POLSKO-JAPOŃSKA
WYŻSZA SZKOŁA
TECHNIK KOMPUTEROWYCH

Piotr Gwiazdowski

Programowanie efektów graficznych w Direct3D

WYDAWNICTWO
PJWSTK

Notka biograficzna

Piotr Gwiazdowski jest absolwentem Politechniki Śląskiej w Gliwicach. W czasie trwania studiów uzyskał wyróżnienie w konkursie „Mobile Your Dreams" za grę „MexMobile". Od 2008 roku zawodowo zajmuje się programowaniem gier komputerowych, głównie na nowoczesne platformy mobilne. Praca ta zaowocowała wydaniem m.in. tytułów „The Big Roll in Paradise" oraz „Sailboat Championship". Jego zainteresowania koncentrują się wokół silników graficznych oraz rozpoznawania obrazów. Obecnie bierze udział w nowatorskim projekcie dotyczącym analizy i interaktywnej syntezy ludzkiego ruchu.

Streszczenie

Niniejsza książka stanowi wprowadzenie w świat niskopoziomowej grafiki trójwymiarowej. Skierowana jest do studentów informatyki zainteresowanych programowaniem nowoczesnych gier komputerowych. Zastosowanym API graficznym jest Direct3D w wersji 10. Książka omawia podstawy matematyczne związane z grafiką trójwymiarową oraz oferuje pomoc w stawianiu pierwszych kroków w świecie niskopoziomowych mechanizmów. Szczegółowo przedstawione wybrane efekty wizualne prezentują różne sposoby wykorzystania możliwości oferowanych przez API, mając dodatkowo na celu zachęcić czytelnika do eksperymentowania we własnym zakresie. Dodatkowo zamieszczono porady, które, mimo iż nie zawsze są bezpośrednio związana z grafiką trójwymiarową, wskazują dobre praktyki w świecie programowania gier.

Ta książka powinna być cytowana jako:

Gwiazdowski, P., 2011. Programowanie efektów graficznych w Direct3D. Warszawa: Wydawnictwo PJWSTK.

Spis treści

1

Wprowadzenie

W ciągu ostatnich lat nastąpił dynamiczny wzrost mocy obliczeniowej kart graficznych. Jednym z czynników warunkujących ten trend było ciągłe zwiększanie się wymagań użytkowników w stosunku do aplikacji zajmujących się renderowaniem grafiki trójwymiarowej w czasie rzeczywistym, czyli z zachowaniem złudzenia płynności animacji. Skutkiem takiego stanu rzeczy było podniesienie się złożoności obliczeniowej programów, a więc także i wymagań sprzętowych. Wzrostowi mocy kart graficznych nie towarzyszył jednak proporcjonalny postęp w dziedzinie jednostek centralnych, wobec czego pojawiły się techniki i rozwiązania pozwalające przenieść część obliczeń z procesora głównego i zagospodarować w ten sposób dostępne możliwości sprzętu. Obecnie kluczem do osiągnięcia wysokiego poziomu generowanej grafiki przy zachowaniu złudzenia płynności jest dokonanie odpowiedniego podziału ról pomiędzy dwiema omawianymi jednostkami wykonawczymi.

1.1 Wymagane umiejętności

Pisząc tę książkę, autorzy przyjęli założenie, że czytelnik posiada następujące umiejętności:

- Znajomość języka C++ na poziomie podstawowym wystarczającym do zrozumienia obsługi API Direct3D lub zaawansowanym do zrozumienia działania narzędzi użytych w przykładach
- Znajomość podstaw WinAPI
- Znajomość podstaw STL
- Znajomość środowiska Visual Studio
- Znajomość matematyki obejmująca trygonometrię i geometrię analityczną

1.2 Wymagania sprzętowe i programowe

Do użycia załączonych przykładów bądź ich odtwarzania niezbędne mogą się okazać:

- Karta graficzna zgodna z Direct3D 10
- System operacyjny Microsoft Windows Vista lub Microsoft Windows 7
- Visual Studio (2008 wzwyż)
- DirectX SDK

1.3 Konwencje stosowane w książce

Komentarze typów, zmiennych i funkcji pisane są pod kątem możliwości późniejszego wygenerowania dokumentacji przy użyciu zewnętrznych programów, np. *doxygen*[1].

Bardzo często się zdarza, że wklejenie całego kodu danej funkcji odciągałoby uwagę od bieżąco omawianego problemu. Dlatego też w niniejszej książce przedstawiane są raczej te fragmenty, które niosą istotne w danym momencie informacje. Jeżeli któraś linia zawiera wielokropek, wówczas oznacza on pominięcie pewnej ilości kodu – być może nawet wyjście z przestrzeni nazw funkcji do przestrzeni globalnej. Aby więc odróżnić lokalne zmienne od globalnych, tym identyfikatorom tych drugich zawsze nadawany jest przedrostek `g_`.

> W takich polach jak to będą znajdować się porady i uwagi często w luźny sposób związane z bieżącą tematyka, ale o znacznym, zdaniem autora, stopniu przydatności.

[1] `www.doxygen.org`

2

Podstawy matematyczne

2.1 Wprowadzenie

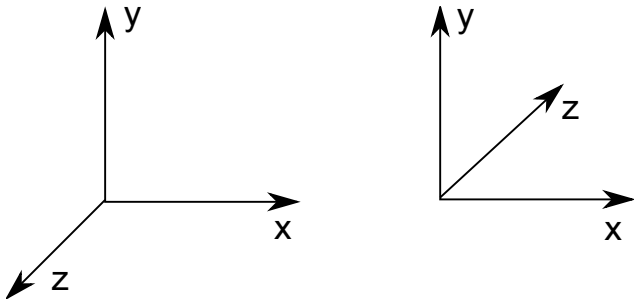
Grafika w pierwszych grach trójwymiarowych (np. Wolfenstein) była tworzona za pomocą bardzo prostych obliczeń i sztuczek. Wrażenie głębi uzyskiwano dzięki prostemu, ale i ciekawemu algorytmowi rzucania promieni (ang. *ray casting*) [7]. Dziś trudno jest wyobrazić sobie grafikę 3D tworzoną bez użycia podstawowych operacji na macierzach i wektorach, dlatego warto lepiej poznać te zagadnienia.

2.2 Układ współrzędnych

Dzięki układowi współrzędnych można określić położenie dowolnego punktu w przestrzeni. Istotne jest rozróżnienie, czy operujemy na układzie lewoskrętnym czy prawoskrętnym. Aby to określić, wystarczy wyprostować palce prawej dłoni wzdłuż hipotetycznej osi OX, a następnie je wygiąć zgodnie ze zwrotem osi OY. Jeśli odwiedziony kciuk wskazuje zwrot osi OZ, wówczas układ testowany jest prawoskrętny, natomiast w przeciwnym przypadku – lewoskrętny. Opisana metoda to tzw. reguła prawej dłoni bazująca na różnym uporządkowaniu osi OX i OY w obu typach układów.

2.3 Wektory

Wektory pełnią bardzo ważną rolę zarówno w grafice, jak i w grach 3D. Wiele wektorowych wielkości fizycznych, takich jak prędkość i siły znajduje swoje odzwierciedlenie w wirtualnych światach. Wektory są również używane do określania względnego położenia, opisywania charakterystyki światła, wykrywania i obsługi kolizji oraz w wielu, wielu innych zastosowaniach.



Rysunek 2.1. Układ prawoskrętny i lewoskrętny

2.3.1 Definicja

Definicja 1 (Wektor [8]) *Wektorem o początku A i końcu B, który oznacza się* $\overrightarrow{AB}$*, nazywa się uporządkowaną parą punktów A i B wyznaczającą w danej przestrzeni odcinek skierowany.*

Punkt w przestrzeni n-wymiarowej można zapisać za pomocą n współrzędnych, a zatem:

$$A = \underbrace{(A_x, A_y, A_z, \ldots)}_{n}$$

$$B = \underbrace{(B_x, B_y, B_z, \ldots)}_{n}$$

W dalszej części książki opisy będą dotyczyły głównie punktów i wektorów trójwymiarowych, gdyż takie są zazwyczaj stosowane w grach komputerowych i grafice 3D.

Współrzędne wektora trójwymiarowego zapisuje się następująco:

$$\overrightarrow{AB} = \begin{bmatrix} B_x - A_x \\ B_y - A_y \\ B_z - A_z \end{bmatrix} = \begin{bmatrix} AB_x \\ AB_y \\ AB_z \end{bmatrix} \tag{2.1}$$

Poziomy zapis również jest poprawny, a czasami wręcz konieczny, żeby zachować formalną poprawność przy mnożeniu razy macierz, gdy wektor jest pierwszym czynnikiem mnożenia.

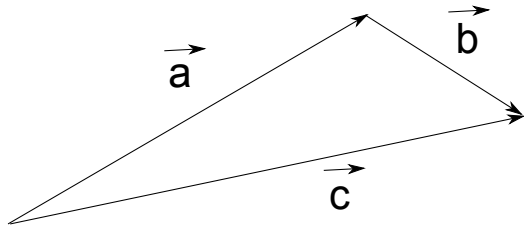
$$\overrightarrow{AB} = [\, AB_x \;\; AB_y \;\; AB_z \,]$$

2.3.2 Proste operacje na wektorach

Jedną z najbardziej podstawowych operacji na wektorach jest ich dodawanie. Wykonanie tej operacji sprowadza się do obliczenia sumy współrzędnych wektorów:

$$\vec{c} = \vec{a} + \vec{b} = \begin{bmatrix} a_x + b_x \\ a_y + b_y \\ a_z + b_z \end{bmatrix}$$

Ważna jest umiejętność wyobrażenia sobie prostych operacji na wektorach, ponieważ umożliwia to pełne zrozumienie wzorów i zależności z wykorzystaniem wektorów. Dodawanie wektorów można intuicyjnie przestawić za pomocą tzw. reguły trójkąta. Należy połączyć koniec pierwszego wektora z początkiem drugiego. Początek pierwszego i koniec drugiego wyznaczy szukany wektor. Przedstawia to rysunek 2.2.

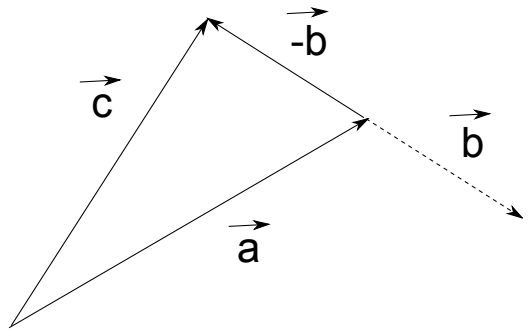


Rysunek 2.2. Dodawanie wektorów

Analogicznie przedstawia się odejmowanie:

$$\vec{c} = \vec{a} - \vec{b} = \begin{bmatrix} a_x - b_x \\ a_y - b_y \\ a_z - b_z \end{bmatrix}$$

W interpretacji graficznej odejmowania wektorów również można skorzystać z reguły trójkąta, uwzględniając fakt, że należy zmienić kierunek odjemnika. Pozostałe reguły pozostają bez zmian. Operację tę przestawia poniższy rysunek 2.3.



Rysunek 2.3. Odejmowanie wektorów

2.3.3 Długość wektora i normalizacja

Wzór na długość wektora oparty jest na twierdzeniu Pitagorasa. Przedstawia się w następujący sposób:

$$|\vec{v}| = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

Dzięki temu wzorowi bardzo łatwo obliczyć znormalizowaną postać $\hat{v}$ wektora $\vec{v}$. Wektor znormalizowany ma ten sam kierunek i zwrot co wektor normalizowany, ale jego długość zawsze wynosi 1. Aby go otrzymać, wystarczy obliczyć długość wektora normalizowanego, a następnie podzielić przez nią każdą jego współrzędną:

$$\hat{v} = \frac{\vec{v}}{|\vec{v}|} = \begin{bmatrix} \frac{v_x}{|\vec{v}|} \\ \frac{v_y}{|\vec{v}|} \\ \frac{v_z}{|\vec{v}|} \end{bmatrix}$$

Wektor znormalizowany nazywany jest również wektorem jednostkowym.

Podczas badania odległości obiektów względem pewnego ustalonego punktu często wystarczy informacja, który z nich leży bliżej. W tym przypadku dużo korzystniej jest liczyć kwadrat odległości i porównywać wyniki. Unika się w ten sposób kosztownego obliczeniowo pierwiastkowania.

2.3.4 Iloczyn skalarny

Definicja 2 (Iloczyn skalarny) *Iloczyn skalarny (ang.* dot product*) jest operacją na dwóch wektorach, która daje w wyniku skalar. Dany jest wzorem:*

$$\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y + a_z b_z$$

Inny wzór na iloczyn skalarny, choć jest bardziej kosztowny obliczeniowo, łatwiej jest zinterpretować geometrycznie:

$$\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}| cos\alpha \tag{2.2}$$

gdzie:

α - kąt między wektorami $\vec{a}$ i $\vec{b}$

Występowanie długości wektorów we wzorze 2.2 prowadzi do poczynienia następującej obserwacji:

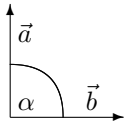
Wniosek 1 *Jeżeli wektory $\hat{a}$ i $\hat{b}$ są znormalizowanymi postaciami wektorów $\vec{a}$ i $\vec{b}$, wówczas dla kąta α między nimi prawdziwa jest zależność:*

$$\hat{a} \cdot \hat{b} = cos\alpha$$

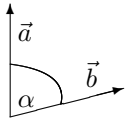
Powyższa obserwacja jest podstawą bardzo wielu algorytmów związanych z grafiką komputerową i decyduje o powszechnym używaniu wektorów znormalizowanych. Przykładowo, aby wyliczyć kąt między wektorami $\vec{a}$ i $\vec{b}$, wystarczy doprowadzić je do postaci znormalizowanych $\hat{a}$ i $\hat{b}$ i wyznaczyć *arccos* z ich iloczynu skalarnego:

$$\alpha = arccos(\hat{a} \cdot \hat{b})$$

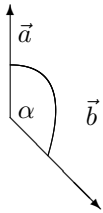
Kilka przydatnych faktów:



jeśli $\alpha = 90°$ to $\vec{a} \cdot \vec{b} = 0$



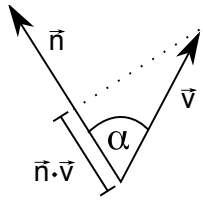
jeśli $\alpha < 90°$ to $\vec{a} \cdot \vec{b} > 0$



jeśli $\alpha > 90°$ to $\vec{a} \cdot \vec{b} < 0$

Przykład jednego z wielu zastosowań iloczynu skalarnego: dane są dwa obiekty poruszające się w pewnych określonych kierunkach $\vec{a}$ i $\vec{b}$. Należy ocenić, w jakim stopniu kierunki te się pokrywają. Aby rozwiązać problem, wystarczy obliczyć iloczyn skalarny tych wektorów. Jeśli oba są jednostkowe, to wynik będzie zawierał się w przedziale $< -1, 1 >$. Im większa liczba, tym kierunek $\vec{a}$ jest bardziej zbliżony do $\vec{b}$. Zazwyczaj jest to informacja wystarczająca, aby odpowiednio zareagować.

Dzięki iloczynowi skalarnemu można szybko obliczyć długość rzutu wektora $\vec{v}$ na dowolną oś. W tym celu należy stworzyć wektor jednostkowy $\vec{n}$ mający kierunek i zwrot zgodny z osią, na którą wektor $\vec{v}$ jest rzutowany. Iloczyn skalarny dwóch rozpatrywanych wektorów da w wyniku szukaną długość. Operacja przedstawiona została na rysunku 2.4.



Rysunek 2.4. Iloczyn skalarny wektora $\vec{v}$ i wektora jednostkowego $\vec{n}$

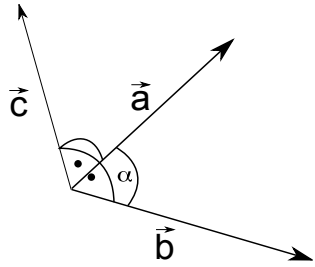
2.3.5 Iloczyn wektorowy

Definicja 3 (Iloczyn wektorowy) *Iloczyn wektorowy (ang.* cross product*) jest działaniem na dwóch wektorach $\vec{a}$ i $\vec{b}$. Wynikiem tego działania jest następny wektor : $\vec{c}$, prostopadły zarówno do $\vec{a}$, jak i do $\vec{b}$. Jego zwrot dobrany jest za pomocą reguły prawej lub lewej dłoni (zależne jest to od skrętności układu odniesienia). Iloczyn wektorowy oblicza się używając wzoru:*

$$\vec{c} = \vec{a} \times \vec{b} = \begin{bmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{bmatrix}$$

Długość nowo powstałego wektora określa iloczyn długości wektorów $\vec{a}$ i $\vec{b}$ oraz sinusa kąta między nimi:

$$|\vec{c}| = |\vec{a}||\vec{b}|sin\alpha$$

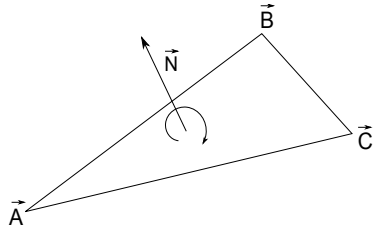


Rysunek 2.5. Iloczyn wektorowy wektorów $\vec{a}$ i $\vec{b}$

2.3.6 Wektor normalny wielokąta i wierzchołka

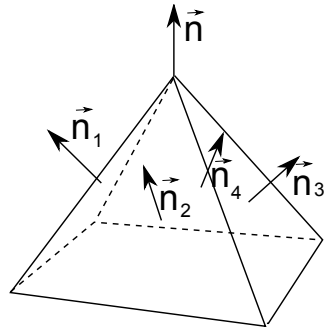
Z formalnego punktu widzenia, wektor normalny to wektor prostopadły do danej płaszczyzny. W grafice komputerowej tego pojęcia używa się w kontekście wielokątów oraz wierzchołków. W pierwszym przypadku definicja jest zgodna

– omawiany wektor jest zawsze prostopadły do płaszczyzny, na której wyznaczona jest dana figura. Co istotne, w tym przypadku, wektor normalny może być obliczony wyłącznie na podstawie pozycji trzech wierzchołków oraz informacji o skrętności układu. W tym celu należy najpierw wyznaczyć wektory dla dwóch krawędzi współdzielących wierzchołek zgodnie ze wzorem 2.1. Kolejność doboru wierzchołków do tej operacji jest określona skrętnością układu, co przedstawia rysunek 2.6. Iloczyn wektorowy wyznaczonych wektorów da, po znormalizowaniu, szukany wektor normalny.



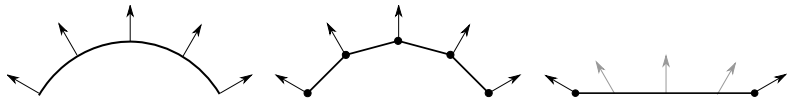
Rysunek 2.6. Wektor normalny trójkąta, jego zwrot określony został za pomocą reguły lewej dłoni

Wektor normalny wierzchołka jest wektorem normalnym powierzchni stycznej do tego wierzchołka. W praktyce często oblicza się go na podstawie średniej wektorów normalnych wielokątów, które współdzielą dany wierzchołek, co przedstawia rysunek 2.7.



Rysunek 2.7. Normalna wierzchołka jako średnia normalnych ścian

W grafice komputerowej możliwość przypisania wierzchołkom figury wektora normalnego skutkuje ciekawymi możliwościami. Jeżeli przyjąć, że wektor normalny dla pewnego algorytmu służy do określania nachylenia płaszczyzny, wówczas okazuje się, że aby oddać wypukłość bądź wklęsłość w ramach płaskiej przecież figury geometrycznej, wystarczy operacja interpolacji liniowej normalnych wierzchołków. Zjawisko to przedstawia rysunek 2.8.



Rysunek 2.8. Zastosowanie wektorów normalnych wierzchołków. Dla wszystkich rzutów powierzchni dla tej samej współrzędnej poziomej wektory normalne są jednakowe. Jaśniejszym kolorem oznaczono wektory powstałe wskutek interpolacji liniowej

2.3.7 Dodatkowe operacje

Działaniem na wektorach, które nie jest zdefiniowane w kanonach matematyki, a które jest bardzo przydatne w dziedzinie grafiki komputerowej, jest mnożenie analogicznych współrzędnych dwóch wektorów. Na potrzeby niniejszej książki wprowadzono zatem operację iloczynu elementów (ang. *component-wise multiplication*) dwóch wektorów n-wymiarowych zdefiniowaną jako:

$$\vec{a} \otimes \vec{b} = \begin{bmatrix} a_0 \\ a_1 \\ ... \\ a_n \end{bmatrix} \otimes \begin{bmatrix} b_0 \\ b_1 \\ ... \\ b_n \end{bmatrix} = \begin{bmatrix} a_0 b_0 \\ a_1 b_1 \\ ... \\ a_n b_n \end{bmatrix}$$

W praktyce nie zawsze wiadomo, czy drugi operand będzie skalarem czy wektorem. W związku z tym, by uprościć zapis, operacja iloczynu elementów jest zdefiniowana również dla wektora n-wymiarowego i skalara jako:

$$\vec{a} \otimes b = \begin{bmatrix} a_0 \\ a_1 \\ ... \\ a_n \end{bmatrix} \cdot b = \begin{bmatrix} a_0 b \\ a_1 b \\ ... \\ a_n b \end{bmatrix}$$

2.4 Macierze

Macierze są nieco bardziej skomplikowane niż wektory. Nie są wykorzystywane w tak wielu różnych aspektach, niemniej jednak są one równie ważne. Zwykle stosuje się je do przedstawiania transformacji geometrycznych

2.4.1 Definicja

Definicja 4 (Macierz [8]) *Macierzą (dokładniej macierzą dwuwskaźnikową) nazywamy każde odwzorowanie M, które każdej parze liczb naturalnych* (i, j), $i = 1, 2, \ldots, s$, $j = 1, 2, \ldots, t$ *przyporządkowuje dokładnie jeden element* m_{ij}, *należący do określonego zbioru* Ω *(np. zbioru liczb rzeczywistych, zbioru liczb zespolonych).*

Macierze zazwyczaj zapisuje się za pomocą tablicy, która ma m wierszy i n kolumn. Każdy element m_{ij} jest w niej odpowiednio uporządkowany - i wskazuje wiersz, j wskazuje kolumnę.

$$\begin{bmatrix} m_{11} & m_{12} & \dots & m_{1j} & \dots & m_{1t} \\ m_{21} & m_{22} & \dots & m_{2j} & \dots & m_{2t} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ m_{i1} & m_{ij} & \dots & m_{ij} & \dots & m_{it} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ m_{s1} & m_{s2} & \dots & m_{sj} & \dots & m_{st} \end{bmatrix}$$

W grafice komputerowej zwykle stosuje się macierze 4×4, dlatego dalsza część rozdziału będzie o nich traktowała. Czasami można się spotkać z macierzami 3×3 (wystarczą, aby opisać skalę i rotację) oraz 4×3 (można nią zapisać skalę, rotację i przesunięcie, ale operowanie nią jest bardziej kłopotliwe). Typowa macierz 4×4 wygląda następująco:

$$M = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix}$$

2.4.2 Mnożenie macierzy

Mnożenie macierzy jest bardzo ważne, gdyż dzięki niemu można w bardzo łatwy sposób składać transformacje.

Definicja 5 (Iloczyn macierzy) *Jeśli macierz A ma dokładnie tyle kolumn, ile macierz B ma wierszy, to ich iloczyn jest określony i dany jest wzorem:*

$$C = AB$$
$$c_{ij} = \sum_{s=1}^{m} a_{i,s} b_{s,j}$$

gdzie:

C - wynik mnożenia macierzy

Wzór ten dostarcza bardzo ważnych informacji. Po pierwsze, wynika z niego, że mnożenie macierzy nie jest przemienne, jest to bardzo ważne, przy składaniu transformacji. Drugim istotnym faktem jest to, że jeśli istnieje działanie $C = AB$, to działanie $C = BA$ może, lecz nie musi być określone.

Wzór ten można naturalnie zastosować w prostszym przypadku - mnożenia macierzy przez wektor i wektora przez macierz. Należy też pamiętać o odpowiednim zapisie wektora - wierszowym lub kolumnowym. Dla macierzy 4×4

wygląda to następująco:

$$\vec{v}M = [\,a\ b\ c\ d\,] \begin{bmatrix} m_{11}\ m_{12}\ m_{13}\ m_{14} \\ m_{21}\ m_{22}\ m_{23}\ m_{24} \\ m_{31}\ m_{32}\ m_{33}\ m_{34} \\ m_{41}\ m_{42}\ m_{43}\ m_{44} \end{bmatrix} = \begin{bmatrix} a_2\ b_2\ c_2\ d_2 \end{bmatrix}$$

$$\begin{aligned} a_2 &= am_{11} + bm_{21} + cm_{31} + dm_{41} \\ b_2 &= am_{12} + bm_{22} + cm_{32} + dm_{42} \\ c_2 &= am_{13} + bm_{23} + cm_{33} + dm_{43} \\ d_2 &= am_{14} + bm_{24} + cm_{34} + dm_{44} \end{aligned}$$

W przypadku, gdy stosuje się kolumnowy zapis wektora, mnożenie razy macierz wygląda następująco:

$$M\vec{v} = \begin{bmatrix} m_{11}\ m_{12}\ m_{13}\ m_{14} \\ m_{21}\ m_{22}\ m_{23}\ m_{24} \\ m_{31}\ m_{32}\ m_{33}\ m_{34} \\ m_{41}\ m_{42}\ m_{43}\ m_{44} \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$$

$$M\vec{v} = \begin{bmatrix} am_{11} + bm_{12} + cm_{13} + dm_{14} \\ am_{21} + bm_{22} + cm_{23} + dm_{24} \\ am_{31} + bm_{32} + cm_{33} + dm_{34} \\ am_{41} + bm_{42} + cm_{43} + dm_{44} \end{bmatrix}$$

Pewna rodzaju niejasność może wynikać z faktu, że opisywane jest mnożenie wektora czterowymiarowego, podczas gdy w grafice używa się głównie wektorów trójwymiarowych. Problem ten został wyjaśniony w punkcie 2.5.6.

2.4.3 Macierz jednostkowa

Definicja 6 (Macierz jednostkowa) *Macierz jednostkowa (ang.* identity matrix*) charakteryzuje się tym, że elementy na jej przekątnej wynoszą 1, natomiast wszystkie pozostałe są zerami, a zatem:*

$$m_{ij} = \begin{cases} 0 \text{ jeżeli } i \neq j \\ 1 \text{ jeżeli } i = j \end{cases}$$

Jednostkowa macierz 4×4 przedstawia się następująco:

$$I = \begin{bmatrix} 1\ 0\ 0\ 0 \\ 0\ 1\ 0\ 0 \\ 0\ 0\ 1\ 0 \\ 0\ 0\ 0\ 1 \end{bmatrix}$$

Wniosek 2 *Unikatową właściwością macierzy jednostkowej jest przemienność mnożenia:*

$$MI = IM = M$$

2.4.4 Macierz transponowana

Definicja 7 (Macierz transponowana) *Macierzą transponowaną macierzy A jest macierz A^T powstająca wskutek zamiany jej wierszy na kolumny i kolumn na wiersze.*

$$B = A^T \qquad \Longleftrightarrow \qquad \forall i,j \qquad b_{ij} = a_{ji}$$

Dla macierzy 4×4:

$$A = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix} \quad A^T = \begin{bmatrix} m_{11} & m_{21} & m_{31} & m_{41} \\ m_{12} & m_{22} & m_{32} & m_{42} \\ m_{13} & m_{23} & m_{33} & m_{43} \\ m_{14} & m_{24} & m_{34} & m_{44} \end{bmatrix}$$

2.4.5 Odwrotność macierzy

Definicja 8 (Macierz odwrotna) *Macierz M^{-1} jest odwrotna do macierzy M, jeśli zachodzi równość:*

$$MM^{-1} = M^{-1}M = I$$

Jest wiele ogólnych metod, dzięki którym można obliczyć macierz odwrotną, ale nie będą one tutaj przedstawiane. Zamiast tego w dalszej części przedstawione zostaną postaci macierzy odwrotnych, które są przydatne w kontekście grafiki komputerowej. Sama konieczność używania macierzy odwrotnych wynika z braku zdefiniowanej operacji dzielenia. W sytuacji, gdy trzeba rozwiązać równanie macierzowe, zastosowanie odwrotności pozwala na przenoszenie czynników macierzowych między stronami znaku równości. Dla przykładowego równania:

$$A = BC$$

dla znanych macierzy A i C oraz szukanej macierzy B proces prowadzący do rozwiązania wygląda następująco:

$$\begin{aligned} AC^{-1} &= BCC^{-1} \\ AC^{-1} &= BI \\ B &= AC^{-1} \end{aligned}$$

> Tylko macierze kwadratowe mogą mieć swoje macierze odwrotne.

2.5 Transformacje

Transformacje są podstawowym sposobem opisu struktury sceny w grafice trójwymiarowej. Intuicyjnie przez transformację można rozumieć sekwencję

przesunięć, obrotów oraz skalowań pewnego obiektu, prowadzącą do uzyskania pożądanego efektu. Powszechnym sposobem opisu tych operacji są macierze o wymiarach 4×4 oraz mnożenie macierzowe. W tym miejscu może rodzić się pytanie o zasadność użycia akurat takiej macierzy – nie jest bowiem możliwe pomnożenie przez nią trójwymiarowego wektora, który jest używany do reprezentacji większości danych geometrycznych. Okazuje się jednak, że korzyści płynące z takiej konwencji przewyższają ewentualne problemy wynikające z nieścisłości.

2.5.1 Translacja

Translacja odpowiada za przesuwanie obiektu o zadany wektor. Macierz dla tej operacji w przypadku wektora przesunięcia $\vec{t}$ przedstawia się następująco:

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \vec{t}_x & \vec{t}_y & \vec{t}_z & 1 \end{bmatrix}$$

Macierz odwrotna dokonująca przesunięcia z przeciwnym zwrotem:

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\vec{t}_x & -\vec{t}_y & -\vec{t}_z & 1 \end{bmatrix}$$

2.5.2 Rotacja

Macierz rotacji określa orientację obiektu w przestrzeni euklidesowej. Można stwierdzić, że ta macierz zawiera trzy wektory $\vec{x}$, $\vec{y}$ i $\vec{z}$, które tworzą układ odniesienia odpowiadający opisywanej orientacji.

$$R = \begin{bmatrix} \vec{x}_x & \vec{x}_y & \vec{x}_z & 0 \\ \vec{y}_x & \vec{y}_y & \vec{y}_z & 0 \\ \vec{z}_x & \vec{z}_y & \vec{z}_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotację można również opisać jako obrót wokół dowolnej osi $\vec{n} = [\, x \; y \; z \,]$ o podany kąt α według następującego wzoru:

$$R = \begin{bmatrix} c + (1-c)x^2 & (1-c)xy + zs & (1-c)xz - ys & 0 \\ (1-c)xy - zs & c + (1-c)y^2 & (1-c)yz - xs & 0 \\ (1-c)xz + ys & (1-c)yz - xs & c + (1-c)z^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$c = cos\alpha$$

$$s = sin\alpha$$

Powyższy wzór upraszcza się znacznie, jeśli wektor $\vec{n}$ będzie zgodny z osią OX, OY lub OZ, czyli będzie wynosił odpowiednio $[1\ 0\ 0]$, $[0\ 1\ 0]$ albo $[0\ 0\ 1]$:

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y = \begin{bmatrix} \cos\alpha & 0 & -\sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ \sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 & 0 \\ -\sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Aby obliczyć macierz odwrotną do macierzy obrotu, wystarczy obliczyć jej macierz transponowaną. Macierze posiadające taką właściwość określa się mianem ortogonalnych.

$$R^{-1} = R^T$$

Macierz, w której zachodzi powyższa własność, nazywa się również macierzą ortogonalną.

Blokada przegubowa

Jednym ze sposobów przedstawienia orientacji obiektu są tzw. kąty Eulera:

- odchylenie (ang. *yaw*) - obrót wokoło osi OY
- nachylenie (ang. *pitch*) - obrót wokoło osi OX
- przechylenie (ang. *roll*) - obrót wokoło osi OZ

Często używane są przy tworzeniu macierzy rotacji. Ich stosowanie obarczone jest jednak ryzykiem. Może się zdarzyć, że utracony zostanie stopień swobody. Zjawisko nazywane jest blokadą przegubową (ang. *gimbal lock*). Efektem jest możliwość obrotu ciała zaledwie wokoło jednej lub dwóch osi. Problem ten występuje, gdy w wyniku obrotów, osie zaczynają się pokrywać (gdyż obroty liczone są oddzielnie). Przykładowo obrót wokół osi OY o -90^o spowoduje, że oś OX pokryje się z osią OZ. Układ stał się zdegenerowany, gdyż obrót wokoło osi OX da identyczny efekt do obrotu wokoło osi OZ.

Kilka słów o kwaternionach

Macierze nie są jedynym stosowanym sposobem przedstawienia obrotów. Czasami dużo korzystniej stosować tzw. kwaterniony. Można powiedzieć, że kwaterniony są uogólnieniem liczb zespolonych. Posiadają część rzeczywistą oraz

trzy składowe urojone. Zwykle przedstawiane są jako suma:

$$q = w + xi + yj + zk$$

gdzie w to część rzeczywista, a i, j i k to części urojone.

Pierwszą osobą, która zaproponowała użycie kwaternionów do opisu orientacji w przestrzeni, był Ken Shoemake (był to rok 1985). Bazował on na obserwacji, że części urojone zachowują się bardzo podobnie do osi [15]. W stosunku do macierzy kwaterniony posiadają kilka zalet. Przede wszystkim do ich opisu wystarczą tylko cztery liczby zmiennoprzecinkowe, co czasami może prowadzić do istotnej optymalizacji pamięciowej. Dodatkowo w operacjach na kwaternionach nie występuje blokada przegubowa. Niestety, obliczenia z ich użyciem mogą być wolniejsze niż w przypadku macierzy (np. obrót punktu - w tym przypadku lepiej zamienić kwaternion na macierz i kontynuować obliczenia w standardowy sposób). Ponadto ciężko jest wyobrazić sobie orientację w przestrzeni za pomocą kwaternionu, dlatego czasami trudniej doszukać się ewentualnych błędów logicznych w programie.

2.5.3 Skalowanie

Skalowanie odpowiada za zmianę rozmiaru obiektu poprzez rozciąganie i ściskanie wzdłuż osi układu współrzędnych. Dla wektora współczynników skalowania $\vec{s}$ macierz dla tej operacji przedstawia się następująco:

$$S = \begin{bmatrix} \vec{s}_x & 0 & 0 & 0 \\ 0 & \vec{s}_y & 0 & 0 \\ 0 & 0 & \vec{s}_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

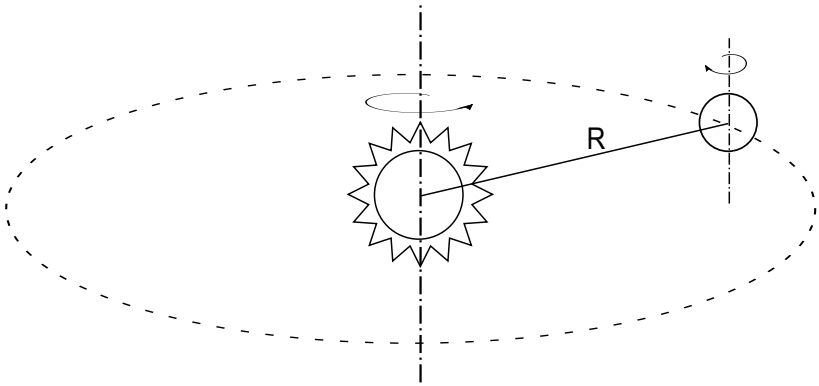
Macierz odwrotna przywracają pierwotną skalę określana jest według poniższej zależności:

$$S^{-1} = \begin{bmatrix} \frac{1}{\vec{s}_x} & 0 & 0 & 0 \\ 0 & \frac{1}{\vec{s}_y} & 0 & 0 \\ 0 & 0 & \frac{1}{\vec{s}_z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.5.4 Składanie transformacji

Do tej pory przedstawione zostały podstawowe transformacje. Na tej podstawie trudno wywnioskować, jak używać ich w bardziej skomplikowanych przypadkach. Przykładowo, jak wymodelować planetę obracającą się zarówno wokoło własnej osi, jak i wokoło gwiazdy, mając do dyspozycji jedynie translację, rotację i skalowanie?

Dzięki macierzowej reprezentacji tworzenie złożonych transformacji sprowadza się do sekwencji mnożeń transformacji atomowych. Transformacje obiektu będą następowały zgodnie z mnożeniem od strony lewej do prawej.



Rysunek 2.9. Przykład składania transformacji

Poniższy przykład pokazuje, jak zamodelować wspomnianą już gwiazdę i obracającą się wokoło niej planetę. Przyjęte zostało założenie, że posiadany model sfery, który posłuży do narysowania obu ciał, ma promień długości 1 jednostki. Najpierw powinno się obliczyć transformację dla gwiazdy, która znajduje się w środku układu odniesienia i nie wykonuje obrotów. Powinna być przykładowo stokrotnie większa niż posiadany model sfery. Dlatego w przypadku gwiazdy wystarczy zwykła macierz skali.

$$G = S(100)$$

gdzie:

G - transformacja gwiazdy
$S(x)$ - macierz skali, gdzie każda oś zostanie przeskalowana o x

Planeta obracająca się wokoło gwiazdy jest już bardziej skomplikowana. Przyjęto założenie, że jest:

1. dziesięciokrotnie mniejsza od gwiazdy
2. obraca się wokoło własnej osi z prędkością jednego obrotu na 5 sekund
3. krąży w odległości 1000 jednostek od gwiazdy z prędkością 1 obrotu na minutę

Transformacja dla planety będzie się przedstawiała następująco:

$$P = S(10) * R_y\left(\frac{360}{5}t\right) * T(1000, 0, 0) * R_y\left(\frac{360}{60}t\right)$$

gdzie:

P - transformacja planety
$R_y(\alpha)$ - macierz obrotu wokoło osi Y o kąt α podany w stopniach
$T(x, y, z)$ - macierz translacji z podanym przesunięciami dla każdej osi
t - czas w sekundach

Przyjęte zostało założenie, że planeta obraca się w obu przypadkach wokoło osi Y. Dlatego przesunięcie musiało być w osi X lub Z (gdyby nastąpiło w osi Y, to planeta nie obracałaby się wokoło gwiazdy). Dzięki ułamkowi $\frac{360}{5}$ ciało będzie obracało się z odpowiednią prędkością - pełny obrót będzie osiągnięty, gdy czas osiągnie wartość 5 ($t = 5$).

Co się stanie, gdy zostanie zmieniona kolejność mnożenia transformacji? Dla oznaczeń użytych transformacji S, R_1, T i R_2 (zgodnie z kolejnością wystąpienia we wzorze):

- SR_1R_2T - obiekt zostanie przeskalowany, obrócony za pomocą dwóch transformacji, a następnie przesunięty, zaś finalnie obiekt będzie obracał się z dużą prędkością tylko wokoło własnej osi, stojąc nieruchomo w punkcie (1000, 0, 0).
- STR_1R_2 - obiekt będzie obracał się z dużą prędkością tylko wokoło środka układu, ponieważ zostanie najpierw przesunięty, a później obrócony.
- R_1TR_2S - skalowanie na samym końcu wpłynie w tym przypadku też na przesunięcie, obiekt będzie się normalnie obracał, jednak będzie poruszał się po okręgu o dziesięciokrotnie większym promieniu.

2.5.5 Intuicyjna interpretacja macierzy 4 × 4

Każdy, kto choć raz pisał program generujący grafikę trójwymiarową, na pewno popełnił trudny do znalezienia błąd w obliczeniach na macierzach. Jest to typowe, dlatego warto umieć wyobrazić sobie transformację, patrząc na wartości elementów macierzy. Wbrew pozorom nie jest to takie trudne. Wystarczy przypomnieć sobie budowę podstawowych transformacji.

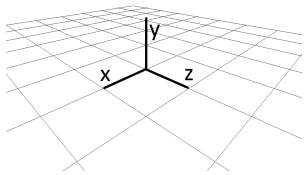
$$T = \begin{bmatrix} \vec{x}_x & \vec{x}_y & \vec{x}_z & 0 \\ \vec{y}_x & \vec{y}_y & \vec{y}_z & 0 \\ \vec{z}_x & \vec{z}_y & \vec{z}_z & 0 \\ \vec{t}_x & \vec{t}_y & \vec{t}_z & 1 \end{bmatrix} \tag{2.3}$$

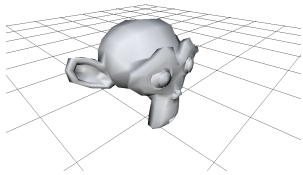
Elementy t_x, t_y, t_z pokazują, w którym miejscu obiekt znajduje się na scenie. Elementy $\vec{a}_x$,$\vec{a}_y$,$\vec{a}_z$ (gdzie $\vec{a}$ to $\vec{x}$, $\vec{y}$ lub $\vec{z}$) tworzą kolejne osie, wyobrażając je sobie, można określić, jaką obiekt ma orientację. Długość kolejnej osi wyznacza skalę. Warto wiedzieć, że dla typowej macierzy rotacji osie te są do siebie wzajemnie prostopadłe, dlatego warto to sprawdzić, gdy coś zaczyna działać niepoprawnie.

> Czasami widać na pierwszy rzut oka, że macierz jest niepoprawna, często jest to bardzo pomocne w szukaniu błędów.

Poniższe obrazki przedstawiają kilka transformacji wraz z ich graficzną reprezentacją.

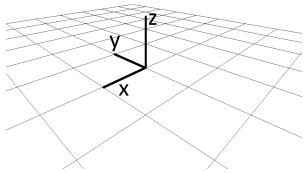
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

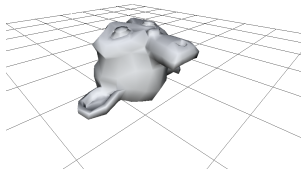




Dla macierzy jednostkowej obiekt znajduje się w środku układu odniesienia (lokalny układ obiektu pokrywa się z nim).

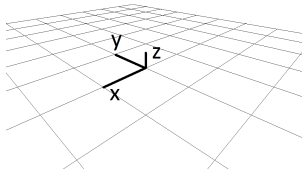
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

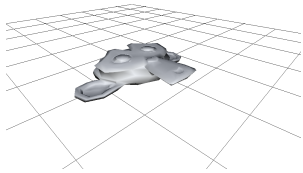




Obiekt został obrócony w osi X o 90 stopni. Widać, że osie lokalnego układu są również obrócone, opisują je pola macierzy.

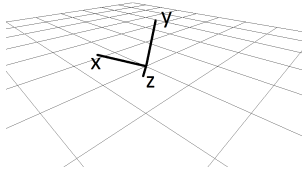
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0.33 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

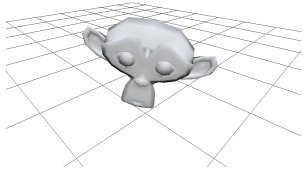




Obiekt został przeskalowany w osi Z o 0.33. Oś ta została skrócona, model zmienił kształt.

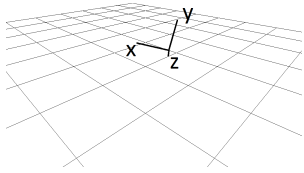
$$\begin{bmatrix} 0.59 & 0.19 & -0.7 & 0 \\ 0.03 & 0.96 & 0.26 & 0 \\ 0.27 & -0.06 & 0.18 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

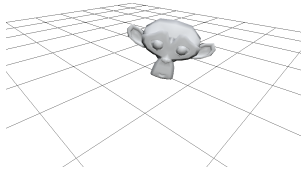




Model został obrócony wokoło każdej z osi. Macierz przestała już być przejrzysta i trudniej wyobrazić sobie transformacje. Niemniej można z grubsza oszacować, jak powinien przedstawiać się obiekt na scenie, co często jest wystarczające.

$$\begin{bmatrix} 0.59 & 0.19 & -0.7 & 0 \\ 0.03 & 0.96 & 0.26 & 0 \\ 0.27 & -0.06 & 0.18 & 0 \\ -2 & -0.5 & -1 & 1 \end{bmatrix}$$





Obiekt został przesunięty o wektor $[\,-2\ -0.5\ -1\,]$. Wartości te łatwo odnaleźć w macierzy.

2.5.6 Transformacja wektora

Do tej pory transformacje opisywane były w kontekście pewnych abstrakcyjnych obiektów, których matematyczna natura nie była przedstawiona. Na potrzeby dalszych rozważań można przyjąć, że każdy obiekt stanowi pewien zbiór punktów w przestrzeni. Ponieważ współrzędne punktów określane są względem środka układu współrzędnych, można je reprezentować przy pomocy wektorów rozciągniętych od punktu $(0; 0; 0)$. Wobec tego transformacja obiektu polega na przemnożeniu wszystkich jego punktów/wektorów razy macierz transformującą. Jedynym nierozwiązanym problemem jest umożliwienie wyznaczenia iloczynu trójwymiarowego wektora i macierzy 4×4.

Przekształcanie pozycji

Żeby znaleźć rozwiązanie, należy prześledzić mnożenie wektora czterowymiarowego $\vec{v} = [\, x \; y \; z \; w \,]$ razy ogólną postać macierzy transformacji T z równania 2.3, przyglądając się szczególnie temu, co dzieje się ze współrzędną w. Wektor przetransformowany $\vec{v}'$ określony jest wzorem:

$$\vec{v}' = \vec{v}T$$

Ponieważ czwarty wiersz macierzy transformacji zawiera ostateczny wektor translacji, nie może on zostać zdegenerowany podczas mnożenia. Czynnikiem, razy który mnożone są składowe transformacji, jest czwarta współrzędna wektora, czyli w. Wynika z tego, że:

$$\vec{v}_w = 1$$

Aby rozważania były spójne, musi zachodzić warunek niezmienności współrzędnej w. Gdyby ulegała ona zmianie, to wówczas działanie kolejnych transformacji byłoby zaburzone przez skalowanie wektora translacji. Bazując na wyprowadzeniach z 2.3 oraz postaci czwartej kolumny macierzy T, można jednak udowodnić, że współrzędna w nigdy nie jest transformowana:

$$\vec{v}'_w = \vec{v}_x \cdot 0 + \vec{v}_y \cdot 0 + \vec{v}_z \cdot 0 + \vec{v}_w \cdot 1 = \vec{v}_w$$

W związku z tym wektor $\vec{v}$ używany podczas transformacji pozycji P powinien mieć postać:

$$\vec{v}_w = [\, P_x \; P_y \; P_z \; 1 \,] \tag{2.4}$$

Przekształcanie kierunku

Poza transformacją wektorów reprezentujących pozycję często konieczne jest przetransformowanie wektora którego zadaniem jest tylko wskazywanie jakiegoś kierunku. Dobrym przykładem może być wektor normalny, który progra-

mista chciałby odwrócić i przeskalować[1] razem z całym obiektem. Użycie postaci zdefiniowanej równaniem 2.4 spowoduje, że wektor będzie potraktowany jak punkt, a więc dodany zostanie do niego wektor translacji. Poza specyficznymi przypadkami spowoduje to utratę cechy prostopadłości do powierzchni obiektu. W znalezieniu rozwiązania ponownie pomaga ogólna postać macierzy transformacji i rozwinięcie mnożenia z 2.3. Rozwiązanie może polegać więc na:

- wyzerowaniu składowej translacji macierzy transformacji,
- wyzerowaniu współrzędnej w,
- pominięciu ostatniego wiersza i kolumny macierzy transformacji i wykonaniu mnożenia trójwymiarowego.

