstuury odpowiadające dokładnie pozycji renderowanego piksela. Mając szerokość i wysokość pojedynczego piksela we współrzędnych tekstuury, możemy poruszać się po obszarze źródłowego obrazu, odczytywać wartości poszczególnych pikseli i liczyć na ich podstawie wynikowy kolor.

Dwie tekstury pomocnicze mają dwukrotnie mniejsze rozmiary od oryginalnego obrazu. Tak często postępuje się przy rozmywaniu obrazu w grafice czasu rzeczywistego, ponieważ wówczas otrzymuje się dwukrotnie większe rozmycie przy czterokrotnie mniejszej liczbie operacji. Dobra jest jeszcze dobierać współrzędne tekstury podczas próbkowania tak, aby trafić dokładnie w środek między dwójką (lub czwórką) teksceli. Wtedy w wyniku biliniowego filtrowania sprzętowego dostaniemy średni kolor dwóch (czterech) teksceli zamiast wartości pojedynczego tekscela i efekt rozmycia będzie większy.

Dodamy w sumie trzy przebiegi – w każdym renderowany będzie prostokąt wypełniający cały ekran. Pierwszy przebieg pobierze dane z dużej tekstury i zapisze do pierwszej z małych tekstur. Drugi pobierze dane z pierwszej z małych tekstur i zapisze wynik rozmycia w poziomie do drugiej z małych tekstur. Trzeci pobierze dane z drugiej z małych tekstur i wyświetli efekt rozmycia jej w pionie już bezpośrednio na ekranie.

Wykonanie:

1. W konstruktorze `ShaderDemo` stwórz trzy tekstury `screen`, `halfscreen1`, `halfscreen2`:

```
auto screenSize = m_window.getClientSize();
m_variables.AddRenderableTexture(m_device,
    "screen", screenSize);
```

Analogicznie dla pozostałych dwóch tekstur, lecz z wymiarami mniejszymi o połowę.

2. Dodaj parametr `viewportDim` przechowujący rozmiar okna:

```
m_variables.AddSemanticVariable("viewportDim",
    VariableSemantic::Vec2ViewportDims);
```

3. Dodaj zmienną `blurScale`, która kontrolować będzie wielkość rozmycia:

```
m_variables.AddGuiVariable("blurScale",
    1.0f, 0.1f, 2.0f);
```

4. Zdefiniuj sampler `blurSampler`. Zadbaj o to, by współrzędne tekstury były obcinane do brzegu (D3D11\_TEXTURE\_ADDRESS\_CLAMP), a filtrowanie było liniowe (D3D11\_FILTER\_MIN\_MAG\_MIP\_LINEAR):

```
SamplerDescription sDesc;
...
m_variables.AddSampler(m_device,
    "blurSampler", sDesc);
```

5. Do rysowania na całej powierzchni okna lub tekstury wykorzystamy model quad, którego używaliśmy przy renderowaniu wody. Ma on już odpowiednie

wymiary ( kwadrat o lewym boku w -1, prawym w 1 wypełni całą szerokość renderowanego obrazu; podobnie z wysokością), jednak musimy zamienić współrzędne Y i Z (kwadrat leży na płaszczyźnie XZ, a chcemy go ustawić na XY). Wyznaczyć musimy też współrzędne tekstury, zawężając wynikowe współrzędne XY z zakresu  $[-1, 1]$  do  $[0, 1]$ .

- Stwórz plik shadera wierzchołków `fullScreenQuadVS.hlsl` (skopiuj `waterVS.hlsl`).
- Usuń zmienne: `modelMtx`, `viewProjMtx`, `waterLevel`.
- Ze struktury `VSOutput` usuń pola `localPos` i `worldPos`. W zamian dodaj:

```
float2 tex : TEXCOORD0;
```

- W funkcji `main` policz położenie wierzchołka, które powinno mieć postać  $[X, Z, 0, 1]^T$
- Wyznacz również współrzędne tekstury postaci:  
 $[(X + 1)/2, (-Z + 1)/2]^T$

Powyższy shader wierzchołków wykorzystamy we wszystkich kolejnych przebiegach.