Potencjał do bycia najlepszym (optymalnym) obliczeniowo ma ostatnie rozwiązanie (9 mnożeń zmiennoprzecinkowych zamiast 16), ale czasem może napotkać problemy w implementacji.

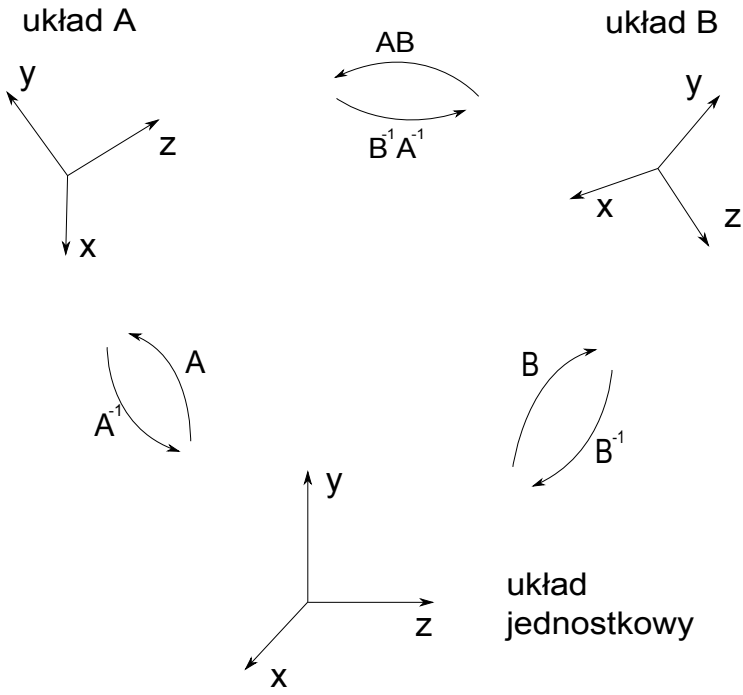
2.5.7 Transformacje odwrotne

Transformacje odwrotne były już tu wstępnie opisane przy opisie macierzy odwrotnej (2.4.5). Pamiętać należy, że transformacje są operacjami odwracalnymi. Transformując punkt z układu A do B wystarczy przemnożyć ten punkt przez odpowiednią macierz transformacji T. Przejście z układu B do A może być przeprowadzone za pomocą odwrotności macierzy T. Ważne jest to, że macierz odwrotna może pochodzić od dowolnej transformacji. W praktyce oznacza to, że możliwa jest bardzo prosta i efektywna zmiana układu odniesienia. Sytuację tę przedstawia rysunek 2.10.

Poniższy przykład obrazuje, jak obliczyć punkty przecięcia prostej z prostopadłościanem, którego transformacja jest określona. Pierwszym pomysłem mogłoby być znalezienie równań każdej z płaszczyzn, porównywanie ich z prostą i na tej podstawie wyciąganie pewnych wniosków. Jest to skomplikowane rozwiązanie, nieefektywne obliczeniowo oraz podatne na błędy. Dużo lepszym sposobem jest przetransformowanie prostej do układu odniesienia prostopadłościanu. Dzięki temu zabiegowi równania płaszczyzn znacznie się uproszczą, ponieważ obliczenia będą prowadzone w układzie, w którym każda ze ścian jest równoległa do jednej z osi. Ogólne równanie płaszczyzny: $Ax+By+Cz+D = 0$ sprowadzi się do równania $x = \frac{A}{D}$ dla ściany równoległej do osi X (gdyż $B = 0$ i $C = 0$).

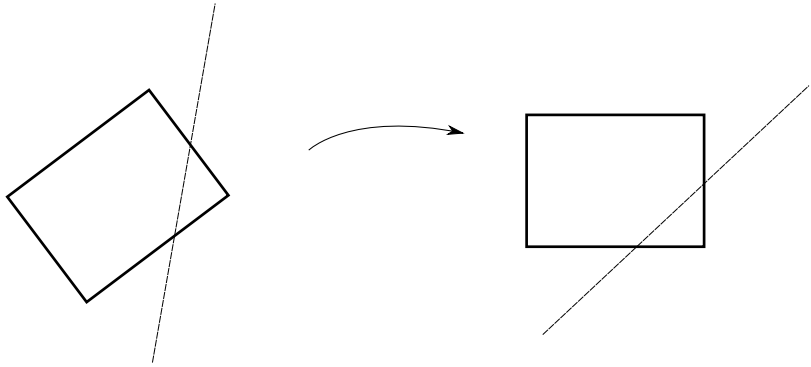
Dla przykładowych równań: $x = \pm 1$, $y = \pm 2$, $z = \pm 3$ wystarczy obliczyć przecięcie prostej z płaszczyznami $z = 3$ oraz $z = -3$, a na koniec sprawdzić, czy otrzymane wartości x_1,y_1 i x_2,y_2 spełniają warunki: $x_1, x_2 \in < -1, 1 >$ i $y_1, y_2 \in < -2, 2 >$. Aby rozwiązanie było poprawne dla każdego przypadku,

[1] Po przeskalowaniu innym niż jednostkowe wektor normalny będzie miał, poza specyficznymi przypadkami, długość różną od jedności, ale zostanie zachowana jego najważniejsza cecha - prostopadłość. Jednostkowość można odbudować przeprowadzając normalizację.



Rysunek 2.10. Przechodzenie pomiędzy układami odniesienia

należy jeszcze rozważyć możliwość, gdy prosta jest równoległa do osi z. Przypadek przedstawia rysunek 2.11 (jest on dwuwymiarowy dla poprawy czytelności, przypadek trójwymiarowy jest identyczny).



Rysunek 2.11. Przechodzenie pomiędzy układami odniesienia

2.6 Przestrzenie

2.6.1 Przestrzeń lokalna i globalna

Lokalne układy należy traktować jako układy związane z obiektami na scenie trójwymiarowej. Skutkiem tego, zmieniają wraz z nimi pozycję i orientację. Środek układu lokalnego zwykle jest umieszczony w wygodnym miejscu obiektu trójwymiarowego, przy czym najczęściej stosuje się do tego współrzędne środka geometrycznego, środka ciężkości bądź też punkt należący do podstawy (o ile dla obiektu można takową wyróżnić). Niezależnie od jego lokalizacji, to właśnie względem środka definowane są wszystkie parametry geometryczne obiektów.

Globalny układ, zwany również przestrzenią świata (ang. *world space*), nie zmienia swojej pozycji i orientacji w czasie. Stanowi on odniesienie, względem którego pozycjonowane są wszystkie obiekty. Jest to więc intuicyjnie rozumiana przestrzeń z wyróżnionym punktem zerowym.

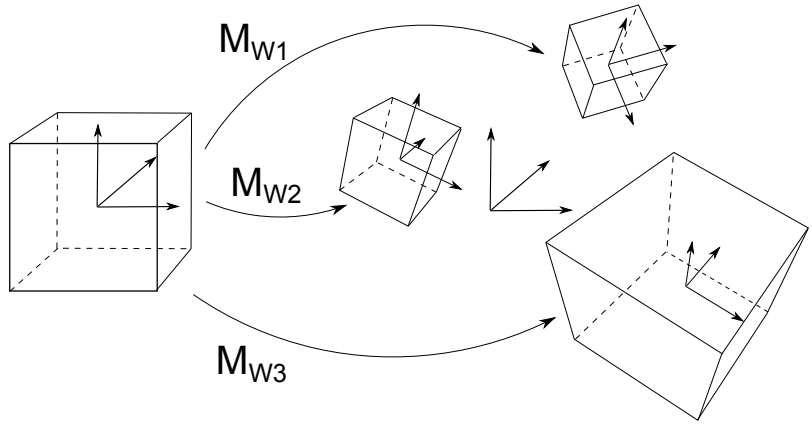
Przejście pomiędzy lokalną i globalną przestrzenią jest nazywane transformacją świata (ang. *world transform*), natomiast macierz ją reprezentująca – macierzą świata (ang. *world matrix*). Tradycyjnie tworzą ją złożenia wielu elementarnych transformacji, jak to było omówione w punkcie 2.5.4. Jeśli obiekt opisany jest bezpośrednio w globalnej przestrzeni, to jego macierz świata jest jednostkowa (przestrzeń lokalna pokrywa się wtedy z globalną).

Wprowadzenie przestrzeni lokalnej ma kilka zalet. Przede wszystkim istnieje możliwość wielokrotnego użycia tego samego obiektu na jednej scenie. By to uzyskać, wystarczy kolejne instancje transformować innymi macierzami świata. Dodatkowo też, obliczenia przeprowadzane w lokalnym układzie mają zazwyczaj prostszą i bardziej intuicyjną postać, ponieważ nie muszą uwzględniać wynikowych obrotów, przesunięć i skalowań. Dzięki omawianemu podziałowi wszystkie te operacje wykonywane są dopiero w momencie przejścia do przestrzeni globalnej.

2.6.2 Przestrzeń widoku

Określenie pola widzenia jest konieczne do prawidłowego przedstawienia sceny 3D. W tym celu umieszcza się kamerę jako osobny obiekt na scenie, jej pozycja i orientacja są konieczne do późniejszego wyznaczenia wynikowego obrazu. Choć transformację kamery można zapisać używając standardowej translacji i rotacji, to macierz, która zostanie wykorzystana w potoku renderowania musi być inna. Potrzebna jest odwrotność tej transformacji, wynika to z tego, że przy tworzenie wynikowego obrazu transformuje się całą scenę do lokalnego układu odniesienia kamery. Podsumowując - jeśli kamera na scenie ma transformację określoną jako C, to macierz widoku $V = C^{-1}$.

> Modyfikując bezpośrednio pola macierzy widoku, pamiętać należy, że jej pola są już inne niż w zwykłej macierzy złożonej z translacji i rotacji.



Rysunek 2.12. Przekształcenia obiektu z przestrzeni lokalnej do przestrzeni świata macierzami transformującymi M_{W1}, M_{W2} i M_{W3}

Macierz widoku ma następującą budowę:

$$\begin{bmatrix} X_x, & Y_x, & Z_x, & 0 \\ X_y, & Y_y, & Z_y, & 0 \\ X_z, & Y_z, & Z_z, & 0 \\ -\vec{X}\cdot\vec{P}, & -\vec{Y}\cdot\vec{P}, & -\vec{Z}\cdot\vec{P}, & 1 \end{bmatrix}$$

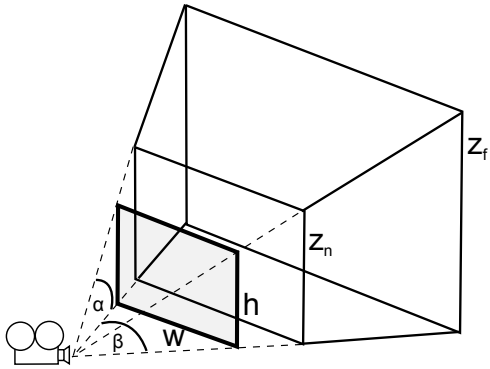
gdzie:

$\vec{X}$, $\vec{Y}$ i $\vec{Z}$ - osie opisujące orientację kamery na scenie
$\vec{P}$ - wektor pozycji kamery

2.6.3 Przestrzeń projekcji

Ponieważ macierz widoku ustala jedynie pozycję i kierunek obserwacji, tak ważne parametry jak zasięg i kąt widzenia wymagają dodatkowego zdefiniowania. W grafice trójwymiarowej fragment sceny, który zostanie odrysowany i rzutowany na płaszczyznę ekranu, określony jest za pomocą bryły widzenia (ang. *viewing frustum*), która w przypadku zastosowania perspektywy ma postać ściętego ostrosłupa o postawie prostokąta (rysunek 2.13). Konstrukt ten charakteryzują następujące wielkości:

- minimalna z_n oraz maksymalna z_f odległość od kamery, czyli pozycja tzw. płaszczyzn przycinania (ang. *clipping planes*)
- współczynnik proporcji (ang. *aspect ratio*) a_r, będący stosunkiem szerokości w do wysokości h
- pola widzenia (ang. *field of view*) α i β dotyczące odpowiednio osi pionowej i poziomej



Rysunek 2.13. Bryła widzenia. Wyróżniona płaszczyzna to okno projekcji

Punkt $[x\ y\ z]$ po rzutowaniu na okno projekcji ma postać $[x'\ y'\ d]$, gdzie d jest odległością okna projekcji od punktu obserwacji. Bazując na rysunku 2.13, można przyjąć, iż widoczność uzależniona jest od spełnienia następujących warunków:

$$-\frac{w}{2} \leqslant x' \leqslant \frac{w}{2} \tag{2.5}$$

$$-\frac{h}{2} \leqslant y' \leqslant \frac{h}{2} \tag{2.6}$$

$$z_n \leqslant z \leqslant z_f \tag{2.7}$$

Znormalizowane współrzędne urządzenia

W praktyce wysokość i szerokość okna projekcji (w i h) nie są, poza teoretycznym opisem modelu, istotne – ważny jest jedynie ich stosunek. Z tego powodu warto przyjąć wartości, dzięki którym obliczenia uproszczą się. Niech $h = 2$, wtedy dla równań 2.5 i 2.6:

$$-a_r \leqslant x' \leqslant a_r$$
$$-1 \leqslant y' \leqslant 1$$

Dodatkowo, aby uczynić przetwarzanie sprzętowe niezależnym od współczynnika proporcji oraz odległości płaszczyzn przycinania, stosuje się dalszą normalizację współrzędnych:

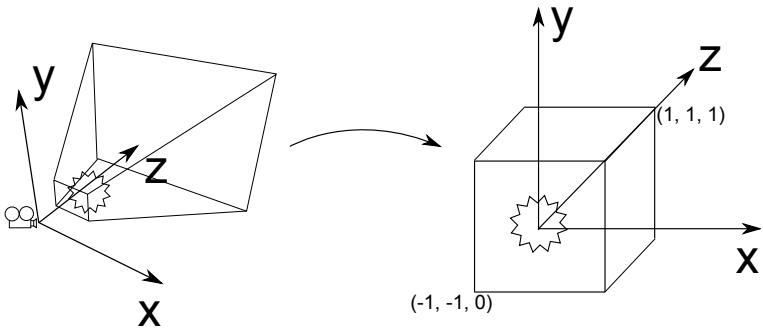
$$-1 \leqslant \frac{x'}{a_r} \leqslant 1$$
$$0 \leqslant f_z(z) \leqslant 1$$

$$f_z :< z_n; z_f > \rightarrow < 0; 1 >$$

gdzie:

f_z - nieliniowa funkcja normalizująca współrzędną z

Znormalizowaną współrzędną z nazywa się też głębokością. Dzięki normalizacji bryła widzenia przekształcona zostaje w prostopadłościan, który określony jest w znormalizowanych współrzędnych urządzenia (ang. *normalized device coordinates*). Warto zwrócić uwagę, że podczas transformacji następuje zrównanie rozmiarów obu podstaw bryły widzenia. Operacja ta odpowiada za uzyskanie efektu perspektywy, ponieważ obiekty znajdujące się bliżej obserwatora zostają rozciągnięte, natomiast bardziej odległe – pomniejszone. Graficzną reprezentację normalizacji przedstawia rysunek 2.14.



Rysunek 2.14. Przejście z bryły widzenia do przestrzeni przycięcia

Macierz projekcji

Za transformację perspektywiczną w lewoskrętnym układzie współrzędnych odpowiada macierz w następującej postaci:

$$M_P = \begin{bmatrix} \frac{ctg\frac{\alpha}{2}}{a_r} & 0 & 0 & 0 \\ 0 & ctg\frac{\alpha}{2} & 0 & 0 \\ 0 & 0 & Q & 1 \\ 0 & 0 & -z_n Q & 0 \end{bmatrix} = \begin{bmatrix} ctg\frac{\beta}{2} & 0 & 0 & 0 \\ 0 & ctg\frac{\alpha}{2} & 0 & 0 \\ 0 & 0 & Q & 1 \\ 0 & 0 & -z_n Q & 0 \end{bmatrix}$$

gdzie:

α - pole widzenia dla osi Y (w radianach)
β - pole widzenia dla osi X (w radianach)
a_r - współczynnik proporcji obrazu
Q - $\frac{z_f}{z_f - z_n}$
z_n - minimalna odległość od kamery
z_f - maksymalna odległość od kamery.

Nie jest możliwe skonstruowanie macierzy, dzięki której można znormalizować koordynaty. Wynika to z nieliniowości odwzorowania $< z_n, z_f >$ do $< 0, 1 >$. Z tego powodu transformacja określona macierzą projekcji dokonuje przekształcenia do przestrzeni projekcji (ang. *projection space*), funkcjonującej również pod nazwą przestrzeni przycinania (ang. *clip space*).

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} M_P = \begin{bmatrix} x \cdot ctg\frac{\beta}{2} & y \cdot ctg\frac{\alpha}{2} & Qz - z_n Q & z \end{bmatrix}$$

Normalizacja możliwa jest dzięki czynnikowi normalizującemu, czyli współrzędnej w wynikowego wektora. Struktura macierzy zapewnia, że jej wartość zawsze będzie równa pierwotnemu koordynatowi z. Podzielenie przez tę wielkość skutkuje uzyskaniem znormalizowanych współrzędnych urządzenia:

$$\begin{bmatrix} \frac{x \cdot ctg\frac{\beta}{2}}{z} & \frac{y \cdot ctg\frac{\alpha}{2}}{z} & Q - \frac{z_n Q}{z} & 1 \end{bmatrix} \tag{2.8}$$

Dobrze oddaje to perspektywiczną naturę projekcji – im większa odległość od kamery, tym większe ściśnięcie wynikowych współrzędnych x i y. Normalizacja zazwyczaj dokonywana jest automatycznie, jednak znajomość tego procesu może być przydatna podczas ręcznego testowania widoczności, niekiedy niezbędnego w implementacji bardziej złożonych algorytmów.

Projekcja ortograficzna

Powyższy opis dotyczy jedynie projekcji perspektywicznej, ponieważ jest ona najczęściej wykorzystywana. Należy jednak pamiętać, że istnieje jeszcze projekcja ortograficzna (ang. *orthographic projection*). W jej przypadku bryła widzenia jest zdefiniowana jako prostopadłościan, przez co nie występuje skalowanie w zależności od odległości, co skutkuje brakiem oddania głębi. Z tego powodu nie może być używana przy tworzeniu realistycznych obrazów. Okazuje się jednak, że jest bardzo użyteczna w różnych inżynierskich programach, gdzie tworzone są schematy. Często wykorzystuje się ją również do rysowania dwuwymiarowych obiektów na ekranie, w tym napisów oraz elementów interfejsu użytkownika.

2.7 Pozostałe definicje matematyczne

2.7.1 Sympleks

Definicja 9 *Sympleks n-wymiarowy w n-wymiarowej przestrzeni euklidesowej to otoczka wypukła $n+1$ afinicznie niezależnych punktów.*

Można powiedzieć, że punkty tworzące sympleks nie leżą na jednej hiperpłaszczyźnie, co dla przestrzeni 2-wymiarowej oznacza nieznajdowanie się na tej samej prostej, a dla przestrzeni 3-wymiarowej – na jednej płaszczyźnie. Intuicyjnie sympleks można rozumieć jako n-wymiarową analogię trójkąta.

Przykładowo:

- Sympleks 0-wymiarowy: punkt.
- Sympleks 1-wymiarowy: odcinek.
- Sympleks 2-wymiarowy: trójkąt.
- Sympleks 3-wymiarowy: czworościan.

2.7.2 Współrzędne barycentryczne

Definicja 10 *Niech $x_0, ..., x_n$ będą wierzchołkami sympleksu n-wymiarowego przestrzeni A. Jeśli dla pewnego punktu $p \in A$ zachodzi równość:*

$$(a_0 + ... + a_n)p = a_0x_0 + ... + a_nx_n \tag{2.9}$$

wtedy współczynniki $a_0, \ldots, a_n$, są współrzędnymi barycentrycznymi punktu p w odniesieniu do punktów $x_0, ..., x_n$.

Stosując za przykład przestrzeń dwuwymiarową, można sobie łatwo wyobrazić, że każdy punkt ze środka trójkąta jest pewną średnią ważoną współrzędnych wierzchołków.

Wniosek 3 *Punkt należy do wnętrza sympleksu, jeżeli:*

$$\forall j \quad (a_j > 0), \quad j = 0, 1, ..., n$$

Wniosek 4 *Punkt należy do krawędzi sympleksu, jeżeli:*

$$\forall j \ (a_j \geqslant 0) \ \wedge \ \exists k \ (a_k = 0), \quad j = 0, 1, ..., n, \ k = 0, 1, ..., n$$

2.7.3 Znormalizowane współrzędne barycentryczne

Definicja 11 *Współrzędne barycentryczne są znormalizowane, jeżeli:*

$$\sum_{j=0}^{n} a_j = 1 \tag{2.10}$$

Wniosek 5 *Dla znormalizowanych współrzędnych barycentrycznych prawdziwa jest zależność:*

$$\forall k \ (a_k = 1 - \sum_{j=0}^{k-1} a_j - \sum_{j=k+1}^{n} a_j), \quad k = 0, 1, ..., n$$

Znormalizowane współrzędne barycentryczne dla sympleksu stopnia n wymagają przechowywania $n - 1$ współrzędnych, gdyż ta pominięta może zostać wyliczona wprost za pomocą równania 5.

2.7.4 Współrzędne barycentryczne dla sympleksu 2-wymiarowego

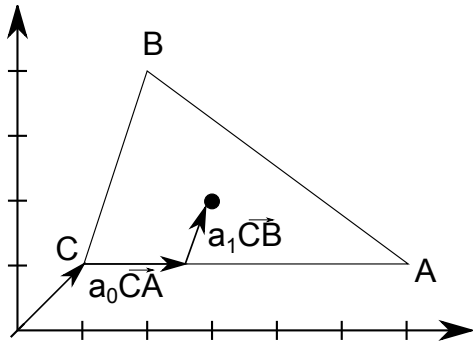
Niech dany będzie sympleks 2-wymiarowy zdefiniowany przez wierzchołki A, B oraz C. Dla każdego punktu p o znormalizowanych współrzędnych barycentrycznych (a_0, a_1, a_0) z równań 2.9 oraz 5 wynika, iż:

$$a_2 = 1 - a_0 - a_1 \tag{2.11}$$

$$p = a_0 A + a_1 B + (1 - a_0 - a_1)C \tag{2.12}$$

$$p = a_0 \vec{CA} + a_1 \vec{CB} + C \tag{2.13}$$

Należy zauważyć, iż z punktu widzenia złożoności obliczeniowej równanie 2.13 jest bardziej optymalne niż równanie 2.12, ponieważ występują w nim tylko dwa mnożenia. Dodatkowo taki opis ma bardzo wygodną i intuicyjną reprezentację geometryczną, którą przedstawia rysunek 2.15. Równania 2.12 i 2.13 występują również pod nazwą interpolacji barycentrycznej.



Rysunek 2.15. Współrzędne euklidesowe wyznaczone ze wzoru 2.13 na podstawie punktów trójkąta ABC oraz znormalizowanych współrzędnych barycentrycznych $a_0 = a_1 = 0.33$

3

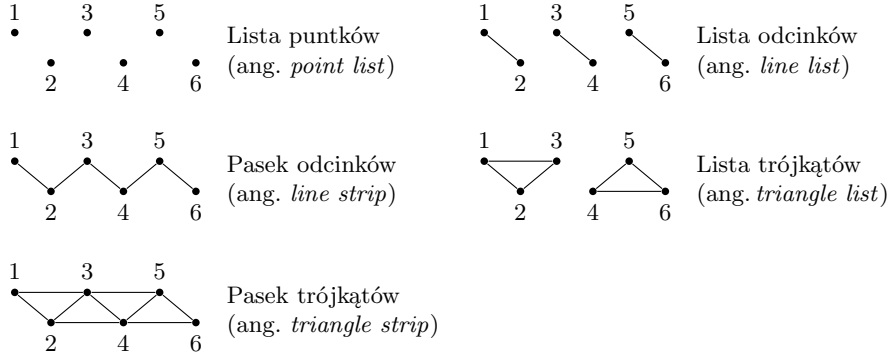
Podstawy niskopoziomowej grafiki 3D

3.1 Wprowadzenie

Programowanie niskopoziomowej grafiki wymaga znajomości pewnych podstawowych, specyficznych dla tej dziedziny pojęć oraz technik. Zamiast operacji na gotowych obiektach bądź prymitywach wymagana jest ręczna obsługa niemal każdego aspektu przetwarzania i generowania obrazu.

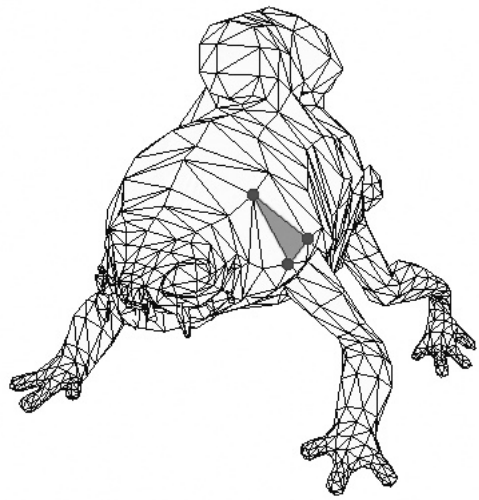
3.2 Wierzchołki, prymitywy i siatki

Na obecnym poziomie geometria reprezentowana jest za pomocą wierzchołków (ang. *vertex*) rozmieszczonych w trójwymiarowej przestrzeni. Ten sam zbiór wierzchołków może tworzyć różne figury geometryczne, w zależności od bieżącego typu prymitywu (ang. *primitive type*). Rysunek 3.1 przedstawia kilka



Rysunek 3.1. Interpretacja wierzchołków w zależności od typu prymitywu

możliwych rezultatów dla różnych typów prymitywów przy jednakowym zbiorze wierzchołków. Częstą praktyką jest tworzenie siatek (ang. *mesh*) z logicznie powiązanych wierzchołków tworzących wielokąty (ang. *polygon*) połączone ze sobą krawędziami. Najczęściej używanymi w tym celu wielokątami są trójkąt i czworokąt. Relacje między wprowadzonymi pojęciami obrazuje rysunek 3.2.



Rysunek 3.2. Przykładowa siatka [9] z trójką wyróżnionych wierzchołków i tworzonym przez nie prymitywem

Dodatkowego wyjaśnienia wymaga pojęcie wierzchołka – zazwyczaj intuicyjnie utożsamia się je wyłącznie z pozycją w przestrzeni. Tymczasem w grafice trójwymiarowej z wierzchołkiem mogą być skojarzone dowolne wielkości, wektorowe bądź skalarne. Przykładowo, często oprócz pozycji dodawana jest również informacja o kolorze, współrzędnych tekstury (znormalizowane współrzędne w prostokątnej tablicy kolorów) oraz o wektorze normalnym.

3.3 Podstawy API graficznych

3.3.1 Potok renderowania

Współczesne karty graficzne według taksonomii Flynna należy zaliczyć do systemów SIMD (ang. *Single Instruction, Multiple Data*). W architekturze tej wiele strumieni danych przetwarzanych jest przez jeden strumień rozkazów, co prowadzi do dużej równoległości obliczeń. Dzięki temu obecnie moc obliczeniowa kart graficznych, zawierających wiele procesorów strumieniowych jest kilkukrotnie wyższa niż procesorów głównych. Przykładowo, teoretyczna moc

obliczeniowa wyrażona we FLOPS'ach (ang. *Floating point operations per second*) najmocniejszego obecnie procesora firmy Intel o nazwie i7-975 wynosi 55.36 GFLOPS[1], natomiast dla karty graficznej firmy NVIDIA GeForce GTX 295 jest równa 1788.48 GFLOPS[2]. Wartości te, z powodu ich głównie marketingowego charakteru, zostały wyznaczone przy bardzo optymistycznych założeniach. Niemniej stanowią pewne źródło odniesienia pozwalające ocenić, jaka przepaść wydajnościowa dzieli oba typy układów.

Potok renderowania (ang. *rendering pipeline*) to sekwencja operacji, jakim poddawane są dane wejściowe w celu uzyskania rastrowego obrazu 2D[3]. Potok jest specyficzny dla danego API graficznego, jednak można wyróżnić wspólne cechy:

- Przetwarzanie w potoku jest jednostronne.
- Etapy potoku dzielą się na programowalne i nieprogramowalne.
- W potoku następuje równoległe przetwarzanie wielu strumieni danych.

3.3.2 Jednostki cieniowania

Jednostka cieniowania (ang. *shader*) to program używany przez programowalną część potoku renderowania. Mimo że obecne wysokopoziomowe języki cieniowania syntaktycznie przypominają język C, architektura typu SIMD wymusza odmienny sposób programowania. Negatywną konsekwencją równoległego przetwarzania danych jest utrudniona implementacja złożonych algorytmów. W jednostkach cieniowania nie można w żaden sposób zapamiętać stanu zmiennej do wykorzystania w innym wywołaniu oraz nie ma możliwości modyfikacji globalnych danych. Zamiast tego wygenerowane dane przesyłane są bezpośrednio do kolejnego etapu potoku renderowania. Prowadzi to do sytuacji, gdzie programy cieniowania są zazwyczaj stosunkowo krótkie i pozbawione wielu rozgałęzień.

3.4 Techniki

Poza znajomością podstawowych pojęć z dziedziny programowania z użyciem API graficznych do łatwiejszego zrozumienia dalszej części książki konieczna jest wiedza na temat najbardziej powszechnych technik i rozwiązań wpływających na sposób tworzenia logiki silników graficznych aplikacji.

3.4.1 Bufory wierzchołków i indeksów

Karty graficzne, jako wyspecjalizowane układy są w stanie przeprowadzać obliczenia w tempie znacznie większym niż procesor główny. W związku z tym

[1] Źródło: `http://www.intel.com/support/processors/sb/cs-023143.htm`
[2] Źródło: `http://www.nvidia.com/object/product_geforce_gtx_295_us.html`
[3] Prostokątna siatka kolorowych punktów

bardzo istotne staje się efektywne zaopatrywanie procesorów strumieniowych w dane. Obecnie karty graficzne z głównego nurtu wytwarzane są w standardzie PCI Express, którego magistrala jest w stanie zapewnić przepustowość rzędu 500 MB/s na pojedynczej linii[4]. Powszechnie używane są sloty umożliwiające dostęp do 16 linii, wobec czego maksymalna teoretyczna prędkość odbioru danych wynosi około 8 GB/s. Dla porównania przepustowość pamięci kart graficznych już w 2006 r. zbliżała się do 100 GB/s[10]. Wobec takiej różnicy, w czasie dostępu ważne jest, aby w jak największym stopniu korzystać z możliwości buforowania danych, przy czym należy wziąć pod uwagę ograniczoną pojemność pamięci układu.

Minimalizacji intensywności transferu geometrii pomiędzy pamięcią operacyjną a kartą graficzną służy mechanizm buforów wierzchołków (ang. *vertex buffers*). Zamiast wysyłania każdej ramki danych do odrysowania wykonywany jest jednorazowy zapis informacji w pamięci karty graficznej. Po takiej operacji odrysowanie wybranych wierzchołków sprowadza się do wskazania tej części bufora, która ma być wykorzystana. Ze względu na ograniczoną pojemność pamięci karty graficznej dla skomplikowanych scen zawierających znaczną ilość geometrii rodzi się problem efektywnego zarządzania buforami.

Metodą pozwalającą zmniejszyć łączny rozmiar przechowywanych danych oraz ilość danych przetwarzanych w potoku jest zastosowanie buforów indeksów (ang. *index buffers*). Miejscem ich zapisu jest też pamięć karty graficznej, jednakże zamiast informacji o wierzchołkach przechowują one indeksy wykorzystywane do konstrukcji kolejnych prymitywów. Dzięki temu istnieje możliwość kilkakrotnego wykorzystania danych i obliczeń dotyczących tego samego wierzchołka, jeśli należy on do więcej niż jednego prymitywu. Oszczędność pamięci wynika z faktu, iż zazwyczaj rozmiar pojedynczego indeksu jest kilkakrotnie mniejszy od rozmiaru danych reprezentujących wierzchołek. Przykładowe zastosowanie tego mechanizmu obrazuje rysunek 3.3.

Rozmiar S_i bufora indeksów dla N prymitywów, przy n_p przypadających wierzchołków na prymityw oraz o s_i rozmiarze indeksu określony jest następująco:

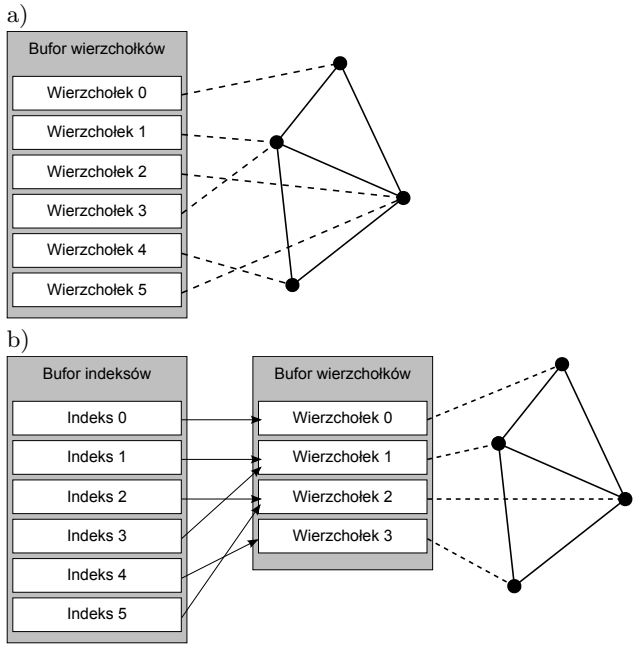
$$S_i = N \cdot n_p \cdot s_i \tag{3.1}$$

Najczęściej indeks ma rozmiar $s_i = 2$. Przy takim założeniu siatka może składać się maksymalnie z 65536 wierzchołków. Aby stworzyć bardziej skomplikowaną siatkę, można albo podzielić ją na podzbiory, albo też użyć indeksu o większym rozmiarze.

3.4.2 Łańcuch wymiany

Obraz wyświetlany na ekranie monitora pochodzi bezpośrednio z karty graficznej z obiektu o nazwie *ang. front buffer*, w którym znajdują się dane bieżącej ramki. Częstotliwość odświeżania ekranu jest wielkością rzędu kilkudziesięciu

[4] Dotyczy to standardu w wersji 2.0.
Źródło: `http://www.pcisig.com/news_room/faqs/pcie3.0_faq`

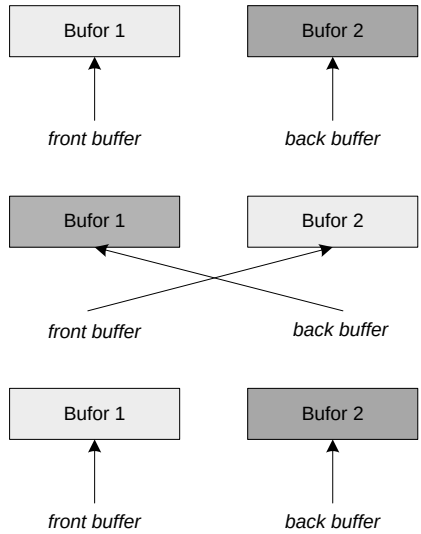


Rysunek 3.3. Czworobok stworzony przy użyciu bufora wierzchołków (a) oraz przy użyciu bufora wierzchołków i bufora indeksów (b)

Hz (obecnie przeciętnie 60 Hz) i, co ważne, sam akt odświeżania nie zachodzi na żądanie aplikacji. Z tej sytuacji rodzi się problem synchronizacji zapisu (przez aplikację) i odczytu (przez monitor) danych bieżącej ramki. Przykładowo, jeżeli aplikacja modyfikuje *front buffer*, to w trakcie odświeżania ekranu wyświetlany obraz będzie składał się z kilku potencjalnie niespójnych części. Rozwiązaniem może być nałożenie blokady na dane ramki podczas odczytu przez monitor tak, żeby wszystkie programy chcące dokonać aktualizacji czekały na jej zniesienie. Niestety, biorąc pod uwagę przytoczoną częstotliwość w idealnym przypadku (tj. zerowy czas samego odświeżania) oznaczałoby to jedynie $\frac{1}{60}$s na cały proces odrysowania sceny dla każdej z aplikacji, co w wielu przypadkach stanowi zdecydowanie zbyt krótki okres.

Powszechnie używanym rozwiązaniem jest mechanizm *surface flipping* działający według założenia, że aplikacja nigdy nie modyfikuje bezpośrednio bieżącej ramki znajdującej się we *front buffer*. Zamiast tego rysowanie odbywa się do niewyświetlanej powierzchni (ang. *off-screen surface*) reprezentowanej przez dane w buforze o nazwie *back buffer*. Kiedy aplikacja zakończy generację ramki i poleci zaprezentowanie obrazu (czyli przeniesienie danych z *back buffer* do *front buffer*) wykonywana jest operacja zamiany ról obu buforów - po tej czynności *front buffer* staje się *back buffer* i na odwrót. Ponieważ zamianie ule-

gają de facto wskaźniki wyznaczające bufory (jak na rysunku 3.4), cały mechanizm prezentacji jest bardzo szybki i możliwy do wykonanie w czasie pomiędzy odświeżeniami ekranu. W praktyce niewyświetlanych powierzchni może być więcej - formowana jest wtedy kolejka. Relaceje pomiędzy buforem bieżącej ramki oraz buforami przejściowymi określane są mianem łańcucha wymiany (ang. *swap chain*). W przypadku jednej niewyświetlanej powierzchni często używa się również nazwy podwójne buforowanie (ang. *double buffering*).



Rysunek 3.4. Dwukrotny *surface flipping*

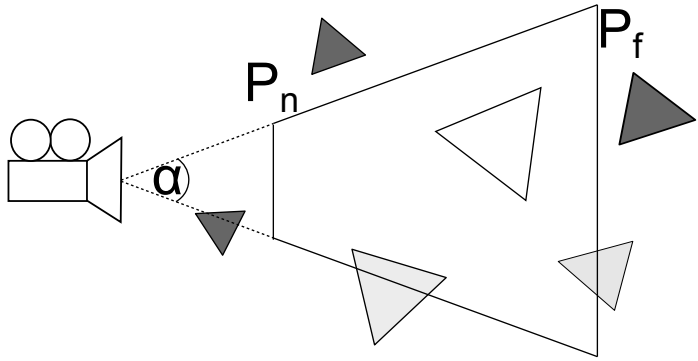
3.4.3 Usuwanie niewidocznych prymitywów

Ogromna moc obliczeniowa nowoczesnych kart graficznych nie jest w żadnym wypadku furtką do stosowania nieoptymalnych rozwiązań. Co więcej, ponieważ charakter przetwarzania wierzchołków jest bardziej ilościowy niż jakościowy (szybkie, proste obliczenia dużej liczby wierzchołków), wraz ze wzrostem liczności danych wejściowych proporcjonalnie zwiększa stracony potencjał obliczeniowy. Z tego powodu powszechne zastosowanie znalazły metody automatycznego zmniejszania ilości danych do przetwarzania w ramach potoku.

Usuwanie prymitywów spoza bryły widzenia

Usuwanie prymitywów spoza bryły widzenia jest najbardziej intuicyjnym sposobem eliminacji zbędnych, z punktu widzenia efektu końcowego, prymitywów. To, co intuicyjnie jest oczywiste, praktycznie okazuje się nietrywialnym prob-

lemem. Okazuje się, że znacznie lepiej i wygodniej jest dokonywać testów nie w intuicyjnych przestrzeniach świata i widoku, lecz w przestrzeni przycinania. Działania potrzebne do ustalenia przynależności do bryły widoku sprowadzają się w niej do jedynie sześciu porównań skalarów [5]. Przykład testów dla kilku prymitywów przedstawia rysunek 3.5.

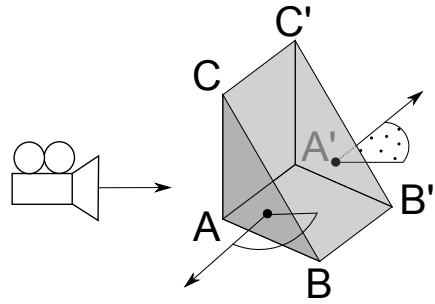


Rysunek 3.5. Testowanie widoczności na przynależności do wnętrza bryły widzenia. Ciemnym kolorem oznaczono elementy niewidoczne – spoza bryły widzenia. Biały kolor nadany jest prymitywom całkowicie widocznym, natomiast pośredni – częściowo widocznym. P_n i P_f to odpowiednio bliska i daleka płaszczyzna przycinania

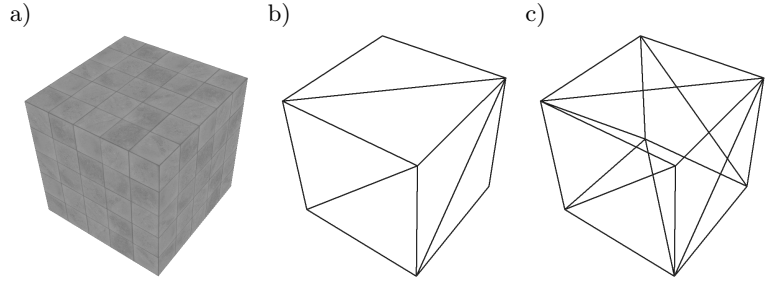
Usuwanie tylnych ścian

Mechanizm usuwania tylnych ścian (ang. *backface culling*) pozwala zmniejszyć liczbę przetwarzanych trójkątów poprzez eliminację tych, które zawsze będą niewidoczne. Technika ta do oceny widzialności nie używa ani informacji o scenie, ani nawet o bieżącej siatce. Nie uwzględnia przez to wszystkich przypadków, ale mimo to pozwala na wyeliminowanie bardzo niskim kosztem obliczeniowym często około połowy prymitywów sceny.

U podstaw tej techniki leży obserwacja, że o ile siatka jest ciągła, to jest każdy prymityw współdzieli wszystkie swoje krawędzie z innymi prymitywami składowymi, albo też tak jest umiejscowiona, że obserwator nigdy nie dostrzeże miejsc nieciągłości, to wtedy, jej tylne względem obserwatora ściany zawsze będą przesłonięte przednimi. Jest to wniosek zgodny z rzeczywistością, natomiast samo testowanie orientacji jest zaskakująco łatwe. Otóż wystarczy bowiem zbadać kąt pomiędzy wektorem poprowadzonym od obserwatora do powierzchni a jej wektorem normalnym. Jeżeli bezwzględna wartość rezultatu jest mniejsza niż 90°, oznacza to, że dana ściana zawsze będzie niewidoczna. Fizycznie ma to swoje uzasadnienie w fakcie, że od tak zorientowanej względem obserwatora powierzchni światło nigdy nie odbije się bezpośrednio w jego kierunku.



Rysunek 3.6. Testowanie widoczności na podstawie kąta między wektorem normalnym a wektorem widoku. Deseniem zaznaczono kąt mniejszy od 90°, czyniący ścianę niewidoczną



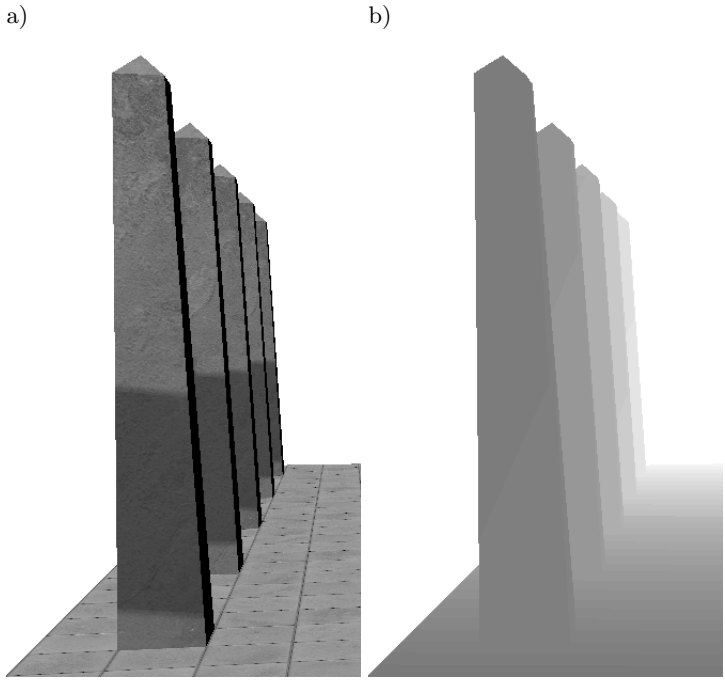
Rysunek 3.7. Przykład zastosowania usuwania niewidocznych ścian: a) sześcian po odrysowaniu; b) obrys przetworzonych trójkątów; c) obrys wszystkich trójkątów składowych

3.4.4 Bufor głębokości

Bufor głębkości (ang. *depth buffer*) lub bufor Z - to mechanizm używany do określania wzajemnego przysłaniania obiektów na scenie trójwymiarowej. Nazwa jest nieprzypadkowa - bufor służy do śledzenia głębokości (ang. *depth*), czyli funkcji odległości pomiędzy obserwatorem a obiektem, który wygenerował dany piksel. Wszystkie potencjalne piksele powstałe wskutek odrysowania nowego obiektu są poddawane następującemu testowi:

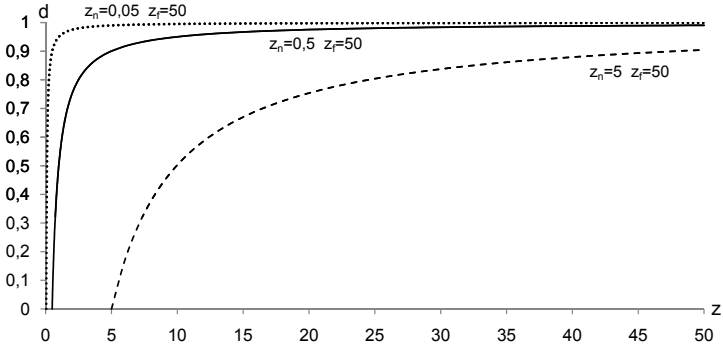
$x :=$ współrzędna x rasteryzowanego piksela
$y :=$ współrzędna y rasteryzowanego piksela
$d :=$ głębokość piksela w punkcie (x, y)
if $d < buforZ(x, y)$ **then**
    $buforZ(x, y) := d$
    $ramka(x, y) :=$ kolor prymitywu w punkcie (x, y)
end if

Dzięki takiemu algorytmowi programista ma gwarancję, że bez sortowania obiektów na scenie (względem ich odległości od kamery) wyświetlana kolejność będzie prawidłowa (patrz rysunek 3.8). W praktyce można dla zadanych obiektów wyłączać możliwość odczytu lub zapisu bufora Z i zmieniać wewnętrzny warunek (tutaj znak mniejszości), uzyskując tym samym pewne efekty graficzne. Dotyczy to zwłaszcza półprzeźroczystych fragmentów sceny, dla których wykonuje się jedynie odczyt głębokości, a zawartość pozostawia się bez zmian.