6. Stwórz shader pikseli dla pierwszego przebiegu `downsamplePS.hlsl`. Struktura otrzymywana na wejściu powinna być analogiczna do rezultatu shadera wierzchołków z poprzedniego punktu.
7. Dodaj do shadera `downsamplePS.hlsl` sampler `blurSampler` oraz teksturę `screen`.
8. W funkcji `main` shadera zwróć kolor pobrany z tekstury według współrzędnych tekstury wyznaczonych w shaderze wierzchołków
9. Funkcja `addPass` posiada dodatkowy parametr pozwalający zmienić bufor docelowy potoku renderowania. Zmiana ta jest zapamiętywana i wszystkie kolejne przebiegi też renderować będą do tego samego bufora, póki któryś nie zmieni go ponownie. Aby zapewnić renderowanie elementów sceny do tekstury w konstruktorze `ShaderDemo` zmodyfikuj istniejący przebieg renderowania pierwszego obiektu — czajnika —, tak by rysowały się one do tekstury `screen`:

```
auto passTeapot = addPass(L"teapotVS.cso",
    L"teapotPS.cso", "screen");
```

10. Na końcu konstruktora klasy `ShaderDemo` dodaj nowy przebieg rysowania `passDownsample`:

```

auto passDownsample = addPass(
    L"fullScreenQuadVS.cso", L"downsamplePS.cso",
    getDefaultRenderTarget());
addModelToPass(passDownsample, quad);

```

Aby sprawdzić, czy zawartość tekstyury jest poprawna, renderujemy ją chwilowo na ekran. Uruchom program i sprawdź rezultat.

11. W kolejnym przebiegu rysowani obraz rozmywać będziemy poziomo. Stwórz dla niego plik shadera pikseli hblurPS.hls1 (skopiuj downsamplePS.hls1)
12. Usuń teksturę `screen` i zamiast niej dodaj `halfscreen1`.
13. Dodaj tablicę wag filtru gaussowskiego:

```

static const float blurWeights[13] = {
    0.002216f,
    0.008764f,
    0.026995f,
    0.064759f,
    0.120985f,
    0.176033f,
    0.199471f,
    0.176033f,
    0.120985f,
    0.064759f,
    0.026995f,
    0.008764f,
    0.002216f
};

```

14. Dodaj do shadera zmienne `blurScale` (typu `float`) oraz `viewportDim` (typu `float2`).
15. W funkcji `main` zsumuj teksele sąsiadujące z obecnym w poziomie, przemnożone przez wagę `blurScale`:

```

float4 color = float4(0.0f, 0.0f, 0.0f, 0.0f);
float2 texelSize = blurScale * 2.0f / viewportDim;
for (int k = 0; k < 13; ++k)
    color += blurWeights[k] * halfscreen1.Sample(
        blurSampler, i.tex + float2(
            ((k - 6) * 2 - 0.5f) * texelSize.x,
            0.0f) );

```

16. W konstruktorze `ShaderDemo` zmodyfikuj przebieg `passDownsample`, tak by renderowany był on do tekstury `halfscreen1`. Poniżej dodaj kolejny przebieg `passHBlur`:

```
auto passHBlur = addPass(L"fullScreenQuadVS.cso",
    L"vblurPS.cso", getDefaultRenderTarget());
addModelToPass(passHBlur, quad);
```

Uruchom program i sprawdź rezultat.

17. Trzeci przebieg odpowiada za rozmycie w pionie. Stwórz dla niego shader pikseli `vblurPS.hls1` (skopiuj `hblurPS.hls1`). Zmień w nim źródłową teksturę z `halfscreen1` na `halfscreen2`.

18. Aby rozmycie odbyło się pionie, musimy w funkcji `main` shadera zamienić wyrażenie:

```
float2(((k - 6) * 2 - 0.5f) * texelSize.x, 0.0f)
```

na:

```
float2(0.0f, ((k - 6) * 2 - 0.5f) * texelSize.y)
```

19. W konstruktorze `ShaderDemo` zmodyfikuj przebieg `passHBlur`, tak by renderowany był do tekstury `halfscreen2`. Poniżej dodaj przebieg `passVBlur`:

```
auto passVBlur = addPass(L"fullScreenQuadVS.cso",
    L"vblurPS.cso", getDefaultRenderTarget());
addModelToPass(passVBlur, quad);
```

Uruchom program i sprawdź rezultat.