Rysunek 3.8. Przykład zastosowania bufora głębokości: a) fragment odrysowywanej sceny; b) analogiczny fragment bufora głębokości

W kontekście testowania głębokości pojawia się również zagadnienie doboru odległości między płaszczyznami przycinania. Głębokość asymptotycznie dąży do jedności, jednak tempo wzrostu zależy bezpośrednio od zakresu widzenia kamery. Im większy jest dystans między płaszczyznami przycinania, tym mniej równomierna jest dystrybucja głębokości (rysunek 3.9). W połączeniu ze skończoną precyzją cyfrowej reprezentacji liczb zmiennoprzecinkowych prowadzić to może do zjawiska tzw. *z-fightingu*, polegającego na wzajemnym przenikaniu się sąsiednich równoległych powierzchni. Z tego powodu dobrą praktyką jest pozycjonowanie płaszczyzn przycinania tak blisko siebie, jak to jest tylko w kontekście zamierzonego efektu wizualnego możliwe.



Rysunek 3.9. Krzywa głębokości w zależności od odległości między płaszczyznami przycinania

4

Podstawy Direct3D 10

4.1 Wprowadzenie

Direct3D to niskopoziomowe API (ang. *application programming interface*) graficzne firmy Microsoft. Biblioteka pozwala na ręczną manipulację układem graficznym, a co za tym idzie, na wykorzystanie jego potencjału obliczeniowego. Teoretycznie, o ile konfiguracja sprzętowa zgodna jest z wymaganiami zadanej wersji Direct3D, to wykonanie tej samej czynności przy użyciu zupełnie odmiennych układów graficznych sprowadza się do tego samego ciągu poleceń. Mimo iż w praktyce wciąż stosuje się optymalizacje pod kątem różnych platform, w znaczny sposób upraszcza to pracę programisty oraz gwarantuje większą przenośność kodu[1]. Niniejsza książka skupia się na wersji Direct3D opatrzonej numerem 10.

4.1.1 Uwagi odnośnie do poprzednich wersji

Dla osób z doświadczeniem w używaniu poprzednich wersji czeka kilka niemiłych niespodzianek. Ponieważ zamierzeniem twórców było ujednolicenie oraz uproszczenie zarówno wewnętrznego działania, jak i interfejsu publicznego, nowa wersja nie jest wstecznie zgodna. Zmiany dotyczą nie tylko nazewnictwa czy przemodelowania struktury API - pewne funkcjonalności, zdaniem twórców zbędne, zostały usunięte, a programiści tym samym zmuszeni do stosowania nowszych technologicznie rozwiązań[2]. Oznacza to, że przenoszenie starego kodu (oraz doświadczeń) może napotkać znaczne problemy. Istotną różnicą jest również fakt zaostrzenia kryteriów zgodności układów sprzętowych. Podczas gdy w poprzednich wersjach programista nie mógł być pewien sprzętowej implementacji wszystkich funkcjonalności API, Direct3D 10 na sztywno definiuje zestaw wymagań względem urządzenia.

[1] Oczywiście wyłącznie w zakresie platform ściśle określonych przez Microsoft

[2] Pełna lista zmian dostępna jest pod adresem `http://msdn.microsoft.com/en-us/library/cc308047`.

4.1.2 Uwagi odnośnie do COM

COM (ang. *Common Object Model*) to model komponentów i komunikacji opracowany przez Microsoft w celu uzyskania niezależności od języka i zgodności wstecznej. Niestety, wiąże się to z utrudnionym i nieintuicyjnym, zwłaszcza przy pierwszym kontakcie, sposobem użycia. Omówieniu i wytłumaczeniu modelu można by poświęcić wiele stron i rozdziałów, ale dla programisty Direct3D niezbędna jest tylko podstawowa wiedza.

Obiekty COM udostępniane są programiście jako interfejsy, przez co większość informacji jest ukryta. Konsekwencją tego faktu oraz wewnętrznego zarządzania pamięcią jest odmienny sposób tworzenia i usuwania obiektów: operator `delete` zastąpiony jest przez metodę `Release`, natomiast `new` przez funkcje bądź metody innych interfejsów. W sensie inżynierii oprogramowania wyczerpany jest wiec wzorzec projektowy metoda wytwórcza. Należy jednak mieć na uwadze, że przy COM nie trzeba stosować dodatkowego mechanizmu „automatycznych" wskaźników, gdyż zliczanie referencji jest wbudowane we wszystkie obiekty. Warunkiem jest jedynie konieczność wywoływania metody `Release`, gdy skończy się pracę z danym obiektem.

Z podporządkowania Direct3D modelowi COM wynika pośrednio niemożność używania bardziej zaawansowanych funkcjonalności dostarczanych przez konkretne języki, takie jak przestrzenie nazw. W związku z tym, aby wyróżnić typy i funkcje biblioteki od pozostałych, projektanci zdecydowali się do nazw wszystkich typy i funkcji dodać przedrostek `D3D10` (w przypadku interfejsów jest to `ID3D10`).

4.1.3 Biblioteki pomocnicze

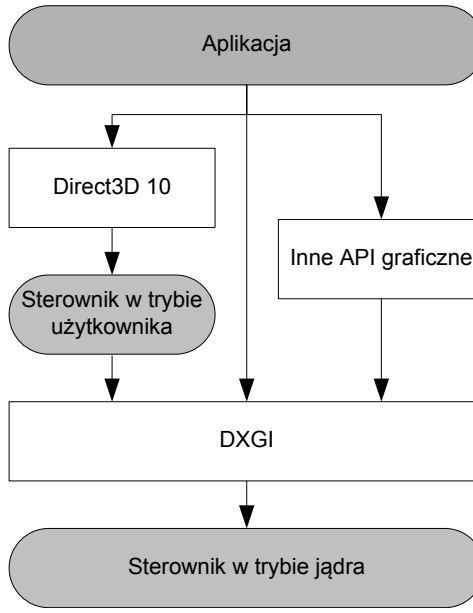
Direct3D jest w znacznym stopniu niezależny od komponentów DirectX. Podczas pracy z API występują jednak sytuacje, gdzie pozostałe składowe biblioteki wydatnie ułatwiają osiągnięcie efektów bądź też wręcz zapewniają nowe niskopoziomowe mechanizmy.

DirectX Graphics Infrastructure

Podstawowym zadaniem DirectX Graphics Infrastructure (DXGI) jest wykonywanie niskopoziomowych zadań, które mogą być realizowane niezależnie od Direct3D. Wobec tego DXGI pośredniczy między aplikacjami i Direct3D a sterownikiem i sprzętem m.in. w zakresie wykrywania urządzeń oraz prezentacji i kontroli obrazu. Relacje te przedstawia rysunek 4.1. Wszystkie interfejsy zdefiniowane w DirectX Graphics Infrastructure mają przedrostek `IDXGI`, natomiast pozostałe typy i funkcje – `DXGI`.

Direct3D Extension

Direct3D Extension (D3DX) - to biblioteka mająca wspierać pozostałe części API. W stosunku do Direct3D poziom abstrakcji jest znacznie wyższy, przez co



Rysunek 4.1. Komunikacja pomiędzy aplikacjami, DXGI oraz systemem

programowanie nabiera bardziej wysokopoziomowego charakteru. Biblioteka zawiera gotowe rozwiązania często spotykanych problemów podczas tworzenia aplikacji 3D.

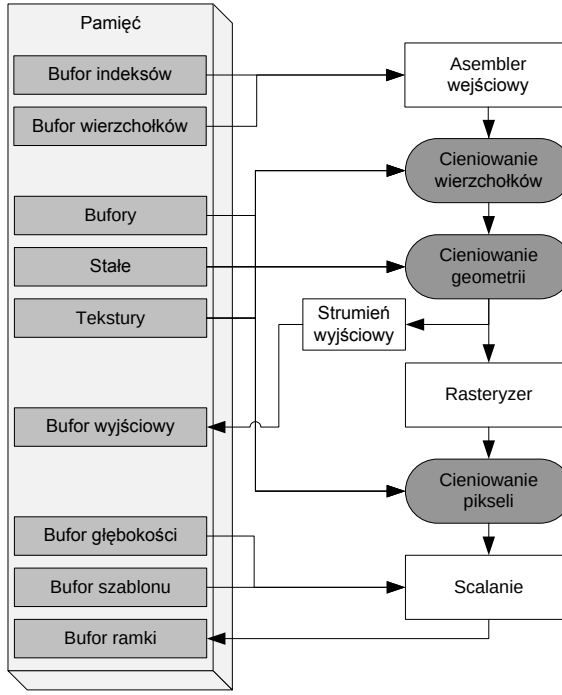
Do oferowanych funkcjonalności można zaliczyć m.in.:

- obliczenia matematyczne dotyczące wektorów, kwaternionów, macierzy i kolorów;
- wczytywanie zasobów takich jak tekstury, siatki i pliki efektów (.fx);
- optymalizacja siatek;
- czcionki 2D i 3D;

Sama biblioteka jest opcjonalna w stosunku do Direct3D, jednak z powodu występowania w tym samym SDK, wszystkie przykładowe programy opisane w tej książce korzystają z niej. Interfejsy zdefiniowane w Direct3D Extension mają przedrostek `ID3DX`, natomiast pozostałe typy i funkcje – `D3DX`.

4.2 Potok renderowania

Przed przystąpieniem do pracy z Direct3D 10 konieczne jest poznanie, chociaż w ogólnym stopniu, jego sposobu przetwarzania danych. Potok renderowania API jest przeznaczony do generowania grafiki w czasie rzeczywistym. Na rysunku 4.2 pokazano przepływ danych pomiędzy kolejnymi etapami.



Rysunek 4.2. Potok renderowania Direct3D 10

4.2.1 Etapy potoku renderowania

- Asembler wejściowy (ang. *input assembler*) odpowiada za dostarczenie danych do etapu cieniowania wierzchołków. Kontrola sposobu tłumaczenia surowych danych z pamięci do postaci skalarów, wektorów i macierzy odbywa się poprzez wskazanie struktury zawierającej układ danych wejściowych (ang. *input layout*). Dodatkowo należy ustawić przetwarzany typ prymitywu. Źródło danych może stanowić zbiór maksymalnie 16 buforów wierzchołków oraz maksymalnie jeden bufor indeksów.
- Cieniowanie wierzchołków (ang. *vertex shader*) to obowiązkowy programowalny etap potoku. Uruchamiany jest dla każdego wejściowego wierzchołka i musi wygenerować dokładnie jeden wierzchołek wyjściowy. Do typowych zastosować można zaliczyć przekształcenie z przestrzeni modelu do przestrzeni świata, widoku lub projekcji.
- Cieniowanie geometrii (ang. *geometry shader*) to nieobowiązkowy programowalny etap potoku renderingu operujący na prymitywach. W odróżnieniu od poprzedzającego go etapu istnieje możliwość wygenerowania zmiennej liczby prymitywów wyjściowych.
- Strumień wyjściowy (ang. *stream output*) to nieobowiązkowy nieprogramowalny etap potoku. Jeśli strumieniowanie danych jest włączone, dane wy-

generowane przez etap cieniowania geometrii zapisywane są w pamięci karty graficznej. Takie informacje mogą być odczytane przez aplikację.
- Rasteryzer (ang. *rasterizer*) odpowiada za wytworzenie obrazu rastrowego na podstawie otrzymanych wierzchołków oraz typu prymitywu. Eliminuje również z dalszego przetwarzania prymitywy na bazie testu tylnych ścian oraz bryły widzenia. Etap rasteryzacji kontrolowany jest za pomocą struktury z informacjami o stanie.
- Cieniowanie pikseli (ang. *pixel shader*) to programowalny nieobowiązkowy etap potoku. Etap otrzymuje interpolowane dane wierzchołków dla każdego piksela powstającego w konsekwencji rzutowania prymitywu na płaszczyznę. Wynikiem jego działania musi być dokładnie jeden kolorowy piksel. Do typowych zastosowań można zaliczyć nakładanie tekstury i obliczanie oświetlenia.
- Etap scalania (ang. *output merger*) odpowiada za ustalenie widoczności piksela przy użyciu technik bufora głębokości oraz bufora maski. Dodatkowo musi przeprowadzić aktualizację każdego z celów renderingu (ang. *render target*).

Potok renderowania jest *de facto* maszyną stanów, a składają się na nią stany każdego z etapów. Dla programowalnych fragmentów stanem będzie kod jednostki cieniowania, dla nieprogramowalnych będą to wartości pewnych parametrów. Co ważne, ten wyraźny podział nie jest tylko ciekawostką, lecz ma odzwierciedlenie w praktycznie każdym aspekcie używania API. Dzięki temu, po zrozumieniu podstawowych zasad obowiązujących w bibliotece i oswojeniu się z niskim poziomem abstrakcji, interfejs publiczny okazuje się być bardzo czytelny i precyzyjny.

4.2.2 Przepływ danych

Poniższy pseudokod pokazuje przybliżony przepływ danych w potoku renderowania. Dla klarowności przyjęto, iż używane są faza cieniowania geometrii i pikseli.

P - prymityw wejściowy
V - wierzchołek prymitywu wejściowego
V_{vs} - wierzchołek transformowany przez VS
P_{temp} - prymityw stworzony przez przetransformowane wierzchołki
P_{gs} - prymityw stworzony przez GS
p - interpolowane dane wierzchołków

for all $P \in InputAssembler$ **do**
    Wyzeruj P_{temp}
    for all $V \in P$ **do**
        $V_{vs} := VertexShader(V)$
        Dodaj V_{vs} do P_{temp}
    end for

```
    for all P_gs ∈ GeometryShader(P_temp) do
        if StreamOutput then
            Dodaj P_gs do bufora wyjściowego
        end if
        for all (x, y) ∈ Rasterizer(P_gs) do
            p := interpolowane atrybuty wierzchołków P_gs w punkcie (x, y)
            OutputMerger[x, y] := PixelShader(p)
        end for
    end for
end for
```

4.2.3 Efekty, techniki i przebiegi

Stan potoku może być ustalany nie tylko bezpośrednio w kodzie aplikacji. Direct3D wprowadza pojęcie efektu (ang. *effect*) – bytu, który pozwala na pewne podniesienie poziomu abstrakcji, zwłaszcza gdy uzyskanie konkretnego rezultatu graficznego wymaga wielokrotnej manipulacji stanem potoku.

Do opisu tego, czym dokładnie jest efekt, najlepiej zacząć od liścia w hierarchii. Jest nim przebieg (ang. *pass*) – pojedynczy zestaw stanów dla wybranych bądź wszystkich etapów potoku. Do stanu zaliczają się ustawienia dla etapów nieprogramowalnych, ale też programy cieniowania. Zazwyczaj jeden przebieg wystarczy, aby geometria mogła być odrysowana. Często jednak do uzyskania bardziej skomplikowanych efektów konieczne jest kilkakrotne przetworzenie tej samej geometrii przez potok renderowania. Zapanować nad tym stanem rzeczy pozwalają techniki (ang. *techniques*) – byty, które agregują i ustalają kolejność przebiegów. Jeżeli technika zawiera więcej niż jeden przebieg, nazywana jest wieloprzebiegową (ang. *multi-pass technique*). Efekt w tej hierarchii znajduje się na szczycie. Jego zadanie nie sprowadza się jednak tylko do agregacji technik, bowiem to w jego przestrzeni definiowane są stany używane później przez przebiegi. Dzięki temu ten sam obiekt stanu może być wykorzystany przez wiele przebiegów, a w konsekwencji przez wiele technik.

Nie należy efektów traktować jako alternatywnego względem standardowego API sposobu na kontrolę potoku. Jest to raczej dobrze przygotowana warstwa abstrakcji i automatyzacji, w której kompetencje w każdej chwili można wkroczyć z poziomu kodu aplikacji. W praktyce, to umiejętna współpraca pomiędzy aplikacją a efektami pozwala na szybkie i niezawodne osiągnięcie bardzo dobrych rezultatów graficznych.

4.3 Inicjalizacja

By móc zainicjować Direct3D 10, konieczne jest posiadanie uchwytu do okna docelowego. Wobec tego najpierw należy przygotować szkielet okienkowej aplikacji albo bezpośrednio używając `WinAPI`, albo za pośrednictwem biblioteki

wyższego poziomu, jak np. MFC. Dopiero wtedy można przystąpić do dalszych czynności, które w tym przypadku będą sprowadzały się do wypełnienia ciała poniższej funkcji.

```
//! \param hWnd Uchwyt do okna
//! \returns S_OK jeżeli się powiodło, kod błędu w przeciwnym wypadku
HRESULT CreateDeviceAndSwapChain(HWND hWnd) {
  HRESULT hr;
  ...
  return S_OK;
}
```

4.3.1 Konfiguracja łańcucha wymiany

Za konfigurację łańcucha wymiany w Direct3D 10 odpowiada struktura DXGI_SWAP_CHAIN_DESC. Rozwiązanie takie jest typowe dla Direct3D - konfigurację lub tworzenie obiektów przeprowadza się przy użyciu mniej lub bardziej złożonych struktur przekazywanych jako parametry. Poniższy kod przedstawia przykładowy sposób ustawienia parametrów łańcucha wymiany.

```
// pobranie rozmiaru okna
RECT clientRect; GetClientRect( hWnd, &clientRect ); UINT width =
clientRect.right - clientRect.left; UINT height = clientRect.bottom
- clientRect.top;
// ustawienie parametrów łańcucha wymiany
DXGI_SWAP_CHAIN_DESC swapChainDesc; ZeroMemory( &swapChainDesc,
sizeof( swapChainDesc ) );
// wyświetlanie w oknie o zadanym uchwycie
swapChainDesc.OutputWindow = hWnd; swapChainDesc.Windowed = TRUE;
// sterownik sam będzie zarządzał buforami; nie można się spodziewać,
// że po zamianie buforów będą w nich poprzednio wprowadzone dane
swapChainDesc.SwapEffect = DXGI_SWAP_EFFECT_DISCARD;
// jeden back buffer
swapChainDesc.BufferCount = 1;
// zadany rozmiar oraz format buforów
swapChainDesc.BufferDesc.Width = width;
swapChainDesc.BufferDesc.Height = height;
swapChainDesc.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
// bufor używany do prezentacji obrazu
swapChainDesc.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
// bez multisamplingu
swapChainDesc.SampleDesc.Count = 1; swapChainDesc.SampleDesc.Quality
= 0;
```

> Chociaż struktury Direct3D często wyposażone są w udogodnienia z rodziny C++, jak np. przeciążone operatory, to jednak domyślne zerujące konstruktory nie są jednymi z nich. Wynika z tego, iż każdą strukturę powinno się zaraz po utworzeniu wyzerować tak, żeby nie zawierała przypadkowych danych. Można tego dokonać albo ręcznie przypisując polom wartość zerową, albo posługując się funkcjami bibliotecznymi typu memset lub ZeroMemory.

Podstruktura DXGI_SAMPLE_DESC (tutaj jako pole SampleDesc) kontroluje mechanizm tzw. próbkowania wielokrotnego (ang. *multisampling*). Dokładny opis działania tej techniki oraz sposób określania możliwych parametrów

znajduje się w dokumentacji Direct3D 10. W niniejszej książce próbkowanie wielokrotne zawsze jest wyłączone, czemu odpowiada przypisanie wartości jak w przytoczonym kodzie.

4.3.2 Urządzenie Direct3D 10

Dysponując wypełnioną strukturą `DXGI_SWAP_CHAIN_DESC` można przejść do stworzenia urządzenia Direct3D 10 (`ID3D10Device`) oraz właściwego łańcucha wymiany (`IDXGISwapChain`). Interfejs `ID3D10Device` to serce aplikacji opartej na omawianym API - to za jego pomocą odbywa się całe sterowanie układem graficznym, począwszy od alokacji danych, przez kontrolę stanu na rysowaniu kończąc. Drugi z interfejsów, `IDXGISwapChain`, reprezentuje bufory, do których następuje rysowanie przed wyświetleniem na ekranie i z tego powodu zapewnia metody pozwalające na reakcję, na przykład zmianę rozmiaru okna bądź wyjścia z trybu pełnoekranowego. Ze względu na charakter książki obsługa tego typu zdarzeń nie jest ani uwzględniona w przykładach, ani omawiana.

W Direct3D 10 inicjacji można dokonać na dwa sposoby. Pierwszy, będący rozwiązaniem "w starym stylu", to posłużenie się funkcją biblioteczną, która tworzy oba obiekty za pomocą jednego tylko wywołania.

```
// zmienne globalne
ID3D10Device* g_device = NULL; IDXGISwapChain* g_swapChain = NULL;
... CHECK_HR(D3D10CreateDeviceAndSwapChain(NULL,
D3D10_DRIVER_TYPE_HARDWARE,
  NULL, D3D10_CREATE_DEVICE_DEBUG, D3D10_SDK_VERSION, &swapChainDesc,
  &g_swapChain, &g_device));
```

Funkcje Direct3D zwracają wartość typu `HRESULT`, która mówi o powodzeniu wykonania bądź napotkanych błędach. Makro `CHECK_HR` używane w przykładach sprawdza rezultat wywołanej funkcji i przerywa wykonywanie, jeżeli napotka błąd. Jego definicja wygląda następująco:

```
#define CHECK_HR(func)    \ do {                        \
  hr = (func);            \
  if (FAILED(hr)) {       \
    DebugBreak();         \
    return hr;            \
  }                       \
} while(0)
```

Zawarcie właściwych poleceń w bloku `do-while` jest praktycznym zastosowaniem idiomu makra wielopoleceniowego (ang. *multi-statement macro*)[16].

Wyjaśnienia wymaga rozróżnienie pomiędzy urządzeniem typu `HARDWARE` a `REFERENCE` (warianty wyliczenia `D3D10_DRIVER_TYPE`). Pierwsza wartość nakazuje Direct3D stworzyć urządzenie, które wszystkie funkcje realizuje sprzętowo. Jest to oczywiście najbardziej wydajny wariant, lecz w przypadku braku posiadania urządzenia graficznego zgodnego ze specyfikacją – nieosiągalny.

Drugi zaś powoduje, że stworzone zostaje urządzenie realizujące programowo wszystkie operacje. Jednak można się spodziewać, że wydajność takiego rozwiązania w stosunku do sprzętowego jest fatalna. Taki wariant jest więc deską ratunkową, gdy bieżąca maszyna nie spełnia wszystkich wymagań. Podkreślić należy, iż taki fragment nie powinien znaleźć się w końcowej aplikacji - urządzenie `REFRENCE` jest przeznaczone tylko dla testów i nie ma nic wspólnego z emulowaniem części funkcjonalności obecnym w starszych wersjach Direct3D. Drugim możliwym zastosowaniem jest testowanie sterowników karty graficznej - warunkiem poprawności jest identyczność obrazów wygenerowanych przez oba typy urządzeń. Wariant programowy wymaga ponadto zainstalowanego DirectX SDK (w odróżnieniu od sprzętowego, który wymaga tylko tzw. *runtime*).

Drugim sposobem na inicjalizację jest użycie komponentu `DXGI`. Dzięki interfejsowi `IDXGIFactory` możliwe jest bardziej szczegółowe i kompleksowe podejście do inicjowania okna z Direct3D, zwłaszcza w maszynie z wieloma układami graficznymi i/lub monitorami. Ponieważ jednak zagadnienia te wykraczają poza zakres tematyczny książki, poniżej znajduje się fragment kodu, który poza operacjami równoważnymi w stosunku do poprzedniego podejścia jawnie sprawdza dostępność obsługi API.

```
// zmienne globalne
ID3D10Device* g_device = NULL; IDXGISwapChain* g_swapChain = NULL;
...
// stworzenie fabryki
IDXGIFactory* factory = NULL;
CHECK_HR(CreateDXGIFactory(__uuidof(IDXGIFactory),
  reinterpret_cast<void**>(&factory)));
// wyliczenie adapterów
IDXGIAdapter* adapter = NULL; for (UINT
i=0;factory->EnumAdapters(i,&adapter)!=DXGI_ERROR_NOT_FOUND;++i) {
  // sprawdzenie czy adapter wspiera D3D10
  LARGE_INTEGER version;
  if (adapter->CheckInterfaceSupport(__uuidof(ID3D10Device), &version) ==
      S_OK) {
    // tak
    break;
  }
  adapter->Release();
}
// stworzenie urządzenia
CHECK_HR(D3D10CreateDevice( adapter, D3D10_DRIVER_TYPE_HARDWARE,
NULL,
  D3D10_CREATE_DEVICE_DEBUG, D3D10_SDK_VERSION, &g_device));
// stworzenie łańcucha
CHECK_HR(factory->CreateSwapChain(g_device, &swapChainDesc,
&g_swapChain));
// zwolnienie obiektów
adapter->Release(); factory->Release();
```

Ponieważ `DXGI` jest osobną częścią biblioteki DirectX, w projekcie należy dodać dodatkowe odwołanie do biblioteki zewnętrznej o nazwie dxgi.lib. Jeżeli się tego nie zrobi, w trakcie linkowania wystąpią błędy.

4.3.3 Konfiguracja etapu scalania

Kolejnym krokiem jest stworzenie buforów oraz skojarzenie ich z odpowiednimi etapami potoku renderowania. W Direct3D 10 zasoby (ang. *resources*) nie mogą być podłączone bezpośrednio do potoku renderowania. Zamiast tego zdefiniowane są interfejsy dziedziczące po `ID3D10View`, które zapewniają warstwę abstrakcji między danymi (zasoby) a potokiem. Taki mechanizm, co jest wielokrotnie podkreślane w dokumentacji, pozwala na walidację zasobów podczas ich tworzenia oraz otwiera drogę ku minimalizacji transferu do karty graficznej (kwestia ta będzie poruszona przy okazji omawiania obiektów stanów w dalszej części książki).

Widok celu renderingu

Zgodnie z tą filozofią należy wskazać urządzeniu widok, który będzie odpowiedzialny za dostęp do bufora docelowego. W tym przypadku jest to obiekt implementujący interfejs `ID3D10RenderTargetView`, natomiast sam bufor pobierany jest z łańcucha wymiany (wspomniany w 3.4.2 *back buffer*). Parametry dowiązania można konfigurować drugim argumentem (wskaźnik na strukturę `D3D10_RENDERTARGETVIEW_DESC`), jednak wykracza to poza tematykę książki.

```
ID3D10RenderTargetView* g_renderTargetView = NULL; ...
// pobieranie bufora z łańcucha wymiany
ID3D10Buffer* renderBuffer = NULL;
CHECK_HR(g_swapChain->GetBuffer(0, __uuidof( ID3D10Texture2D ),
reinterpret_cast<void**>(&renderBuffer)));
// stworzenie widoku bufora
CHECK_HR(g_device->CreateRenderTargetView(renderBuffer, NULL,
  &g_renderTargetView));
// zwolnienie bufora
renderBuffer->Release();
```

> Akcesory interfejsów COM zwiększają liczniki referencji zwracanych/ustawianych obiektów. Aby uniknąć wycieków pamięci, ważne jest, aby wywoływać metodę `Release` w momencie, gdy dany obiekt nie będzie już potrzebny.

Widok bufora głębokości

Odmiennie niż w poprzednim przypadku, bufor głębokości trzeba stworzyć od podstaw. W pierwszej kolejności należy pozyskać zasób docelowy, w tym przypadku jest to tekstura. Jej parametry określane są poprzez odpowiednie wypełnienie struktury `D3D10_TEXTURE2D_DESC`. Drugą czynnością jest stworzenie widoku `ID3D10DepthStencilView`.

```
ID3D10Texture2D* g_depth = NULL; ID3D10DepthStencilView* g_depthView
= NULL; ...
// tekstura stanowiąca bufor głębokości
```

```
D3D10_TEXTURE2D_DESC depthTextureDesc; ZeroMemory(&depthTextureDesc,
sizeof(depthTextureDesc));
// rozmiar równy rozmiarowi okna
depthTextureDesc.Width = width; depthTextureDesc.Height = height;
// bez mipmapingu
depthTextureDesc.MipLevels = 1;
// jednowymarowa tekstura
depthTextureDesc.ArraySize = 1;
// informacje o strukturze i sposobie użycia
depthTextureDesc.Format = DXGI_FORMAT_D32_FLOAT;
depthTextureDesc.Usage = D3D10_USAGE_DEFAULT;
depthTextureDesc.BindFlags = D3D10_BIND_DEPTH_STENCIL;
// bez multisamplingu
depthTextureDesc.SampleDesc.Count = 1;
depthTextureDesc.SampleDesc.Quality = 0;
CHECK_HR(g_device->CreateTexture2D(&depthTextureDesc, NULL,
&g_depth));

// widok bufora głębokości
D3D10_DEPTH_STENCIL_VIEW_DESC depthDesc; ZeroMemory(&depthDesc,
sizeof(depthDesc)); depthDesc.Format = depthTextureDesc.Format;
depthDesc.ViewDimension = D3D10_DSV_DIMENSION_TEXTURE2D;
CHECK_HR(g_device->CreateDepthStencilView(g_depth,
  &depthDesc, &g_depthView));
```

W trakcie tworzenia zasobów często występować będą obok siebie pola `Usage` (typ `D3D10_USAGE`) oraz `CPUAccessFlags` (typ `UINT`, ale wartość powinna być sumą logiczną wartości wyliczenia `D3D10_CPU_ACCESS_FLAG`). Kombinacja tych pól definiuje, w jaki sposób można dokonywać odczytu i zapisu danych zasobu. W ogólnym przypadku im bardziej zasób jest zamknięty na odczyt i napis, tym bardziej podatny jest na różne optymalizacje. Szczegóły znajdują się w dokumentacji, jednak przy dobieraniu wartości należy kierować się następującymi zasadami:

- GPU modyfikuje i/lub odczytuje zasób: `USAGE_DEFAULT`
- CPU modyfikuje zasób: `USAGE_DYNAMIC` i flaga `CPU_ACCESS_WRITE`
- CPU i GPU nie modyfikują zasobu: `USAGE_IMMUTABLE`
- CPU odczytuje zasób: `USAGE_STAGING` i flaga `CPU_ACCESS_READ`

W przypadku bufora głębokości odpowiednią wartością pola `Usage` jest wartość `DEFAULT`, gdyż układ graficzny musi mieć możliwość zarówno zapisu, jak i odczytu danych. Procesor główny nie powinien mieć, w celach optymalizacyjnych, dostępu do zawartości bufora, więc pole `CPUAccessFlags` jest wyzerowane. Należy zwrócić tu uwagę na pole `BindFlags` (suma logiczna stałych z wyliczenia `D3D10_BIND_FLAG`) – jego wartość precyzyjnie określa, gdzie dany zasób może być użyty w potoku renderowania.

Przypisanie widoków

Widoki celu renderowania oraz bufora głębokości należy przypisać do fazy scalania (ang. *output merger*). Należy zwrócić uwagę, iż o ile może być tylko jeden bufor głębokości, to celów renderowania można przekazać aż do maksimum ustalonego stałą `D3D10_SIMULTANEOUS_RENDER_TARGET_COUNT`. Przy odpowiedniej współpracy z etapami cieniowania możliwe jest przykładowo

wyrenderowanie tej samej sceny dla kilku różnych położeń kamery za pomocą jednego tylko wywołania funkcji rysującej.

```
// ustawianie widoku
g_device->OMSetRenderTargets(1, &g_renderTargetView, g_depthView);
```

Metody `ID3D10Device` odnoszące się do różnych faz potoku renderowania zawierają przedrostki będące akronimami ich pełnych nazw. Przykładowo, metody dotyczące etapu scalania (*output merger*) będą miały przedrostek `OM`.
Dodatkowo przypisywanie widoków do potoku renderowania nie zwiększa ich liczby referencji. Z tego powodu, aby poprawnie zwolnić zasoby, wskaźnik do widoku należy przechowywać poza lokalnym zasięgiem nazw (np. jako zmienną globalną lub pole typu).

4.3.4 Konfiguracja rasteryzera

Ostatnim etapem inicjacji jest ustalenie fragmentu okna, w który będzie odrysowywany z użyciem Direct3D - region ten nosi nazwę rzutni (ang. *viewport*). Ustawienia dokonuje się poprzez odpowiednie wypełnienie struktury `D3D10_VIEWPORT`. Prostokąt, który wyznacza, jest zorientowany względem lewego górnego rogu okna docelowego. Wartości pól `MinDepth` i `MinDepth` muszą zawierać się w zakresie $< 0.0; 1.0 >$. Podobnie jak w poprzednim punkcie, istnieje możliwość przekazania tablicy rzutni, a możliwości z tym związane są analogiczne.

```
// ustawienie viewportu
D3D10_VIEWPORT viewport; viewport.TopLeftX = 0; viewport.TopLeftY =
0; viewport.Width = width; viewport.Height = height;
viewport.MinDepth = 0.0f; viewport.MaxDepth = 1.0f;
g_device->RSSetViewports(1, &viewport);
```

Rzutnie tradycyjnie wykorzystywało się do gier działających w tzw. trybie *split screen*. Innym zastosowaniem może być rysowanie minimapy, czyli obrazu sceny widzianej pod pionowym kątem z dużej wysokości. Logika obu efektów jest z grubsza identyczna i polega na zmianie aktywnej rzutni oraz pozycji kamery.

4.4 Tworzenie geometrii

Programowanie z użyciem Direct3D, co można było zauważyć w poprzednim punkcie, skupia się wokół konfiguracji i dostarczania danych potokowi renderowania. Poniżej przedstawiony jest sposób przygotowania danych pojedynczego trójkąta równobocznego. Ilość kodu i wiedzy potrzebnej do tak na pozór trywialnego zadania może być zaskakująca, jednak model przyjęty w Direct3D

jest bardzo skalowalny i jednorodny, przez co po przebiciu się przez konfigurację rozwijanie kodu rysującego nie napotyka znacznych przeszkód.

Trójkąt wyznaczają trzy wierzchołki rozmieszczone w przestrzeni. Jak zostało wspomniane w rozdz. 3.2, standardem jest dodawanie do wierzchołków atrybutów tak, aby uzyskanie pewnych efektów graficznych z nimi związanych było łatwiejsze lub wręcz po prostu możliwe. Prostym przykładem może być wypełnianie wnętrza figury gradientem[3]. O ile po przeczytaniu książki czytelnik z pewnością wskaże kilka możliwych realizacji tego zadania, to kanonicznym sposobem jest dodanie atrybutu koloru do każdego z wierzchołków.

Poniższe dane opisują trójkąt

```
//! Deklaracja wierzchołka.
struct Vertex {
  D3DXVECTOR3 position;
  D3DXVECTOR3 color;
};

//! Dane trójkąta równobocznego o długości krawędzi 1
//! i ze środkiem w punkcie (0; 0; 0)
Vertex g_triangle[] = {
  { D3DXVECTOR3( 0.0f, 0.577f, 0), D3DXVECTOR3(1.0f, 0.0f, 0.0f) },
  { D3DXVECTOR3( 0.5f,-0.289f, 0), D3DXVECTOR3(0.0f, 1.0f, 0.0f) },
  { D3DXVECTOR3(-0.5f,-0.289f, 0), D3DXVECTOR3(0.0f, 0.0f, 1.0f) }
}
```

Gdy pozycja wierzchołków nie jest poddawana transformacjom, odnosi się do przestrzeni przycinania. W tym przypadku współrzędne lewego dolnego rogu okna to $(-1; -1; 0)$, natomiast prawego górnego to $(1; 1; 0)$. Komponent z oznacza znormalizowaną głębokość, więc może być dowolną wartością z przedziału $< 0; 1 >$. Nie wpływa ona na rozmiar obiektu na ekranie – jest to po prostu wartość poddawana testowi w buforze głębokości.

4.4.1 Tworzenie bufora wierzchołków

Każda geometria przed odrysowaniem musi być umieszczona w buforze wierzchołków. Jest to odmiana w stosunku do poprzednich wersji Direct3D, gdzie pewne prymitywy graficzne można było odrysować bez angażowania się w niskopoziomową logikę. Wraz z wersją opatrzoną numerem 10, w celach optymalizacyjnych, każda geometria przed odrysowaniem musi być umieszczona w buforze wierzchołków (mechanizm opisany w punkcie 3.4.1).

Tworzenie buforów sprowadza się do odpowiedniego wypełnienia struktury `D3D10_BUFFER_DESC` – niesie ona informacje o rozmiarze, typie i przeznaczeniu obiektu. W zależności od zastosowania wypełnienia może wymagać także opcjonalna struktura `D3D10_SUBRESOURCE_DATA`, która pozwala na skopiowanie pewnych danych z pamięci operacyjnej w momencie tworzenia bufora w celu

[3] Płynne przejście tonalne między dwoma lub więcej kolorami.

jego inicjalizacji. Mając wypełnione obie struktury oraz dane źródłowe, można przystąpić do tworzenia buforów.

```
ID3D10Buffer* g_vertexBuffer = NULL; ...
// definicja bufora
D3D10_BUFFER_DESC bufferDesc; memset(&bufferDesc, 0,
sizeof(bufferDesc));
// niezmienne, odczytywane przez GPU
bufferDesc.Usage = D3D10_USAGE_DEFAULT; bufferDesc.CPUAccessFlags =
0;
// jako bufor wierzchołków
bufferDesc.BindFlags = D3D10_BIND_VERTEX_BUFFER;
// rozmiar bufora
bufferDesc.ByteWidth = sizeof(g_triangle);
// definicja danych inicjalizacyjnych
D3D10_SUBRESOURCE_DATA initData; ZeroMemory(&initData,
sizeof(initData));
// dane trójkąta
initData.pSysMem = g_triangle;
// tworzenie bufora
CHECK_HR(d3ddevice->CreateBuffer(&bufferDesc, &initData,
&g_vertexBuffer));
```

4.4.2 Tworzenie układu wierzchołków

Ponieważ pole `InitData` struktury `D3D10_SUBRESOURCE_DATA` jest typu `const void*`, w momencie kopiowania danych wierzchołków tracona jest informacja odnośnie ich typu i atrybutów. Bez dodatkowego mechanizmu opisującego zawartość bufor stanowi wyłącznie tablicę bajtów o określonym rozmiarze, która potencjalnie może być zinterpretowana w dowolny sposób. Metadane pod postacią układu danych wejściowych (ang. *input layout*) definiują rozkład atrybutów wierzchołków w pamięci: ich typ, rozmiar oraz kolejność. Jest to więc funkcja analogiczna do tej, jaką pełnią definicje typów użytkownika w językach programowania wysokiego poziomu. Różnica występuje w samym sposobie deklaracji oraz precyzji i jednoznaczności obu technik.

W Direct3D każdy atrybut wierzchołków opisywany jest przez strukturę `D3D10_INPUT_ELEMENT_DESC`. W praktyce, ponieważ wierzchołek zazwyczaj posiada więcej niż jeden atrybut, konieczne jest stworzenie tablicy elementów. Typ `D3D10_INPUT_ELEMENT_DESC` posiada następujące pola:

- `SemanticName`: znaczenie (semantyka) danych; atrybut będzie identyfikowany w potoku renderowania pod tą nazwą
- `SemanticIndex`: indeks dodawany do nazwy jako przyrostek; wartość 0 równoważna jest z brakiem indeksu
- `Format`: stała z wyliczenia `DXGI_FORMAT`, określająca typ atrybutu
- `InputSlot`: indeks bufora wierzchołków, z którego pochodzić ma atrybut
- `AlignedByteOffset`: przesunięcie atrybutu w bajtach względem początku danych wierzchołka
- `InputSlotClass`: określa, czy atrybut dotyczy wierzchołka, czy grupy wierzchołków (instancji); mechanizm ten zostanie omówiony w dalszej części książki

- `InstanceDataStepRate`: konfiguruje mechanizm instancji; mechanizm ten zostanie omówiony w dalszej części książki

Deklaracja sporządzona przy użyciu tablicy omawianych struktur może, ale nie musi, być całkowicie zgodna z deklaracją typu wierzchołka w języku wysokiego poziomu. Przykładowo, można pominąć niektóre pola lub używać ich w potoku renderowania pod inną nazwą. Dodatkowo, jeżeli atrybut jest typu o rozmiarze lub formacie nieprzewidzianym przez wyliczenie `DXGI_FORMAT`, konieczne jest sporządzenie wielu elementów tablicy, przy czym każdy opisywać musi inną porcję danych. Przykład tutaj może stanowić macierz 4×4, którą trzeba w ten sposób rozłożyć na cztery wektory. W przypadku używanego w tym rozdziale typu `Vertex` tablica opisująca atrybuty wyglądać może następująco:

```
// Układ danych wejściowych.
D3D10_INPUT_ELEMENT_DESC inputLayoutDesc[] = {
  { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
      D3D10_INPUT_PER_VERTEX_DATA, 0 },
  { "COLOR", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
      D3D10_INPUT_PER_VERTEX_DATA, 0 }
};
```

Układ wejściowy reprezentowany jest interfejsem `ID3D10InputLayout`. Do jego stworzenia, poza przygotowanym opisem, konieczna jest jeszcze sygnatura etapu potoku renderowania, do którego trafiają dane z buforów - cieniowania wierzchołków. Etap ten oczekuje danych w zadanym, określonym przez wybrany program, formacie, przez co atrybuty muszą być mapowane na odpowiednie wejścia. Sygnatura dostarczana jest przez strukturę `D3D10_PASS_DESC`, która opisuje jeden przebieg efektu (interfejsy efektów omówione zostaną w dalszej części książki). Przykładowo, dysponując poprzednio stworzoną tablicą struktur `D3D10_INPUT_ELEMENT_DESC` oraz techniką jednoprzebiegową, stworzenie układu wierzchołków może wyglądać następująco:

```
ID3D10EffectTechnique* g_technique = NULL; ID3D10InputLayout*
g_inputLayout = NULL; ...
// liczba elementów opisu struktury bufora
UINT
numElements=sizeof(inputLayoutDesc)/sizeof(D3D10_INPUT_ELEMENT_DESC);
// pobranie informacji o efekcie
D3D10_PASS_DESC passDesc;
g_technique->GetPassByIndex(0)->GetDesc(&passDesc);
// tworzenie układu
CHECK_HR(g_device->CreateInputLayout(inputLayoutDesc, numElements,
  passDesc.pIAInputSignature, passDesc.IAInputSignatureSize,
  &g_inputLayout));
```

4.5 Tworzenie efektów

O tym, w jaki sposób zostanie wyświetlona geometria decyduje efekt (interfejs `ID3D10Effect`). Pierwszym krokiem ku jego stworzeniu jest przygotowanie kodu w języku HLSL. Przy projektowaniu efektów należy pamiętać o jednym mechanizmie działającym „za kulisami” – o automatycznej, liniowej interpolacji

atrybutów wierzchołków prymitywu przed etapem cieniowania pikseli. Dzięki temu, w przypadku wypełnienia gradientem, logika efektu sprowadza się głównie do przekazania programowi pikseli kolorów każdego z wierzchołków trójkąta.

4.5.1 Cieniowanie wierzchołków

Biorąc pod uwagę spodziewaną zawartość bufora wierzchołków, należy zadeklarować, jakich danych wejściowych będzie wymagała jednostka cieniowania wierzchołków. W przypadku trójkąta wypełnionego gradientem będą to pozycja i kolor wierzchołka. Kod jednostki cieniowania może więc wyglądać następująco:

```
void SimpleVertexShader(float3 position : POSITION, float3 color :
COLOR,
  out float4 outPosition : SV_POSITION, out float3 outColor : COLOR)
{
    outPosition = float4(position, 1);
    outColor = color;
}
```

Definicja parametru jednostki cieniowania różni się nieco od tego znanego z C i C++. Poza nazwą typu może znajdować się modyfikator dostępu określający sposób, w jaki można odczytywać lub zmieniać wartość. Dostępne modyfikatory to:

- `in`: parametr wejściowy, traktować go należy jako kopię przekazywaną; jest to domyślny modyfikator
- `out`: parametr wyjściowy, można na nim wykonywać wyłącznie operacje zapisu
- `inout`: parametr wejściowo-wyjściowy, odpowiednik referencji z C++
- `uniform`: stała, która jest taka sama dla wszystkich danych przetwarzanych w danym przebiegu; parametr tego typu nie pochodzi z danych geometrii, lecz jest definiowany na etapie kompilacji jednostki

Dodatkowo parametrom jednostki cieniowania towarzyszy semantyka opisująca znaczenie danych. W powyższym przykładzie dane wejściowe używają semantyk `POSITION` i `COLOR`, ponieważ takie atrybuty zostały zdefiniowane w układzie wierzchołków (punkt 4.4.2). Jakie w takim razie mają znaczenie oznaczenia wartości wyjściowych (oznaczonych modyfikatorem `out`)? Umożliwiają one poprawne mapowanie wyników etapu wierzchołków na wejścia następnego etapu potoku, czyli cieniowania geometrii lub rasteryzera, który przekazuje je dalej do cieniowania pikseli. Standard HLSL opisuje predefiniowane semantyki o specjalnym znaczeniu – ich nazwy opatrzone są przedrostkami `SV_` (od *System-value*). Jedną z nich jest `SV_Position`, która mówi potokowi, że dane przypisane atrybutowi reprezentują pozycję w przestrzeni przycinania, czyli wymaganą przez etap rasteryzacji do poprawnego działania. Ponieważ przygotowując dane, wierzchołki zostały już rozmieszczone w tej

przestrzeni, nie ma potrzeby przeliczania pozycji. Biorąc pod uwagę, że przekazany wektor musi być czterowymiarowy, konieczne jest dodanie dodatkowej współrzędnej. Zgodnie z wyprowadzeniami z punktu 2.6.3, aby nie nastąpiło perspektywiczne zniekształcenie obrazu, należy przekazać w niej jedynkę.