## 8.2 Filtr bloom

Nieznacznie zmodyfikujemy przebieg `passDownsample` tak, żeby zostawić tylko najjaśniejsze fragmenty obraz. W nowym przebiegu `passComposite` wynik rozmycia dodamy do pierwotnego obrazu tak, żeby uzyskać poświatę wokół najjaśniejszych fragmentów.

Wykonanie:

1. W konstruktorze `ShaderDemo` dodaj zmienną `cutoff`:

```
m_variables.AddGuiVariable("cutoff",
    0.72f, 0.1f, 1.0f);
```

2. Zmodyfikuj shader `downsamplePS.hls1`: dodaj zmienną `cutoff`, a funkcję `main` zmień tak by kopiowała z tekstury tylko te wartości, dla których jasność tekscela:

```
float luminance = dot(color.rgb,  
                      float3(0.3f, 0.58f, 0.12f));
```

jest większa niż wartość `cutoff`. W przeciwnym wypadku zwracać powinna kolor ze wszystkimi kanałami ustawionymi na 0:

```
return luminance < cutoff ?  
    float4(0.0f, 0.0f, 0.0f, 0.0f) : color;
```

Uruchom program i sprawdź rezultat.

3. Potrzebny jest nam ostatni przebieg, który połączy oryginalny obraz z tekstyry `screen` z wynikami powyższych działań. Stwórz dla niego shader pikseli `compositePS.hlsl` (kopią `downsamplePS.hlsl`). Usuń zmienną `cutoff` i dodaj teksturę `halfscreen1`.
4. W funkcji `main` shadera zwróć sumę kolorów pobranych z obu tekstur.
5. W konstruktorze `ShaderDemo` zmodyfikuj przebieg `passVBlur`, tak by rysowany był do tekstuury `halfscreen1`. Ze względu na to, że jest ona wypełniona (a właściwie wypełniony jest powiązany z nią bufor głębokości) musimy zapewnić jej wyczyszczenie przed wykonaniem przebiegu, czym steruje ostatni opcjonalny parametr funkcji `addPass`:

```
auto passVBlur = addPass(L"fullScreenQuadVS.cso",  
                          L"vblurPS.cso", "halfscreen1", true);
```

6. Poniżej dodaj przebieg `passComposite`:

```
auto passComposite = addPass(  
    L"fullScreenQuadVS.cso", L"compositePS.cso",  
    getDefaultRenderTarget());  
addModelToPass(passComposite, quad);
```

Uruchom program i sprawdź rezultat.

7. Jasne fragmenty obrazu mogą zostać prześwietlone. Aby zniwelować ten efekt w shaderze `compositePS.hlsl` zwracaną sumę można podzielić przez  $(1.0f + \text{bloom.a})$  (gdzie `bloom` to kolor pobrany z tekstuury `halfscreen1`).

## 9 Odbicia w przestrzeni ekranu

W dotychczasowym rozwiążaniu promienie załamanie i odbite ignorowały czajnik i sprężynę. Metoda wyznaczania odbić skomplikowanych siatek prezentowana w zadaniu z pokojem (mapowanie środowiska) zakładało, że lustrzana powierzchnia znajduje się daleko od odbijającej się geometrii, co nie jest prawdą w tym przypadku. Inna metoda z zadania z motyle umożliwiała odbicie w płaskim lustrze, jednakże powierzchnia wody jest pofaładowana. Wyznaczanie przejęć promieni w przestrzeni trójwymiarowej ma sens (pod względem wydajnościowym) tylko dla prostych modeli, takich jak sześcian otaczającym scenę.

Możemy jednak uprościć sobie zakres poszukiwań jeżeli zauważymy, że odbijające się powierzchnie są (większości) widoczne w innym miejscu na ekranie. Ewentualne punkty przecięć leżą na promieniu a ich obraz w buforze koloru powinien znaleźć się wzdłuż rzutu promienia na powierzchnię ekranu. Ponadto punkty widoczne na ekranie posiadają w buforze głębokości zakodowaną odległość od kamery (właściwie to wartość odwrotnie proporcjonalną do współrzędnej z w układzie kamery).

Idea metody polega więc na prześledzeniu pikseli wzdłuż rzutu promienia na ekran i porównywanie jego głębokości z wartościami zapisanymi w z-buforze. W pierwszym podejściu za punkt przecięcia uznamy pierwszy pikseli gdzie wartość w z-buforze będzie mniejsza.

Wykonanie:

1. W konstruktorze `ShaderDemo` dodaj zmienną `nearZ` — odległość do bliższej płaszczyzny obcinania:

```
m_variables.AddSemanticVariable("nearZ",
    VariableSemantic::FloatNearPlane);
```

2. Dodaj również dwie tekstury na kopię bufora koloru oraz bufora głębokości. Obie powinny mieć wymiary ekranu i jeden poziom mipmapy.

Format pikseli pierwszej tekstury będzie taki sam jak bufora koloru. W przypadku drugiej, domyślnie używany przez nas w buforze głębokości format `DXGI_FORMAT_D24_UNORM_S8_UINT` nie nadaje się dobrze dla tekstury wejściowej shadera pikseli. Dlatego w kopii użyjemy kompatybilnego formatu `DXGI_FORMAT_R24_UNORM_X8_TYPELESS`, który głębokość zwróci jako kanał czerwony, a kanał bufora szablonu zignoruje.

```
m_variables.AddTexture(m_device, "screenColor",
    Texture2DDescription(screenSize.cx, screenSize.cy,
        DXGI_FORMAT_R8G8B8A8_UNORM, 1));
m_variables.AddTexture(m_device, "screenDepth",
    Texture2DDescription(screenSize.cx, screenSize.cy,
        DXGI_FORMAT_R24_UNORM_X8_TYPELESS, 1));
```

3. Na potrzeby rysowania wody chcemy, aby w teksturach `screenColor` oraz `screenDepth` znalazły się kopie bufora koloru i głębokości po narysowaniu czajnika i sprężyny (lecz bez sześciianu). Upewnij się więc, że przebiegi renderowania dodawane są w odpowiedniej kolejności (tj. czajnik → sprężyna → woda → sześciian → ...).

Aby zapewnić kopiowanie do tekstur dodaj (tuż po stworzeniu przebiegu renderowania wody) następujące instrukcje:

```
copyRenderTarget(passWater, "screenColor");
copyDepthBuffer(passWater, "screenDepth");
```

4. W shaderze pikseli powierzchni wody (plik `waterPS.hlsl`) dodaj globalne deklaracje nowych zmiennych i tekstur:

```
texture2D screenColor;
texture2D screenDepth;
matrix viewProjMtx;
float2 viewportDim;
float nearZ;
static float maxDistance = 30.0f;
```

5. Dodaj definicję funkcji `screenSpaceRayCast` w której zaimplementujemy algorytm śledzenia promienia po powierzchni ekranu:

```
float4 screenSpaceRayCast(float3 wOrg, float3 wDir)
{
    ...
}
```

Parametrami funkcji będą początek `wOrg` i kierunek `wDir` promienia wyrażone w układzie sceny.

6. Wyznaczmy punkt końcowy promienia w odległości `maxDistance` wzdłuż promienia oraz rzuty obu krańców promienia na ekran, przekształcając je przez macierze widoku i kamery:

```
//world-space ray end
float3 wEnd = wOrg + wDir * maxDistance;
//NDC ray endpoints
float4 ssOrg = mul(viewProjMtx, float4(wOrg, 1.0f));
float4 ssEnd = mul(viewProjMtx, float4(wEnd, 1.0f));
```

7. Wzdłuż promienia na ekranie interpolować będziemy wartość zależną od odwrotności współrzędnej  $z$  w układzie kamery (po przekształceniu krańców znajdują się one w czwartej współrzędnej `ssOrg` i `ssEnd`). Jednakże jeżeli promień biegnie w kierunku kamery nie chcemy aby przeciął on płaszczyznę

$z = 0$  (występuje tam nieciągłość). Dlatego przyciąć go musimy do bliższej płaszczyzny obcinania:

```
if (ssEnd.w < nearZ) {
    float toNearZ = maxRayDistance *
        (nearZ - ssOrg.w) / (ssEnd.w - ssOrg.w);
    wEnd = wOrg + wDir * toNearZ;
    ssEnd = mul(viewProjMtx, float4(wEnd, 1.0f));
}
```