Alternatywnym sposobem opisu danych wejściowo-wyjściowych jest użycie struktur. Ich użycie pozwala osiągnąć bardziej zwięzłą i przejrzystą strukturę, zwłaszcza jeżeli wierzchołek posiada wiele atrybutów. Z tego powodu zazwyczaj stosuje się właśnie tę konwencję, a fakt ten znajduje również odzwierciedlenie w niniejszej książce.

```
//! Definicja wierzchołka wejściowego.
struct VertexInput {
  float3 position : POSITION;
  float3 color : COLOR;
};

//! Definicja wierzchołka wyjściowego.
struct VertexOutput {
  float4 position : SV_POSITION;
  float3 color : COLOR;
};

VertexOutput SimpleVertexShader(VertexInput input) {
  VertexOutput output;
  output.position = float4(input.position, 1);
  output.color = input.color;
  return output;
}
```

4.5.2 Cieniowanie geometrii

Do prostego odrysowania geometrii etap jej cieniowania zazwyczaj nie jest potrzebny, gdyż wszelkie konieczne operacje można wykonać z zyskiem dla wydajności na etapie wierzchołków. Cieniowanie geometrii staje się użyteczne, gdy potrzebna jest informacja o całym prymitywie. W bieżącym przykładzie brak programu byłby równoznaczny z następującą jednostką:

```
[maxvertexcount(3)] void SimpleGeometryShader(triangle VertexOutput
input[3], inout TriangleStream<VertexOutput> output ) {
  output.Append(input[0]);
  output.Append(input[1]);
  output.Append(input[2]);
  output.RestartStrip();
}
```

Pewnym niecodziennym konstruktem językowym jest obowiązkowy atrybut `maxvertexcount`, mówiący o tym, ile wierzchołków maksymalnie może wytworzyć jednostka. Dodatkowo dla danych wejściowych konieczne jest zdefiniowanie typu prymitywu i, co za tym idzie, liczności jego wierzchołków. Danymi wyjściowymi zawsze jest strumień z modyfikatorem inout. Co ciekawe, strumień może być dowolnego typu spośród `PointStream`, `LineStream` i `TriangleStream` niezależnie od rodzaju prymitywu wejściowego. Przykładowo, zmiana kodu na

```
[maxvertexcount(6)] void SimpleGeometryShader(triangle VertexOutput
input[3], inout LineStream<VertexOutput> output ) {
  output.Append(input[0]);
  output.Append(input[1]);
  output.RestartStrip();
  output.Append(input[1]);
  output.Append(input[2]);
  output.RestartStrip();
  output.Append(input[2]);
  output.Append(input[0]);
  output.RestartStrip();
}
```

spowoduje, że do rasteryzera przesłany zostanie obrys trójkąta. Przy zmianie typu prymitywu wyjściowego należy pamiętać o ewentualnej zmianie atrybutu `maxvertexcount` oraz uporządkowaniu sekwencji wywołań metod `Append` i `RestartStrip`, odpowiednio dodających wierzchołki do bufora i przesyłających je do rasteryzera.

4.5.3 Cieniowanie pikseli

Format danych wejściowych w przypadku cieniowania pikseli jest konsekwencją danych wyjściowych poprzedniego etapu potoku: muszą zgadzać się zarówno typy, jak i semantyki. Ponieważ, jak już zostało wspomniane, cała logika związana z wyliczeniem koloru wewnątrz trójkąta wykonywana jest na etapie interpolacji atrybutów wierzchołków, w omawianym przypadku zadaniem programu jest tylko przekazanie piksela do etapu scalania. Do tego celu służy semantyka `SV_Target`.

```
float4 SimplePixelShader(VertexOutput input) : SV_Target {
  // kopiowanie koloru, alpha = 1
  return float4(input.color, 1.0);
}
```

4.5.4 Definicja techniki

Stworzone programy cieniowania należy przypisać do odpowiednich etapów potoku, tworząc tym samym jego nowy stan (przebieg efektu). Przed tą asocjacją jednostki cieniowania należy skompilować (z punktu widzenia HLSL), podając przy tym wersję tzw. *shader model*, czyli zestawu funkcjonalności, który wymagany jest od układu graficznego. Gdy jakiś programowalny etap nie jest używany, należy przypisać mu wartość `NULL`.

Dysponując zdefiniowanym przebiegiem (bądź przebiegami), należy przypisać go do techniki, która będzie kontrolowała całość zmian stanu potoku. Poniższy kod definiuje technikę, przebieg oraz skompilowane programy.

```
technique10 Render {
  pass SimplePass
  {
    SetVertexShader( CompileShader(vs_4_0, SimpleVertexShader()) );
    SetGeometryShader( CompileShader(gs_4_0, SimpleGeometryShader()) );
    SetPixelShader( CompileShader(ps_4_0, SimplePixelShader()) );
  }
}
```

Jeżeli dana jednostka używana jest więcej razy niż w jednym przebiegu, to za każdym razem kompilowana jest na nowo. W przypadku wielu technik oraz skomplikowanych programów czas oczekiwania może okazać się na tyle długi, że w znaczny sposób utrudnia to testowanie wprowadzanych zmian. W takim przypadku najlepiej stworzyć osobno tak zwany obiekt jednostki `shader object`, a następnie przypisywać go danym przebiegom.

```
VertexShader vsSimple = CompileShader(vs_4_0, SimpleVertexShader());
GeometryShader gsSimple = CompileShader(gs_4_0,
SimpleGeometryShader()); PixelShader psSimple =
CompileShader(ps_4_0, SimplePixelShader());

technique10 Render {
  pass SimplePass
  {
    SetVertexShader( vsSimple );
    SetGeometryShader( gsSimple );
    SetPixelShader( psSimple );
  }
}
```

4.5.5 Kompilacja

W przypadku prostych programów wygodne jest definiowanie źródeł bezpośrednio w kodzie aplikacji (jako łańcuch ASCII), gdyż nie wymaga to żadnych operacji na plikach. Rozsądniejszym rozwiązaniem, zwłaszcza w przypadku większych projektów, jest przechowywanie efektów w plikach tekstowych z rozszerzeniem .fx. Dzięki temu można np. testować i kompilować efekty poza samą aplikacją, co usprawnia proces ich rozwijania.

Wczytywanie oraz kompilacja efektów przy użyciu samego Direct3D jest dość skomplikowana, ale programiście pomaga tutaj biblioteka D3DX. Funkcja `D3DX10CreateEffectFromMemory` pozwala na stworzenie efektu dla łańcucha ASCII, natomiast `D3DX10CreateEffectFromFile` tworzy go na podstawie zadanego pliku .fx. W przykładach przygotowanych na rzecz książki efekty wczytuje się wyłącznie z zewnętrznych plików. Poniższy kod demonstruje sposób wczytywania efektów oraz pobierania zadanej techniki.

```
#include <sstream> #include <tchar.h> ... ID3D10Effect* g_effect =
NULL; ID3D10EffectTechnique* g_technique = NULL; ...
// flagi kompilacji shaderów
DWORD shaderFlags = D3D10_SHADER_ENABLE_STRICTNESS; #if defined(
DEBUG ) || defined( _DEBUG ) shaderFlags |= D3D10_SHADER_DEBUG |
D3D10_SHADER_SKIP_OPTIMIZATION; #endif
// wczytanie efektu z pliku
ID3D10Blob * errors = NULL; hr = D3DX10CreateEffectFromFile(
"Effect.fx", NULL, NULL, "fx_4_0",
  shaderFlags, 0, g_device, NULL, NULL, &g_effect, &errors, NULL );

// jeżeli były błędy można je wypisać
if ( FAILED(hr) ) {
  // sformatowanie komunikatu o błędzie
  std::ostringstream output;
  output << "Blad␣odczytywania␣pliku␣efektu␣(id:" << hr << ")";
  if ( errors ) {
```

```
    output << "\n" << reinterpret_cast<char*>(errors->GetBufferPointer());
    errors->Release();
  }
  MessageBox( NULL, output.str().c_str(), "Error", MB_OK );
  return hr;
}
// pobranie techniki
g_technique = g_effect->GetTechniqueByName( "Render" );
// czy się udało?
if ( !g_technique->IsValid() ) {
  return E_FAIL;
}
```

Kompilacja dużych efektów może trwać dość długo, dlatego też w DirectX SDK znajduje się narzędzie `fxc` mogące skompilować efekt poza aplikacją. Sposób komplikacji plików .fx z poziomu Visual Studio opisany jest w załączniku C.

4.6 Odrysowywanie

Dysponując urządzeniem Direct3D oraz geometrią w buforze i efektem, który jest w stanie ją obsłużyć, można przystąpić do budowania funkcji renderującej. W jej ciele wykorzystane zostaną obiekty stworzone w ramach inicjalizacji oraz wczytania efektów.

```
HRESULT RenderFrame() {
  HRESULT hr = S_OK;
  ...
  return S_OK;
}
```

Pierwszym krokiem jest pozbycie się pozostałości z poprzedniego odrysowania. Aby wyczyścić bufor, do którego odbywać będzie się renderowanie, konieczne jest wywołanie metody `ClearRenderTargetView`, podając przy tym widok oraz kolor (obowiązkowo czterokomponentowy). Dodatkowo konieczne jest wyczyszczenie bufora głębokości metodą `ClearDepthStencilView` dla zadanego widoku i wartości głębokości (1 to wartość maksymalna).

```
// czyszczenie zawartości bufora renderingu szarym kolorem
D3DXVECTOR4 clearColor(0.33f, 0.33f, 0.33f, 1.0f);
g_device->ClearRenderTargetView(g_renderTargetView, clearColor);
// czyszczenie bufora głębokości
g_device->ClearDepthStencilView(g_depthView, D3D10_CLEAR_DEPTH,
1.0f, 0);
```

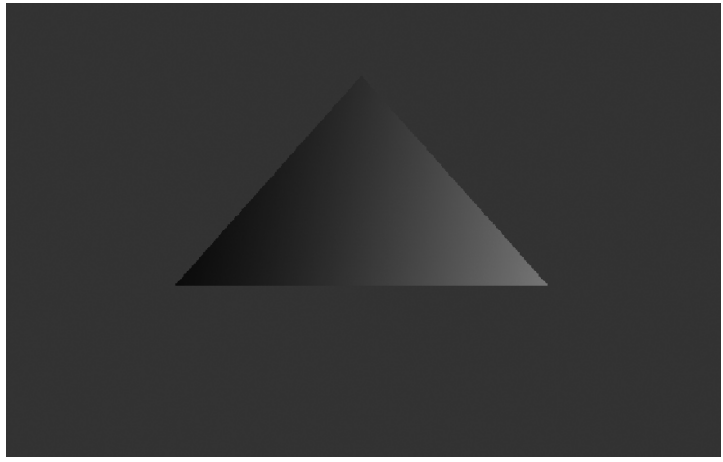
Kolejny krok to wybranie źródła geometrii, czyli konfiguracja asemblera danych wejściowych. Składają się na nią trzy czynności: wskazanie bufora (lub buforów) wierzchołków (`IASetVertexBuffers`), układu wierzchołków (`IASetInputLayout`) oraz typu prymitywu (`IASetPrimitiveTopology`). Wszystkie te czynności, z uwagi na fakt odrysowywania wyłącznie jednego trójkąta, mogłyby zostać wykonane jednorazowo na etapie inicjalizacji. Ponieważ w praktyce

sceny są bardziej skomplikowane i zawierają więcej źródeł geometrii, z których każde potencjalnie może mieć odmienny układ i topologię, bardziej właściwe jest umieszczanie logiki konfiguracji asemblera przed samym odrysowaniem konkretnego bufora.

```
// ustawienie bieżącego bufora wierzchołków
UINT stride = sizeof(Vertex); UINT offset = 0;
g_device->IASetVertexBuffers(0, 1, &g_vertexBuffer, &stride,
&offset);
// ustawienie układu wierzchołków
g_device->IASetInputLayout(g_inputLayout);
// ustawienie typu prymitywu
g_device->IASetPrimitiveTopology(D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
```

Po wybraniu danych można zacząć właściwe odrysowywanie, które w Direct3D 10 sprowadza się do nadawania potokowi stanów zdefiniowanych w technice. Liczbę przebiegów można ustalić za pomocą struktury `D3D10_TECHNIQUE_DESC`, która jest wypełniana przez wywołanie metody `ID3D10 EffectTechnique::GetDesc`. Zmiana stanu potoku następuje w momencie wywołania metody `ID3D10EffectPass::Apply`. Ponieważ w tym momencie urządzenie posiada wszystkie dane potrzebne do odrysowania, może nastąpić wywołanie metody rysującej, czyli `Draw`.

```
// liczba wierzchołków do odrysowania
UINT vertexCount = sizeof(g_triangle)/sizeof(Vertex);
// odrysowanie z użyciem wszystkich przebiegów techniki
D3D10_TECHNIQUE_DESC techniqueDesc;
CHECK_HR(g_technique->GetDesc(&techniqueDesc)); for(UINT i = 0; i <
techniqueDesc.Passes; ++i) {
  // nadanie stanu zdefiniowanego w przebiegu
  CHECK_HR(g_technique->GetPassByIndex(i)->Apply(0));
  // odrysowanie
  g_device->Draw(vertexCount, 0);
}
```



Rysunek 4.3. Trójkąt odrysowany w przestrzeni przycinania

Po zakończeniu odrysowania ostatniej geometrii może nastąpić zaprezentowanie bufora renderingu.

```
// zaprezentowanie bufora
CHECK_HR(g_swapChain->Present(0, 0));
```

Na rysunku 4.3 pokazano rezultat. Warto pamiętać, że ponieważ zdarzenie okna nie jest obsłużone, w przypadku zmiany rozmiaru trójkąt ulegnie rozciągnięciu bądź ściśnięciu, ponieważ z punktu widzenia Direct3D rozmiar się nie zmienił. Odrysowany obraz w pierwotnej rozdzielczości jest skalowany tak, aby zapełnić dostępny obszar.

4.7 Transformacje

Omawiany dotychczas trójkąt miał wierzchołki rozmieszczone w przestrzeni przycinania. Takie podejście, ze względu na możliwość pozycjonowania obiektów na znormalizowanej względem rozmiaru płaszczyźnie, może dobrze funkcjonować w zakresie grafiki 2D. W przypadku, kiedy reprezentowana ma być scena 3D, określanie pozycji w przestrzeni przycinania jest niepraktyczne i wymagające obliczeniowo. Zamiast tego wierzchołki definiowane są w lokalnej przestrzeni danego obiektu. Aby uprościć zarządzanie sceną, zakłada się często, że lokalne przestrzenie mają swój punkt odniesienia, czyli $(0; 0; 0)$, w miejscu środka ciężkości siatki. Innymi słowy, współrzędne wierzchołków są wektorami pociągniętymi od środka obiektu. Za transformację z lokalnego układu odniesienia do przestrzeni przycinania odpowiada macierz świat-widok-projekcja, czyli iloczyn macierzy świata, widoku i projekcji.

4.7.1 Tworzenie bufora indeksów

Sporządzony program wymaga oczywiście modyfikacji. Przede wszystkim należy zmodyfikować geometrię tak, aby reprezentowała bryłę zamiast figury geometrycznej. Najszybszą metodą jest zamienienie trójkąta w czworościan - teoretycznie wymaga to dodania tylko jednego wierzchołka. Tymczasem uwzględniając rozważania z rozdziału 3.4.1, okazuje się, że konieczne byłoby duplikowanie już istniejących wierzchołków w celu osobnego zbudowania wszystkich czterech ścian. Zamiast tego można posłużyć się buforem indeksów. Definicja geometrii i indeksów może wyglądać następująco:

```
//! Dane czworoboku foremnego o długości krawędzi 1 i
//! ze środkiem w punkcie (0;0;0)
Vertex g_tetrahedron[] = {
  { D3DXVECTOR3( 0.0f, 0.577f,-0.272f), D3DXVECTOR3(1.0f, 0.0f, 0.0f) },
  { D3DXVECTOR3( 0.5f,-0.289f,-0.272f), D3DXVECTOR3(0.0f, 1.0f, 0.0f) },
  { D3DXVECTOR3(-0.5f,-0.289f,-0.272f), D3DXVECTOR3(0.0f, 0.0f, 1.0f) },
  { D3DXVECTOR3( 0.0f,   0.0f, 0.544f), D3DXVECTOR3(1.0f, 1.0f, 1.0f) }
};

//! Indeksy ścian czworoboku.
```

```
USHORT g_tetrahedronIndices [] = {
  1, 2, 0,
  3, 1, 0,
  3, 2, 1,
  3, 0, 2,
};
```

Tworzenie bufora indeksów jest bardzo podobne do tworzenia bufora wierzchołków. Różnica polega na innej wartości pola `BindFlags` oraz typu użytego do liczenia rozmiaru.

```
ID3D10Buffer* g_indexBuffer = NULL; ...
// używa wcześniej stworzonych struktur
ZeroMemory(&bufferDesc, sizeof(bufferDesc));
// niezmienne, odczytywane przez GPU
bufferDesc.Usage = D3D10_USAGE_DEFAULT; bufferDesc.CPUAccessFlags =
0;
// jako bufor indeksów
bufferDesc.BindFlags = D3D10_BIND_INDEX_BUFFER;
// rozmiar bufora
bufferDesc.ByteWidth = sizeof( g_tetrahedronIndices );
// definicja danych inicjalizacyjnych
ZeroMemory(&initData, sizeof(initData)); initData.pSysMem =
g_tetrahedronIndices;
// tworzenie bufora
CHECK_HR(g_device->CreateBuffer(&bufferDesc, &initData,
&g_indexBuffer));
```

4.7.2 Zmienne efektu

Transformacja pozycji wierzchołków z lokalnego układu odniesienia do przestrzeni przycinania wymaga pomnożenia pozycji razy macierz świat-widok-projekcja. Istnieje możliwość programowego wykonania tego zadania – oprócz mnożenia wymagałoby to aktualizacji bufora co ramkę. Ponieważ takie rozwiązanie w przypadku bardziej złożonej geometrii wiązałoby się z ogromnym kosztem obliczeniowym oraz koniecznością ciągłego przesyłania dużej ilości danych do karty graficznej, w praktyce operacje tego typu wykonuje się w jednostce cieniowania wierzchołków.

Pierwszym krokiem w tym przypadku będzie utworzenie zmiennej w efekcie, która reprezentowałaby macierz. W Direct3D 10 zmienne grupowane są w obiektach określanych mianem buforów stałych (ang. *constant buffers*). Nazwa ta ma źródło w fakcie, iż z punktu widzenia jednostek cieniowania wartości zmiennych są stałe (tylko aplikacja może je modyfikować pomiędzy wywołaniami funkcji rysującej, natomiast w trakcie trwania przebiegu potoku wartości są niezmienne). Kryterium grupowania powinna być częstotliwość aktualizacji, ponieważ wtedy otwiera się pole do optymalnego wymiaru transferu danych do urządzenia graficznego. Wynika to z samej specyfiki buforów – bowiem dokonanie zmiany jednej tylko wartości powoduje przesłanie całej jego zawartości. Wobec tego, jeśli jedna zmienna ulega aktualizacji częściej niż pozostałe w ramach tego samego bufora, następuje wysłanie nadmiarowych informacji. Przy skomplikowaniu prezentowanego przykładu wystarczy jednak tylko jeden bufor.

Poniższy kod tworzy bufor o nazwie `ChangesPerObject` oraz przedstawia zmodyfikowaną jednostkę cieniowania wierzchołków:

```
//! Bufor zmieniajacy sie raz na obiekt.
cbuffer ChangesPerObject {
  //! Macierz swiat-widok-projekcja.
  matrix g_worldViewProjection;
} ... VertexOutput SimpleVertexShader(VertexInput input) {
  VertexOutput output;
  output.position = mul(float4(input.position, 1.0), g_worldViewProjection)
      ;
  output.color = input.color;
  return output;
}
```

Oczywiście zamiast iloczynu w buforze stałych mogłyby znajdować się osobne macierze dla świata, widoku oraz projekcji. W tym momencie w jednostce cieniowania następowały po sobie trzy mnożenia wektora razy macierz, osobne dla każdej z przestrzeni. Należy jednak wziąć pod uwagę fakt, że dla wszystkich wierzchołków z danego przebiegu macierze te będą identyczne. W związku z tym nie warto jest tracić cykli etapu cieniowania na dwa zbędne mnożenia, skoro stosunkowo małym nakładem czasu można po stronie aplikacji wyznaczyć łączną transformację. Dodatkowo, przy dokonywaniu przekształceń pozycji należy pamiętać o dodaniu do niego czwartej współrzędnej, koniecznie o wartości 1 (więcej w punkcie 2.5.6).

4.7.3 Zmienne efektu w aplikacji

Direct3D definiuje kilka interfejsów, które ułatwiają modyfikację zmiennych z buforów stałych – wszystkie one rozszerzają interfejs bazowy, którym jest `ID3D10EffectVariable`. Przykładowo, dla macierzy zdefiniowany jest typ `ID3D10EffectMatrixVariable`. Wskaźnik do obiektu implementującego ten interfejs można pobrać za metodą `ID3D10Effect::GetVariableByName` (istnieją również warianty identyfikujące na podstawie indeksów lub semantyk). Uzyskanie dostępu do poprzednio zdefiniowanej zmiennej można zrealizować następująco:

```
ID3D10EffectMatrixVariable* g_worldViewProjectionVariable = NULL;
... g_worldViewProjectionVariable =
g_effect->GetVariableByName("g_worldViewProjection")->AsMatrix();
// sprawdzenie poprawności
if ( !g_worldViewProjectionVariable->IsValid() ) {
    return E_FAIL;
}
```

Co ciekawe, metody pobierające zmienne efektów nigdy nie zwracają `NULL`, nawet w razie nieodnalezienia tej poszukiwanej. Kolejną ciekawostką jest fakt, że nie implementują interfejsu `IUnknown`, a więc nie mają metody `Release`. Nie można ich wobec tego zwolnić, a zwalnianie operatorem `delete` powoduje wyrzucenie wyjątku. Można więc uznać, że prawdziwe instancje przechowywane są przez `ID3D10Effect`. Odwołania do nich są właściwe tylko do momentu, kiedy obiekt efektu nie zostanie zwolniony. Podobne właściwości charakteryzują interfejs `ID3D10EffectTechnique`.

4.7.4 Obliczanie macierzy świata-widoku-projekcji

Macierz świata

Żeby zademonstrować, jaki jest efekt transformacji, najlepiej pokazać obiekt w ruchu. W programowaniu grafiki trójwymiarowej ruch obiektu jest tożsamy z faktem, że jego macierz świata zmienia się w czasie. Logikę aktualizacji sceny w przykładowej implementacji zamknięto w poniższej funkcji.

```
void Update( float time ) {
  ...
}
```

> Praktyką nie do przecenienia jest rozdzielenie logiki aktualizacji sceny od logiki jej rysowania. Poza większą niezależnością od API graficznego dużo łatwiejsza staje się implementacja takich zachowań jak pauza bądź replay. Dodatkowo kod przygotowany zgodnie z tą zasadą jest bardziej przejrzysty, a składowe mają bardzo jasno określone kompetencje, na czym zyskuje skalowalność i niezawodność projektu.

Za pomocą D3DX macierz świata można konstruować funkcjami z rodziny `D3DXMatrix*`. Wszystkie te funkcje jako pierwszy parametr przyjmują wskaźnik na macierz docelową, który ostatecznie jest zwracany jako rezultat, co daje możliwość tworzenie zagnieżdżonych wywołań. Uzupełniająco można używać operatorów dostarczanych przez typ `D3DXMATRIX`, jednak należy pamiętać, że operatory binarne (dwa operandy) zawsze tworzą kopię obiektu, co może w przypadku skomplikowanych działań prowadzić do wielu zbędnych operacji kopiowania. Poniższy kod najpierw obraca czworościan o 90°względem osi X, aby „położyć" go na podstawie, a następnie wprawia w ruch po orbicie o małym promieniu.

```
// zmienne globalne
D3DXMATRIX g_worldMatrix; ... static const float Deg2Rad =
float(D3DX_PI) / 180;
// kąt obrotu; co 4 sekundy pełny obrót
float angle = 90 * time * Deg2Rad;
// budowa macierzy składowych
D3DXMATRIX rotationY, translation, rotationX;
D3DXMatrixRotationY(&rotationY, angle);
D3DXMatrixRotationX(&rotationX, -90 * Deg2Rad);
D3DXMatrixTranslation(&translation, 0.5f, 0.0f, 0.0f);
// odpowiednik:
// g_worldMatrix = rotationX * translation * rotationY
// z wykorzystaniem jednoargumentowych operatorów
g_worldMatrix = rotationX; g_worldMatrix *= translation;
g_worldMatrix *= rotationY;
```

Macierz widoku

Macierz widoku jest bezpośrednio zależna od położenia i orientacji kamery. W związku z tym jeżeli kamera jest dynamiczna, wymagana jest również aktualizacja w postaci wyznaczenia nowej macierzy transformującej. Prezentowany

przykład nie pozwala na kontrolę położenia kamery, wobec czego obliczenia przeprowadzana są tylko zaraz po inicjalizacji urządzenia Direct3D. Samo tworzenie macierzy widoku jest, dzięki D3DX, bardzo uproszczone i sprowadza się do wywołania funkcji `D3DXMatrixLookAtLH` z odpowiednimi, intuicyjnymi parametrami.

```
D3DXMATRIX g_viewMatrix; ...
// pozycja kamery
D3DXVECTOR3 position(0.0f, 1.0f, -2.5f);
// punkt na który kamera patrzy
D3DXVECTOR3 target(0.0f, 0.0f, 0.0f);
// wektor reprezentujący kierunek "do góry"
D3DXVECTOR3 worldUp(0.0f, 1.0f, 0.0f);
D3DXMatrixLookAtLH(&g_viewMatrix, &position, &target, &worldUp);
```

Macierz projekcji

Perspektywiczną macierz projekcji konstruuje się za pomocą globalnej funkcji `D3DXMatrixPerspectiveFovLH`. Argumentami są: kąt widzenia (dla kierunku y), współczynnik proporcji oraz odległości płaszczyzn przycinania od kamery. Zmiany w macierzy projekcji zachodzą bardzo rzadko i zwykle podyktowane są chęcią osiągnięcia specyficznego efektu graficznego. Z tego powodu w przedstawianym przykładzie obliczenia zachodzą w funkcji inicjującej.

```
D3DXMATRIX g_projectionMatrix; ...
// budowa macierzy projekcji o polu widzenia 45 stopni (w radianach pi/4)
// i płaszczyznach przycinania oddalonych od 1 i o 1000 od kamery
D3DXMatrixPerspectiveFovLH(&g_projectionMatrix,
  static_cast<float>(D3DX_PI/4),
  static_cast<float>(width)/height, 0.1f, 1000.0f);
```

> Dociekliwy czytelnik na pewno zastanowi się nad sensem przyrostka `LH` dla funkcji wyliczającej macierz widoku oraz projekcji. W rzeczywistości jest to skrót od *left-handed*, co sugeruje, że owe wersje przeznaczone są dla układów lewoskrętnych. Na lekcjach matematyki zazwyczaj przedstawiany jest model prawoskrętny, co może prowadzić do pewnych nieporozumień. Wybór systemu lewoskrętnego podyktowany jest faktem, że jest to domyślna orientacja dla Direct3D. Gdyby jednak z jakiś powodów musiałby być zastosowany model prawoskrętny, wówczas do dyspozycji są odpowiedniki wyżej wymienionych funkcji z sufiksem `RH`.

Macierz świat-widok-projekcja

Przy wyliczaniu iloczynu należy pamiętać, że z powodu uwzględnienia macierzy świata jest specyficzny dla każdego obiektu na scenie. W przedstawionym przykładzie macierz świat-widok-projekcja wyliczana jest przed odrysowaniem geometrii zgodnie z poniższym kodem.

```
// wyliczenie i ustawienie macierzy świat-widok-projekcja
D3DXMATRIX worldViewProjection = g_worldMatrix * g_viewMatrix *
g_projectionMatrix;
g_worldViewProjectionVariable->SetMatrix(worldViewProjection);
```

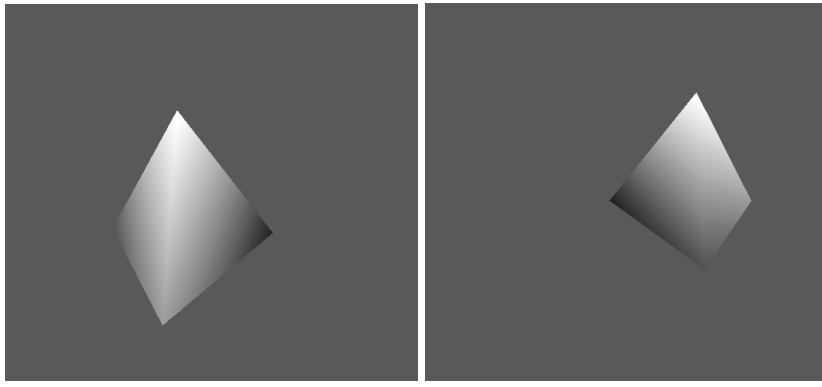
4.7.5 Modyfikacja funkcji rysującej

Ponieważ w omawianym przykładzie wprowadzono bufor indeksów konieczna jest modyfikacja funkcji rysującej. Pierwszą czynnością, którą należy wykonać jest przypisanie bufora do odpowiedniego etapu potoku renderowania. W przypadku indeksów nie ma na szczęście potrzeby definiowania oraz tworzenia układu danych – rolę tę spełnia stała pochodząca z wyliczenia `DXGI_FORMAT` (tylko `R16_UINT` lub `R32_UINT`) podawana podczas samego ustawiania.

```
// ustawienie bufora indeksów
g_device->IASetIndexBuffer(g_indexBuffer, DXGI_FORMAT_R16_UINT, 0);
```

Zmianie musi ulec również sama pętla obsługi techniki. Liczbę wierzchołków do odrysowania ustalić trzeba nie na podstawie długości bufora wierzchołków, lecz na podstawie indeksów. Dodatkowo wywołania metody `Draw` należy zastąpić wywołaniami `DrawIndexed`. Rezultat prezentowany jest poniżej na rysunku 4.4.

```
// liczba wierzchołków do odrysowania
UINT indexCount = sizeof(g_tetrahedronIndices)/sizeof(USHORT);
// odrysowanie z użyciem wszystkich przebiegów techniki
D3D10_TECHNIQUE_DESC techniqueDesc;
CHECK_HR(g_technique->GetDesc(&techniqueDesc)); for(UINT i = 0; i <
techniqueDesc.Passes; ++i) {
  // nadanie stanu zdefiniowanego w przebiegu
  CHECK_HR(g_technique->GetPassByIndex(i)->Apply(0));
  // odrysowanie
  g_device->DrawIndexed(indexCount, 0, 0);
}
```



Rysunek 4.4. Dwie klatki animacji czworościanu krążącego po wąskiej orbicie

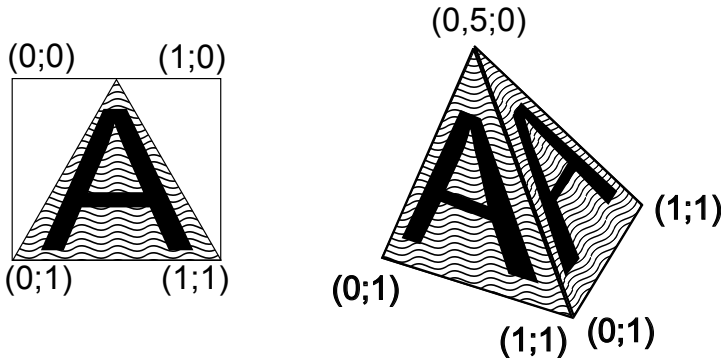
4.8 Teksturowanie

W rzeczywistym świecie rzadko który obiekt pokryty całkowicie jest jednolitym bądź równomiernie gradientowym kolorem – na powierzchni obiektów widać zagłębienia, przebarwienia albo po prostu zmianę koloru. Aby oddać ten stan przy użyciu dotychczasowej techniki, trzeba by bardzo zagęścić siatkę tak, aby prymitywy były na tyle małe, żeby interpolacja kolorów nie była widoczna. Innym rozwiązaniem byłoby zastosowanie jakiegoś proceduralnego mechanizmu w jednostce pikseli bazującego np. na pozycji w przestrzeni przycinania. W praktyce stosuje się mapowanie tekstur (ang. *texture mapping*), czyli nakładanie obrazu rastrowego na trójwymiarową geometrię.

4.8.1 Współrzędne tekstur

Czynnikiem decydującym o sposobie nakładania obrazu na geometrię są współrzędne tekstury (ang. *texture coordinates*). Poszczególne teksele[4] nie są adresowane jak komórki pamięci, ale za pomocą znormalizowanych wartości zmiennoprzecinkowych, czyli liczb z przedziału $< 0.0; 1.0 >$. Dla tekstur dwuwymiarowych współrzędne te w Direct3D nazywane są u (oś pozioma) i v (oś pionowa). Punkt $(0; 0)$ znajduje się w lewym górnym rogu obrazu, przez co oś v skierowana jest w dół. Dzięki takiemu podejściu współrzędne, np. $(0,5; 0,5)$ zawsze odpowiadają środkowi tekstury, niezależnie od jej rozmiaru.

Dodanie współrzędnych tekstur do wierzchołków wymaga modyfikacji definicji typu $Vertex$ oraz opisu struktury bufora. Niestety, wraz z tą zmianą musi zwiększyć się również liczba wierzchołków. Wynika to z faktu, że ten sam ze względu na pozycję wierzchołek może należeć do kilku ścian, dla których nie ma ciągłości w mapowaniu tekstur (sytuację tę obrazuje rysunek 4.5).



Rysunek 4.5. Problem teksturowania siatek przy nieciągłym mapowaniu. Tekstura (po lewej stronie) mapowana tak samo na dwie ściany skutkuje koniecznością powielania wierzchołków

[4] Najmniejszy dyskretny element tekstury, potocznie nazywany również pikselem

```
//! Definicja wierzchołka
struct Vertex {
  //! Pozycja.
  D3DXVECTOR3 position;
  //! Kolor (rgb).
  D3DXVECTOR3 color;
  //! Współrzędne tekstury.
  D3DXVECTOR2 texcoord;
};

//! Dane czworoboku foremnego o długości krawędzi 1 i
//! ze środkiem w punkcie (0;0;0
Vertex g_tetrahedron[] = {
  // podstawa mapowana osobno
  { D3DXVECTOR3( 0.0f, 0.577f,-0.272f), D3DXVECTOR3(1.0f, 0.0f, 0.0f),
      D3DXVECTOR2(0.0f,0.0f) },
  { D3DXVECTOR3( 0.5f,-0.289f,-0.272f), D3DXVECTOR3(0.0f, 1.0f, 0.0f),
      D3DXVECTOR2(0.0f,0.0f) },
  { D3DXVECTOR3(-0.5f,-0.289f,-0.272f), D3DXVECTOR3(0.0f, 0.0f, 1.0f),
      D3DXVECTOR2(0.0f,0.0f) },
  // wierzchołek naprzeciwko podstawy
  { D3DXVECTOR3( 0.0f,   0.0f, 0.544f), D3DXVECTOR3(1.0f, 1.0f, 1.0f),
      D3DXVECTOR2(0.5f,0.0f) },
  // pozostałe
  { D3DXVECTOR3( 0.5f,-0.289f,-0.272f), D3DXVECTOR3(0.0f, 0.0f, 1.0f),
      D3DXVECTOR2(0.0f, 1.0f) },
  { D3DXVECTOR3( 0.0f, 0.577f,-0.272f), D3DXVECTOR3(1.0f, 0.0f, 0.0f),
      D3DXVECTOR2(1.0f, 1.0f) },
  { D3DXVECTOR3(-0.5f,-0.289f,-0.272f), D3DXVECTOR3(0.0f, 1.0f, 0.0f),
      D3DXVECTOR2(0.0f, 1.0f) },
  { D3DXVECTOR3( 0.5f,-0.289f,-0.272f), D3DXVECTOR3(0.0f, 0.0f, 1.0f),
      D3DXVECTOR2(1.0f, 1.0f) },
  { D3DXVECTOR3( 0.0f, 0.577f,-0.272f), D3DXVECTOR3(1.0f, 0.0f, 0.0f),
      D3DXVECTOR2(0.0f, 1.0f) },
  { D3DXVECTOR3(-0.5f,-0.289f,-0.272f), D3DXVECTOR3(0.0f, 1.0f, 0.0f),
      D3DXVECTOR2(1.0f, 1.0f) },
};

//! Indeksy ścian czworoboku.
USHORT g_tetrahedronIndices [] = {
  0,1,2,
  3,4,5,
  3,6,7,
  3,8,9
}; ...

//! Układ danych wejściowych.
D3D10_INPUT_ELEMENT_DESC inputLayoutDesc[] = {
  { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
      D3D10_INPUT_PER_VERTEX_DATA, 0 },
  { "COLOR", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
      D3D10_INPUT_PER_VERTEX_DATA, 0 },
  { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 24,
      D3D10_INPUT_PER_VERTEX_DATA, 0 }
};
```

Jak widać, wprowadzanie nowych atrybutów przy stosowaniu wyłącznie unikatowych wierzchołków sprawia, że dane stają się trudne do intuicyjnej interpretacji. Z tego też powodu jest to ostatnia w tej książce geometria podana *explicite*.

4.8.2 Modyfikacja efektu

Na skutek wprowadzenia dodatkowych atrybutów zmodyfikować należy definicje wierzchołków w pliku efektu. Nie warto próbkować koloru tekstury w jednostce wierzchołków – wynikowy kolor byłby interpolowany, co nie różniłoby się znacznie od wcześniej zaimplementowanego kolorowania gradientowego. Zamiast tego należy przekazać dalej współrzędne tak, aby w wyniku ich uśredniania jednostka cieniowania pikseli mogła próbkować teksturę w ramach całego prymitywu.

```
//! Definicja wierzchołka wejściowego.
struct VertexInput {
  float3 position : POSITION;
  float3 color: COLOR;
  float2 texcoords : TEXCOORD;
};
//! Definicja wierzchołka wyjściowego.
struct VertexOutput {
  float4 position : SV_POSITION;
  float3 color : COLOR;
  float2 texcoords : TEXCOORD;
};
```

Oprócz powyższego należy zadeklarować obecność tekstury oraz zdefiniować stan próbkowania (ang. *sampler state*), czyli sampler. Dokładne informacje na temat konfiguracji próbkowania znajdują się w dokumentacji. Na potrzeby przykładu wystarczające jest filtrowanie liniowe reprezentowane polem `Filter` o wartości `MIN_MAG_MIP_LINEAR`.

```
//! Tekstura.
Texture2D g_texture;
//! Sampler.
SamplerState linearSampler {
    Filter = MIN_MAG_MIP_LINEAR;
};
```

> Nienumeryczne typy jak np. tekstury i stany próbkowania nie mogą być umieszczane w buforach stałych.

Ostatnim krokiem jest uwzględnienie tekstury przy liczeniu koloru piksela. Przy łączeniu kolorów (w tym wypadku pochodzącego z wierzchołka i odczytanego z tekstury) wykorzystuje się zazwyczaj iloczyn elementów zdefiniowany w punkcie 2.3.7.

```
VertexOutput SimpleVertexShader(VertexInput input) {
  VertexOutput output;
  output.position = mul(float4(input.position, 1.0), g_worldViewProjection)
      ;
  output.color = input.color;
  output.texcoords = input.texcoords;
  return output;
} ... float4 SimplePixelShader(VertexOutput input) : SV_Target {
```

```
  // kopiowanie koloru, alpha = 1
  float4 Cv = float4(input.color, 1.0);
  float4 Ct = g_texture.Sample(linearSampler, input.texcoords);
  return Cv * Ct;
}
```

4.8.3 Wczytanie i ustawienie tekstury

Poniższy kod prezentuje sposób uzyskiwania dostępu do zmiennej reprezentującej teksturę w efekcie. Jak można zaobserwować, jest on niemal bliźniaczo podobny do już przedstawionego pobierania zmiennej-macierzy. Ponieważ ta czynność zawsze będzie wyglądała bardzo podobnie, więcej nie będzie już przytaczana w postaci kodu źródłowego.

```
ID3D10EffectShaderResourceVariable* g_textureVariable = NULL; ...
g_textureVariable =
g_effect->GetVariableByName("g_texture")->AsShaderResource();
// sprawdzenie poprawności
if ( !g_textureVariable->IsValid() ) {
    return E_FAIL;
}
```

Wczytanie tekstury, dzięki D3DX, może sprowadzać się do wywołania tylko jednej funkcji – `D3DX10CreateShaderResourceViewFromFile` lub ewentualnie jej odpowiedników jako źródło bądź plik zasobów. Jak sama nazwa wskazuje, tworzony jest od razu widok zasobu, gotowy do użycia przez potok renderowania. Dokładny opis zarówno parametrów funkcji, jak i jej bardziej niskopoziomowych odpowiedników znajduje się w dokumentacji.

```
ID3D10ShaderResourceView* g_textureView = NULL; ...
// wczytanie tekstury
CHECK_HR(D3DX10CreateShaderResourceViewFromFile(g_device,
"SimpleTexture.jpg", NULL, NULL, &g_textureView, NULL));
```

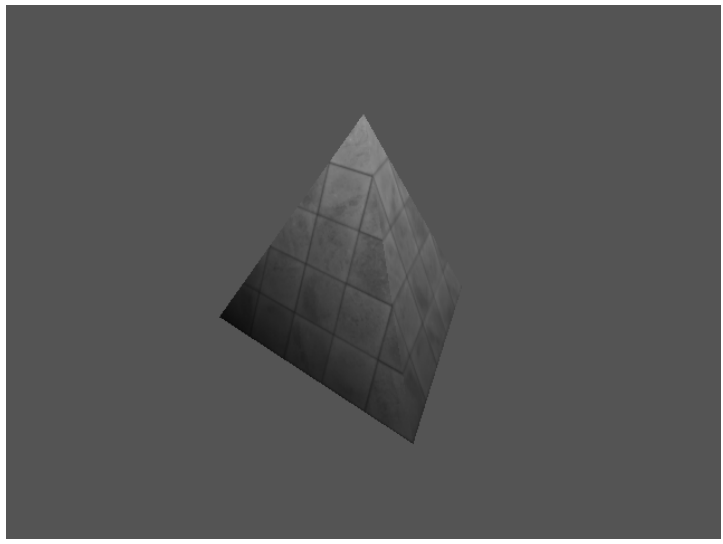
Ostatni krok to przypisanie zmiennej widoku tekstury. W aplikacji z jednym obiektem z przypisaną jedną teksturą można to zrobić zaraz, po stworzeniu widoku i pobraniu tekstury. Takie uproszczenie mści się jednak dłuższym czasem koniecznym do modyfikacji kodu, gdy doda się do sceny kolejny obiekt, tym razem już z inną teksturą. Dlatego też, aby utrwalać dobre wzorce, widok w przykładzie ustawiany jest każdorazowo przed odrysowaniem.

```
g_textureVariable->SetResource(g_textureView);
```

Reszta instrukcji pętli renderującej nie musi ulegać zmianom. Rezultat teksturowania przykładowym obrazem przedstawia rysunek 4.6.

4.9 Zarządzanie stanami

W poprzednich przykładach konfiguracja etapów rasteryzacji oraz scalania ograniczała się odpowiednio do wskazania rozmiaru ramki oraz buforów docelowych. Direct3D umożliwia kontrolę pewnych aspektów przetwarzania danych



Rysunek 4.6. Oteksturowany czworościan z uwzględnieniem koloru wierzchołków

w tej nieprogramowalnej części potoku. W odróżnieniu od poprzednich wersji API brak jest jednak funkcji, która przełączałaby pojedyncze składowe stanu etapu.

Poszczególne stany atomowe są zgrupowane w tzw. obiekty stanów (ang. *state objects*), które dostarczają wszelkich informacji o pewnym aspekcie danego etapu. Łączenie tych obiektów z potokiem odbywa się, podobnie jak w przypadku buforów i zasobów, za pośrednictwem widoków. Jest to znaczna zmiana w stosunku do poprzednich wersji API, a jej wprowadzenie zostało podyktowane względami wydajnościowymi. Przykładowo, dzięki kompletności i niezależności każdy obiekt stanu może być przechowywany w pamięci urządzenia graficznego. Zmiana stanu etapu w takiej sytuacji wymaga tylko przełączenia wskaźnika na bieżąco wybrany. Jeżeli więc zastosować się do dokumentacji, która zaleca tworzenie wszystkich potrzebnych obiektów stanowych przed pierwszym odrysowaniem, narzut związany z przełączaniem trybów rysowania może być znikomy.

Wszystkie obiekty stanów mogą być albo tworzone na poziomie aplikacji, albo definiowane w pliku efektu. Różnica pomiędzy oboma sposobami polega wyłącznie na nieco innej konwencji nazewnictwa. Źródłem nieporozumień może być fakt, że w kodzie aplikacji konieczne jest ustawienie wartości wszystkich pól struktur opisowych, natomiast w przypadku definicji w efekcie nie jest to wymagane. Nie oznacza to jednak, że w drugim przypadku istnieje możliwość stworzenia stanu częściowego, tj. zmieniającego tylko niektóre atomowe ustawienia, obowiązuje bowiem zasada, że jeżeli pole nie jest w jawny sposób podane, wówczas jest mu nadawana domyślna, specyficzna dla niego wartość.