8. Samo mnożenie nie wystarczy, musimy jeszcze znormalizować współrzędne afiniczne. Podzielenie przez czwartą współrzędną dałoby nam wartość 1 we współrzędnej  $w$  wynikowego wektora. Jako że ten element wektora nie będzie wykorzystywany, użyjemy go jednak do przechowania odwrotności  $z$  punktu w układzie kamery, która przyda nam się później.

```
ssOrg = float4(ssOrg.xyz, 1.0f) / ssOrg.w;
ssEnd = float4(ssEnd.xyz, 1.0f) / ssEnd.w;
```

9. Wynikowe współrzędne  $xy$  punktów na ekranie wyrażone są w NDC (zakres  $[-1, 1] \times [-1, 1]$ , oś  $Y$  skierowana ku górze), nam natomiast będą potrzebne współrzędne piksela. Musimy je więc przekształcić (przesunąć o 1, podzielić przez 2, pomnożyć przez wymiary tekstuury i odwrócić  $y$ ):

```
ssOrg.xy = (float2(ssOrg.x, -ssOrg.y) + 1.0f) *
    viewportDim / 2.0f;
ssEnd.xy = (float2(ssEnd.x, -ssEnd.y) + 1.0f) *
    viewportDim / 2.0f;
```

10. Do śledzenia promienia po powierzchni ekranu pomiędzy punktami `ssOrg` i `ssEnd` użyjemy algorytmu DDA. Przy liniowej interpolacji na powierzchni obrazu, wartości z przestrzeni trójwymiarowej interpolować musimy proporcjonalnie do zmiany odwrotności współrzędnej  $z$  w układzie kamery. Na szczęście dzięki normalizacji w punkcie 8 mamy już to zrobione.

```
float2 delta = (ssEnd.xy - ssOrg.xy);
bool coordSwap = false;
if (abs(delta.x) < abs(delta.y))
{
    //DDA will iterate vertically, swap coordinates
    delta = delta.yx;
    ssOrg.xy = ssOrg.yx;
    ssEnd.xy = ssEnd.yx;
    coordSwap = true;
}
```

```

//Iteration direction:
float stepDir = sign(delta.x);
float4 dP = stepDir * (ssEnd - ssOrg) / delta.x;
float4 P = ssOrg;
float endX = stepDir * ssEnd.x;
for (; (P.x * stepDir) < endX; P += dP)
{
    ...
}

```

11. W pętli pobrać możemy wartość z bufora głębokości w punkcie `P.xy`, porównać ją z głębokością promienia (`P.z` przesunięte o pół kroku wzdłuż promienia). Jeżeli promień jest za powierzchnią, zwróćmy kolor tego punktu.

Ze względu na ograniczenia języka HLSL nie możemy odczytywać danych tekstury znaną nam funkcją `Sample` (filtrowanie tekstury wymaga informacji z sąsiednich pikseli — aby zapewnić że są one dostępne, kompilator musiałby rozwinąć pętlę, co w przypadku naszego algorytmu nie jest możliwe). Odczyt danych bez filtrowania umożliwia metoda `Load`, która przyjmuje parametr typu `int3` (współrzędne piksela i poziom mipmapy). Jak łatwo zauważyc podać musimy współrzędne jako indeksy całkowite (tak jakbyśmy adresowali tablicę dwuwymiarową), a nie używane zwykle zmiennopozycyjne współrzędne tekstury znormalizowane do zakresu  $[0, 1]$ . To wyjaśnia, czemu w punkcie 9 skalowaliśmy współrzędne do wymiarów obrazu.

Jeśli nie znajdziemy przecięcia wzdłuż długości zwróćmy kolor przezroczysty

```

for (; (P.x * stepDir) < endX; P += dP)
{
    int3 pxCoord = int3(coordSwap ? P.yx : P.xy, 0);
    float screenZ = screenDepth.Load(pxCoord).r;
    float rayZ = P.z + 0.5f * dP.z;
    if (screenZ < rayZ)
        return screenColor.Load(pxCoord);
}
return float4(0.0f, 0.0f, 0.0f, 0.0f);

```

12. W funkcji `main` shadera możemy użyć naszej nowej funkcji do znalezienia koloru dla promienia odbitego, który należy zmieszać z kolorem odbitym `reflColor` z mapy środowiska. Pamiętaj, że początek promienia jak i kierunek wyrażone powinny być w układzie sceny.

```

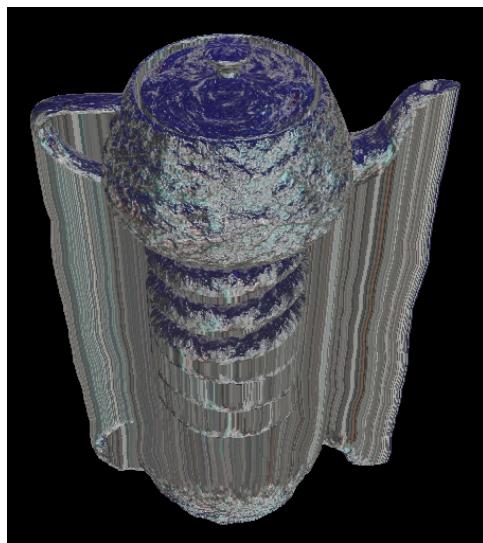
float4 ssReflColor =
    screenSpaceRayCast(i.worldPos, reflDir);
reflColor = lerp(reflColor,
    ssReflColor.rgb, ssReflColor.a);

```

13. Podobnie możemy postąpić z promieniem załamanym:

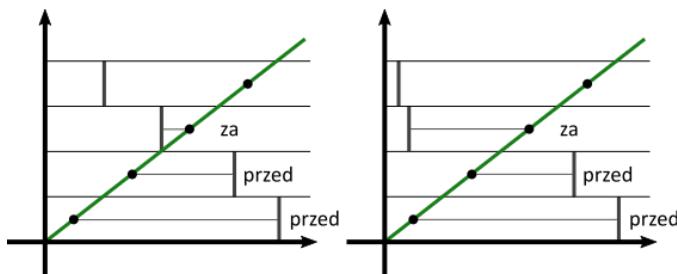
```
float4 ssRefrColor =  
    screenSpaceRayCast(i.worldPos, refrDir);  
refrColor = lerp(refrColor,  
    ssRefrColor.rgb, ssRefrColor.a);
```

Jeżeli uruchomimy teraz program, łatwo zauważyc, że rezultat jest daleki od ideału. W funkcji `main` shadera możemy testowo zwrócić kolor `ssReflColor` aby lepiej zilustrować problem. Rezultat wyglądać będzie jak na rysunku 7.



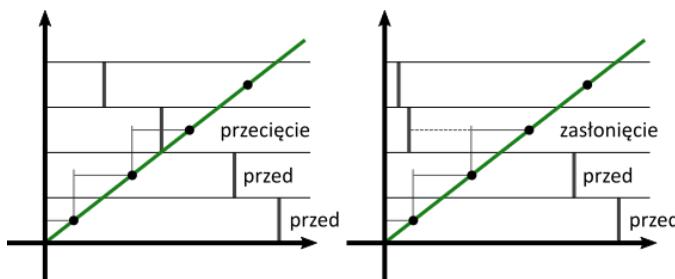
Rysunek 7: Niepoprawne odbicia promieni przechodzących za obiektem

Wynika to z faktu, że nasze obecne podejście wykryje przecięcie promienia z powierzchnią nie zależnie od tego, czy promień faktycznie przeciął powierzchnię, czy też przeszedł daleko za nią (por. rys. 8).



Rysunek 8: Brak rozróżnienia przecięcia od przejścia za powierzchnią

Problem można zniwelować jeżeli jako przecięcie traktować będziemy gdy głębokość pobrana z bufora leżeć będzie pomiędzy głębokością promienia z poprzedniego i obecnego kroku algorytmu DDA, co ilustruje rysunek 9. W przypadku, gdy promień jest zasłonięty, to nie jesteśmy w stanie stwierdzić co się dzieje z promieniem dalej: czy przecina jakąś powierzchnię niewidoczną na ekranie, czy też przechodzi całkiem za obiektem go przysłaniającym. Musimy więc przerwać poszukiwania i potraktować promień jakby w nic nie trafił.