4.9.1 Stan rasteryzera

Głównym zadaniem rasteryzera, poza dyskretyzacją sceny, jest obliczanie wartości głębokości oraz eliminacja niektórych prymitywów z dalszego przetwarzania. Niektóre z tych aspektów jego działania dają się kontrolować za pomocą obiektu stanowego, który można stworzyć w następujący sposób:

```
ID3D10RasterizerState* rastWireframeCullBack;
// definicja stanu
D3D10_RASTERIZER_DESC rastDesc; rastDesc.FillMode =
D3D10_FILL_WIREFRAME; rastDesc.CullMode = D3D10_CULL_BACK;
rastDesc.DepthClipEnable = true; rastDesc.FrontCounterClockwise =
false; rastDesc.DepthBias = false; rastDesc.DepthBiasClamp = 0;
rastDesc.SlopeScaledDepthBias = 0; rastDesc.ScissorEnable = false;
rastDesc.MultisampleEnable = false; rastDesc.AntialiasedLineEnable =
false;
// stworzenie widoku stanu
g_device->CreateRasterizerState( &rastDesc, &rastWireframeCullBack
);
```

Analogiczna definicja efektu:

```
RasterizerState rastWireframeCullBack {
    FillMode = WIREFRAME;
    CullMode = BACK;
    DepthClipEnable = true;
};
```

Na etapie rasteryzacji, gdy obligatoryjnie znana jest pozycja w przestrzeni przycinania, potok renderowania przed faktycznym rzutowaniem i dyskretyzacją prymitywów dokonuje eliminacji niewidocznych powierzchni na podstawie algorytmów przedstawionych w punkcie 3.4.3. Pole `CullMode` pozwala na kontrolę nad mechanizmem usuwania tylnych ścian. Dla domyślnej wartości `BACK` algorytm usuwa tylne, natomiast w przypadku wybrania `FRONT` przednie względem obserwatora ściany. Przypisanie `CullMode` wartości `NONE` całkowicie wyłącza mechanizm usuwania niewidocznych powierzchni. Warto też pamiętać o możliwości zmiany działania tej techniki, zwłaszcza gdy bazuje się na siatkach pobranych z sieci. Często występujące błędy w postaci nieodpowiedniej kolejności wierzchołków powodują, że dla niektórych trójkątów widać ich „przód", zaś dla pozostałych „tył". W takich sytuacjach przydaje się bardziej intuicyjny sposób określania widoczności – dla domyślnych ustawień trójkąt będzie odrysowany, jeżeli na płaszczyźnie ekranu jego wierzchołki pod względem kolejności ułożone będą zgodnie z kierunkiem ruchu wskazówek zegara.

Na odrysowywanie trójkątów razem z ich wnętrzem (`SOLID` wyzwala pole `FillMode`, wartość domyślna) lub, uwzględniając wyłącznie krawędzie (`WIREFRAME`), dzięki czemu można rysować szkielety siatek (ang. *wireframe*). Myląca może być właściwość `DepthClipEnable` – nie dotyczy ona mechanizmów testów głębokości w ujęciu pkt 3.4.4, lecz możliwości odrzucania prymitywów, których wierzchołki mają głębokość mniejszą lub większą od umiejscowienia, odpowiednio, bliskiej i dalekiej płaszczyzny przycinania. Znaczenie pozostałych pól i ich wartości domyślne opisane są dokładnie w dokumentacji.

4.9.2 Stan mieszania

Prowadząc rozważania nad kolorami pikseli, zazwyczaj używa się w ich kontekście wektora czterowymiarowego. Pierwsze trzy składniki to znormalizowany kolor w formacie RGB, czwarty zaś to tak zwany kanał alfa. Współczynnik ten zazwyczaj określa stopień, w jakim piksel wynikowy powinien być zmieszany z wartością już obecną w buforze ramki. To, jak dokładnie wygląda ten proces określa stan mieszania (ang. *blend state*).

> W przypadku, gdy kanał alfa nie jest potrzebny (np. materiał ma być nieprzenikliwy), można w jego miejscu przekazywać inne dane, takie jak maski rysowania bądź inna skalarna właściwość materiału.

W Direct3D kolor wynikowy wyrażany jest wzorem:

$$C_{n+1} = f_b(C_{src} \otimes F_{src}, C_n \otimes F_n)$$

gdzie:

C_{n+1} - nowy kolor piksela w buforze ramki
C_{src} - kolor bieżąco rasteryzowanego piksela
C_n - obecny kolor piksela w buforze ramki
F_{src} - współczynnik mieszania dla bieżąco rasteryzowanego piksela
F_n - współczynnik mieszania dla piksela w buforze ramki
f_b - funkcja mieszająca

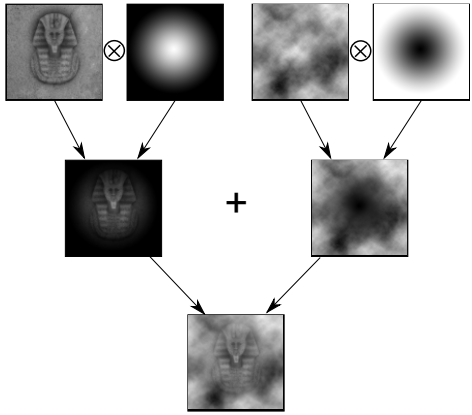
Dobór ustawień mieszania jest niezależny dla kanałów RGB i kanału alfa. W drugim przypadku zawężają się nieco możliwości konfiguracyjne, co dokładnie opisuje dokumentacja. W dalszej części rozdziału omówiony będzie przypadek ustawiania parametrów mieszania dla składowych RGB.

Współczynniki mieszania F_{src} oraz F_n mogą, ale nie muszą, być związane z wartością odczytaną z kanału alfa. Dostępne typy mieszania określa wyliczenie `D3D10_BLEND`, a dokładny ich opis znajduje się w dokumentacji. Warto wyróżnić najczęściej stosowane współczynniki:

- wartość pobrana kanału alfa
- dopełnienie wartości pobranej kanału alfa, czyli $1 - a$
- jeden lub zero
- kolor piksela

Zdefiniowania wymaga jeszcze funkcja mieszająca. Zgodnie z wyliczeniem `D3D10_BLEND_OP` może to być suma, obie różnice lub kolor, którego składowe są większymi bądź mniejszymi spośród analogicznych elementów operandów. Zazwyczaj stosuje się dodawanie, gdyż prowadzi ono do efektu intuicyjnie rozumianego częściowego przesłaniania się obiektów.

W Direct3D 10 za całkowity stan operacji mieszania odpowiada `BlendState`. Szczegóły odnośnie jego właściwości, domyślnych wartości i sposobie



Rysunek 4.7. Przykład mieszania. Współczynnikiem dla źródłowych pikseli (po lewej) jest ich kanał alfa, natomiast dla zawartości ramki – odwrotność tego samego kanału

ustawienia znajdują się w dokumentacji. Poniżej znajduje się fragment kodu obrazujący w wielkim skrócie proces tworzenia stanu na poziomie aplikacji:

```
D3D10_BLEND_DESC blendDesc; blendDesc.BlendOp = D3D10_BLEND_OP_ADD;
... ID3D10BlendState* blendState;
device->CreateBlendState(&blendDesc, &blendState); ... float
blendFactor[4] = { 0, 0, 0, 0 }; device->OMSetBlendState(blendState,
blendFactor, 0xFFFFFFFF);
```

Dla efektu:

```
BlendState SrcAlphaBlendingAdd {
  BlendOpAlpha = ADD;
  ...
}; technique XXX {
    pass XXX {
        SetBlendState(SrcAlphaBlendingAdd, float4(0,0,0,0), 0xFFFFFFFF);
        ...
    }
}
```

Warto pamiętać, że mieszanie zazwyczaj nie jest przemienne. Dotyczy to przypadków, gdy oba współczynniki są wyznaczane na podstawie tylko jednego z operandów, tak jak ma to miejsce na rysunku 4.7. Z tego powodu w celu uzyskania określonego efektu często konieczne jest uporządkowanie kolejności odrysowania według głębokości. Rozwiązaniem może być również taki dobór parametrów, aby przemienność występowała – przykładem może tu być mieszanie addytywne, czyli dla $F_{src} = a_{src}$, $F_n = 1$. Ponieważ wiąże się to zazwyczaj z pewną, czasami zamierzoną, zmianą zachodzącą w wynikowym obrazie, wybór docelowej metody powinien być podyktowany zarówno względami wydajnościowymi, jak i artystycznymi.

4.9.3 Stan głębokości-szablonu

Etap scalania, poza aktualizacją buforów ramek, ma za zadanie przeprowadzanie testów oraz aktualizacji buforów głębokości i szablonów. Pewną kontrolę nad tymi aspektami potoku umożliwia obiekt stanu `DepthStencilState`, definiowany następująco:

```
D3D10_DEPTH_STENCIL_DESC depthStencilDesc;
depthStencilDesc.DepthEnable = true; depthStencilDesc.DepthWriteMask
= D3D10_DEPTH_WRITE_MASK_ALL; depthStencilDesc.DepthFunc =
D3D10_COMPARISON_LESS; depthStencilDesc.StencilEnable = false; ...
ID3D10DepthStencilState* depthEnabledState;
device->CreateDepthStencilState( &depthStencilDesc,
&depthEnabledState ); ... device->OMSetDepthStencilState(
depthEnabledState, 0 );
```

W efekcie:

```
DepthStencilState depthEnabledState {
    DepthEnable = true;
    DepthWriteMask = ALL;
    DepthFunc = LESS;
    StencilEnable = false;
};
```

Stan głębokości

Mechanizm testów głębokości omówiony został w rozdziale 3.4.4, warto jednak omówić dostępne opcje konfiguracji. Przede wszystkim pole `DepthEnable` wyłącza wszystkie testy. Bardziej atomową kontrolę umożliwia `DepthFunc`, czyli opcja określająca, dla jakiego warunku test głębokości należy uznać za zaliczony. Domyślna wartość, czyli LESS, każe zapisywać tylko te piksele, których głębokość jest mniejsza niż wartość w buforze. Wyliczenie `D3D10_COMPARISON_FUNC` zawiera wszystkie możliwe warunki, przy czym szczególną uwagę należy zwrócić na wartość `ALWAYS`, której wybranie spowoduje zaliczenie testu dla każdego piksela. Sposób aktualizacji zawartości bufora kontroluje z kolei pole `DepthWriteMask`. Domyślna wartość `ALL` pozwala na zapis dla wszystkich pikseli, które uprzednio przeszły test; wartość `ZERO` wyłącza możliwość zapisu dla wszystkich pikseli, niezależnie od wyniku testu. Właściwość ta jest wykorzystywana w praktyce, gdy dochodzi do rysowania przezroczystych obiektów, a chce się uniknąć kosztownego i nietrywialnego sortowania względem odległości od obserwatora. Należy też pamiętać, że stan mieszania jest całkowicie niezależny od stanu głębokości – nawet jeżeli z funkcji mieszania wyniknie, że piksel powinien być niewidoczny (np. zerowe alfa), to zostanie on poddany testowi głębokości ze wszystkimi tego konsekwencjami, włącznie z ewentualnym zapisem nowej wartości do bufora.

Bufor szablonu

Bufor szablonu (ang. *stencil buffer*) współdzieli pewne cechy z buforem głębokości. Oba obiekty mają tę samą rozdzielczość, oba są używane do bloko-

wania zapisywania pewnych pikseli do bufora ramki oraz oba akumulują pewne informacje. To, co wyróżnia bufor szablonu, to fakt, że dane przez niego gromadzone i wykorzystywane bez kontekstu są niemożliwe do zinterpretowania, gdyż są specyficzne dla danej aplikacji. Reguła odrzucania pikseli i akumulacji wartości zarazem zdefiniowana jest następująco:

$Reference$ - wartość odniesienia
$ReadMask$ - maska odczytu
$Comparision$ - funkcja porównująca
S - bieżąca wartość w buforze szablonu
if Piksel zaakceptowany przez buforowanie głębokości **then**
  if $Comparison(Reference \wedge ReadMask, S \wedge ReadMask)$ **then**
    $S := Update(Pass)$
    Zaakceptuj piksel
  else
    $S := Update(Fail)$
    Odrzuć piksel
  end if
else
  $S := Update(DepthFail)$
end if

Wartości $Reference$ i $ReadMask$ oraz funkcje $Comparison$ i $Update$ to elementy konfiguracji stanu.

Klasycznym zastosowaniem bufora szablonu jest, *nomen omen*, generacja szablonu, według którego piksele będą odrzucane albo akceptowane. W tym celu najpierw odrysowuje się siatkę i dla każdego piksela, który przeszedł test głębokości, ustawiana jest wartość w buforze. Dysponując taką maską, można, po zmianie warunku akceptacji piksela, odrysowywać wyłącznie w regionie określonym siatką-szablonem. Obecnie bufor szablonu znajduje zastosowanie m.in. w technikach generacji cieni wolumetrycznych [1] lub tzw. cieniowaniu odroczonym (ang. *deferred shading*) [14]. Szczegółowy opis pól odpowiedzialnych za kontrolę tego aspektu etapu scalania znajduje się w dokumentacji.

Mechanizm bufora szablonów stanowi przydatne narzędzie do optymalizacji wydajnościowej, gdyż dzięki niemu można przetwarzać w jednostkach cieniowania pikseli wyłącznie te fragmenty, które tego potrzebują. Przykładowo, implementując światłocień można w ten sposób wykluczyć obiekty, które nie wymagają tego typu obliczeń.

4.9.4 Pozostałe

W Direct3D 10 występują jeszcze dwa typy noszące cechy obiektów stanu, ale rzadko używane są w sposób podobny do poprzednio wymienionych. Pierwszy

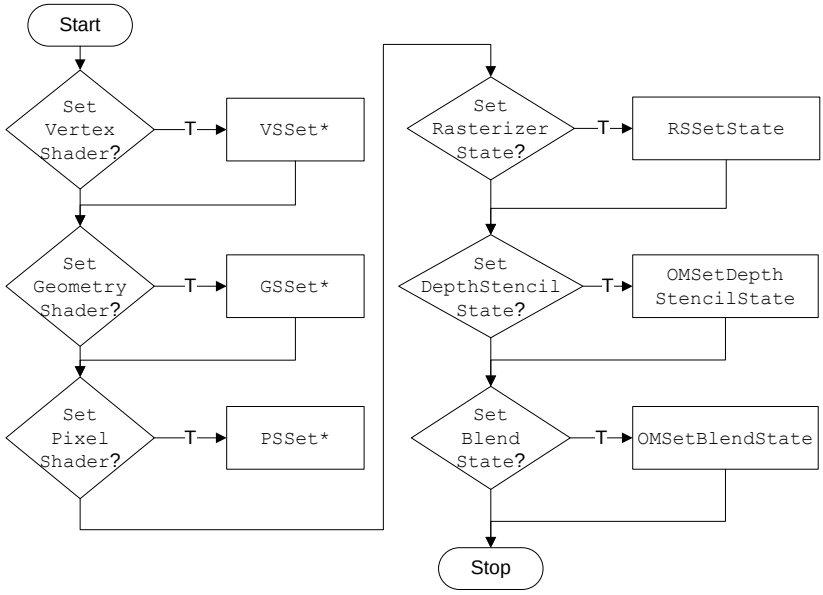
z nich to konfiguracja asemblera wejściowego reprezentowana przez interfejs `ID3D10InputLayout`. Drugim takim typem jest stan próbkowania tekstury definiowany w efekcie.

Zrozumienie tego, co tak naprawdę dzieje się w przy wywołaniu metody `ID3D10EffectPass::Apply`, znacznie pomaga w projektowaniu pętli rysującej nastawionej na uniwersalność. W książce nie są omawiane metody z przedrostkami `VS`, `GS` i `PS`, które reprezentują etapy cieniowania. Ich obsługa jest dość skomplikowana i żmudna, ale już sama ich obecność to sygnał, że stan całego potoku można kontrolować ręcznie bez użycia efektów. Wygodne i sprawdzające się w praktyce jest założenie, że metoda `ID3D10EffectPass::Apply` składa się z sekwencji nadań stanów przy użyciu odpowiednich metod `ID3D10Device` tym etapom, które są wymienione w przebiegu (jak na rysunku 4.8).
W związku z tym, nawet jeżeli przebieg definiuje np. specyficzny stan rasteryzera, to po wywolaniu `ID3D10EffectPass::Apply` można go odczytać i zmodyfikować lub użyć zupełnie innego stanu. Łatwo dzięki temu zaimplementować pętlę rysującą, która będzie w stanie np. wyrenderować szkielet wszystkich siatek. Trzeba pamiętać, że zachodzi też sytuacja odwrotna – na nic są modyfikacje stanu metodami `ID3D10Device`, jeżeli przebieg definiuje swoje własne stany.
Takie zachowanie wyjaśnie też, dlaczego zawsze powinno się ustawiać bądź zerować jednostkę cieniowania geometrii – w innym przypadku istnieje możliwość „odziedziczenia" stanu tego etapu z poprzedniego odrysowania, co z oczywistych powodów może spowodować błędy wyświetlania.

4.9.5 Zastosowanie zmiany stanów

Przykładem efektu graficznego bazującego na wszystkich opisanych powyżej obiektach stanów może być wzbogacone odrysowywanie tylnych ścian siatek. W standardowym przypadku wystarczy zmienić ustawienie `CullMode` stanu rasteryzera, jednak uzyskany efekt graficzny może być trudny do interpretacji. Wobec tego można dodać obrys wszystkich prymitywów, niezależnie od ich orientacji. Dzięki temu wypełniane będą tylne ściany, ale też można będzie na podstawie szkieletu wnioskować na temat kształtu siatki. Dodatkowo, aby ostateczny efekt był bardziej wiarygodny i lepiej oddawał zaglądania „wgłąb" ciała, przednie ściany można odrysować z silnym mieszaniem tak, aby ich obecność była delikatnie odzwierciedlona.

Zadanie można oczywiście zrealizować na poziomie aplikacji konstruując i przełączając się między odpowiednimi stanami. Znacznie łatwiejsze, szybsze oraz bardziej przejrzyste jest jednak tworzenie stanów w efektach. W związku z tym w przykładzie zdefiniowano następujące stany:



Rysunek 4.8. Schemat blokowy metody `ID3D10EffectPass::Apply`

```
//! Wszystkie ściany, szkielet.
RasterizerState rsNoCullWireframe {
    FillMode = WIREFRAME;
    CullMode = NONE;
};
//! Usuwanie przednich ścian.
RasterizerState rsCullFrontSolid {
    CullMode = FRONT;
};
//! Brak zapisu do bufora głębokości.
DepthStencilState dssNoWrite {
  DepthWriteMask = ZERO;
};
//! Alpha blending.
BlendState bsAlphaBlending {
  BlendEnable[0] = TRUE;
  SrcBlend = SRC_ALPHA;
  DestBlend = INV_SRC_ALPHA;
  BlendOp = ADD;
  SrcBlendAlpha = ONE;
  DestBlendAlpha = ZERO;
  BlendOpAlpha = ADD;
};
```

Jak widać, przy znajomości domyślnych parametry stanów tworzenie nowych sprowadza się wyłącznie do zaznaczenie zmian. Powinna to też być pewna podpowiedź dlaczego nie zdefiniowano stanu dla, przykładowo, `CullMode` wynoszącego `BACK`, skoro konieczność jego użycia wynika bezpośrednio z polecenia. Otóż stan można w prosty sposób wyzerować, podając funkcji ustawiającej wartość `NULL`. W związku z tym instrukcje

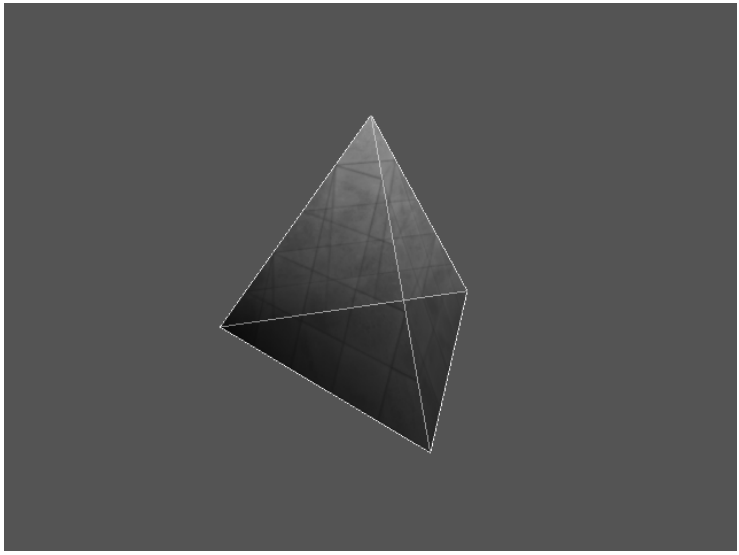
```
SetRasterizerState( NULL ); SetDepthStencilState( NULL, 0);
SetBlendState(NULL, float4(0,0,0,0), 0xFFFFFFFF)
```

powodują wczytanie domyślnych stanów etapów. Właściwość tę wykazują również metody dostępne z poziomu aplikacji.

Ostatecznie technika odpowiadająca za rysowanie wzbogaconej siatki będzie składać się z trzech przebiegów, z których każdy stanowi inną kombinację stanów.

```
float4 AlphaShader(VertexOutput input, uniform float alpha) :
SV_Target {
  float4 Cv = float4(input.color, alpha);
  float4 Ct = g_texture.Sample(linearSampler, input.texcoords);
  return Cv * Ct;
}


float4 ColorShader(VertexOutput input, uniform float4 color) :
SV_Target {
  return color;
}


VertexShader vsSimple = CompileShader(vs_4_0, SimpleVertexShader());
technique10 RichBackfaces {
  // rysowanie tylko tylnych ścian plus wyłączenie zapisu
  // głębokości
  pass PassBackfaces
  {
    SetVertexShader( vsSimple );
    SetGeometryShader( NULL );
    SetPixelShader( CompileShader(ps_4_0, AlphaShader(1.0)) );
    SetRasterizerState( rsCullFrontSolid );
    SetDepthStencilState( dssNoWrite, 0);
  }

  // zmiana stany rasteryzera na rysowanie szkieletu
  // wszystkich ścian białym kolorem
  pass PassWireframe
  {
    SetVertexShader( vsSimple );
    SetGeometryShader( NULL );
    SetPixelShader( CompileShader(ps_4_0, ColorShader(float4(1,1,1,1))) );
    SetRasterizerState( rsNoCullWireframe );
  }

  // przywrócenie domyślnych stanów potoku oraz włączenie
  // mieszania
  pass PassFrontfaces
  {
    SetVertexShader( vsSimple );
    SetGeometryShader( NULL );
    SetPixelShader( CompileShader(ps_4_0, AlphaShader(0.5)) );
    SetRasterizerState( NULL );
    SetDepthStencilState( NULL, 0 );
    SetBlendState( bsAlphaBlending, float4(1,1,1,1), 0xFFFFFFFF );
  }
}
```

Jedyną wymaganą zmianą w kodzie jest modyfikacja nazwy pobieranej techniki. Ostateczny rezultat przedstawia rysunek 4.9.



Rysunek 4.9. Czworościan odrysowany za pomocą techniki „wzbogaconych tylnych ścian”

5

Tworzenie środowiska testowego

Przykłady stworzone na potrzeby poprzedniego rozdziału nie korzystały z żadnych dodatkowych bibliotek, poza dołączonymi do SDK D3DX oraz DXGI. Powodem takiego stanu rzeczy była chęć pokazania dokładnego przebiegu inicjalizacji oraz obsługi urządzenia Direct3D. Kolejne przykłady zawarte w książce korzystają z bardziej rozwiniętego środowiska testowego, które implementując znaczną część funkcjonalności niezwiązanych bezpośrednio z efektami graficznymi, w znaczący sposób wpływają na uproszczenie kodu.

5.1 Standard Template Library

Standard Template Library (STL) to standardowa biblioteka C++ zawierająca kontenery, algorytmy oraz iteratory (łączniki pomiędzy kontenerami a algorytmami). Wszystkie te konstrukty dostarczane są w postaci szablonów (ang. *template*) w przestrzeni nazw (ang. *namespace*) `std`. Biblioteka, ze względu na bardzo zaawansowane konstrukcje językowe, jest trudna do opanowania, jednak w przykładach jej zastosowanie ograniczone zostało do podstawowych kontenerów (`std::vector`, `std::map` i `std::set`), łańcuchów (`std::string`), strumieni (pochodne `std::istream`) oraz wybranych algorytmów.

5.2 Microsoft Foundation Classes

Microsoft Foundation Classes (MFC) to biblioteka, która opakowując funkcjonalność Windows API w klasy C++ i dostarczając różnorakiego kodu użytkowego (kolekcje, serializacja), może stać się szkieletem aplikacji pracującej w systemie Windows. W przykładach używana jest właśnie część użytkowa, gdyż obsługę okna i zdarzeń przejmuje na siebie DXUT.

Często wykorzystywanym komponentem biblioteki jest klasa `CComPtr` (nagłówek `atlbase.h`). Używanie tej klasy zamiast wskaźników dla lokalnych interfejsów COM praktycznie całkowicie zwalnia programistę z obowiązku ręcznego wywoływania metody `IUnknown::Release`, nawet w przypadku błędów.

Wynika to ze specyfiki C++ - destruktory lokalnych obiektów wywoływane są niezależnie od tego, czy bieżąca funkcja została zakończona zwróceniem wyniku (słowo kluczowe `return`) lub wyrzuceniem wyjątku (`throw`), a to właśnie w destruktorze `CComPtr` następuje zmniejszanie licznika referencji.

Korzyści płynące z zrzucenia części odpowiedzialności programisty na mechanizmy językowe pokazuje przykład. Jest to fragment kodu przedstawiający uzupełniony o obsługę błędów proces tworzenia urządzenia i łańcucha wymiany z użyciem DXGI.

```
IDXGIFactory* factory = NULL; IDXGIAdapter* adapter = NULL; /* ...
ustawienie wskaźników ... */
// stworzenie urządzenia
hr = D3D10CreateDevice( adapter, D3D10_DRIVER_TYPE_HARDWARE, NULL,
  D3D10_CREATE_DEVICE_DEBUG, D3D10_SDK_VERSION, &g_device);
// obsług błędu
if ( FAILED(hr) ) {
  // zwolnienie lokalnych zasobów
  adapter->Release();
  factory->Release();
  return hr;
}
// stworzenie łańcucha
hr = factory->CreateSwapChain(g_device, &swapChainDesc,
&g_swapChain);
// zwolnienie lokalnych zasobów
adapter->Release(); factory->Release(); if ( FAILED(hr) ) {
  return hr;
}
```

Drugi fragment jest logicznie równoważny pierwszemu, lecz widoczną różnicą jest brak konieczności zwalniania obiektów w przypadku błędów (dzieje się to automatycznie). Dzięki temu programista może skupić się na rozwijaniu funkcjonalności, a nie na łataniu wycieków pamięci w przypadku zaistnienia wyjątkowych sytuacji.

```
#define CHECK_HR(func) hr = (func); if (FAILED(hr)) return hr; ... {
  CComPtr<IDXGIFactory> factory;
  CComPtr<IDXGIAdapter> adapter;
  /* ... ustawienie wskaźników ... */
  // stworzenie urządzenia
  CHECK_HR(D3D10CreateDevice( adapter, D3D10_DRIVER_TYPE_HARDWARE, NULL,
    D3D10_CREATE_DEVICE_DEBUG, D3D10_SDK_VERSION, &g_device));
  // stworzenie łańcucha
  CHECK_HR(factory->CreateSwapChain(g_device, &swapChainDesc, &g_swapChain)
      );
  // zasoby są zwalniane automatycznie przez destruktor CComPtr,
  // nawet jeśli funkcja się zakończy na makrze CHECK_HR
}
```

5.3 DirectX Utility Toolkit

DirectX Utility Toolkit (DXUT) to szkielet większości samouczków i przykładów zawartych w DirectX SDK. Biblioteka upraszcza m.in. wybór i tworzenie urządzenia Direct3D oraz okna i obsługę jego zdarzeń. Dodatkowo, w części

opcjonalnej, DXUT udostępnia standardowe modele kamer, prosty model GUI oraz obsługę dźwięku.

Integracja z DXUT odbywa się przez wskazanie funkcji zwrotnych (ang. *callback*), które mają być wywołane w określonych momentach. Przykładowo, poniższy kod przypisuje pustą funkcję użytkownika `OnD3D10CreateDevice` do zdarzenia utworzenia urządzenia Direct3D – ciało tej funkcji jest miejscem, gdzie powinna dokonać się wszelka dodatkowa inicjalizacja, taka jak wczytanie siatek, tekstur i efektów.

```
// definicja callbacka
HRESULT CALLBACK OnD3D10CreateDevice( ID3D10Device* device,
  const DXGI_SURFACE_DESC* surfaceDesc, void* userContext )
{
    return S_OK;
} /* ... */
// przypisania callbacka do zdarzenia
DXUTSetCallbackD3D10DeviceCreated( OnD3D10CreateDevice );
```

Samo tworzenie okna i urządzenia jest nieporównywalnie mniej skomplikowane niż w przypadku używania „gołego" Windows API – poniższy fragment tworzy okno o rozmiarze 640x480 pikseli.

```
DXUTCreateWindow(L"Przykładowe okno")); DXUTCreateDevice( true,
640, 480 );
```

Przedrostek `L` przed łańcuchami wynika z faktu, iż DXUT działa wyłącznie w trybie Unicode. Oznacza to, że większość stałych tekstowych musi być definiowana z tym przedrostkiem, natomiast zamiast typów `char*` oraz `std::string` trzeba używać odpowiednio `WCHAR*` i `std::wstring`.

Samouczki i przykłady z DirectX SDK załączają do swoich projektów kod źródłowy DXUT. Takie rozwiązanie zwiększa czas kompilacji oraz utrudnia wprowadzanie własnych modyfikacji, gdyż nie propagują się one na wszystkie korzystające z narzędzia programy. Z uwagi na to, przykłady przygotowane na potrzeby niniejszej książki korzystają ze wspólnej, statycznej biblioteki DXUT oraz jej opcjonalnej części DXUTOptional.

Jak każda biblioteka, DXUT nie jest pozbawiona niedoskonałości. Dzięki otwartej architekturze istnieje jednak możliwość naprawy lub modyfikacji jej logiki. W związku z tym, że dołączone do książki źródła są nieznacznie zmienione w stosunku do oryginału, dołączona została lista zmian w formacie `diff` znajdująca się w pliku `/etc/DXUT.diff`.

Począwszy od następnego rozdziału makro `CHECK_HR` zastępowane będzie przez wbudowany w DXUT odpowiednik, czyli `V_RETURN` i makro `V`, które wyłącznie testuje rezultat, bez zwracania wartości w przypadku błędu.

5.4 Wczytywanie modeli OBJ

W dotychczasowych przykładach geometria była zaszyta w kodzie źródłowym w postaci definicji tablic wierzchołków i indeksów. Podejście takie, chociaż pozwala na szybkie stworzenie testowej aplikacji, jest niepraktyczne. Jako powody można wymienić konieczność kompilacji kodu po każdej zmianie geometrii i mało intuicyjną, trudną w interpretacji reprezentację. W praktyce do tworzenia siatek używa się zaawansowanych edytorów graficznych. W tej sytuacji zadaniem programisty staje się stworzenie (bądź użycie istniejącego) kodu do odczytu pliku z modelem. Kryteriów wyboru obsługiwanego typu pliku siatek jest wiele, z najważniejszych można wymienić solidne wsparcie przez edytory graficzne, możliwości oraz dostępność gotowych modeli.

5.4.1 Opis struktury

Format OBJ (rozszerzenie .obj)[1] służy do przechowywania danych statycznych siatek. Dzięki prostej strukturze jest wspierany przez większość edytorów 3D. Dodatkowo, z racji tekstowej reprezentacji danych wierzchołków, łatwo poddaje się testowaniu i modyfikacjom. Każda linia składa się z polecenia oraz danych. Format definiuje wiele poleceń, jednak dla klarowności omówione i zaimplementowane będą wyłącznie te podstawowe, zdefiniowane w tablicy 5.1.

Tablica 5.1. Podstawowe polecenia OBJ

Polecenie	Opis	Przykład
`#`	Komentarz	`# To jest komentarz`
`v`	Dodanie pozycji	`v 0.123 0.456 0.789`
`vt`	Dodanie współrzędnych tekstury	`vt 0.0 1.0`
`vn`	Dodanie wektora normalnego	`vn 0.0 1.0 0.0`
`f`	Ściana	`f 1/2/3 4/5/6 7/8/9`
`mtllib`	Wybór biblioteki materiałów	`mtllib materials.mtl`
`usemtl`	Wybór materiału	`usemtl inside`

W pliku OBJ najpierw definiowane są osobno atrybuty wierzchołków: pozycja, współrzędne tekstury oraz wektor normalny wierzchołka. Na etapie polecenia `f` składane są z nich same wierzchołki, a z tych ściany. Indeksy atrybutów oddziela się znakiem `\`, przy czym współrzędne tekstury i wektor normalny są opcjonalne (możliwe kombinacje przedstawia tabela 5.2). Zarówno przy ręcznym tworzeniu plików OBJ, jak i przy ich odczycie należy pamiętać, że indeksacja rozpoczyna się od jedynki. Polecenia `mtllib` oraz `usemtl` nie mają bezpośredniego związku z geometrią, ale ich obsługa na tym etapie jest konieczna z punktu widzenia kolejnego rozdziału. Pierwsze z nich wskazuje plik biblioteki materiałów, natomiast drugie wybiera bieżący materiał, co oznacza,

[1] Specyfikacja: `http://local.wasp.uwa.edu.au/~pbourke/dataformats/obj/`

że wszystkie kolejne ściany powinny go używać aż do momentu kolejnego jego wystąpienia.

Tablica 5.2. Możliwe definicje wierzchołków polecenia OBJ.

Atrybuty wierzchołka	Format
Pozycja	$\mathtt{i}_p$
Pozycja + współrzędne tekstury	$\mathtt{i}_p/\mathtt{i}_t$
Pozycja + wektor normalny	$\mathtt{i}_p//\mathtt{i}_n$
Pozycja + współrzędne tekstury + wektor normalny	$\mathtt{i}_p/\mathtt{i}_t/\mathtt{i}_n$

$\mathtt{i}_i$ - indeks pozycji, $\mathtt{i}_t$ - indeks współrzędnych tekstur, $\mathtt{i}_n$ - indeks wektora normalnego

Ściana musi składać się co najmniej z trzech wierzchołków, a może ich być więcej. Często modele OBJ budowane są z prostokątnych ścian, a ponieważ Direct3D nie wspiera prymitywów czterowierzchołkowych, w tym przypadku konieczny jest podział na dwa trójkąty na etapie odczytywania. W dalszej części praca będzie zmierzać do poprawnego odczytania pliku reprezentującego sześcian. Poniżej znajduje się jego uproszczona wersja, bez wektorów normalnych.

```
mtllib cube_materials  v -0.5  0.5  0.5 v -0.5 -0.5  0.5 v  0.5 -0.5
0.5 v  0.5  0.5  0.5 v -0.5  0.5 -0.5  v -0.5 -0.5 -0.5  v  0.5 -0.5
-0.5 v  0.5  0.5 -0.5  vt 0.0 0.0 vt 1.0 0.0  vt 1.0 1.0  vt 0.0 1.0
usemtl shiny f 1/1 2/2 3/3 4/4 usemtl textured f 8/1 7/2 6/3 5/4 f
4/1 3/2 7/3 8/4  f 5/1 6/2 2/3 1/4  f 5/1 1/2 4/3 8/4  f 2/1 6/2 7/3
3/4
```

5.4.2 Wczytywanie danych

W odróżnieniu od wektorów (reprezentujących pozycje, współrzędne tekstury i wektory normalne) oraz łańcuchów (typ `std::string`), na próżno szukać w bibliotekach DirectX oraz bibliotece STL typów nadających się do reprezentacji ściany i, pośrednio, wierzchołka. Poniżej znajduje się przykładowa reprezentacja danych możliwych do wczytania z plików OBJ.

```
struct ObjData {
  //! ID wierzchołka.
  struct ObjVertex
  {
    //! Indeks pozycji.
    int position;
    //! Indeks koordynatów tekstury.
    int texcoord;
    //! Indeks wektora normalnego.
    int normal;
    //! Operator porównania, umożliwa używanie jako klucza w kontenerze std
        ::map
```

```
    //! oraz umieszczanie w kontenerze std::set
    inline bool operator<(const ObjVertex& rhs) const
    {
      // wywołanie standardowej funkcji porównującej obszary pamięci
      return memcmp(&position, &rhs.position, sizeof(ObjVertex)) < 0;
    }
  };

  //! Prymityw składowy siatki.
  struct ObjFace
  {
    //! Wierzchołki składowe.
    ObjVertex v[3];
  };

  //! Pozycje.
  std::vector<D3DXVECTOR3> positions;
  //! Współrzędne tekstury.
  std::vector<D3DXVECTOR2> texcoords;
  //! Wektory normalne.
  std::vector<D3DXVECTOR3> normals;
  //! ściany.
  std::vector<ObjFace> faces;
  //! Biblioteka materiałów.
  std::string materialLib;
  //! Materiał w znaczeniu formatu OBJ  to nazwa
  //! oraz indeks ściany, od której obowiązuje
  typedef std::pair<std::string, int> Material;
  //! Nazwy materiałów oraz indeksy ścian, od których obowiązują.
  std::vector<Material> materials;
};
```

Typ `ObjVertex` posiada operator mniejszości, dzięki czemu może stanowić klucz w kontenerach `std::map` oraz `std::set`. Właściwość ta będzie wykorzystywana później do jednoznacznej identyfikacji wierzchołka i potencjalnej optymalizacji pamięciowej.

Mając ustalony typ danych, można zaimplementować odczyt z pliku. Ponieważ format OBJ jest tekstowy, wygodnie jest zastosować strumienie wejściowe (pochodne `std::istream`) z biblioteki STL, ze względu na prostotę obsługi oraz niezawodność. Dodatkowo umożliwia to np. wczytywanie modelu bezpośrednio z wejścia standardowego (`std::cin`) lub ze strumieni bazujących na pamięci. Na potrzeby kolejnych przykładów rozszerzono strukturę `ObjData` o statyczną metodę `Create` wypełniającą dane przekazanej struktury na podstawie strumienia. Jej szkielet przedstawia się jak poniżej.

```
void ObjData::Create( std::istream& input, ObjData& data ) {
  /* ... */
  // pętla odczytująca
  std::string command;
  while (input >> command) {
    /* obsluga polecenia */
    // przejście do następnej linii
    input.ignore(INT_MAX, '\n');
  }
}
```

Warunek pętli może wyglądać jak powyżej, gdyż `std::istream` posiada przeładowany operator rzutowania na typ `void*` zwracający `NULL` wyłącznie wtedy, gdy strumień znajduje się w nieodpowiednim stanie. Obsługa polece-

nia, podobnie jak jego wczytanie, bazuje wyłącznie na operatorach strumieniowych, przez co programista nie musi pilnować rozmiaru odczytanych danych. Przykładowo, pozycję odczytać można następująco:

```
if ( command == "v" ) {
  // odczyt pozycji
  D3DXVECTOR3 position;
  input >> position.x >> position.y >> position.z;
  data.positions.push_back(position);
}
```

> Sekwencja porównań typu `std::string` do stałych łańcuchowej jest daleka od optymalności. Przede wszystkim operator porównania musi każdorazowo obliczyć długość łańcucha. Dodatkowo polecenia można by sprawdzać w kolejności leksykograficznej tak, aby minimalizować liczbę niepotrzebnych porównań.

Jedynie odczyt ściany wiąże się z nieco bardziej skomplikowaną logiką, ponieważ może występować zmienna liczba wierzchołków (trzy lub cztery). Dodatkowo zmienność występuje również w zakresie indeksów, które przypisane są do każdego z wierzchołków. Kod będący w stanie odczytać te ostatnie wyglądać może jak poniżej. Dzięki temu, że atrybuty w pliku OBJ indeksowane są od jedności, wartość 0 jest wartością specjalną, świadczącą o braku danej składowej.

```
ObjVertex vertex; input >> vertex.position; vertex.texcoord =
vertex.normal = 0; if ( input.peek() == '/' ) {
  input.ignore();
  if ( input.peek() != '/' ) {
    input >> vertex.texcoord;
  }
  if ( input.peek() == '/' ) {
    input.ignore();
    input >> vertex.normal;
  }
}
```

W omawianym przykładzie typ `ObjData` zajmuje się wyłącznie wczytaniem danych, bez tłumaczenia ich na interfejsy gotowe do użycia przez urządzenie Direct3D. Takie rozwiązanie powoduje, że moduł wczytywania jest uniwersalny oraz istnieje możliwość wykorzystania go w innych projektach, gdzie API graficzne może być inne.

5.4.3 Wyznaczanie wierzchołków, indeksów i atrybutów

D3DX dostarcza interfejs `ID3DX10Mesh`, który reprezentuje siatkę. Nie jest to wyłącznie opakowanie bufora wierzchołków oraz indeksów – interfejs wspiera podział geometrii na podzbiory, tworzenie informacji o sąsiedztwie, optymalizację danych oraz testowanie przecinania (ang. *intersection*) z promieniem.

Zanim jednak przystąpi się do tworzenia siatki, należy przygotować dane dla bufora wierzchołków oraz indeksów. W przypadku dysponowania wypełnioną strukturą `ObjData` konieczne jest przetłumaczenie indeksów atrybutów na faktyczne dane. Typy docelowe przedstawiają się następująco:

```
class ObjMesh { public:
  //! Wierzchołek urzywany przez siatkę.
  struct Vertex
  {
    //! Pozycja.
    D3DXVECTOR3 position;
    //! Współrzędne tekstury.
    D3DXVECTOR2 texcoord;
    //! Wektor normalny.
    D3DXVECTOR3 normal;
  };

  //! Prymityw składowy siatki.
  union Face
  {
    //! Reprezetnacja jako indeksy.
    struct {
      UINT i0;
      UINT i1;
      UINT i2;
    };
    //! Reprezentacja jako tablica indeksów.
    UINT i[3];
  };

... private:
  //! Wierzchołki.
  std::vector<Vertex> vertices;
  //! Indeksy.
  std::vector<Face> faces;
... }
```

Przed tłumaczeniem danych z plików OBJ na dane geometrii należy uwzględnić kilka czynników. Przede wszystkim w praktyce kombinacje pozycja/współrzędne/wektory często się powtarzają, gdyż ściany współdzielą ze sobą niektóre wierzchołki. Otwiera to pole do optymalizacji pamięciowej poprzez zapisywanie tylko unikatowych wystąpień oraz stworzenie bufora indeksów do wyznaczania samych prymitywów. Dzięki zdefiniowaniu operatora mniejszości dla typu `ObjData::ObjVertex` możliwe jest używanie go jako klucza w słowniku `std::map`, co zostało wykorzystane do sprawdzania unikatowości wierzchołków.

```
ObjData data; ...
// słownik z już wczytanymi wierzchołkami
// klucz to wartość odczytana z pliku OBJ, natomiat wartość
// to indeks w tablicy vertices
typedef std::map<ObjData::ObjVertex, size_t> VerticesMap;
VerticesMap uniqueVertices;

// konstrukcja właściwych danych
typedef std::vector<ObjData::ObjFace> FacesList; for (
FacesList::iterator it = data.faces.begin();
      it != data.faces.end(); ++it ) {
  // wyznaczenie indeksów wierzchołków
  ObjData::ObjFace & objFace = (*it);
  Face face;
```

```
  for (int i = 0; i < 3; ++i) {
    // czy taki wierzchołek już istnieje?
    ObjData::ObjVertex & objVertex = objFace.v[i];
    VerticesMap::iterator found = uniqueVertices.find(objVertex);
    if ( found != uniqueVertices.end() ) {
      // tak, pobieramy jego indeks
      face.i[i] = found->second;
    } else {
      // stworzenie wierzchołka
      // trzeba pamiętać, że OBJ indeksuje od jedynki!
      Vertex vertex;
      // pozycja musi być zdefiniowana
      vertex.position = data.positions[objVertex.position-1];
      // jeżeli indeks koordynatów albo wektora jest równy 0, to znaczy,
      // że go nie podano i należy nadać domyślną wartość
      vertex.texcoord = objVertex.texcoord ?
        data.texcoords[objVertex.texcoord-1] : D3DXVECTOR2(0,0);
      vertex.normal = objVertex.normal ?
        data.normals[objVertex.normal-1] : D3DXVECTOR3(0,0,0);
      vertices.push_back(vertex);
      // dodanie klucza do słownika oraz do ściany
      face.i[i] = uniqueVertices[objVertex] = vertices.size() - 1;
    }
  }
  // dodanie do listy ścian
  faces.push_back(face);
}
```

Teoretycznie na tym etapie można zakończyć tłumaczenie danych odczytanych z pliku OBJ. W praktyce jednak często pożądane jest stworzenie informacji o podzbiorach. Podzbiór to grupa ścian, która może być odrysowana korzystając z tego samego stanu potoku, czyli między innymi z tych samych tekstur oraz efektów. Dzięki wprowadzeniu tego mechanizmu możliwe jest różnicowanie wyglądu poszczególnych fragmentów obiektu. Przykładowo, szyby samochodu powinny być rysowane w inny sposób niż jego karoseria. Zamiast przygotowywać osobne siatki dla obu części, wygodniej jest zdefiniować dwa podzbiory w ramach tego samego obiektu. W przypadku siatek OBJ kryterium podziału są materiały, które zostaną omówione dokładniej w kolejnym rozdziale. Wobec tego oraz faktu, że dany materiał może być wielokrotnie przypisany różnym grupom ścian, konieczne jest wyznaczenie unikatowego identyfikatora każdemu z nich.

```
//! Nazwy materiałów. Indeks materiału oznacza również jego ID.
std::vector<std::string> materials; ...
// wypełnienie listy materiałów tylko unikatowymi nazwami materiału
for (size_t i = 0; i < data.materials.size(); ++i) {
  // dodanie do listy tylko jeżeli nazwy materiału tam jeszcze nie ma
  const std::string& name = data.materials[i].first;
  if ( std::find(materials.begin(), materials.end(), name) == materials.end
      () ) {
    materials.push_back(name);
  }
}
```

Interfejs `ID3DX10Mesh` wyznacza podzbiory na podstawie listy atrybutów, tj. tablicy, która każdemu trójkątowi przyporządkowuje numer właściwego jemu podzbioru. Aby osiągnąć ten efekt, konieczne jest zmodyfikowanie pętli tworzącej dane ścian i wierzchołków i określanie materiału (i pośrednio atrybutu) dla bieżąco odczytywanego prymitywu.

```
//! Atrybuty ścian. Tutaj: id materiałów.
std::vector<UINT> attributes; ...
std::vector<ObjData::Material>::iterator nextMaterial =
data.materials.begin(); size_t attribute = 0;

// konstrukcja właściwych danych
typedef std::vector<ObjData::ObjFace> FacesList; for (
FacesList::iterator it = data.faces.begin();
      it != data.faces.end(); ++it ) {
  // sprawdzenie, czy rozpoczął obowiązywanie następny materiał
  // indeks bieżącej ściany jest równy rozmiarowi tablicy już wczytanych ś
       cian
  while ( nextMaterial != data.materials.end() && nextMaterial->second <=
       faces.size() ) {
    // wyszukanie w liście unikatowych materiałów
    std::vector<std::string>::iterator found = std::find(materials.begin(),
         materials.end(), nextMaterial->first);
    // określenie indeksu, a więc id
    attribute = std::distance(materials.begin(), found);
    ++nextMaterial;
  }

  // przypisanie atrybutu bieżącej ścianie
  attributes.push_back(attribute);
  /* dalsza część jak porzednio */
}
```

5.4.4 Tworzenie siatki

Dysponując listą wierzchołków, indeksów oraz atrybutów, można stworzyć siatkę `ID3DX10Mesh`. W pierwszej kolejności konieczne jest wywołanie funkcji `D3DX10CreateMesh` i zadeklarowanie tym samym rozmiaru oraz formatu danych. Nietypowym wymaganiem jest konieczność przekazania semantyki elementu opisu bufora odpowiadającego za pozycję.

```
//! Typ opisu danych wierzchołków.
typedef std::vector<D3D10_INPUT_ELEMENT_DESC> InputDesc;
//! Opis danych wierzchołków.
InputDesc inputDesc;
//! Siatka.
ID3DX10Mesh* mesh; ...

// tworzenie układu elementów
D3D10_INPUT_ELEMENT_DESC vertexElements[] = {
  { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
       D3D10_INPUT_PER_VERTEX_DATA, 0 },
  { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 12,
       D3D10_INPUT_PER_VERTEX_DATA, 0 },
  { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 20,
       D3D10_INPUT_PER_VERTEX_DATA, 0 },
}; inputDesc.assign(vertexElements, vertexElements+3);

// tworzenie siatki
CHECK_HR( D3DX10CreateMesh( device,
  &inputDesc[0],
  inputDesc.size(),
  inputDesc[0].SemanticName,  // semantyka pozycji
  vertices.size(),    // liczba wierzchołków
  faces.size(),       // liczba ścian
  D3DX10_MESH_32_BIT, // indeksy 32-bitowe
  &mesh) );
```

```
// ustawienie danych wierzchołków
CHECK_HR( mesh->SetVertexData(0, &vertices[0]) );

// ustawienie danych indeksów
CHECK_HR( mesh->SetIndexData(&faces[0], faces.size()*3) );

// ustawienie danych atrybutów
CHECK_HR( mesh->SetAttributeData(&attributes[0]) );

// zatwierdzenie zmian
CHECK_HR( mesh->CommitToDevice() );
```

Powyższy kod stworzy poprawnie działającą siatkę. Jej używanie jest jednak utrudnione wobec konieczności ręcznego wyznaczania indeksów podzbiorów w trakcie rysowania. Ponadto nie uwzględniono problemu możliwej nieoptymalnej organizacji ścian w buforze indeksów ze względu na liczbę zmian stanu potoku. Istnieje jednak możliwość spowodowania, żeby to obiekt implementujący interfejs `ID3DX10Mesh` sam dokonywał zliczania oraz sortowania atrybutów.