Rysunek 9: Promień przecinający powierzchnię lub przechodzący za nią

Wykonanie:

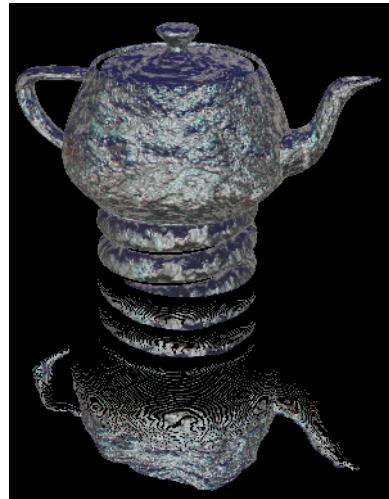
14. W funkcji `screenSpaceRayCast`, tuż przed pętlą `for` zadeklaruj zmienną `prevZ`, gdzie przechowywać będziemy głębokość promienia z poprzedniej iteracji.

```
float prevZ = P.z;
```

15. Wewnątrz pętli sprawdzać będziemy, czy `screenZ` leży w pomiędzy `rayZ` i `prevZ`. Test musi jednak uwzględnić przypadek, gdy promień biegnie w kierunku kamery (tj. `rayZ < prevZ`). Jeżeli znaleźliśmy przecięcie, zwracamy kolor jak poprzednio. Gdy wykryjemy zasłonięcie, musimy przerwać pętle.

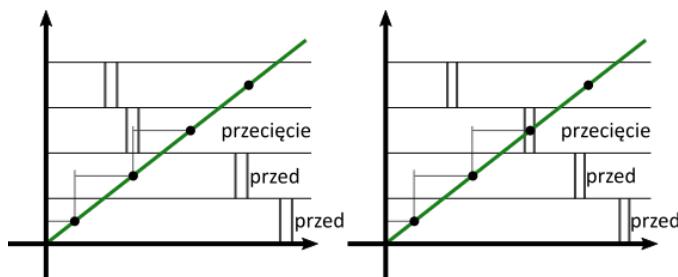
```
float maxZ = max(rayZ, prevZ);
float minZ = min(rayZ, prevZ);
prevZ = rayZ;
if (screenZ < maxZ)
{
    if (screenZ > minZ)
        return screenColor.Load(pxCoord);
    else break;
}
```

Po wprowadzeniu powyższych zmian rezultat powinien wyglądać jak na rysunku 10 (ponownie zakładając, że tymczasowo zwracamy tylko `ssReflColor` z funkcji `main`).



Rysunek 10: Promień przecinający powierzchnię lub przechodzący za nią

Widoczne przerwy wynikają z tego, że bufor głębokości jest zdyskretyzowany, każdemu pikselowi przypisuje jedną wartość głębokości, mimo iż reprezentuje on fragment powierzchni na którym wartość  $z$  w układzie kamery nie jest stała. Aby rozwiązać ten problem możemy dać każdemu pikselowi bufora pewną stałą grubość. Przecięcie znajdziemy wtedy, gdy zakresy głębokości bufora głębokości (plus stała grubość) oraz promienia w danym pikselu się nakładają (por. rys. 11).



Rysunek 11: Promień przecinający powierzchnię o niezerowej grubości

Wspomniana grubość piksela to długość zakresu wartości współrzędnej  $z$  w układzie kamery. Jednakże w buforze głębokości wartości te są przekształcone przez macierz perspektywy (i normalizację współrzędnych aficznich). Aby je odzyskać wystarczy jednak przemnożyć pewien punkt zawierający wartość z bufora głębokości w trzeciej współrzędnej przez odwrotność macierzy perspektywy (i wynik znormalizować).