Metoda `ID3DX10Mesh::Optimize` zapewnia szeroki wachlarz możliwości automatycznej modyfikacji danych siatki pod zadanym kątem. Operacje przez nią wykonywane definiuje przekazana suma logiczna flag pochodzących z wyliczenia `_D3DXMESHOPT`. W przypadku sortowania podzbiorów wymaganą flagą jest `D3DXMESHOPT_ATTRSORT` (pozostałe wartości opisane są dokładnie w dokumentacji). Przed samą optymalizacją należy przygotować informacje o krawędziach oraz o ścianach, które je współdzielą. Nie trzeba jednak dokonywać tego ręcznie, gdyż jest metoda `ID3DX10Mesh::GenerateAdjacencyAndPointReps`. Jej parametr określa minimalną odległość, o jaką muszą być oddalone wierzchołki, aby nie być uznawanymi za tożsame.

Należy mieć na uwadze fakt, że po operacji optymalizacji nie można polegać w zależności od dobranych flag, na kolejności wierzchołków, indeksów, jak również atrybutów uzyskanych przy odczycie pliku OBJ. Jeżeli takie dane będą jednak z jakiegoś powodu wymagane, interfejs udostępnia metody pozwalające na pobranie wskaźników na wewnętrzne bufory. Po dokonaniu optymalizacji można pobrać informacje o nowym rozkładzie podzbiorów. Służy do tego metoda `ID3DX10Mesh::GetAttributeTable`, która wypełnia tablicę struktur `D3DX10_ATTRIBUTE_RANGE` – stanowią one bardziej zwartą postać listy atrybutów, gdyż jawnie definiują przedziały. Wszystkie opisane czynności należy wykonać przed zatwierdzeniem siatki metodą `ID3DX10Mesh::Commit`.

```
//! Fragmenty siatki.
std::vector<D3DX10_ATTRIBUTE_RANGE> sections; /* ... */
// optymalizacja
CHECK_HR( mesh->GenerateAdjacencyAndPointReps( 1e-6f ) ); CHECK_HR(
mesh->Optimize( D3DXMESHOPT_ATTRSORT | D3DXMESHOPT_VERTEXCACHE,
NULL, NULL ) );

// pobranie liczby podzbiorów
UINT attributesCount; mesh->GetAttributeTable( NULL,
&attributesCount );
// alokacja miejsca, pobranie podzbiorów
sections.resize(attributesCount); mesh->GetAttributeTable(
&sections[0], &attributesCount );
```

5.4.5 Rysowanie siatki

Z racji obecności podzbiorów pętla rysująca musi ulec zmianie. Ponieważ interfejs `ID3DX10Mesh` zapewnia automatyczne ustawianie buforów wierzchołków i indeksów, zadanie programisty sprowadza się do zapewnienia połączenia z efektem (układ wierzchołków, zmienne efektu) oraz wskazania numeru podzbioru. Biorąc pod uwagę, że cała funkcjonalność opisana w bieżącym rozdziale została zamknięta w klasie `ObjMesh`, modyfikacja funkcji rysującej może wyglądać następująco:

```
ObjMesh * g_mesh = NULL; ...
// ustawienie układu wierzchołków
g_device->IASetInputLayout(g_vertexLayout);
// ustawienie typu prymitywu
g_device->IASetPrimitiveTopology(D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
// odrysowanie z użyciem wszystkich przebiegów techniki
D3D10_TECHNIQUE_DESC techniqueDesc;
CHECK_HR(g_technique->GetDesc(&techniqueDesc)); for (UINT subset =
0; subset < g_mesh->GetSubsetsCount(); ++subset) {
  for(UINT i = 0; i < techniqueDesc.Passes; ++i) {
    // nadanie stanu zdefiniowanego w przebiegu
    CHECK_HR(g_technique->GetPassByIndex(i)->Apply(0));
    // odrysowanie
    g_mesh->GetMesh()->DrawSubset(subset);
  }
}
```

Metoda `ObjMesh::GetSubsetsCount` zwraca liczbę podzbiorów (rozmiar pojawiającej się wcześniej w przykładach tablicy `sections`), natomiast metoda `ObjMesh::GetMesh` zwraca wskaźnik na `ID3D10XMesh`.

5.5 Wczytywanie materiałów MTL

Materiał to w grafice trójwymiarowej zespół właściwości przypisanych danej powierzchni. Przykładowo, materiał może decydować o kolorze (w odróżnieniu od poprzednich przykładów, gdzie to kolor pochodził wprost z wierzchołków), wskazywać teksturę oraz określać, jak światło powinno być odbijane. Wprowadzenie abstrakcyjnych materiałów jest kolejnym krokiem ku uwolnieniu programisty od obowiązków stricte artystycznych, gdyż pozwala na zmianę sposobu wyświetlania obrazu bez modyfikacji kodu programu. O ile siatki powiązane są z efektami poprzez dane wierzchołków, to materiały zajmują się modyfikacją zmiennych globalnych oraz (ewentualnie) wyborem technik.

5.5.1 Opis struktury

Pliki MTL (rozszerzenie .mtl)[2] to biblioteki materiałów towarzyszące plikom OBJ. Tutaj również zawartość ma postać tekstową, która logicznie jest sekwencją poleceń. Podstawowe instrukcje prezentuje tabela 5.3. Dodatkowo, aby

[2] Specyfikacja: `http://local.wasp.uwa.edu.au/~pbourke/dataformats/mtl/`

umożliwić lub uprościć kontrolę nad pewnymi efektami graficznymi, na potrzeby książki wprowadzono też dodatkowe polecenia. Wszystkie rozpoczynają się znakiem komentarza `#`, więc z punktu widzenia standardu nie wprowadzają niekompatybilnych elementów. Lista nowych instrukcji wraz z ich opisem znajduje się w tabeli 5.4. Omówienie znaczenia poleceń standardowych, jak i rozszerzonych wykracza poza bieżący materiał, dlatego też w niniejszym rozdziale nastąpi wyłącznie zaprezentowanie przykładowego sposobu wczytywania danych oraz interpretacji pliku. Warto jeszcze dodać, że każde polecenie w ramach materiału jest opcjonalne.

Tablica 5.3. Podstawowe polecenia MTL

Polecenie	Opis	Przykład
`#`	Komentarz	`# To jest komentarz`
`newmtl`	Rozpoczęcie nowego materiału	`usemtl inside`
`Kd`	Kolor dla światła rozproszonego	`Kd 1.0 0.0 0.0`
`Ka`	Kolor dla światła otoczenia	`Ka 0.0 0.0 0.0`
`Ks`	Kolor dla światła zwierciadlanego	`Ks 1.0 1.0 1.0`
`d`	Przeźroczystość	`diss 1.0`
`Ns`	Wykładnik odbicia światła zwierciadlanego	`Ns 32`
`map_Kd`	Tekstura światła rozproszonego	`map_Kd texture.jpg`
`map_Ka`	Tekstura światła otoczenia	`map_Ka texture.jpg`
`map_Ks`	Tekstura dla światła zwierciadlanego	`map_Ks specular.jpg`
`disp`	Tekstura mapy przemieszczeń	`disp displacement.jpg`
`bump`	Tekstura wypukłości	`bump bump.jpg`
`bump`	Tekstura odbić	`refl refl.jpg`
`illum`	Tryb odbijania światła	`illum 2`

Tablica 5.4. Niestandardowe polecenia MTL

Polecenie	Opis	Przykład
`#roughness`	Porowatość powierzchni (izotropiczna)	`#roughness 1`
`#roughness`	Porowatość powierzchni (anizotropiczna)	`#roughness 1 1`
`#tess`	Domyślny stopień kafelkowania	`#tess 10`
`#disp`	Skala nanoszenia przemieszczeń	`#disp 5`

Jedna biblioteka może być używana przez dowolną liczbę siatek, przez co mogą w niej znajdować się zbędne z punktu widzenia konkretnego modelu materiały. Odwrotna sytuacja nie może mieć miejsca – wszystkie materiały zadeklarowane poleceniami `usemtl` w pliku OBJ muszą znaleźć swoje odpowiedniki w pliku MTL. Dzieje się tak dlatego, gdyż można wskazać nazwę wyłącznie jednej biblioteki (polecenie `mtllib`). Poniżej znajduje się bardzo uproszczona zawartość przykładowej biblioteki MTL.

```
# materiał wypełniany teksturą newmtl textured map_Kd texture.bmp

# materiał połyskliwy newmtl shiny Kd 0.0 0.0 0.0 Ks 1.0 1.0 1.0
Ns 32 #roughness 0.5 0.1
```

5.5.2 Wczytywanie danych

Analizując wybrane polecenia pliku MTL można w łatwy sposób stworzyć strukturę zdolną przechować wszystkie dane. Na tym etapie, podobnie jak to było w przypadku plików OBJ, warto w imię uniwersalności i przejrzystości wstrzymać się z tłumaczeniem wartości na interfejsy Direct3D. Wobec tego zmienne wektorowe i skalarne wczytywane są bezpośrednio do odpowiednich typów, natomiast nazwy tekstur - do łańcuchów `std::string`.

```
//! Dane wczytane z pliku MTL.
struct MtlData {
  //! Dane materiału.
  struct MtlMaterial
  {
    //! Nazwa materiału.
    std::string name;
    //! Kolor dla światła rozproszonego.
    D3DXVECTOR3 diffuseColor;
    //! Kolor bez światła otoczenia.
    D3DXVECTOR3 ambientColor;
    //! Kolor bez światła odbitego.
    D3DXVECTOR3 specularColor;
    //! Współczynnik mieszania z tłem.
    float dissolve;
    //! Współczynnik odbijania światła.
    int specularExponent;
    //! Tekstura dla światła rozproszonego.
    std::string diffuseMap;
    //! Tekstura dla światła otoczenia.
    std::string ambientMap;
    //! Tekstura dla światła odbitego.
    std::string specularMap;
    //! Tekstura dla mapy przemieszczeń.
    std::string displacementMap;
    //! Tekstura dla mapy wypukłości/normalnych.
    std::string bumpMap;
    //! Tekstura odbić.
    std::string reflectionMap;
    //! Model oświetlenia.
    int illumination;

    //! Chropowatość (izo- lub anizotropiczna).
    D3DXVECTOR2 roughness;
    //! Skala nanoszenia przemieszczeń.
    float displacementScale;
    //! Domyślny współczynnik kafelkowania.
    int tessellationFactor;
  };

  //! Typ słownika materiałów.
  typedef std::map<std::string, MtlMaterial> MaterialsDict;
  //! Materiały.
  MaterialsDict materials;
};
```

Zadanie odczytu sprowadza się więc do odpowiedniego wypełnienia słownika `MtlData::materials`, po jednym wpisie dla każdego wystąpienia polecenia `newmtl`. Kwestią, która może tu budzić zastanowienie, jest opcjonalność każdego z atrybutów materiału. Pociąga to za sobą konieczność albo rozpoznawania braku wystąpienia poleceń, albo zdefiniowana domyślnych wartości. Ponieważ rozstrzygnięcie tej kwestii jest problematyczne, w omawianym przykładzie skalary i wektory mają przy tworzeniu materiału (polecenie `newmtl` nadawane pewne typowe stany), natomiast łańcuchy reprezentujące nazwy tekstur zostają wyzerowane, co wskazuje na brak ich wyspecyfikowania. Szkielet pętli odczytującej zawartej w metodzie `MtlData::Create` przedstawiony został poniżej.

```
void MtlData::Create( std::istream& input, MtlData& data ) {
  MtlMaterial defaultMaterial = { /* ... */ };
  MtlMaterial current;
  bool materialStarted = false;

  // pętla odczytująca
  std::string command;
  while (input >> command) {
    if ( command == "newmtl" ) {
      // skopiowanie poprzedniego materiału do słownika
      if ( materialStarted ) {
        data.materials[current.name] = current;
      }
      // wyzerowanie nowego materiału i wczytanie nazwy
      materialStarted = true;
      current = defaultMaterial;
      input >> current.name;
    } else if ( materialStarted ) {
      ...
    }

    // pominięcie linii do końca
    input.ignore(INT_MAX, '\n');
  }

  // jeżeli materiał był aktywny, trzeba go skopiować do słownika
  if ( materialStarted ) {
    data.materials[current.name] = current;
  }
}
```

5.5.3 Tworzenie materiału

Niestety, D3DX nie dostarcza uniwersalnego interfejsu materiału. Przed stworzeniem własnego odpowiednika należy zastanowić się, co jest potrzebne, aby typ `MtlData::MtlMaterial` mógł być zrozumiały dla Direct3D? Skalary oraz wektory mogą być bez żadnych modyfikacji przypisane zmiennym efektu. Jedynie ścieżki do tekstur wymagają, żeby przy ich pomocy wczytać teksturę, czyli żeby stworzyć instancję typu implementującego poznany już interfejs `ID3D10ShaderResourceView`. Wobec tego najszybszym sposobem jest rozszerzenie `MtlData::MtlMaterial` o brakujące zmienne oraz stworzenie metody mogącej na podstawie ścieżki odczytać teksturę.

```
//! Dane materiału.
struct Material : public MtlData::MtlMaterial {
  ID3D10ShaderResourceView* diffuseMapView;
  ID3D10ShaderResourceView* ambientMapView;
  ID3D10ShaderResourceView* specularMapView;
  ID3D10ShaderResourceView* displacementMapView;
  ID3D10ShaderResourceView* bumpMapView;
  ID3D10ShaderResourceView* reflectionMapView;
};
//! Słownik materiałów.
typedef std::map<std::string, Material> MaterialsDict;
//! Materiały.
MaterialsDict materials; ... HRESULT LoadTexture(ID3D10Device*
device, const std::string& path,
    ID3D10ShaderResourceView *& view)
{
  view = NULL;
  return path.empty() ? S_OK : D3DX10CreateShaderResourceViewFromFileA(
    device, path.c_str(), NULL, NULL, &view, NULL);
}
```

W ogólnym przypadku przed wczytaniem tekstury należałoby sprawdzić, czy nie została już wcześniej załadowana dla jakiegoś innego materiału.

Ostatnim parametrem jest referencja do wskaźnika, który należy ustawić. Należy zauważyć, iż w przypadku pustej ścieżki funkcja zwróci `S_OK`, a wskaźnik zostanie wyzerowany. Wynika to z faktu, że nie każdy materiał musi dostarczać tekstury każdego typu (i tak w praktyce właśnie jest). Jeżeli jednak podana zostanie ścieżka, a odczyt się nie powiedzie, zostanie zwrócony błąd za wywołaniem funkcji D3DX. Wobec powyższego kodu przetłumaczenie struktury `MtlData` do słownika `materials` wyglądać może następująco:

```
MtlData data; ...
// przetworzenie danych wczytanych z pliku .mtl
for ( MtlData::MaterialsDict::iterator it = data.materials.begin();
  it != data.materials.end(); ++it) {
  // skopiowanie wszystkich danych z oryginalnego materiału
  Material material;
  reinterpret_cast<MtlData::MtlMaterial&>(material) = it->second;
  // wczytanie tekstur
  material.diffuseMapView = material.ambientMapView = material.
      specularMapView =
    material.displacementMapView = material.bumpMapView = material.
        reflectionMapView = NULL;
  CHECK_HR(LoadTexture(device, material.diffuseMap,
    material.diffuseMapView));
  CHECK_HR(LoadTexture(device, material.ambientMap,
    material.ambientMapView));
  CHECK_HR(LoadTexture(device, material.specularMap,
    material.specularMapView));
  CHECK_HR(LoadTexture(device, material.displacementMap,
    material.displacementMapView));
  CHECK_HR(LoadTexture(device, material.bumpMap,
    material.bumpMapView));
  CHECK_HR(LoadTexture(device, material.reflectionMap,
    material.reflectionMapView));
  materials[material.name] = material;
}
```

Proces ładowania tekstur można by uczynić bardziej skalowalnym przez odpowiednie stablicowanie zmiennych reprezentujących nazwy oraz widoki. Przykładowo, widoki w typie `Material` mogą być złączone w unię:

```
union {
  struct {
    ID3D10ShaderResourceView* diffuseMapView;
    ID3D10ShaderResourceView* ambientMapView;
    ID3D10ShaderResourceView* specularMapView;
    ID3D10ShaderResourceView* displacementMapView;
    ID3D10ShaderResourceView* bumpMapView;
    ID3D10ShaderResourceView* reflectionMapView;
  }
  ID3D10ShaderResourceView* views[6];
};
```

Gdyby wykonać analogiczną czynność dla zmiennych reprezentujących nazwy w typie `MtlData::MtlMaterial` (ponieważ typ `std::string` nie może być przechowywany w unii, wymagałoby to zamiany typu łańcuchów na tablice `char[]` o zadanym rozmiarze), wtedy przy zachowaniu jednakowej kolejności sekwencja wywołań `LoadTexture` z jawnie podanymi argumentami mogłaby być zastąpiona uniwersalną pętlą.

5.5.4 Powiązanie z efektem

Tak jak dane siatki ostatecznie zostały powiązane z buforami wierzchołków i indeksów, tak atrybuty materiału muszą zostać powiązane ze zmiennymi efektu. W D3DX nie istnieje niestety odpowiednik `ID3DX10Mesh` dla materiałów, dlatego też trzeba operować na poziomie poszczególnych zmiennych. Istotnymi problemami w implementacji są następujące fakty:

- Nazwy zmiennych efektów nie są znane na etapie kompilacji.
- Materiał może mieć zdefiniowane tylko wybrane atrybuty.
- Efekt może zawierać zmienne reprezentujące tylko wybrane atrybuty.

Najbardziej elastycznym rozwiązaniem byłoby dodanie mechanizmu metadanych do efektu, który dostarczałby informacji o nazwach i obecności zmiennych. Pewną namiastką tej koncepcji są semantyki. Dzięki nim można oznaczać, a później, już na poziomie aplikacji, rozpoznawać zmienne niezależnie od ich identyfikatorów. Zamiast w ten sposób komplikować logikę programu można również postąpić zupełnie na odwrót, czyli zmniejszyć stopień dowolności w definiowaniu efektu poprzez stosowanie pewnej ustalonej konwencji nazewnictwa. Podejściem, które łączy zalety obu rozwiązań, jest dodanie efektu jako jednej z własności materiału. Ponieważ oba obiekty byłyby ze sobą w widoczny sposób powiązane, dopilnowanie zbieżności nazw i wystąpień mogłoby być realizowane nawet w sposób programowy, a konwencje stosowane w jednej parze materiał-efekt nie wpływałyby na pozostałe. Takie rozwiązanie można zaobserwować w wysokopoziomowych środowiskach do tworzenia gier.

Specyfikacja plików MTL, która nie przewiduje wskazywania plików efektów oraz sam charakter przykładów powodują, że najłatwiejszą i najbardziej przejrzystą metodą wiązania atrybutów materiałów ze zmiennymi jest ustalenie konwencji nazewnictwa zmiennych. Zaproponowana reguła ma postać:

```
<typ> mat_<nazwa ze struktury MtlData::MtlMaterial>;
```

Kwestią, która zostanie później poruszona, jest obecność bądź nie konkretnych zmiennych – nie każdy efekt może wspierać wielokrotne teksturowanie, w szczególności rzadko stosowaną teksturą dla światła otoczenia. Przyjmując, że nie wszystkie atrybuty materiału muszą znaleźć odwzorowanie w efekcie, przykładowy blok definicji zmiennych powinien wyglądać następująco:

```
cbuffer ChangesPerMaterial {
  float3 mat_diffuseColor;
  float mat_dissolve;
  bool mat_diffuseMapEnabled;
} Texture2D mat_diffuseMap;
```

Zastanowienie może budzić obecność zmiennej `mat_diffuseMapEnabled`. Wracając do przykładu tekstury światła otoczenia – nie wszystkie materiały muszą ją zawierać. Co w sytuacji, kiedy efekt zakłada, że taka tekstura jest obecna? Bez zareagowania na tę sytuację odczytana wartość będzie nieprawidłowa. Ponieważ obiekty tekstur nie mają cech arytmetyki wskaźnikowej (nie można ich porównać do zera, żeby sprawdzić, czy są ustawione), konieczne jest wprowadzenie zmiennej logicznej informującej o dostępności zasobu. Dzięki temu możliwe jest rozgałęzianie jednostek cieniowania i zapobieżenie nieprawidłowemu odczytowi. Podobny problem nie występuje w przypadku skalarów i wektorów, o ile zmiennym nada się takie wartości, by nie wpływały na uzyskiwany efekt. Przykładowo, jeżeli kolory wierzchołków, materiału i tekstury są mnożone w celu uzyskania końcowego rezultatu, wartością „neutralną" będzie `(1;1;1)`.

> W praktyce dodawanie zmiennych logicznych do jednostek cieniowania nie jest dobrym pomysłem – wydajność mierzy się tam w ilości instrukcji. Operacja testu logicznego oraz nieużywane rozgałęzienie kodu to „stracone" instrukcje z punktu widzenia efektu końcowego. Lepszym rozwiązaniem jest stworzenie kilku technik, z których każda uwzględnia inną kombinację dostępności zasobów. Dzięki temu test logiczny wykonywany byłby przez program jednorazowo na ramkę, a nie dla każdego wierzchołka/prymitywu/piksela (w zależności od jednostki cieniowania). Takie podejście utrudnia zarządzenie i rozwijanie kodu, więc na potrzeby przykładów stosowane jest gorsze, ale bardziej przejrzyste rozwiązanie.

Do wiązania atrybutów materiału ze zmiennymi efektu służą pochodne interfejsu `ID3D10EffectVariable`. Przykładowy blok deklaracji kilku takich obiektów przedstawiony jest poniżej. Jak widać, ich nazwy to połączenie nazwy oryginalnego atrybutu oraz przyrostka `Variable`.

```
ID3D10EffectVectorVariable* diffuseColorVariable;
ID3D10EffectScalarVariable* dissolveVariable;
ID3D10EffectScalarVariable* diffuseMapEnabledVariable;
ID3D10EffectShaderResourceVariable* diffuseMapVariable;
```

Zmienne z efektu pobiera się metodą `ID3D10Effect::GetVariableByName` (lub innymi odpowiednikami). Co warte nadmienienia - nigdy nie zwraca ona wartości NULL. Zamiast tego na otrzymanym obiekcie należy wywołać metodę `ID3D10EffectVariable::IsValid`, która sprawdza poprawność. Ponieważ tej informacji nie widać w inspektorze podczas debugowania, najlepiej taki wskaźnik po prostu wyzerować, jeżeli wystąpi błąd. Cały proces można zautomatyzować, o czym świadczy poniższy kod:

```
//! Przypisuje zmienną o zadanej nazwie do zadanego wskaźnika.
//! \param effect Efekt.
//! \param name Nazwa zmiennej.
//! \param variable Wskaźnik docelowy. Zostanie wyzerowany przy błędzie.
template <class T> void BindVariable(ID3D10Effect* effect, const
char* name, T& variable) {
  // wywołanie odpowiedniej wersji metody CastVariable
  CastVariable( effect->GetVariableByName(name), variable );
  if ( !variable->IsValid() ) {
    variable = NULL;
  }
}
//! Pomocnicza funkcja działająca w połączeniu z BindVariable.
void CastVariable(ID3D10EffectVariable* generic,
ID3D10EffectScalarVariable*& variable) {
  variable = generic->AsScalar();
}
//! Pomocnicza funkcja działająca w połączeniu z BindVariable.
void CastVariable(ID3D10EffectVariable* generic,
ID3D10EffectVectorVariable*& variable) {
  variable = generic->AsVector();
}
//! Pomocnicza funkcja działająca w połączeniu z BindVariable.
void CastVariable(ID3D10EffectVariable* generic,
ID3D10EffectShaderResourceVariable*& variable) {
  variable = generic->AsShaderResource();
} ... BindVariable(effect, "mat_diffuseColor",
diffuseColorVariable); BindVariable(effect, "mat_dissolve",
dissolveVariable); BindVariable(effect, "mat_diffuseMapEnabled",
diffuseMapEnabledVariable); BindVariable(effect, "mat_diffuseMap",
diffuseMapVariable);
```

5.5.5 Zastosowanie materiału

Materiał to w rzeczywistości część stanu potoku renderowania – należy więc go ustawiać przed odrysowaniem obiektu z niego korzystającego. Przy modyfikacji każdej ze zmiennych efektu konieczne jest sprawdzenie, czy wskaźnik na nią nie jest zerowy oraz ewentualne zbadanie rezultatu ustawiania wartości. Wraz ze wzrostem liczby zmiennych szybko rosłaby też ilość kodu, czyniąc go mniej przejrzystym. Można więc, podobnie jak w przypadku wiązania metodą `BindVariable`, go uprościć, stosując odpowiednie przeładowania. Tutaj odpowiednikami `CastVariable` z poprzedniego rozdziału są różne wersje `SetVariable`. Poniżej zaprezentowano szkielet metody `Apply` wybierającej

materiał o zadanej nazwie oraz dwa przeładowania `SetVariable` (pozostałe są równie trywialne).

```
HRESULT SetVariable( ID3D10EffectVectorVariable* variable,
D3DXVECTOR3 value ) {
  return variable ? variable->SetFloatVector(value) : S_OK;
} HRESULT SetVariable( ID3D10EffectScalarVariable* variable, bool
value ) {
  return variable ? variable->SetBool( value ? TRUE : FALSE ) : S_OK;
} ... HRESULT Apply( const std::string& name ) {
  HRESULT hr;
  // czy dany materiał istnieje?
  std::map<std::string, Material>::iterator found = materials.find(name);
  if ( found == materials.end() ) {
    return E_FAIL;
  }

  // ustawienie zmiennych efektu
  Material& material = found->second;
  CHECK_HR(SetVariable(diffuseColorVariable, material.diffuseColor));
  CHECK_HR(SetVariable(dissolveVariable, material.dissolve));
  CHECK_HR(SetVariable(diffuseMapEnabledVariable, material.diffuseMapView
      != NULL));
  CHECK_HR(SetVariable(diffuseMapVariable, material.diffuseMapView));
  /* ... */
  return S_OK;
}
```

Jak ustawić zmienną reprezentującą teksturę, jeżeli dla danego materiału jej nie wczytano? Można przekazać `NULL` ale, niestety, w znacznym stopniu utrudniona zostanie diagnostyka. Otóż Direct3D w trybie debug do konsoli wypisuje komunikaty i ostrzeżenia, które bardzo pomagają w przypadku zaistnienia jakiegoś błędu. Wyzerowanie zasobu, przy pozostawieniu instrukcji próbkowania w jednostce cieniowania spowoduje całkowite zaciemnienie tego źródła informacji, gdyż co ramkę będzie tam wypisywana wiadomość o tym, że może nastąpić nieprawidłowy odczyt. Zastosowanie rozgałęzień dynamicznych, jak w zaproponowanym rozwiązaniu, nie rozwiązuje tego problemu, gdyż z punktu widzenia Direct3D wprowadzone zmienne logiczne nie muszą być powiązane z obecnością tekstury. Wobec tego, aby uniknąć tych problemów, w kolejnych przykładach nieobecnym teksturom przypisywany jest wskaźnik na teksturę o rozmiarze 1×1.

Ostatnim krokiem jest podłączenie materiału do funkcji rysującej. Trzeba tutaj pamiętać, że pętla do tej pory podporządkowana technice oraz przebiegom musi uwzględniać nadrzędność materiału. Przedstawiony wyżej kod zaimplementowano w klasie `MtlMaterials`, którą w połączeniu z poprzednio zdefiniowanym typem `ObjMesh` można wykorzystać w następujący sposób:

```
// zmienne globalne
ObjMesh * g_mesh; MtlMaterials * g_materials; ...
D3D10_TECHNIQUE_DESC techniqueDesc;
CHECK_HR(technique->GetDesc(&techniqueDesc)); for (UINT subset = 0;
subset < mesh->GetSubsetsCount(); ++subset) {
```

```
  // pobieranie materiału
  MtlMaterials::Material material;
  if (SUCCEEDED(g_materials->GetMaterial(g_mesh->GetSubsetMaterial(subset),
        material))) {
    // zastosowanie materiału
    g_materials->Apply(material);
    for(UINT i = 0; i < techniqueDesc.Passes; ++i) {
      // nadanie stanu zdefiniowanego w przebiegu
      CHECK_HR(g_technique->GetPassByIndex(i)->Apply(0));
      // odrysowanie
      g_mesh->GetMesh()->DrawSubset(subset);
    }
  }
}
```

Metoda `MtlMaterials::GetSubsetMaterial` wypełnia strukturę informacjami o materiale, o zadanym id, zwracając przy tym `S_OK`, jeżeli nie wystąpiły żadne błędy. Trzeba pamiętać, że powyższy przykład jest zdegenerowany przez założenie występowania tylko jednej techniki (pod wskaźnikiem `g_technique`). W praktyce wybór techniki następuje na podstawie informacji o bieżącym materiale.

5.6 Koncepcja uniwersalnego efektu

Ponieważ z założenia obsługiwane będą modele w formacie OBJ, można z góry przyjąć pewny format wierzchołka oraz przygotować bazowe jednostki cieniowania, które będzie można rozwijać w kolejnych przykładach. Należy jeszcze raz podkreślić, że zaprezentowany kod będzie zazwyczaj daleki od optymalności, ale nie na poziomie kodu, lecz organizacji. Dzięki temu poświęceniu uzyskano większą skalowalność oraz prostotę jednostek. Aby uniknąć niejasności, jawne przykłady nieoptymalności zazwyczaj będą w widoczny sposób komentowane.

Poza zmiennymi definiowanymi przez materiał, bazowy efekt składa się z następujących komponentów. W podstawowej postaci jednostka cieniowania wierzchołków wykorzystuje wyłącznie informacje o kolorze dla światła rozproszonego (w praktyce pełni ona rolę „domyślnego” koloru).

```
//! Sampler tekstur.
SamplerState linearSampler {
    Filter = MIN_MAG_MIP_LINEAR;
};

//! Dane wierzchołka.
struct VertexInput {
  float3 position : POSITION;
  float2 texcoords : TEXCOORD;
  float3 normal : NORMAL;
};

//! Dane zwracane przez jednostkę wierzchołków.
struct VertexOutput {
  float4 clipPosition : SV_POSITION;
  float2 texcoords : TEXCOORD;
};
```

```
//! Bufor zmieniajacy sie raz na obiekt
cbuffer ChangesPerObject {
  //! Macierz swiat-widok-projekcja.
  matrix g_worldViewProjection;
}

//! Uniwersalny shader wierzchołków.
VertexOutput VSGeneric(VertexInput input) {
  VertexOutput output;
  output.clipPosition = mul(float4(input.position, 1),
      g_worldViewProjection);
  output.texcoords = input.texcoords;
  return output;
}

//! Uniwersalny shader pikseli.
float4 PSGeneric(VertexOutput input) : SV_Target {
  // kolor końcowy
  float4 result = float4(0,0,0,mat_dissolve);

  // kolor dla światła rozproszonego jako kolor bazowy
  float3 diffuse = mat_diffuseColor;
  [branch]
  if ( mat_diffuseMapEnabled ) {
    // mapa dla światła rozproszonego jako jedyna może dostarczać alfę
    float4 color = mat_diffuseMap.Sample(linearSampler, input.texcoords);
    diffuse *= color.rgb;
    result.a *= color.a;
  }

  // nadanie koloru pikselowi
  result.xyz = diffuse;
  return result;
}

// Kompilacja shaderów
VertexShader vsGeneric = CompileShader( vs_4_0, VSGeneric() );
PixelShader psGeneric = CompileShader( ps_4_0, PSGeneric() );

//! Technika ogólna, zdolna odrysować wszystko.
technique10 Generic {
  pass Pass0
  {
    SetVertexShader( vsGeneric );
    SetGeometryShader( NULL );
    SetPixelShader( psGeneric );
  }
}
```

Jak widać, w jednostce cieniowania wierzchołków następuje test logiczny zmiennej materiału. Optymalnym rozwiązaniem byłoby albo stworzenie dwóch osobnych jednostek, albo też dodanie parametru z modyfikatorem `uniform`. W obu przypadkach wymagałoby to jednak stworzenia dodatkowej techniki, co komplikuje logikę po stronie aplikacji.

Do sterowania kompilacją rozgałęzień służą atrybuty `[branch]` oraz `[flatten]`. W zależności od ich obecności lub na podstawie rozstrzygnięcia kompilatora rozgałęzienia mogą być traktowane jako:

- faktyczne odnogi programu, gdzie odpowiedni kod wykonuje się dopiero po sprawdzeniu warunku (atrybut `branch`),
- jeden ciąg instrukcji, z ustaleniem ostatecznego rezultatu na podstawie operacji logicznych (atrybut `flatten`).

Dobór atrybutów powinien być wypadkową doświadczenia programisty i sumiennie dokonywanego sprawdzenia rezultatów kompilacji. Przykładowo, poniżej przedstawione są rezultaty kompilacji prostego kodu dla różnych atrybutów [13].

```
float4 main(float4 t: TEXCOORD0) : SV_Target {
  if (t.x > t.y)
    return t.xyzw;
  else
    return t.wzyx;
}
```

brak lub `[branch]`	`[flatten]`

`lt r0.x, v0.y, v0.x if_nz r0.x`
`  mov o0.xyzw, v0.xyzw`
`  ret`
`else`
`  mov o0.xyzw, v0.wzyx`
`  ret`
`endif`
	`lt r0.x, v0.y, v0.x movc o0.xyzw, r0.xxxx, v0.xyzw, v0.wzyx ret`

Odpowiednikami omówionych atrybutów dla pętli są `[unroll]` i `[loop]`.

6

Oświetlenie

Obok teksturowania oświetlenie jest głównym czynnikiem wprowadzającym pierwiastek realizmu do generowanych scen. Niestety, przy obecnym poziomie zaawansowania technicznego fizycznie poprawne odwzorowanie tego zjawiska jest niemożliwe dla aplikacji mających działać w czasie rzeczywistym. Podobnie jak to jest z innymi problemami w grafice trójwymiarowej, niemożność ta maskowana jest przez pewne proste, lecz pomysłowe metody imitacji zachowania się światła. Wspólną cechą modelu fizycznego i uproszczonego jest jednak fakt, iż o charakterze oświetlenia decydują relacje pomiędzy źródłem a właściwościami powierzchni obiektów – materiałami.

Rysunek 6.1. Ta sama siatka odrysowana bez i z uwzględnieniem światła

Prezentowane rozważania dotyczą tzw. lokalnego modelu oświetlenia (ang. *local illumination model*). Oznacza to, że każdy obiekt oświetlany jest niezależnie, przez co uwzględnia wyłącznie promienie pochodzące bezpośrednio od

pierwotnych źródeł. Jest to znaczne uproszczenie w stosunku do modelu fizycznego, gdzie promień światła natrafiający na przeszkodę zostaje w części zaabsorbowany i w części odbity, przez co może potencjalne oświetlić wiele obiektów. Na lepszą imitację tego zjawiska pozwala globalny model oświetlenia (ang. *global illumination model*), który uwzględnia zawartość całej sceny, przez co możliwe jest uwzględnienie wielu wtórnych źródeł światła. Obecnie globalne modele nie są wykorzystywane w czasie rzeczywistym ze względu na dużo większą złożoność obliczeniową w stosunku do modelu lokalnego. Dodatkowo strumieniowa specyfika przetwarzania w jednostkach cieniowania sprzyja zastosowaniu uproszczonego rozwiązania.

6.1 Źródła światła

W grafice komputerowej źródło określa charakter rozchodzenia się promieni w przestrzeni. Dodatkowo, tak jak w modelu fizycznym, źródło może generować światło o różnych barwach. Poniżej opisano trzy najczęściej spotykane typy źródeł światła: punktowe, równoległe (kierunkowe) i skierowane.

Punktowe

Źródła światła punktowego (ang. *point light*) emitują promienie równomiernie we wszystkich kierunkach. Formalizując, dla każdego punktu w przestrzeni P istnieje promień wychodzący ze źródła S reprezentowany wektorem $\vec{SP}$. W dalszych rozważaniach wektor światła będzie oznaczany jako $\vec{l}$.

Zgodnie z regułami fizyki, natężenie światła powinno maleć wraz z odległością od źródła. Najbardziej zgodne z fizycznym modelem jest osłabienie (ang. *attenuation*) odwrotnie proporcjonalne do kwadratu odległości:

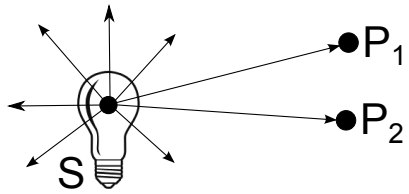
$$f_{att} = \frac{1}{|\vec{l}|^2}$$

Okazuje się jednak, iż w grafice komputerowej nie daje to zadowalających rezultatów. Z tego powodu rezygnuje się z pełnej zgodności z modelem fizycznym na rzecz większej kontroli nad tym aspektem światła. Typowym rozwiązaniem jest obliczanie osłabienia zgodnie ze wzorem:

$$f_{att} = min\left(\frac{1}{a_0 + a_1|\vec{l}| + a_2|\vec{l}|^2}, 1\right) \tag{6.1}$$

gdzie:

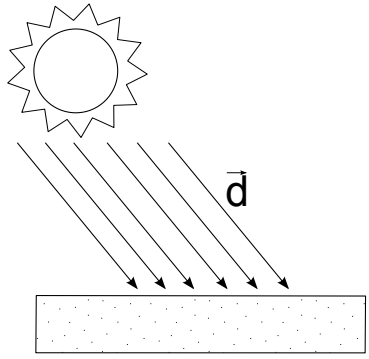
- f_{att} - współczynnik osłabienia
- a_0 - stała osłabiania
- a_1 - liniowy współczynnik osłabiania
- a_2 - kwadratowy współczynnik osłabiania



Rysunek 6.2. Punktowe źródło światła

6.1.1 Równoległe

Światło równoległe (ang. *parallel light*) lub kierunkowe (ang. *directional light*) stanowi przybliżenie bardzo odległych punktowych źródeł światła. Odległość ta sprawia, iż z punktu widzenia powierzchni padające promienie są w przybliżeniu równoległe. Przykładem może być Słońce, którego światło, z racji oddalenia źródła od Ziemi o setki milionów kilometrów, taktowane jest jako równoległe.



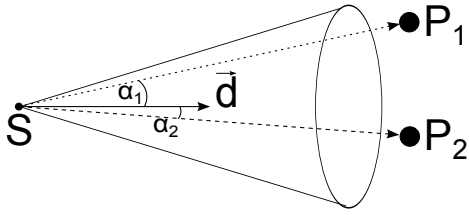
Rysunek 6.3. Równoległe źródło światła

Właściwościami równoległego źródła światła jest brak określonej pozycji w przestrzeni oraz ściśle określony kierunek padania $\vec{s}$, równy dla wszystkich emitowanych promieni. Dodatkowo, z racji założenia dużych odległości od źródła, nie stosuje się osłabienia światła wraz z odległością, gdyż rozmiar sceny jest zazwyczaj o wiele rzędów mniejszy od dystansu, który promienie musiałyby przebyć, aby osiągnąć równoległość.

6.1.2 Skupione

Źródła światła skupionego (ang. *spotlight*) można przedstawić jako zdegenerowane źródło punktowe, które emisję prowadzi jedynie w zakresie pewnego

stożka. Skutkiem tego jest prawidłowość, która mówiła o promieniu biegnącym w linii prostej do każdego punktu w przestrzeni, odnosi się to wyłącznie do promieni, których kąt emisji jest mniejszy od kąta rozwarcia. Światła skupione charakteryzują się również tym, że największa energia emitowana jest w kierunku prostopadłym do hipotetycznej podstawy stożka (rysunek 6.4).



Rysunek 6.4. Skupione źródło światła. Promień $S\vec{P}_1$ niesie z sobą mniejszą energię wskutek większego nachylenia wobec wektora kierunkowego światła $\vec{s}$, co zostało oznaczone bardziej przerywaną linią w stosunku do promienia $S\vec{P}_2$

W praktyce przyjmuje się, że kąt emisji zawsze wynosi 180°. Efekt stożka uzyskiwany jest za pomocą osłabiania intensywności wykładniczą funkcją cosinusa kąta między wektorem kierunkowym a promieniem. Im większy wykładnik, tym większy stopień skupienia światła, a więc tym słabsze są promienie emitowane dla dużych kątów.

$$f_{spec} = max(cos(\alpha)^s, 0) \tag{6.2}$$

gdzie:

f_{spec} - współczynnik osłabienia światła stożkowego
α - kąt między promieniem a wektorem kierunkowym
s - wykładnik

Ponieważ charakterystyka źródła skupionego jest zbliżona do punktowego, ostateczny stopień osłabienia promienia powinien uwzględniać również współczynnik wynikający z odległości (6.1).

6.2 Składowe światła

Wszystkie obiekty w świecie rzeczywistym są widzialne dlatego, że odbiło się od nich światło, które następnie trafiło do siatkówki oka. Dysponując zaś odpowiednią mocą obliczeniową, można by dokładnie prześledzić drogę każdego promienia od źródła do obserwatora zgodnie z regułami fizyki, jednak obecnie jest to niewykonalne dla aplikacji czasu rzeczywistego. Zamiast tego w grafice komputerowej wyróżnia się tradycyjnie trzy tzw. składowe światła, które mogą trafić do obserwatora. To dzięki ich kombinacji oraz sposobowi wyznaczania możliwe jest stosunkowo wierne odzwierciedlenie odbicia światła od wielu materiałów.

6.2.1 Światło otoczenia

Jak zostało wspomniane w rozdziale 6, prezentowany model światła nie uwzględnia wtórnych źródeł. Uproszczenie to urasta do rangi problemu, gdy przychodzi do renderowania np. zamkniętych pomieszczeń, gdzie z powodu niewielkiej odległości pomiędzy ścianami promienie mogą być wielokrotnie odbijane i rozświetlać tym samym pierwotnie zacienione powierzchnie. Najprostszym rozwiązaniem tego problemu jest wprowadzenie składowej światła otoczenia (ang. *ambient light*), która oświetla wszystkie obiekty z jednakowym natężeniem niezależnie od ich położenia. Jest to więc symulacja sytuacji, gdy światło odbija się od tak wielu powierzchni wiele razy, że działa jednakowo na całej scenie.

Zbyt duża jasność otoczenia prowadzi do zmniejszenia widocznego wpływu pozostałych składowych, na czym traci realizm sceny. Należy mieć to szczególnie na uwadze, gdy światło otoczenia traktuje się jako jedną z własności źródła. Wynika to z faktu, że gdy na scenie znajduje się wiele świateł, wówczas ich składowe się sumują. Aby uniknąć problemu akumulacji jasności otoczenia, często usuwa się reprezentujący ją atrybut ze źródeł, a zamiennie stosuje się pojedynczy parametr dla całej sceny.

6.2.2 Odbicie zwierciadlane

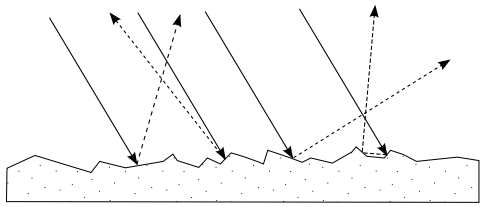
Fizyka uczy, że zgodnie z tak zwanym prawem odbicia, kąty padania i odbicia, definiowane jako kąty pomiędzy promieniem padającym i odbitym a wektorem normalnym, są równe. Odbicie zgodne z tą regułą nosi nazwę zwierciadlanego (ang. *specular*) i dotyczy płaskich powierzchni zarówno w mikro-, jak i w makroskali. Jego występowanie można zaobserwować na przykładzie luster bądź innych bardzo równych i wypolerowanych ciał. Odbicie to charakteryzuje zasada, że jeżeli obserwator nie będzie przecinał kierunku promieni odbitych, światło nie dotrze do niego tą drogą.

W kontekście lokalnego modelu oświetlenia przez modelowanie odbicia światła rozumie się nie sam proces analitycznego wyznaczania i śledzenia nowego kierunku promieni, lecz określenie, jaka porcja energii ze źródła pierwotnego dotrze do obserwatora po natrafieniu na obiekt. W przypadku powierzchni zwierciadlanych stosuje się więc bezpośrednio bądź pośrednio prawo odbicia. Można więc uogólnić, że w grafice komputerowej połysk jest tym większy, im bardziej wektor widoku (od punktu na powierzchni do obserwatora) jest zbliżony do wektora odbicia.