Wykonanie:

16. W konstruktorze klasy `ShaderDemo` dodaj zmienne `projInvMtx` zawierającą odwrotność macierzy rzutowania oraz `depthThickness`. Tą drugą chcemy mówić modyfikować z poziomu interfejsu programu:

```
m_variables.AddSemanticVariable("projInvMtx",
    VariableSemantic::MatPInv);
m_variables.AddGuiVariable("depthThickness",
    0.05f, 0.0f, 0.5f);
```

17. W pliku shadera pikseli wody `waterPS.hlsl` dodaj deklaracje powyższych zmiennych:

```
matrix projInvMtx;
float depthThickness;
```

18. Dodaj funkcję `linearizeDepth`, która odkoduje współrzędne  $z$  w układzie kamery z bufora głębokości:

```
float linearizeDepth(float depth)
{
    float4 p = float4(0.0f, 0.0f, depth, 1.0f)
    p = mul(projInvMtx, p);
    return p.z / p.w;
}
```

19. W funkcji `screenSpaceRayCast` chcemy porównywać współrzędne  $z$ . Użyskanie ich dla promienia nie stanowi problemu, gdyż w kroku 8 odwrotność  $z$  zapamiętaliśmy w czwartej współrzędnej. Przed pętlą `for` zmienić trzeba tylko początkową wartość `prevZ`:

```
float prevZ = P.z;

na:

float prevZ = 1/P.w;
```

20. Podobnie wewnątrz pętli aktualną wartość  $z$  zmienić należy z:

```
float rayZ = P.z + 0.5f * dP.z;
```

na:

```
float rayZ = 1.0f / (P.w + 0.5f * dP.w);
```

21. Współrzedną z ekranu odzyskujemy funkcją `linearizeDepth`:

```
float screenZ =  
    linearizeDepth(screenDepth.Load(pxCoord).r);
```

22. A na koniec modyfikujemy sam test uwzględniając grubość powierzchni:

```
if (screenZ - depthThickness < maxZ)  
{  
    ...  
}
```

Po wprowadzeniu powyższych zmian uruchom program. Modyfikując zmienną `depthThickness` można obserwować jaki wpływ ma ona na rezultat. Zbyt małe wartości nie eliminują przerw w odbiciu, jednak za duże zbyt je zniekształcają. Dla naszej sceny odpowiednia wartość wynosi około 0,05.

Wprowadzone zmiany nie eliminują wszystkich artefaktów — w szczególności nie uda nam się znaleźć koloru jeżeli promienie przecinają powierzchnie zasłonięte lub zwrócone tyłem do kamery. Jednakże w połączeniu z innymi efektami (jak ilustruje to rysunek 12) wszelkie niedokładności są prawie niezauważalne.



Rysunek 12: Wynikowy efekt

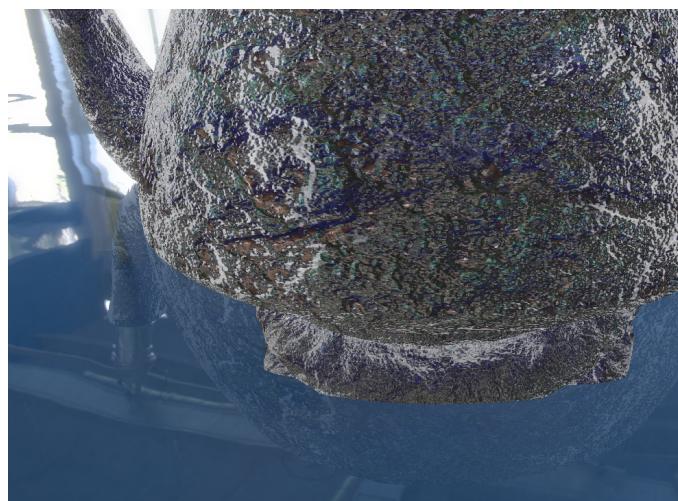
Prezentowana technika wymaga by obiekty były widoczne na ekranie. Jak pokazują rysunki 13, 14 i 15 nie uda się pokazać załamanych i odbitych obrazów fragmentów geometrii wychodzących poza ekran. O ile jest to nie do uniknięcia, to można uniknąć gwałtownego obcięcia obrazów poprzez implementację stopniowego zanikania zwracanego koloru im bliżej znalezione przecięcie znajduje się krawędzi ekranu.



Rysunek 13: Pełne obrazy odbite i załamane



Rysunek 14: Obcięty obraz załamany



Rysunek 15: Obcięte obrazy odbite i załamane