6.2.3 Odbicie rozproszone

Gdy powierzchnia jest chropowata, zachodzi zjawisko rozproszenia światła. Wynika ono z faktu, że nierówną powierzchnię tworzą losowo rozmieszczone płaszczyzny i krzywizny. Jeśli do każdej z nich zastosować prawo odbicia, okazuje się, że w skali makro światło pierwotnie padające w jednym wybranym

kierunku jest rozpraszane w przybliżeniu równomiernie we wszystkie strony. W związku z tym do każdego obserwatora, który widzi oświetloną powierzchnię, dotrze taka sama wiązka energii ze źródła pierwotnego.



Rysunek 6.5. Rozproszenie światła przy padaniu na chropowatą powierzchnię

W rzeczywistości większość dostrzeganego światła odbitego ma charakter rozproszony. W aplikacjach 3D ta składowa również dominuje do tego stopnia, że to, co kiedyś było nazywane po prostu teksturą obiektu, obecnie nazywane jest często mapą dyfuzji (ang. *diffuse map*).

6.3 Wyznaczenie oświetlenia

Zakładając, że światło jest podzielone na pewną liczbę pasm, to rejestrowane przez obserwatora natężenie składowej λ wyrazić można jako:

$$I_\lambda = R_a I_{a\lambda} C_{a\lambda} + f(R_d I_{d\lambda} C_{d\lambda} + R_s I_{s\lambda} C_{s\lambda})$$

$$f = \begin{cases} 1 & \text{dla światła kierunkowego} \\ f_{att} & \text{dla światła punktowego} \\ f_{att} f_{spot} & \text{dla światła skupionego} \end{cases}$$

gdzie:

$I_{a\lambda}$ - natężenie światła otoczenia
$I_{d\lambda}$ - natężenie światła rozproszonego
$I_{s\lambda}$ - natężenie światła zwierciadlanego
$C_{a\lambda}$ - współczynnik odbicia światła otoczenia
$C_{d\lambda}$ - współczynnik odbicia światła rozproszonego
$C_{s\lambda}$ - współczynnik odbicia światła zwierciadlanego
R_a - procent energii otoczenia odbitej w kierunku obserwatora
R_d - procent energii rozproszonej odbitej w kierunku obserwatora
R_s - procent energii zwierciadlanej odbitej w kierunku obserwatora

Składowe identyfikowane są tu na podstawie pierwszej litery ich angielskich nazw, a więc wynoszą a dla światła otoczenia, d dla odbicia rozproszonego i s dla odbicia zwierciadlanego. W dalszej części rozdziału światło jest podzielone na trzy pasma: czerwień (R), zieleń (G) i niebieskie (B). Dzięki czemu jego

opis staje się zgodny z reprezentacją kolorów pikseli w Direct3D. Dodatkowo równanie oraz jego współczynniki nabierają bardziej intuicyjnego charakteru:

$$I = I_a \otimes C_a + f(R_d I_d \otimes C_d + R_s I_s \otimes C_s) \tag{6.3}$$

$$f = \begin{cases} 1 & \text{dla światła kierunkowego} \\ f_{att} & \text{dla światła punktowego} \\ f_{att} f_{spot} & \text{dla światła skupionego} \end{cases}$$

gdzie:

I_a, I_d, I_s - kolory składowych światła
C_a, C_d, C_s - kolory materiału dla zadanych składowych
R_a, R_d, R_s - procent energii składowych trafiający do obserwatora

Warto zauważyć, że kolory materiałów pełnią rolę taką, jak w rzeczywistości. Przykładowo, jeżeli materiał ma czarny kolor $(0; 0; 0)$, wykonanie operacji $\otimes$ na kolorze światła spowoduje pochłonięcie całej jego energii. Z kolei kolor idealnie biały $(1; 1; 1)$ odbije całą wiązkę padającego światła, ale niekoniecznie w kierunku obserwatora.

Jedynymi niewiadomymi w równaniu 6.3 są współczynniki określające procent energii trafiającej do obserwatora. Kolejne punkty omawiają zaledwie kilka spośród popularnych modeli oświetlenia, czyli algorytmów modelujących odbicia. Zeby zachować spójność, wszystkie zaprezentowane w języku HLSL funkcje zwracają zawsze trójwymiarowy wektor określony następująco:

$$R = [\, R_a \; R_d \; R_s \,]$$

Ponieważ we wszystkich przedstawionych modelach $R_a = 1$, składowa ta jest pominięta w przedstawionych wzorach.

6.3.1 Model Lamberta

Model Lamberta, charakteryzujący się niską złożonością obliczeniową oraz prostotą w implementacji, jest jednym z najczęściej używanych modeli oświetlenia. Zakłada on, że odbite światło jest rozpraszane równomiernie we wszystkich kierunkach, a jego intensywność zależy od kąta padania, przy czym maksimum osiąga przy kącie prostym. Innymi słowy każdy obserwator, niezależnie od pozycji, widzi obiekt oświetlony w ten sam sposób. Ta specyfika powoduje, że model Lamberta dobrze odzwierciedla gładkie, matowe powierzchnie. Dodatkowo, jak się później okaże, Model Lamberta stanowi podstawę wielu innych algorytmów, w tym również opisujących materiały połyskliwe.

$$R_d = max(\vec{l} \cdot \vec{n}, 0)$$
$$R_s = 0$$

gdzie:

$\vec{n}$ - wektor normalny powierzchni oświetlanej
$\vec{l}$ - wektor kierunkowy światła

W języku HLSL zdefiniowana jest funkcja `saturate`, która dokonuje przycięcia wartości do przedziału $< 0; 1 >$. W obliczeniach oświetlenia często używa się jej zamiast funkcji *min* i *max*, gdyż przekłada się na mniejszą liczbę instrukcji maszynowych.

```
float3 LambertModel(float3 l, float3 n) {
  // swiatlo otoczenia zawsze 1, polysk zawsze 0
  return float3(1, saturate(dot(l, n)), 0);
}
```

Przy tworzeniu jednostek cieniowania zawsze warto mieć w zasięgu ręki listę funkcji wbudowanych HLSL, dostępną pod adresem `http://msdn.microsoft.com/en-us/library/ff471376.aspx`. Wiele podstawowych operacji na skalarach, wektorach i macierzach posiada już swoje implementacje i to z nich należy korzystać, ponieważ zazwyczaj są szybsze niż samodzielna realizacja.

6.3.2 Model Orena-Nayara

Niektóre materiały rozpraszają światło inaczej, niż by to wynikało z modelu Lamberta. Według Orena i Nayara[11] odpowiedzialność za to ponosi chropowatość, która powoduje, że płaskie z pozoru powierzchnie można przedstawić jako zbiór drobnych płaskich stykających się ze sobą obszarów o różnym nachyleniu. Zarówno w rzeczywistym, jak i wirtualnym świecie obszary te często są tak małe, że dysponujące skończoną czułością narzędzie rejestrujące (odpowiednio siatkówka i ramka docelowa) nie jest w stanie ich rozróżnić, w rezultacie czego powstaje uśredniony obraz. Pomimo faktu, że każdy fragment odbija światło zgodnie z prawami fizyki, różne nachylenie i kąty padania promieni dla każdego z nich powodują, że wygląd chropowatej powierzchni zmienia się w zależności od pozycji obserwatora.

Model Orana-Nayara, będący de facto generalizacją modelu Lamberta, uwzględnia to zjawisko. W wyniku jego zastosowania uzyskiwane oświetlenie jest bardziej równomierne i pozbawione gwałtownych zmian. Sama chropowatość powierzchni, a więc również kąty odbicia, określone są poprzez aproksymowany rozkład normalny o wartości średniej 0 z parametryzowanym odchyleniem standardowym σ. Co warte odnotowania, jeżeli $\sigma = 0$, to światło będzie rozpraszane zgodnie z modelem Lamberta. Oryginalny model Orena-Nayara jest skomplikowany i koncepcyjnie, i obliczeniowo. Sami autorzy zaproponowali jednak uproszczoną realizację, pomijającą niektóre aspekty wewnętrznych odbić w ramach nierówności:

$$R_d = max(0, \vec{l} \cdot \vec{n})(A + B \cdot max(0, \gamma) \cdot sin\alpha \cdot tg\beta)$$
$$R_s = 0$$

$$A = 1 - 0.5\frac{\sigma^2}{\sigma^2 + 0.33}$$
$$B = 0.45\frac{\sigma^2}{\sigma^2 + 0.09}$$
$$\gamma = (\vec{v} - \vec{n}(\vec{v} \cdot \vec{n})) \cdot (\vec{l} - \vec{n}(\vec{l} \cdot \vec{n}))$$
$$\alpha = max(acos(\vec{v} \cdot \vec{n}), acos(\vec{l} \cdot \vec{n}))$$
$$\beta = min(acos(\vec{v} \cdot \vec{n}), acos(\vec{l} \cdot \vec{n}))$$
$$\sigma \in < 0; 1 >$$

gdzie:

$\vec{n}$ - wektor normalny powierzchni oświetlanej
$\vec{l}$ - wektor kierunkowy światła
$\vec{v}$ - wektor widoku (od powierzchni do obserwatora)
σ - współczynnik chropowatości

Przykładowa implementacja w HLSL może wyglądać następująco:

```
float3 OrenNayarModel(float3 l, float3 n, float3 v, float roughness)
{
  float VdotN = dot(v, n);
  float LdotN = dot(l, n);

  // wewnętrznie używany jest kwadrat chropowatości
  roughness = roughness * roughness;
  float A = 1.0f - 0.5f * (roughness / (roughness + 0.57f));
  float B = 0.45f * (roughness / (roughness + 0.09));

  // obliczenie współczynników
  float acosVdotN = acos(VdotN);
  float acosLdotN = acos(LdotN);
  float alpha = max( acosVdotN, acosLdotN );
  float beta  = min( acosVdotN, acosLdotN );
  float gamma = dot(v - n * VdotN, l - n * LdotN);

  float3 result;
  result.x = 1.0;
  result.y = max(0.0f, LdotN)*(A + B*max(0.0f, gamma)*sin(alpha)*tan(beta))
      ;
  result.z = 0.0;

  return result;
}
```

6.3.3 Model Phonga

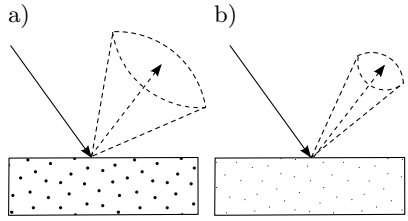
Najbardziej rozpowszechnionym i niemal kanonicznym modelem oświetlenia połyskliwego jest model opracowany przez Phonga (w 1975 roku). Jest on *de facto* rozszerzeniem modelu Lamberta o składnik, który jest odbijany zgodnie z regułami optyki, czyli pod kątem równym kątowi padania. Dodatkowo, intensywność połysku zależy od kąta obserwacji powierzchni, uzyskując maksimum dla promieni odbitych równolegle do kierunku widzenia obserwatora. Sama połyskliwość dla materiału regulowana jest przez wykładnik rezultatu

cosinusa kąta między wektorem odbicia a wektorem widoku – im większy połysk, tym bardziej odbicie zmierza do idealnego. Równanie modelu Phonga przedstawia się więc następująco:

$$R_d = max(\vec{l} \cdot \vec{n}, 0)$$
$$R_s = max((\vec{r} \cdot \vec{n})^g, 0)$$
$$\vec{r} = 2 * (\vec{l} \cdot \vec{n}) * \vec{n} - \vec{l}$$

gdzie:

$\vec{v}$ - wektor od powierzchni obserwowanego obiektu do kamery
$\vec{l}$ - wektor kierunkowy światła padającego na powierzchnię
$\vec{n}$ - wektor normalny do powierzchni
g - współczynnik połysku



Rysunek 6.6. Interpretacja odbicia zgodna z modelem Phonga. Materiał z a) posiada mniejszy współczynnik połyskliwości niż z b)

HLSL udostępnia funkcję `reflect`, która odbija wektor od powierzchni reprezentowanej przez przekazany wektor normalny. Dzięki temu implementacja modelu Phonga sprowadza się do wyliczenia wektora odbicia oraz przełożenia iloczynu skalarnego między nim a wektorem kierunkowym obserwatora na składową połysku.

```
float3 PhongModel(float3 l, float3 n, float3 v, int gloss) {
  // x - światło otoczenia (zawsze 1)
  float3 result = float3(1,0,0);
  // y - światło rozproszone jak w modelu Lamberta
  result.y = saturate(dot(l, n));
  // wektor odbicia światła
  float3 r = normalize(reflect(-l, n));
  // z - światło odbicia
  result.z = pow(saturate(dot(r, v)), gloss);
  return result;
}
```

6.3.4 Model Blinna-Phonga

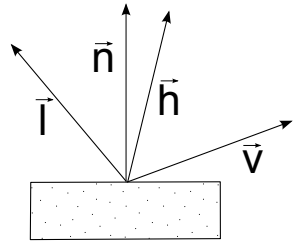
W praktyce często rezygnuje się z modelu Phonga na rzecz nieco mniej wymagającego obliczeniowo modelu Blinna-Phonga. Optymalizacja polega na re-

zygnacji z wyznaczania wektora odbicia na rzecz tzw. wektora połówkowego, będącego znormalizowaną średnią wektorów widoku i światła – obliczenia te generują mniej instrukcji niż funkcja `reflect`. Wobec tego faktu intensywność połysku zależy od kąta pomiędzy wektorem normalnym a wektorem połówkowym. Ponieważ jest to uproszczony sposób na uzyskanie informacji o tym, jak bardzo oddalony jest wektor widoku od faktycznego wektora odbicia, rezultaty nie będą identyczne z tymi uzyskanymi za pomocą modelu Phonga. Dzięki odpowiedniemu doborowi parametrów można jednak uzyskać znaczne podobieństwo. Rysunek 6.7 przedstawia zależności między omawianymi wielkościami. Równanie modelu Blinna-Phonga [4] przedstawia się następująco:

$$R_d = max(\vec{l} \cdot \vec{n}, 0)$$
$$R_s = max((\vec{n} \cdot \vec{h})^g, 0)$$
$$\vec{h} = \frac{\vec{l} + \vec{v}}{|\vec{l} + \vec{v}|}$$

gdzie:

$\vec{n}$ - wektor normalny do powierzchni
$\vec{l}$ - wektor kierunkowy światła padającego na powierzchnię
$\vec{v}$ - wektor od powierzchni obserwowanego obiektu do kamery
g - współczynnik połysku



Rysunek 6.7. Wyznaczanie wektora połówkowego $\vec{h}$

Sama implementacja jest bardzo podobna do tej z modelu Phonga. W wyniku kompilacji powstanie mniej instrukcji maszynowych, ponieważ operacja odbicia zastąpiona została prostym dodaniem wektorów.

```
float3 BlinnPhongModel(float3 l, float3 n, float3 v, int gloss) {
  // x - światło otoczenia (zawsze 1)
  float3 result = float3(1,0,0);
  // światło rozproszone jak w modelu Lamberta
  result.y = saturate(dot(l, n));
  // wektor połówkowy
  float3 h = normalize(v + l);
  // z - połysk
  result.z = pow(saturate(dot(n, h)), gloss);
  return result;
}
```

Pamiętać trzeba też o pewnej negatywnej konsekwencji używania modelu Blinna-Phonga – powierzchnie, które powinny być niewidoczne ($\vec{n} \cdot \vec{l} < 0$), mogą – w wyniku uproszczenia modelu - zostać oświetlone połyskiem. Model Phonga, z racji większej wierności prawom fizyki, jest nieco bardziej odporny na takie błędy. HLSL dostarcza funkcji `lit`, która jest de facto implementacją modelu Blinna-Phonga uzupełnioną o sprawdzanie widoczności:

$$I_s = v * max((\vec{n} \cdot \vec{h})^n, 0)$$
$$v = \begin{cases} 0 \text{ jeżeli } \vec{n} \cdot \vec{l} < 0 \\ 1 \text{ jeżeli } \vec{n} \cdot \vec{l} >= 0 \end{cases}$$

Implementacja modelu dzięki funkcji `lit` staje się trywialna:

```
float3 BlinnPhongModelLit(float3 l, float3 n, float3 v, int gloss) {
  float3 h = normalize(v + l);
  return lit(dot(l, n), dot(n, h), gloss).xyz;
}
```

Niestety, generuje ona więcej instrukcji maszynowych – nawet od implementacji modelu Phonga. Ręczne przeprowadzenie obliczeń może zmniejszyć ich ilość, ale wciąż okazuje się, że wprowadzenie tych poprawek niweluje główną zaletę algorytmu Blinna-Phonga, czyli mniejszą złożoność obliczeniową.

6.3.5 Model Warda

Dotychczas omawiane modele światła były izotropowe. W grafice komputerowej terminem tym określa się powierzchnie, których właściwości nie zmieniają się wraz z obrotem wokół własnego wektora normalnego. Ta cecha pozwalała uzyskać wiarygodny obraz gładkich lub porowatych powierzchni, w których nierówności mają charakter jednorodny (są takie same we wszystkich kierunkach). Przeciwieństwem izotropii jest anizotropia, a więc zależność od orientacji własnej powierzchni. Przykładami anizotropowych materiałów mogą być np. farba nakładana grubym pędzlem lub też zarysowana metalowa płytka. W obu przypadkach powstałe nierówności będą skierowane w większości w jednym kierunku, wzdłuż którego następowało malowanie bądź zarysowywanie. Z powodu tej niejednorodności powierzchni zmienia się zachowanie światła. I tak, jeżeli rzut wektora oświetlenia będzie równoległy do dominującego kierunku nierówności, promienie odbiją się w przybliżeniu zgodnie z regułami Phonga. Gdy jednak dokona się obrotu powierzchni o 90°i zagłębienia staną się prostopadłe, światło będzie w nie wpadać i wielokrotnie się odbijać od ich ścian.

Model Warda [17] pozwala na oddanie anizotropowych powierzchni. W odróżnieniu od poprzednio zaprezentowanych rozwiązań nie bazuje w bezpośredni sposób na fizycznych właściwościach światła takich jak symetryczność kąta odbicia. Autor zamiast tego zaproponował empiryczną aproksymację rezultatów obserwowanych w świecie rzeczywistym. Chropowatość jest więc kontrolowana przez dwa parametry, które określają stopień nierówności w dwóch

prostopadłych do siebie kierunkach. Kierunki te, z racji bycia stycznymi (ang. *tangent*) do bieżącej powierzchni nazywane są, odpowiednio, *tangent* i *bitangent*. Najłatwiejszym sposobem na ich wyznaczenie jest zastosowanie wektora odniesienia $\vec{\epsilon}$ i przeprowadzenie następujących obliczeń bezpośrednio w jednostkach cieniowania:

$$\vec{t} = \vec{n} \times \vec{\epsilon}$$
$$\vec{b} = \vec{n} \times \vec{t}$$

gdzie:

$\vec{\epsilon}$ - wektor odniesienia, np. [1 0 0]
$\vec{t}$ - tangent
$\vec{b}$ - bitangent
$\vec{n}$ - wektor normalny powierzchni

Sam Ward, oprócz wzoru będącego empiryczną aproksymacją obserwowanego zjawiska, zaproponował też nieco uproszczoną, ale mniej wymagającą obliczeniowo zależność. Zgodnie z nią, natężenie światła odbitego L_s wyrażone jest wzorem:

$$R_d = max(\vec{l} \cdot \vec{n}, 0)$$
$$R_s = e^{\beta} * \left(4 * \pi * \alpha_t * \alpha_b * \sqrt{(\vec{n} \cdot \vec{l}) * (\vec{n} \cdot \vec{v})}\right)^{-1}$$
$$\beta = -2 * \frac{\left(\frac{\vec{h} \cdot \vec{t}}{\alpha_t}\right)^2 + \left(\frac{\vec{h} \cdot \vec{b}}{\alpha_b}\right)^2}{1 + \vec{h} \cdot \vec{n}}$$

gdzie:

$\vec{t}$ - tangent
$\vec{b}$ - bitangent
$\vec{n}$ - wektor normalny powierzchni
$\vec{l}$ - wektor światła
$\vec{v}$ - wektor widoku
$\vec{h}$ - wektor połówkowy
α_t - chropowatość w kierunku określonym przez $\vec{t}$
α_b - chropowatość w kierunku określonym przez $\vec{b}$

Warto nadmienić, że jeżeli $\alpha_t = \alpha_b$, to powyższa zależność upraszcza się i opisuje powierzchnię izotropową. Przy występowaniu tego warunku uzyskiwane rezultaty są zbliżone do tych przewidzianych modelem Phonga.

```
float3 WardModel(float3 l, float3 n, float3 v, float3 t, float3 b,
float alphaT, float alphaB ) {
  // dodanie małych wartości do chropowatości, aby uniknąć dzielenia przez
      zero
  alphaT += 1e-5f;
  alphaB += 1e-5f;
  // wektor połówkowy
  float3 h = normalize(v + l);
```

```
  // obliczenie cosinusów
  float VdotN = dot( v, n );
  float LdotN = dot( l, n );
  float HdotN = dot( h, n );
  float HdotT = dot( h, t );
  float HdotB = dot( h, b );
  // obliczenie bety
  float betaT = HdotT / alphaT;
  float betaB = HdotB / alphaB;
  betaT *= betaT;
  betaB *= betaB;
  float beta = -2 * (betaT + betaB) / (1 + HdotN);
  // obliczenie s
  float den = 4 * PI * alphaT * alphaB * sqrt(LdotN * VdotN);
  float s = exp(beta) / den;
  // rezultat
  float LdotNSaturated = saturate(LdotN);
  float3 result;
  result.x = 1.0;
  result.y = LdotNSaturated;
  result.z = saturate(LdotNSaturated * s);
  return result;
}
```

6.4 Zastosowanie modeli

Wiedząc już, jakie wielkości charakteryzujące światło oraz materiał są potrzebne, można przystąpić do sporządzenia funkcji, która uwzględniając typ źródła będzie dokonywała obliczeń zgodnie z wybranym modelem. Żeby uczynić kod bardziej przenośnym i intuicyjnym, przygotowano następujące typy danych:

```
//! Struktura reprezentująca światło.
struct Light {
  int type;
  float3 position;
  float3 direction;
  float3 color;
  float3 attenuation;
  float spotPower;
};

//! Struktura z informacją o powierzchni.
struct Surface {
  int lightModel;
  float3 position;
  float3 normal;
  int gloss;
  float3 tangent;
  float3 bitangent;
  float2 roughness;
};
```

Pole `Light.color` nie jest bezpośrednio używane w obliczeniach samego natężenia, lecz zostało wprowadzone, by zamknąć wszystkie informacje o źródle w jednej strukturze. Jego wartość omówiona będzie w kolejnych punktach. Dalszy ciąg tego rozdziału będzie natomiast poświęcony implementacji następującej funkcji:

```
float3 CalculateLight(Light light, Surface surface, float3
eyePosition) {
  ...
}
```

Na podstawia sygnatury oraz użytych typów można wnioskować, iż powinna ona obliczać natężenia konkretnych składowych z uwzględnieniem specyficznych cech poszczególnych źródeł. Funkcja nie będzie korzystać również ze zmiennych materiału (przedrostek `mat_`). Dzięki tym samonarzuconym ograniczeniom kod nabiera cech uniwersalności i przenośności, gdyż bazuje jedynie na podstawie przekazanych parametrów. Jednostka wywołująca musi jedynie zapewnić parametry oraz własnoręcznie dokonać mieszania kolorów na podstawie natężeń.

Każdy model światła wymaga, oprócz wektora normalnego powierzchni, który znajduje się już w definicji typu `Surface`, wektora kierunkowego światła. Jest to wielkość zależna od typu światła, wyliczana w następujący sposób:

```
// ustalenie orientacji światła
float3 lightNormal; float lightDistance; if ( light.type ==
LIGHT_TYPE_PARARELL ) {
  // kierunek pobierany z właściwości światła; odległości nie trzeba liczyć
  lightNormal = -light.direction;
} else {
  // od światła do punktu
  lightNormal = light.position - surface.position;
  // ręczna normalizacja (zapamiętanie odległości, może być potrzebna)
  lightDistance = length(lightNormal);
  lightNormal /= lightDistance;
}
// wektor do obserwatora
float3 eyeNormal = normalize(eyePosition - surface.position);
```

Kolejnym krokiem jest zastosowanie modelu wskazanego przez pole `Surface.lightModel`. Kod reprezentujący tę czynność jest wyjątkowo mało interesujący, gdyż składa się na niego sekwencja instrukcji warunkowych. Aby zachować ciągłość funkcji, został on przytoczony poniżej.

```
// wybranie modelu światła
float3 result; if (surface.lightModel == LIGHT_MODEL_LAMBERT) {
  result = LambertModel(lightNormal, surface.normal);
} else if (surface.lightModel == LIGHT_MODEL_OREN_NAYAR) {
  result = OrenNayarModel(lightNormal, surface.normal, eyeNormal, surface.
      roughness.x);
} else if (surface.lightModel == LIGHT_MODEL_PHONG) {
  result = PhongModel(lightNormal, surface.normal, eyeNormal, surface.gloss
      );
} else if (surface.lightModel == LIGHT_MODEL_BLINN_PHONG) {
  result = BlinnPhongModel(lightNormal, surface.normal, eyeNormal, surface.
      gloss);
} else if (surface.lightModel == LIGHT_MODEL_WARD) {
  result = WardModel(lightNormal, surface.normal, eyeNormal, surface.
      tangent, surface.bitangent, surface.roughness.x, surface.roughness.y
      );
} else {
  // nieokreślony model
  result = float3(1,1,1);
}
```

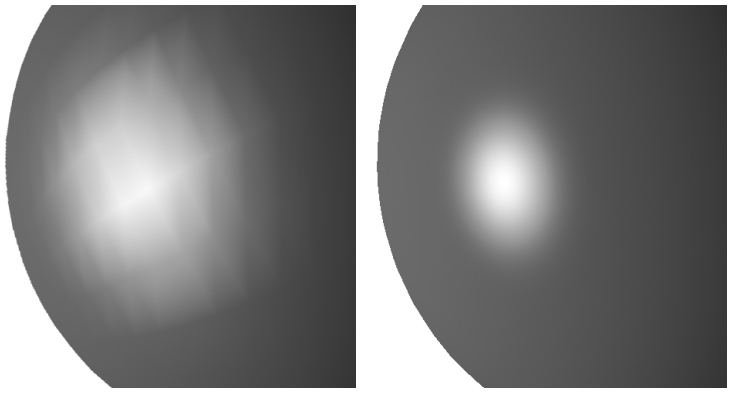
Po obliczeniu natężeń składowych światła należy jeszcze uwzględnić współczynniki osłabiania, jeżeli źródło tego wymaga. Należy przy tym pamiętać, że nie powinno się skalować jasności otoczenia.

```
// osłabienie światła w zależności od odległości
if ( light.type != LIGHT_TYPE_PARARELL ) {
  // równoległe nie jest osłabiane
  result.yz /= dot(light.attenuation,
    float3(1.0f, lightDistance, lightDistance*lightDistance));
  if ( light.type == LIGHT_TYPE_SPOT ) {
    // dodatkowe osłabienie światła skupionego
    result.yz *= pow(max(dot(-lightNormal, light.direction), 0.0f),
      light.spotPower);
  }
} return result;
```

Czy instrukcje warunkowe w jednostkach cieniowania, a w szczególności ich sekwencja, nie powinny być unikane? Zdecydowanie powinny, ponieważ drastycznie zwiększają liczbę wynikowych instrukcji. Wyjątkiem są sytuacje, gdy rozgałęzienia mogą być statycznie (na etapie kompilacji) wykluczone. W pokazanych przykładach testowane są jedynie pola `Light.type` oraz `Surface.lightModel`. Jeżeli polom tym wcześniej przypisano stałą literalną bądź zmienną oznaczoną modyfikatorem `uniform` kompilator wytnie z kodu porównania, których wynik jest znany na etapie kompilacji oraz, co za tym idzie, niedostępne rozgałęzienia. Niestety, kompilator nie może wnioskować o wartości zmiennych globalnych (z buforów stałych), więc powinno się ich unikać.

6.5 Aktualizacja jednostek cieniowania

Nie było to jawnie wcześniej zaznaczone, ale z kontekstu można było wywnioskować, że wszystkie obliczenia światła dokonywane są w przestrzeni świata. Istnieje oczywiście możliwość przenoszenia światła i jego właściwości do lokalnej przestrzeni obiektu, ale wiąże się to z kosztownym wyznaczaniem macierzy odwrotnej. Dodatkowo należy sobie odpowiedzieć na pytanie, gdzie obliczenia powinny być wykonywane. Zdecydowanie szybsze są rozwiązania bazujące na etapie wierzchołków, gdyż wiąże się to ze stosunkowo niewielką liczbą wywołań jednostki. Problemem w tym przypadku jest fakt, że natężenie światła nie powinno zmieniać się liniowo w zakresie trójkąta, ponieważ obliczenia bazują na funkcjach trygonometrycznych. W rezultacie, na obrazie końcowym, wyraźnie widać granice pomiędzy prymitywami. Popularniejszym obecnie rozwiązaniem jest wyliczanie światła na etapie cieniowania pikseli, gdyż zapewnia możliwość przeprowadzania obliczeń dla najmniejszej jednostki podziału – piksela. Należy jednak dążyć, aby maksymalnie odciążyć ten etap od operacji, które mogą być wykonane wcześniej, a ich rezultaty z powodzeniem zinterpolowane. Róż-



Rysunek 6.8. Różnica między obliczaniem światła per wierzchołek a per piksel

nicę wizualną między cieniowaniem per wierzchołek i per piksel przedstawia rysunek 6.8

Poniżej przedstawiono aktualizacje wprowadzone do jednostek cieniowania, mające na celu umożliwienie cieniowania per piksel. Aby móc przekształcać pozycje i wektory do przestrzeni świata, konieczne jest dodanie odpowiedniej macierzy do zmiennych globalnych oraz obsłużenie jej po stronie aplikacji (na tym etapie nie powinno to stanowić problemu dla czytelnika).

```
struct VertexOutput {
  ...
  float3 worldNormal : NORMAL;
  float3 worldPosition : POSITION;
}; ...
//! Bufor zmieniajacy sie raz na obiekt
cbuffer ChangesPerObject {
  ...
  matrix g_world;
} ... VertexOutput VSGeneric(VertexInput input) {
  VertexOutput output;
  ...
  output.worldPosition = mul(float4(input.position, 1), g_world).xyz;
  output.worldNormal = mul(input.normal, (float3x3)g_world);
  return output;
}
```

Jednostka cieniowania pikseli do obliczeń potrzebuje informacji o świetle oraz pozycji obserwatora w przestrzeni świata. Z tego powodu dodano następujące zmienne:

```
//! Bufor zmieniajacy sie raz na ramkę
cbuffer ChangesSometimes {
  float3 g_eyePosition;
  float3 g_ambient;
  Light g_light;
}
```

Obsługa zmiennych efektu, które są strukturami, jest nieco problematyczna. Celem, do którego należy niewątpliwie dążyć, jest stworzenie identycznych pod względem rozkładu danych typów w kodzie aplikacji. Na przeszkodzie stoją jednak inne zasady wyrównywania pół używane przez kompilator efektów i kompilator C++. Gdy oba typy nie są w tym zakresie zsynchronizowane, musi prędzej czy później dojść do odczytania błędnych danych.
Problem należy rozwiązać albo rezygnując ze struktur w efektach na rzecz wbudowanych typów, albo znając dokładny rozkład jednego z typów, przystosować drugi do współpracy z nim. W języku HLSL istnieje specjalna składnia służąca do jawnego pozycjonowania pól, opisana dokładnie w dokumentacji. W przypadku języków C++ brakuje takiej funkcjonalności języka, ale problem można obejść, dodając nieużywane pola wyrównujące. Dokładny rozkład pól w typach HLSL można poznać studiując listing wykreowany podczas kompilacji (załącznik C):

```
//   struct
//   {
//
//       int type;               // Offset:   16
//       float3 position;        // Offset:   20
//       float3 direction;       // Offset:   32
//       float3 color;           // Offset:   48
//       float3 attenuation;     // Offset:   64
//       float spotPower;        // Offset:   76
//
//   } g_light;                  // Offset:   16 Size:    64
```

Definicja analogicznej struktury w C++ musi uwzględniać fakt, że między dwoma wektorami trójwymiarowymi odstęp wynosi nie 12, lecz 16 bajtów. Wobec tego typ po stronie aplikacji może przedstawiać się następująco:

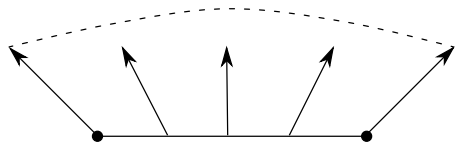
```
struct Light {
  int type;
  D3DXVECTOR3 position;
  D3DXVECTOR3 direction;
  char __padding0[4];
  D3DXVECTOR3 color;
  char __padding1[4];
  D3DXVECTOR3 attenuation;
  float spotPower;
};
```

Do ustawiania/odczytywania takiej struktury może służyć metoda `ID3D10EffectVariable::SetRawValue`.

Aby uniknąć narzutu związanego z obsługą sekwencji instrukcji warunkowych w funkcji `CalculateLight`, jednostce cieniowania pikseli został dodany parametr z modyfikatorem `uniform` określający wybrany model światła. Dzięki temu, wszystkie testy związane z tym aspektem są wykonywane w trakcie kompilacji, a niepotrzebne rozgałęzienia są usuwane. Zanim jednak nastąpią obliczenia, należy wyznaczyć kolory materiału dla zadanych składowych:

```
float4 PSGeneric(VertexOutput input, uniform int lightModel) :
SV_Target {

  // kolor końcowy
  float4 result = float4(0,0,0,mat_dissolve);
  // ustawienie kolorów dla różnych składowych światła
  float3 ambient = mat_ambientColor;
  float3 diffuse = mat_diffuseColor;
  float3 specular = mat_specularColor;
  [branch]
  if ( mat_diffuseMapEnabled ) {
    // mapa dla światła rozproszonego jako jedyna może dostarczać alfę
    float4 color = mat_diffuseMap.Sample(linearSampler, input.texcoords);
    diffuse *= color.rgb;
    result.a *= color.a;
  }
  [branch]
  if ( mat_specularMapEnabled ) {
    specular *= mat_specularMap.Sample(linearSampler, input.texcoords).rgb;
  }
```

Kolejnym krokiem będzie ustawienie właściwości powierzchni. Przy przypisywaniu wektora normalnego należy pamiętać, że na etapie cieniowania pikseli wektory, które wcześniej były jednostkowe, zazwyczaj tracą tę właściwość (istotę problemu obrazuje rysunek 6.9). Dlatego też praktycznie zawsze każdy wektor, co do którego jest wymaganie jednostkowości, musi zostać jawnie znormalizowany.



Rysunek 6.9. Zmiana długości wektorów podczas interpolacji liniowej. Gdyby interpolowane wektory miały jednakową długość jak brzegowe, wówczas czubki ich grotów zakreślałyby okrąg (kreskowany)

```
  // określenie właściwości powierzchni
  Surface surface;
  surface.position = input.worldPosition;
  surface.normal = normalize(input.worldNormal);
  surface.tangent = normalize(cross(surface.normal, float3(1,0,0)));
  surface.bitangent = cross(surface.normal, surface.tangent);
  surface.gloss = mat_specularExponent;
  surface.roughness = mat_roughness;
  surface.lightModel = lightModel;
```

Pytanie może rodzić zasadność obliczania wektorów stycznych do powierzchni, skoro są one wykorzystywane tylko dla jednego modelu. Czy instrukcje te nie powinny znaleźć się w bloku warunkowym, skoro i tak byłby statycznie testowany? Generalnie tak właśnie powinno być, ale ta sytuacja prezentuje kolejną właściwość kompilatora – usuwanie obliczeń, które nic nie wnoszą do wyniku. Dzięki uczynieniu z modelu światła parametru `uniform` kompilator może prześledzić dokładnie użycie wszystkich pól typu `Surface` oraz odrzucić

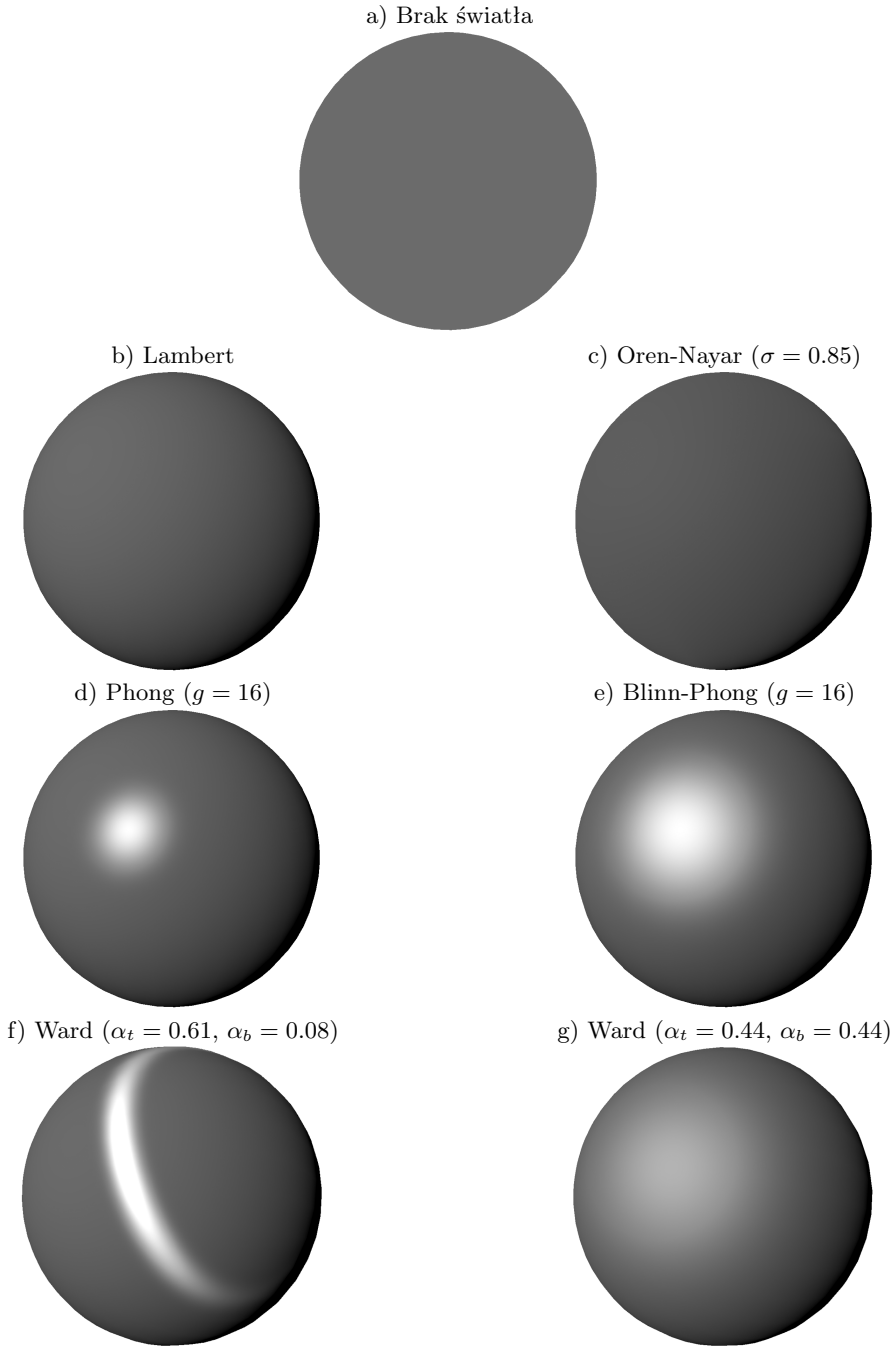
instrukcje związane z tymi, które nie zostaną uwzględnione w ostatecznym wyniku. Aby się upewnić, czy faktycznie zachodzi takie zjawisko, należy przejrzeć listing skompilowanego efektu (więcej na ten temat w załączniku C).

Aby zachować pewną uniwersalność, identyfikator typu modelu może przyjąć dodatkowe dwie wartości. `NONE` oznacza, że materiał nie używa światła i ma zawsze kolor równy zdefiniowanemu kolorowi dla światła rozproszonego. Wartość `UNKNOWN` zapewnia całkowitą uniwersalność jednostce cieniowania, ponieważ wówczas model światła dobierany jest na podstawie danych dostarczonych przez sam materiał. W tym przypadku używane jest pole `mat_illumination` odwzorowujące polecenie `illum` materiałów MTL – zgodnie zaś ze standardem wartość 2 oznacza, że obiekt może odbijać światło w sposób zwierciadlany. Wybór modelu uwzględnia również sprawdzanie, czy materiał jest chropowaty. Podobna procedura sprawdzania może być zaimplementowana w kodzie aplikacji, z tym że wtedy owocem wyboru byłaby odpowiednia specjalizowana technika.

```
  if ( lightModel == LIGHT_MODEL_NONE ) {
    result.rgb = diffuse;
  } else {
    // czy dynamicznie określany typ oświetlenia?
    if (lightModel == LIGHT_MODEL_UNKNOWN) {
      // czy podano chropowatość? (wartość ujemna oznacza, że nie)
      if ( mat_roughness.x >= 0 ) {
        // model uwzględniający chropowatość
        surface.lightModel = (mat_illumination==2 ? LIGHT_MODEL_WARD :
            LIGHT_MODEL_OREN_NAYAR);
      } else {
        // model nieuwzględniający chropowatości
        surface.lightModel = (mat_illumination==2 ? LIGHT_MODEL_PHONG :
            LIGHT_MODEL_LAMBERT);
      }
    }
    // obliczenie natężenia światła
    float3 light = CalculateLight(g_light, surface, g_eyePosition);
    // ustalenie koloru końcowego
    result.rgb = light.x * ambient * g_ambient;
    result.rgb += (light.y * diffuse + light.z * specular) * g_light.color;
  }
  return result;
}
```

Aby przekonać się do zasadności twierdzeń na temat kompilatora oraz modyfikatorów `uniform`, wystarczy rzucić okiem na listing skompilowanego efektu. Okazuje się, że wariant dla modelu `UNKNOWN` zawiera średnio około dwukrotnie więcej instrukcji niż jednostki skompilowane z inną stałą. Zadaniem, które pozostało do wykonania, jest stworzenie odpowiednich technik:

```
PixelShader psGeneric = CompileShader( ps_4_0,
PSGeneric(LIGHT_MODEL_UNKNOWN) ); PixelShader psGenericLambert =
CompileShader( ps_4_0, PSGeneric(LIGHT_MODEL_LAMBERT) ); ...
technique10 Generic {
  pass Pass0
  {
    SetVertexShader( vsGeneric );
    SetGeometryShader( NULL );
    SetPixelShader( psGeneric );
  }
```



Rysunek 6.10. Przykład oświetlania tej samej powierzchni z różnym modelem oświetlenia

```
} technique10 LightLambert {
  pass Pass0
  {
    SetVertexShader( vsGeneric );
    SetGeometryShader( NULL );
    SetPixelShader( psGenericLambert );
  }
} ...
```

Procedura wyznaczania odpowiedniej techniki w kodzie aplikacji nie zostanie tutaj omówiona, gdyż w kontekście przedstawionych informacji jej implementacja jest trywialna. Przykładowe efekty otrzymywane za pomocą różnych modeli światła przedstawia rysunek 6.10.

7

Mapy normalnych

Dotychczas szeroko pojęte dane geometryczne używane na etapie cieniowania piksela pochodziły z interpolowanych wierzchołków. Jest to intuicyjne rozwiązanie, jednak czasami w celu uzyskania konkretnego efektu warto uwolnić się od tego mechanizmu. Przykładem są tu tekstury – ich odczyt następuje dopiero na etapie cieniowania fragmentów. Gdyby wykonywać to wcześniej, wówczas kolor piksela stanowiłby średnią ważoną kolorów pobranych w wierzchołkach, co w oczywisty sposób zmniejszyłoby szczegółowość i jakość ostatecznego obrazu. To, co nie powinno zostać zinterpolowane, można w miarę możliwości zapisać w postaci obrazka, a na etapie cieniowania pikseli odczytać korzystając z zinterpolowanych współrzędnych tekstury.

Mapowanie normalnych (ang. *normal mapping*) to jedna z podstawowych technik służąca do stwarzania iluzji większej szczegółowości siatek niż to wynika z danych ich geometrii. Uogólniając, mapy normalnych to tekstury, w których składowe tekseli to współrzędne wektorów normalnych, które powinny być użyte przy obliczeniach światła. Podobnie jak odczyt zwyczajnych tekstur dopiero na etapie cieniowania pikseli pozwala na zróżnicowanie koloru w ramach prymitywu, tak próbkowanie mapy normalnych umożliwia precyzyjniejsze i bardziej szczegółowe wyznaczenie oświetlenia. Sama konwersja odczytanego teksela RGB na wektor wygląda następująco:

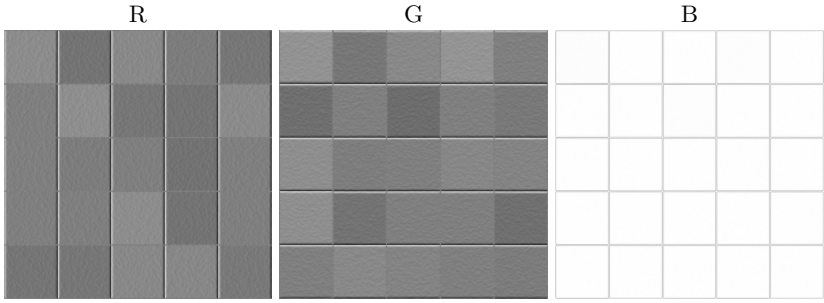
$$\vec{n_T} = 2 * (C_n - 0.5)$$

gdzie:

$\vec{n_T}$ - wektor normalny z mapy normalnych
C_n - znormalizowany kolor odczytany z mapy normalnych

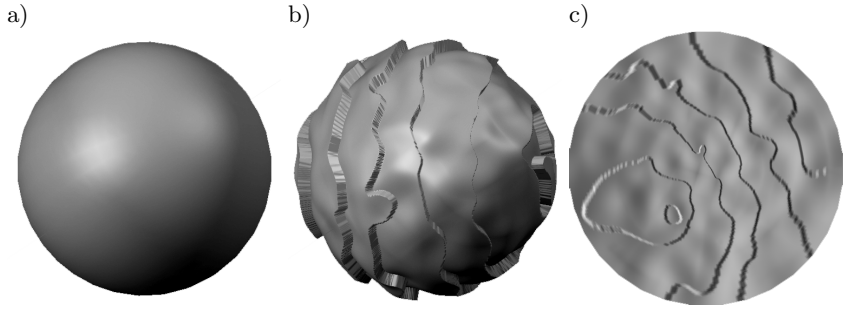
7.1 Tworzenie map normalnych

Mapy normalnych zazwyczaj tworzy się na dwa sposoby. Pierwszy z nich to tzw. wypalanie (ang. *baking*), udostępniane przez edytory 3D. Technika ta ba-



Rysunek 7.1. Przykładowa mapa normalnych dekomponowana na poszczególne kanały

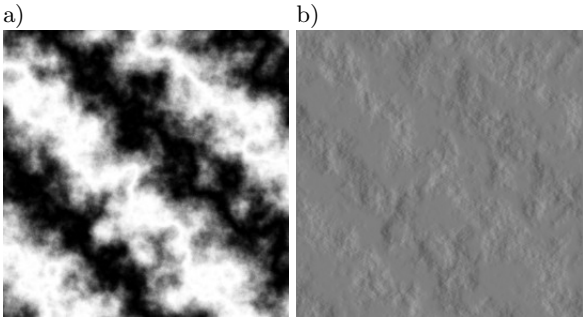
zuje na dwóch wariantach siatki – wysoko szczegółowym oraz uproszczonym, używanym we właściwej aplikacji. W znacznym uogólnieniu proces ten polega na próbkowaniu z dużą częstotliwością ścian prostego modelu oraz odczycie wektora normalnego ze skomplikowanego wariantu w analogicznym miejscu. Pobrana wartość zapisywana jest do tekstury mapowanej zgodnie ze współrzędnymi z uproszczonej siatki.



Rysunek 7.2. Wypalanie normalnych. Próbkując prostą siatkę (a) oraz odczytując wektory normalne ze szczegółowej siatki (b), generuje się mapę normalnych (c, kanał R)

Innym rozwiązaniem jest wyliczenie mapy normalnych na podstawie mapy wysokości. Mapa wysokości (ang. *heightmap*) to rastrowy obraz z jednym kanałem, gdzie każdy piksel zawiera informację o odległości od pewnej powierzchni. Obraz taki można przedstawić jako bitmapę w skali szarości, gdzie czarny kolor oznacza minimalną odległość, natomiast biały maksymalną, tak jak na rysunku 7.3a. Stosując takie algorytmy wizji komputerowej, jak detekcja krawędzi operatorami gradientowymi, można, bazując na różnicach wysokości, wyznaczyć wektor normalny właściwy zadanemu pikselowi. Przykład zastoso-

wania takiej metody obrazuje rysunek 7.3b. Mapę normalnych na podstawie mapy wysokości można obliczyć funkcją `D3DX10ComputeNormalMap` albo odpowiednim edytorem grafiki 2D (np. wtyczką NVIDIA do Adobe Photoshop) lub 3D. Uzupełnieniem obu metod może być ręczna obróbka w edytorze grafiki 2D, co pozwala na szybkie wprowadzanie dodatkowych szczegółów.



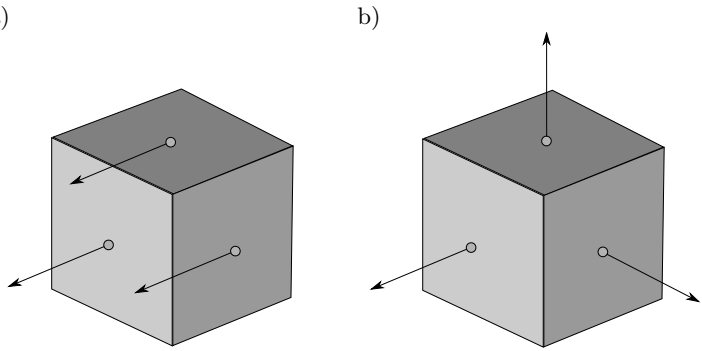
Rysunek 7.3. Przykładowa mapa wysokości (a) oraz kanał R wyznaczonej na jej podstawie mapy normalnych (b)

7.2 Teoretyczne podstawy przestrzeni stycznych

Wektor odczytany z tekstury nie może być bezpośrednio użyty w obliczeniach. Wynika to z faktu, że jest on zazwyczaj zdefiniowany w przestrzeni stycznych (ang. *tangent space*), zwanej również przestrzenią tekstury (ang. *texture space*). Wówczas taki wektor jest określony względem ściany, do której zostaje przyłożony.

Przykładowo, odczytany został wektor $\vec{n_T} = [\,0\ 0\ 1\,]$. Założywszy, że pomija się transformację macierzą świata, jaki kierunek będzie wskazywany? Odpowiedzi nie można udzielić nie znając orientacji mapowanej płaszczyzny oraz sposobu nakładania tekstury. Jeżeli ściana będzie zwrócona idealnie zgodnie z osią OZ, wówczas wektor rzeczywiście będzie wskazywał zamierzony kierunek. Każde odstępstwo od tej idealności spowoduje jednak, że zmianie powinna ulec interpretacja odczytanych danych (rysunek 7.4a i 7.4b). Wobec tego przed użyciem wektora odczytanego z mapy normalnych konieczne jest przeniesienie go z przestrzeni tekstury do przestrzeni lokalnej.

Przestrzeń stycznych wyznaczają, poza oryginalnym wektorem normalnym powierzchni, dwa styczne do prymitywu wektory – tangent i prostopadły do niego bitangent. Ich punkt zaczepienia jest kwestią przyjętej konwencji. W dalszych przykładach będzie to pierwszy wierzchołek ściany (o indeksie 0). Wektory tangent i bitangent powinny być tak wyznaczone, aby mapowały przesunięcie w ramach prymitywu na zmianę współrzędnych tekstury. Opisuje to poniższa zależność:



Rysunek 7.4. Wpływ orientacji ściany na odczytany wektor normalny $n\vec{}_T = [0\ 0\ 1]$. Nieuwzględnienie orientacji ściany skutkuje błędną interpretacją wektora odczytanego z tekstury (a). Obok (b) znajdują się poprawnie zinterpretowane wektory

$$P - P_0 = (u - u_0)\vec{t} + (v - v_0)\vec{b} \tag{7.1}$$

gdzie:

P - punkt na prymitywie
P_0 - pozycja wierzchołka odniesienia
u, v - współrzędne tekstury punktu
u_0, v_0 - współrzędne tekstury punktu odniesienia
$\vec{t}$ - tangent
$\vec{b}$ - bitangent

Podstawiając do równania 7.1 wierzchołki 1 i 2, tworzy się układ sześciu równań z sześcioma niewiadomymi, którego rozwiązanie wygląda następująco (wyprowadzenia pozostawione Czytelnikowi):

$$\begin{bmatrix} \vec{t}_x & \vec{t}_y & \vec{t}_z \\ \vec{b}_x & \vec{b}_y & \vec{b}_z \end{bmatrix} = \frac{1}{\Delta u_1 \Delta v_2 - \Delta u_2 \Delta v_1} \begin{bmatrix} \Delta v_2 & -\Delta v_1 \\ -\Delta u_2 & \Delta u_1 \end{bmatrix} \begin{bmatrix} \Delta x_1 & \Delta y_1 & \Delta z_1 \\ \Delta x_2 & \Delta y_2 & \Delta z_2 \end{bmatrix}$$

$$(\Delta x_1, \Delta y_1, \Delta z_1) = P_1 - P_0$$
$$(\Delta x_2, \Delta y_2, \Delta z_2) = P_2 - P_0$$
$$(\Delta u_1, \Delta v_1) = (u_1 - u_0, v_1 - v_0)$$
$$(\Delta u_2, \Delta v_2) = (u_2 - u_0, v_2 - v_0)$$

Przy użyciu wektorów $\vec{t}$, $\vec{b}$ i $\vec{n}$ (oryginalnego wektora normalnego) można skonstruować macierz TBN, która dokonuje transformacji z przestrzeni lokalnej do przestrzeni stycznych.

$$M_T = \begin{bmatrix} \vec{t}_x & \vec{b}_x & \vec{n}_x \\ \vec{t}_y & \vec{b}_y & \vec{n}_y \\ \vec{t}_z & \vec{b}_z & \vec{n}_z \end{bmatrix} \tag{7.2}$$

Ponieważ jednak celem jest odwrotne przekształcenie, tj. wektora z przestrzeni stycznych do przestrzeni obiektu, konieczne jest wyznaczenie macierzy odwrotnej. Najmniejszym kosztem można to zrobić, jeżeli macierz jest ortogonalna – wówczas wystarczy zwykła transpozycja. Niestety, wektory $\vec{t}$ i $\vec{b}$ nie będą zazwyczaj ani jednostkowe, ani prostopadłe. Rozwiązaniem jest zastosowanie ortogonalizacji oraz normalizacji. W przypadku wektora $\vec{t}$ obliczenia wyglądać mogą następująco:

$$\vec{t'} = \vec{t} - (\vec{n} \cdot \vec{t})\vec{n}$$

Dysponując prostopadłymi jednostkowymi wektorami $\vec{t'}$ i $\vec{n}$, można wyznaczyć ortogonalny wektor $\vec{b'}$ za pomocą iloczynu wektorowego:

$$\vec{b'} = \vec{n} \times \vec{t'} \tag{7.3}$$

W efekcie można przyjąć, że przybliżoną odwrotnością macierzy 7.2, a zarazem transformacją z przestrzeni stycznych do przestrzeni obiektu jest macierz

$$M_L = \begin{bmatrix} t'_x & t'_y & t'_z \\ b'_x & b'_y & b'_z \\ n_x & n_y & n_z \end{bmatrix}$$

Podsumowując, dopiero po mnożeniu n_T razy M_L wektor normalny staje się zorientowany w przestrzeni obiektu, dzięki czemu można go poddać transformacji macierzą świata i następnie użyć w obliczeniach światła.

$$n_W = n_T \cdot M_L \cdot M_W \tag{7.4}$$

gdzie:

n_W - wektor normalny w przestrzeni świata
M_W - macierz świata

7.3 Wyznaczanie tangentów

Przed aktualizacją jednostek cieniowania konieczne jest dostarczenie im danych o stycznych. W idealnym przypadku siatka wyeksportowana przez grafika z edytora 3D zawierałaby te informacje, jednak z uwagi na fakt, iż format OBJ nie pozwala na dodawanie tangentów do wierzchołków, konieczne jest przeprowadzenie obliczeń już w trakcie uruchomienia programu. Najmniej inwazyjną metodą aktualizacji logiki typu `ObjMesh` jest dodanie nowego bufora wierzchołków, który będzie przechowywał jedynie tangenty.

Zanim stworzy się bufor konieczne jest wyliczenie odpowiednich wektorów. Zgodnie ze wzorem 7.3 nie ma konieczności wyliczania bitangentów na tym etapie. Warto zaznaczyć, że tangent przypisany do wierzchołka jest pewnym umownym uproszczeniem – do tej pory ta wielkość rozważana była w sytuacji całego prymitywu. Sytuację można przyrównać do przypadku wektorów normalnych w wierzchołkach, które zazwyczaj są uśrednieniem normalnych przyległych ścian. Przyjęcie tej samej filozofii w przypadku stycznych pozwala zachować jedną konwencję oraz uniknąć wyraźnych różnic między sąsiadującymi prymitywami. Biorąc pod uwagę powyższe uwagi oraz wyprowadzenia z poprzedniego rozdziału, funkcja obliczająca tangenty może wyglądać następująco:

```
void ObjMesh::CalculateTangents(const std::vector<Vertex>& vertices,
                                const std::vector<Face>& faces,
                                std::vector<D3DXVECTOR3>& tangents)
{
  // inicjalizacja
  tangents.assign(vertices.size(), D3DXVECTOR3(0,0,0));
  // pętla po wszystkich ścianach
  for ( std::vector<Face>::const_iterator it = faces.begin();
    it != faces.end(); ++it ) {
    // pobranie wierzchołków
    const Face& face = *it;
    const Vertex& v0 = vertices[face.i0];
    const Vertex& v1 = vertices[face.i1];
    const Vertex& v2 = vertices[face.i2];
    // obliczenie delt
    float dx1 = v1.position.x - v0.position.x;
    float dx2 = v2.position.x - v0.position.x;
    float dy1 = v1.position.y - v0.position.y;
    float dy2 = v2.position.y - v0.position.y;
    float dz1 = v1.position.z - v0.position.z;
    float dz2 = v2.position.z - v0.position.z;
    float du1 = v1.texcoord.x - v0.texcoord.x;
    float du2 = v2.texcoord.x - v0.texcoord.x;
    float dv1 = v1.texcoord.y - v0.texcoord.y;
    float dv2 = v2.texcoord.y - v0.texcoord.y;
    // wyznacznik macierzy [ du0 dv0; du1 dv1 ]
    float det = 1 / (du1 * dv2 - du2 * dv1);
    // tangent
    D3DXVECTOR3 t((dv2 * dx1 - dv1 * dx2) * det,
                  (dv2 * dy1 - dv1 * dy2) * det,
                  (dv2 * dz1 - dv1 * dz2) * det);
    // sumowanie tangentów (do uśrednienia współdzielonych
    // wierzchołków podczas normalizacji)
    tangents[face.i0] += t;
    tangents[face.i1] += t;
    tangents[face.i2] += t;
  }
  // normalizacja i~ortogonalizacja
  for (size_t i~= 0; i~< vertices.size(); ++i) {
    D3DXVECTOR3 n = vertices[i].normal;
    D3DXVECTOR3& t = tangents[i];
    // ortogonalizacja
    D3DXVec3Subtract(&t, &t, D3DXVec3Scale(&n, &n, D3DXVec3Dot(&n, &t)));
    // normalizacja
    D3DXVec3Normalize(&t, &t);
  }
}
```

Drugim krokiem jest aktualizacja opisu struktury bufora. Ponieważ, jak zostało wspomniane wcześniej, dane tangentów będą się znajdować w osobnym

buforze, trzeba to odpowiednio zaznaczyć polem `InputSlot`, czyli indeksem źródła. Poniższy kod definiuje przykładowy opis (pole `InputSlot` to czwarta wartość począwszy od lewej strony).

```
// tworzenie układu elementów
D3D10_INPUT_ELEMENT_DESC vertexElements[] = {
  { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
      D3D10_INPUT_PER_VERTEX_DATA, 0 },
  { "TANGENT", 0, DXGI_FORMAT_R32G32B32_FLOAT, 1, 0,
      D3D10_INPUT_PER_VERTEX_DATA, 0 },
  { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 12,
      D3D10_INPUT_PER_VERTEX_DATA, 0 },
  { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 20,
      D3D10_INPUT_PER_VERTEX_DATA, 0 },
}; elements.assign(vertexElements,
 vertexElements + sizeof(vertexElements)/sizeof(D3D10_INPUT_ELEMENT_DESC));
```

Każda automatyzacja może potencjalnie rodzić problemy, które ciężko przewidzieć. Posługiwanie się `ID3DX10Mesh` zamiast buforami jest znacznie wygodniejsze choćby z faktu, że nie trzeba podawać rozmiaru wierzchołka dla każdego bufora (por. rozdz. 4.6 i 5.4.5); obiekt implementujący interfejs sam dokonuje pewnych obliczeń, bazując na przekazanej definicji układu elementów. Wiąże się to z wymaganiem, które nie jest nigdzie sformułowane – konieczne jest układania elementów opisu zgodnie z ich offsetem, nawet jeśli dotyczą innych źródeł. Wobec tego opis w postaci

```
D3D10_INPUT_ELEMENT_DESC vertexElements[] = {
  { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT,
    0, 0, D3D10_INPUT_PER_VERTEX_DATA, 0 },
  { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT,
    0, 12, D3D10_INPUT_PER_VERTEX_DATA, 0 },
  { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT,
    0, 20, D3D10_INPUT_PER_VERTEX_DATA, 0 },
  { "TANGENT", 0, DXGI_FORMAT_R32G32B32_FLOAT,
    1, 0, D3D10_INPUT_PER_VERTEX_DATA, 0 }
};
```

mimo że jest poprawny, spowodowałby błędne obliczanie przesunięć przez obiekt implementujący `ID3DX10Mesh` i, co za tym idzie, niespodziewane błędy w renderingu.

Dodanie dodatkowego bufora wierzchołków do siatki wymaga wywołania metody `SetVertexData` z parametrami będącymi kolejno indeksem źródła oraz danymi do inicjalizacji. Po następującym wywołaniu

```
V_RETURN(mesh->SetVertexData(1, &tangents[0]) );
```

aktualizacja kodu po stronie aplikacji może być zakończona.

7.4 Wyznaczanie wektora normalnego

Aktualizację efektu należy rozpocząć od uzupełnienia definicji typów wierzchołków o tangent. Należy przy tym pamiętać, żeby semantyka przypisana nowemu polu odpowiadała tej ustalonej na etapie opisu zawartości buforów.

```
//! Dane wierzchołka.
struct VertexInput {
  float3 pos : POSITION;
  float2 tex : TEXCOORD;
  float3 normal : NORMAL;
  float3 tangent : TANGENT;
};
```

Równania 7.4 w niezmienionej postaci można używać wyłącznie w jednostce cieniowania fragmentów. Należy jednak zauważyć, że iloczyn macierzy tangentów i świata można przedstawić w następujący sposób:

$$M_o \cdot M_w = \begin{bmatrix} T'_x & T'_y & T'_z \\ B'_x & B'_y & B'_z \\ N_x & N_y & N_z \end{bmatrix} \cdot M_w = \begin{bmatrix} (T'_w)_x & (T'_w)_y & (T'_w)_z \\ (B'_w)_x & (B'_w)_y & (B'_w)_z \\ (N_w)_x & (N_w)_y & (N_w)_z \end{bmatrix}$$

$$T'_w = T \cdot M_w$$
$$B'_w = B \cdot M_w$$
$$N_w = N \cdot M_w$$

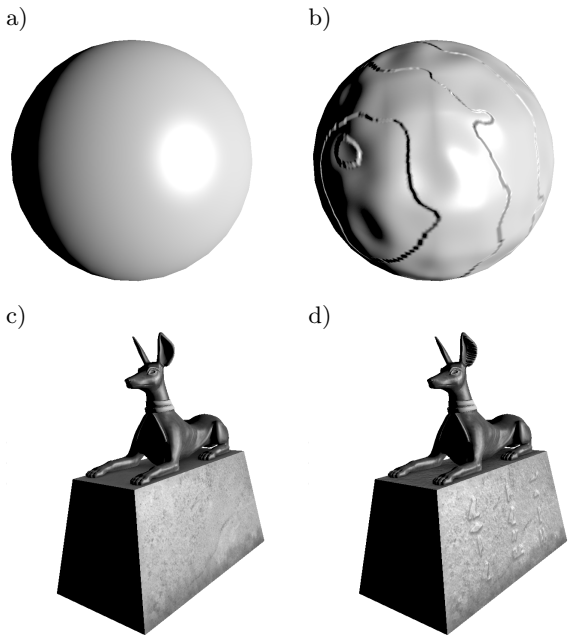
Wektor N_w obliczany był już przy okazji obliczeń światła, więc jego pojawienie się w macierzy wynikowej nie generuje dodatkowych kosztów, natomiast wektor B'_w można alternatywnie wyrazić jako iloczyn wektorowy N_w i T'_w. Okazuje się zatem, że zamiast mnożyć dwie macierze o wymiarach 3x3 można pomnożyć tylko jeden wektor oraz wyliczyć jeden iloczyn wektorowy. Dodatkowo nie ma przeszkód, aby wyliczenie T'_w przenieść do jednostek cieniowania wierzchołków.

```
//! Dane zwracane przez jednostkę wierzchołków.
struct VertexOutput {
  float4 pos : SV_POSITION;
  float2 tex : TEXCOORD;
  float3 worldNormal : NORMAL;
  float3 worldPos : POSITION;
  float4 worldTangent : TANGENT;
};

//!
VertexOutput VSTransform(VertexInput input) {
  VertexOutput output;
  /* ... */
  output.worldTangent = mul(input.tangent, (float3x3)g_world);
  return output;
}
```

Na etapie cieniowania pikseli jest zatem konieczne odtworzenie iloczynu $M_o \cdot M_w$ na podstawie przekazanych już danych. Nie można zapomnieć o przeprowadzeniu normalizacji i ortogonalizacji wektorów – interpolacja tangentów i normalnych między wierzchołkami z bardzo dużą dozą prawdopodobieństwa może pozbawić je tej potrzebnej właściwości. Po wyznaczeniu macierzy zostaje pomnożyć poddany wcześniej przeskalowaniu zgodnie z równaniem 7.1 wektor odczytany z tekstury, żeby otrzymać wektor normalny zdefiniowany w przestrzeni świata. Pozostałe obliczenia związane ze światłem nie ulegają zmianom w stosunku do poprzednich wersji. Ponieważ nie każdy materiał musi posiadać

mapy normalne, nad całością czuwa flaga `mat_bumpMapEnabled` określająca, czy można dokonać odczytu z tekstrury `mat_bumpMap`.

```
float4 PSLight(VertexOutput input, uniform int lightModel) :
SV_Target {
  Surface surface;
  /* ... */
  // czy jest mapa normalnych?
  if (mat_bumpMapEnabled) {
    // odczyt wektora normalnego z mapy normalnych
    float3 Nt = mat_bumpMap.Sample( linearSampler, input.tex ).rgb;
    // skalowanie liniowe z zakresu <0,1> do <-1,1>.
    Nt = 2.0f*Nt - 1.0f;
    // korekta wektorów tak, aby były ortogonalne (interpolacja
    // zaburzą tę właściwościwość)
    float3 N = normalize(input.worldNormal);
    float3 T = normalize(input.worldTangent - dot(input.worldTangent, N)*N)
        ;
    float3 B = cross(N, T);
    // macierz Mo * Mw
    float3x3 MlMw = float3x3(T, B, N);
    // wyznaczenie wektora normalnego
    surface.normal = normalize(mul(Nt, MlMw));
  } else {
    // nie ma - bazowanie na interpolowanej wartości
    surface.normal = normalize(input.worldNormal);
  }
}
```



Rysunek 7.5. Ta sama siatka odrysowana bez (a, c) i z (b, d) mapą normalnych. Na podstawie mapy normalnych można oddawać pofałdowanie (b) albo dodawać detale (d)

Na rysunku 7.5 przedstawiono różnicę pomiędzy przykładową siatką odrysowaną bez i z mapą normalnych. Jak widać, omawiana technika pozwala na znaczne zwiększenie realizmu sceny bez żmudnych obliczeń czy też zwiększania skomplikowania geometrii. Z tych powodów mapy normalnych są obecnie w powszechnym użyciu.

8

Mapy odbić

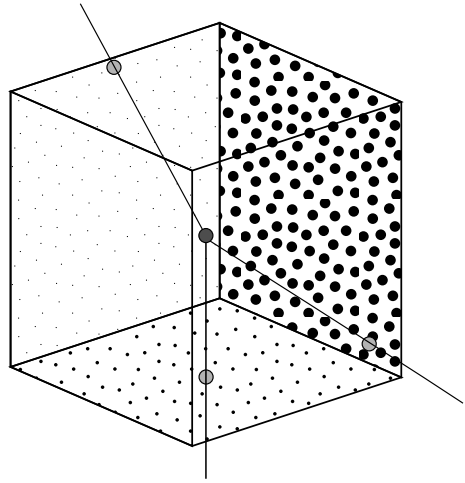
Do tej pory zwierciadlana składowa światła uwzględniała wyłącznie promienie pochodzące od źródła pierwotnego, w czego efekcie na powierzchni obiektu można było zaobserwować charakterystyczne rozbłyski. Model taki dobrze przybliża rzeczywistość, gdy źródła wtórne są bardzo słabe bądź odległe. Gdy jednak istnieje sąsiedztwo obiektu, od którego promienie mogą się odbić, na powierzchni połyskliwego ciała powinien być widoczny ich obraz. W przeciwnym razie ta dysproporcja pomiędzy uwzględnieniem źródeł pierwotnych oraz wtórnych bardzo rzutuje na wiarygodność sceny. Ponieważ z racji złożoności obliczeniowej zastosowanie metod rodem z globalnego modelu oświetlenia jest niewykonalne, opracowano bardzo pomysłową metodę pozwalającą na uzyskanie zadowalających rezultatów.

8.1 Tekstury sześcienne

Tekstura sześcienna (ang. *cube map*) to zestaw sześciu tekstur, z których każda jest w całości przypisana jednej wewnętrznej ścianie hipotetycznej sześciennej siatki. Współrzędne nie są dłużej określone wektorem dwuwymiarowym, lecz trójwymiarowym. Zgodnie z jego kierunkiem ze środka siatki wypuszczana jest półprosta, która przecinając którąś ze ścian, jednoznacznie wyznacza miejsce próbkowania (jak na rysunku 8.1). Prezentowany model zakłada niejawnie, że zarówno hipotetyczna siatka, jak i wektor kierunkowy określone będą w tym samym układzie odniesienia.

Niespodzianką może być, że tekstury sześcienne w Direct3D reprezentowane są interfejsem `ID3D10Texture2D` – tym samym co zwykła tekstura dwuwymiarowa. D3DX automatyzuje proces ich wczytywania, o ile znajdują się one w pliku DDS (akronim od DirectDraw Surface). Metadane odpowiednio przygotowanego pliku[1] pozwalają funkcjom typu `D3DX10CreateTextureFromFile`

[1] Szczegółowy opis tworzenia tekstur sześciennych znajduje się w załączniku D.



Rysunek 8.1. Przykładowe miejsca próbkowania tekstury sześciennej

na poprawną interpretację odczytywanych danych. Ta zgodność interfejsów i funkcji ma dużą zaletę – kod służący do odczytu obu typów zasobów może być wspólny.

Gdy zachodzi potrzeba własnoręcznej alokacji i wypełnienia tekstury sześciennej, proces ten wygląda podobnie jak w przypadku tekstury dwuwymiarowej. Ponieważ to struktura `D3D10_TEXTURE2D_DESC` opisuje format tworzonego obiektu implementującego `ID3D10Texture2D`, należy wprowadzić tam zmiany uwzględniające faktyczną mnogość tekstur w zakresie zasobu. Dodatkowo też, jeżeli tekstura ma być zainicjalizowana pewnymi danymi w momencie jej tworzenia, konieczne jest przekazanie sześcioelementowej tablicy struktur `D3D10_SUBRESOURCE_DATA`. To, który indeks opisuje który kierunek, określa wyliczenie `D3D10_TEXTURECUBE_FACE`.

```
ID3D10Texture2D* cubeTexture;
// opis tekstury
D3D10_TEXTURE2D_DESC texDesc; texDesc.ArraySize = 6;
texDesc.MiscFlags = D3D10_RESOURCE_MISC_TEXTURECUBE; /* pozostałe
ustawienia */
// opis danych inicjalizacyjnych
D3D10_SUBRESOURCE_DATA initialData[6]; /* wskazanie danych
inicjalizacyjnych */
// stworzenie tekstury
device->CreateTexture2D(&texDesc, initialData, &cubeTexture);
```

W odróżnieniu od kodu aplikacji, w efektach tekstury sześcienne reprezentowane są osobnym typem danych – `TextureCube`. Ponadto należy pamiętać, że próbkowanie odbywa się przy użyciu trójwymiarowego wektora.

```
TextureCube textureCube; SamplerState sampler = { /* ... */ }; /*
... */ float4 color = textureCube.Sample(sampler, float3(1,0,0));
```

8.2 Tworzenie map otoczenia

Głównym zastosowaniem tekstur sześciennych są: mapowanie środowiska (ang. *environment mapping*) lub mapowanie odbić (ang. *reflection mapping*). Pozwalają one na aproksymację wyglądu zwierciadlanych powierzchni w zadanym otoczeniu poprzez nanoszenie na nie wcześniej wyznaczonego obrazu tego sąsiedztwa.

Odpowiednia tekstura sześcienna powstaje podczas procesu projekcji sceny, polegającym na mapowaniu sześciu rzutów przestrzeni na sześcian. Obserwator umieszczany jest w punkcie, z którego roztaczany będzie widok. Następnie scena odrysowywana jest sześciokrotnie, za każdym razem z wektorem widoku zgodnym z kierunkiem kolejnej osi ($\pm X$, $\pm Y$ oraz $\pm Z$) oraz z 90°polem widzenia (pionowym i poziomym). Wygenerowane obrazy nie są wyświetlane, lecz zapisywane w teksturze, która przypisana jest bieżącemu kierunkowi. Po ostatnim odrysowaniu, z racji uwzględnienia wszystkich kierunków oraz 90°pola widzenia, tekstury są komplementarne i zawierają kompletny zapis pola widzenia w zadanej pozycji w przestrzeni (por. rysunek 8.2).

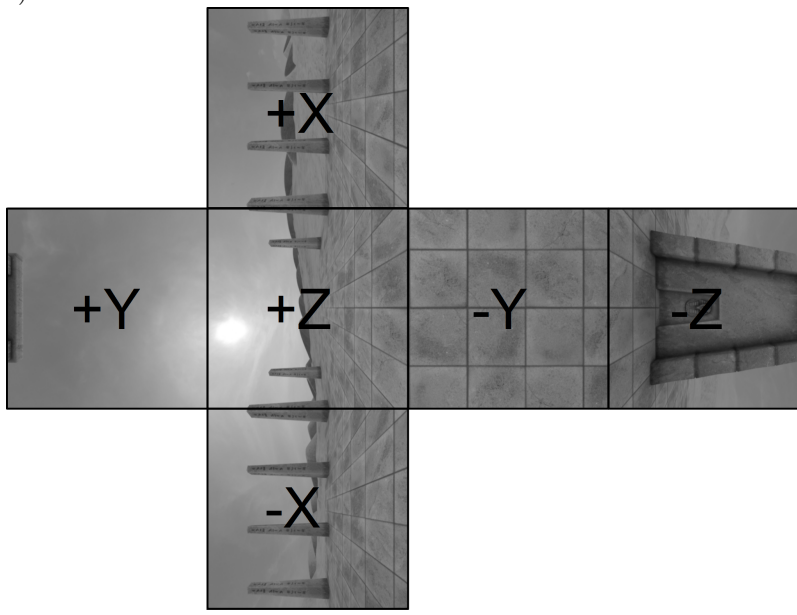
Wątpliwości powinna budzić możliwość obliczania wielu map w czasie rzeczywistym. Zakładając, że widocznych jest n połyskliwych obiektów, potrzeba aż $6n$ odrysowań sceny tylko do sporządzenia map środowiska dla każdego z nich. Aby zniwelować ogromne koszty obliczeniowe, można rozważyć kilka uproszczeń, które pozwalałyby na osiągnięcie zadowalającego efektu w rozsądnych ramach czasowych:

- Dynamiczna, czyli aktualizowana co ramkę, mapa odbić może być generowana jedynie dla obiektów bardzo połyskliwych, absorbujących uwagę użytkownika.
- Jeżeli obiekt oraz jego otoczenie są w przeważającej mierze statyczne, to mapę odbić można przygotować w zewnętrznej aplikacji lub też aktualizować ją rzadko.
- Dla obiektów częściowo połyskliwych występujących w podobnym otoczeniu można przygotować jedną wspólną mapę środowiska. Przykładem mogą być otwarte tereny z jednorodną roślinnością. Zazwyczaj taką mapę można również przygotować w zewnętrznej aplikacji.
- Jeżeli powierzchnia jest pofałdowana albo niedużych rozmiarów, można odrysować otoczenie, używając uproszczonych, mniej wymagających obliczeniowo technik, gdyż ewentualne straty jakości nie będą bardzo widoczne.

8.3 Modelowanie odbić

Obliczenia w dotychczas zaprezentowanych modelach oświetlenia zawsze polegały na analizie kierunku padania światła. Na podstawie relacji między wektorem oświetlenia a wektorem normalnym powierzchni wyznaczany był procent

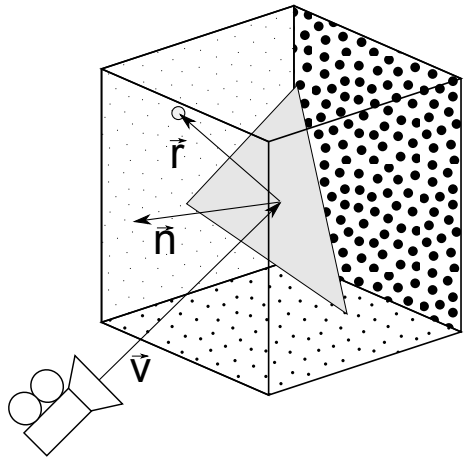
a)



b)

Rysunek 8.2. Przykładowa mapa otoczenia z (a) i bez (b) oznaczeń reprezentowanych kierunków.

energii odbijany w stronę obserwatora. W przypadku modelowania za pomocą map środowiska wtórnych źródeł światła zazwyczaj jest za dużo, aby obliczenia przeprowadzać w ten sposób. Zamiast tego przyjmuje się, że promienie docierające do obserwatora odbijane są w idealnie zwierciadlany sposób. Dzięki temu założeniu drogę promienia można odtworzyć na podstawie wektora widoku oraz wektora normalnego. Wyznaczony wektor padania prowadzi wprost do źródła, co prezentuje rysunek 8.3. Specyfika tekstur sześciennych sprawia, że jest to jednocześnie wskazanie miejsca próbkowania w celu pobrania koloru światła.



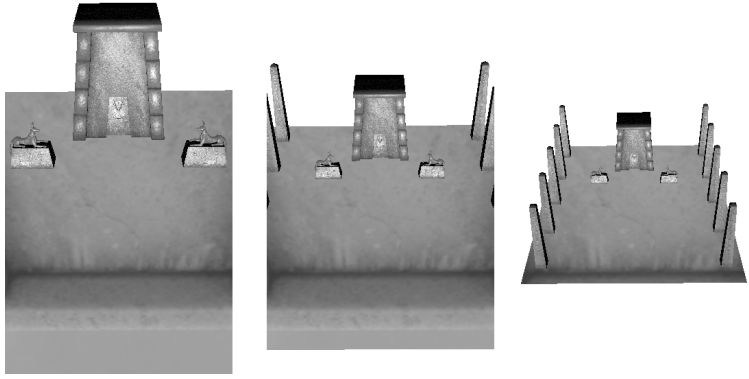
Rysunek 8.3. Modelowanie odbić przy użyciu mapy środowiska. Miejsce próbkowania zaznaczone jaśniejszym kolorem

W proponowanej implementacji kolor pobrany z tekstury sześciennej mnożony jest razy współczynnik pochłaniania światła zwierciadlanego. Wynika to po części z faktu, iż specyfikacja materiałów MTL nie przewiduje występowania analogicznej właściwości dla światła środowiska. Dodatkowo, ponieważ w obliczeniach wykorzystuje się prawo odbicia, więc użycie zmiennej przeznaczonej pierwotnie dla odbić zwierciadlanych światła ze źródła pierwotnego jest uzasadnione. Nic poza przestrzeganiem specyfikacji materiału nie stoi jednak na przeszkodzie, aby wprowadzić osobny współczynnik pochłaniania światła środowiska.

Wszystkie wymagane obliczenia można wykonać w jednostce cieniowania pikseli. W odróżnieniu od tradycyjnych modeli oświetlenia, których obliczenia bazują na iloczynach skalarnych znormalizowanych wektorów, tym razem nie ma konieczności przeprowadzania normalizacji. Dodatkowo z racji nieskomplikowania oraz braku specjalnych wymagań logika odbić środowiska łatwo łączy się z już istniejącymi jednostkami, w tym z obsługującą oświetlenie oraz mapy normalnych, rozwijaną w poprzednich przykładach.

```
float4 PSGeneric(VertexOutput input, uniform int lightModel) :
SV_Target {
  ...
  // dodanie składowej środowiska
  [branch]
  if (mat_reflectionMapEnabled) {
    float3 V = input.worldPos - g_eyePosition;
    float3 R = reflect(V, surface.normal);
    float3 envColor = mat_reflectionMap.Sample(linearSampler, R).rgb;
    result.rgb += mat_reflectance * envColor;
  }
  return result;
}
```

Bazowanie jedynie na wektorze normalnym oraz na pozycji obserwatora nie sprawdza się w przypadku rozległych, płaskich powierzchni. Wektor odbicia ma wówczas małą zmienność, a ponieważ każdy punkt na płaszczyźnie przeprowadza obliczenia tak, jakby był w środku hipotetycznego sześcianu, próbkowany jest jedynie niewielki zakres mapy otoczenia (rysunek 8.4). Rozwiązaniem problemu może być branie pod uwagę odległości danego punktu od środka obiektu [3], jednakże omówienie tego zagadnienia wykracza poza ramy tej książki.



Rysunek 8.4. Nakładanie odbić na płaskie powierzchnie dla różnych ujęć. Niezależnie od odległości kamery odbicie zawsze nakładane jest w ten sam sposób

Na rysunku 8.5 przedstawiono przykładowe efekty uzyskiwane za pomocą map odbić.

Rysunek 8.5. Przykłady zastosowań mapy otoczenia. Gładkie powierzchnie uzyskuje się, stosując najprostszy model oświetlenia (a), pofałdowane stosując mapy normalnych (b). Dzięki zastosowaniu wielu materiałów w ramach tej samej siatki tylko część może odbijać otoczenie (c)

9

Rysowanie tła

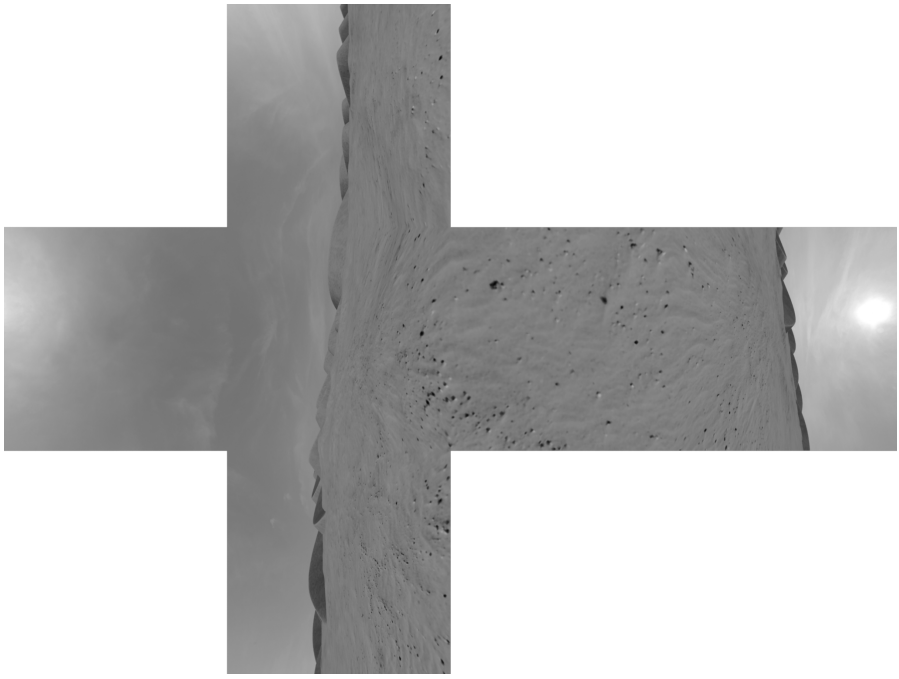
W grach komputerowych, których akcja odgrywa się w wirtualnej, otwartej przestrzeni, jednym z czynników wpływających na postrzegany realizm jest wierne oddanie wyglądu nieba. Wiarygodna symulacja zjawisk atmosferycznych, tak niezbędnych jak na przykład chmury [2], jest nietrywialnym zadaniem i wykracza poza ramy tej książki. Zamiast tego, zostaną zaprezentowane ogólne idee i proste rozwiązania, mogące służyć do stworzenia bardziej przekonywującego efektu.

9.1 Mapa środowiska jako tło

Częstym zastosowaniem statycznych map środowiska (rys. 8.2) jest tworzenie złudzenia nieba lub dalekiego, nieosiągalnego krajobrazu. Aby uzyskać efekt, po wyrenderowaniu nieprzezroczystej części sceny wykonuje się odrysowanie siatki nieba (sklepienia) od środka pokrytej teksturą sześcienną zawierającą tło. Mapa środowiska użyta w kolejnych przykładach przedstawiona jest na rysunku 9.1.

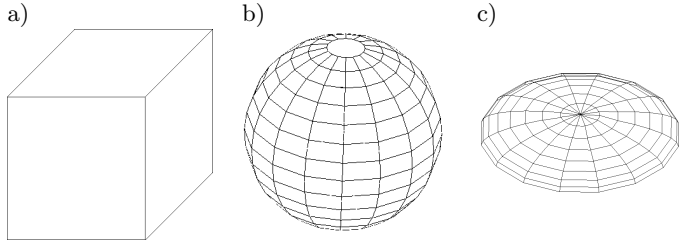
> Dlaczego tło powinno się odrysowywać jako ostatnie z nieprzezroczystej części sceny? Wynika to z pewnej kalkulacji. Jeżeli tło rysowane jest jako ostatnie, wtedy jego duże połacie będą prawdopodobnie zasłonięte już odrysowanymi obiektami, przez co duża liczba pikseli nie przejdzie testu głębokości. W efekcie etap cieniowania pikseli przetworzy mniej danych wejściowych. Jeżeli natomiast tło będzie odrysowywane jako pierwsze, to z racji pustego bufora głębokości wszystkie piksele przejdą test i zostaną poddane dalszemu przetwarzaniu.

Tradycyjnie siatkami używanymi do reprezentacji otoczenia są sześcian (ang. *skycube*), sfera (ang. *skysphere*) oraz kopuła (ang. *skydome*). Wszystkie



Rysunek 9.1. Przykładowa mapa tła

one zostały przedstawione na rysunku 9.2. Wybór używanego typu siatki uzależniony jest od algorytmu próbkowania oraz od typu mapy środowiska, która niekoniecznie musi być w tym przypadku teksturą sześcienną. Warto też zauważyć, że kopuła jest zdegenerowaną sferą pozbawioną połowy wierzchołków oraz rozciągniętą w osiach OX i OZ.



Rysunek 9.2. Przykładowe siatki otoczenia. Sześcian (a) najlepiej mapuje tekstury sześcienne, sfera (b) oraz kopuła (c) - sferyczne

9.2 Dobór skali

Ponieważ w rzeczywistości obserwatora od horyzontu do chmur dzieli bardzo duża odległość, intuicyjnie można założyć, że siatka nieba w wirtualnym świecie również powinna mieć tę właściwość. W praktyce jednak, rysowanie bardzo dużych siatek napotyka na problemy związane z odpowiednim doborem płaszczyzn przycinania opisanymi w macierzy projekcji oraz możliwością wystąpienia zjawiska *z-fightingu*, omówionego w punkcie 3.4.4.

Problemom związanym z transformacją do przestrzeni przycinania można zaradzić, ręcznie manipulując albo macierzą, albo wyliczoną pozycją. Aby uczynić obliczenia łatwiejszymi, można założyć, że środek siatki reprezentującej tło zawsze ma tę samą pozycję co obserwator. Wówczas można wykorzystać właściwość rzutu perspektywicznego sprawiającą, że dla dowolnej skali rozmieszczenie wierzchołków dowolnej siatki na płaszczyźnie ekranu będzie zawsze jednakowe, o ile obserwator ma pozycję zgodną z lokalnymi współrzędnymi $(0; 0; 0)$. Jedyną różnicą będzie inna zawartość bufora głębokości. Można więc odrysować siatkę nieba, nie przejmując się jej rozmiarem, dokonując jedynie modyfikacji w zakresie jej głębokości d, która dla punktu P w przestrzeni przycinania wyraża się wzorem:

$$d = \frac{P_z}{P_w}$$

Warunkiem osiągnięcia maksimum jest więc:

$$d = 1 \quad \Longleftrightarrow \quad P_z = P_w$$

W przypadku spełnienia powyższego warunku piksele, niezależnie od rozmiaru siatki, zostaną umieszczone na dalekiej płaszczyźnie przycinania, a więc w maksymalnej możliwej odległości od obserwatora.

9.3 Mapowanie środowiska

Owocem powyższych rozważań jest efekt, który dokonuje modyfikacji głębokości, bezpośrednio zmieniając współrzędną z z przestrzeni przycinania. Dodatkowo, ponieważ z założenia siatka używa mapy sześciennej, rolę współrzędnych tekstury pełnią lokalne pozycje wierzchołków. Ważne jest, że zgodnie z rysunkiem 8.1 owe pozycje nie muszą być normalizowane.

```
//! Tekstura skyboxa.
TextureCube g_cubeMap;
//! Liniowy sampler.
SamplerState linearSampler {
  Filter = MIN_MAG_MIP_LINEAR;
  AddressU = Wrap;
  AddressV = Wrap;
};
```

```
//! Wierzchołek wyjściowy.
struct VertexOutput {
    float4 clipPosition : SV_POSITION;
    float3 texcoords : TEXCOORD;
};
//!
VertexOutput VSSkybox(in float3 position : POSITION) {
  VertexOutput output;
  // współrzędne tekstury - pozycja w lokalnym układzie odniesienia
  output.texcoords  = position;
  // ponieważ s=z/w, to aby wierzchołek był na far plane: 1=z/w -> z = w
  output.clipPosition = mul(float4(position, 1.0f), g_worldViewProjection).
      xyww;
  return output;
}
//!
float4 PSSkybox(VertexOutput input) : SV_Target {
    return g_cubeMap.Sample(linearSampler, input.tex);
}
```

Zwróćmy teraz uwagę na dwie pomijane dotąd kwestie. Pierwszą z nich jest modyfikacja funkcji tekstu głębokości. Standardowo test przechodzą wartości mniejsze niż aktualnie przechowywana w buforze, więc gdy będzie on co ramkę czyszczony wartością 1, wówczas wszystkie piksele wygenerowane przez zaproponowany efekt zostaną odrzucone. Najbezpieczniejszym i najsolidniejszym rozwiązaniem tego problemu jest zmiana funkcji głębokości tak, aby uznawała również wartości równe bieżącej. Alternatywnie można również przeskalować przekształconą współrzędną z tak, aby iloraz $\frac{z}{w}$ był nieznacznie mniejszy od jedności. Drugi problem związany jest z domyślnym zachowaniem potoku, który polega na odrzucaniu na etapie rasteryzera wewnętrznych ścian siatek. Ponieważ tym razem to właśnie one mają być wyświetlone, można albo przy przygotowywaniu siatki uwzględnić to wymaganie, albo zmienić domyślne zachowanie Direct3D na eliminowanie zewnętrznych ścian. Uniwersalnym, ale potencjalnie mało wydajnym rozwiązaniem jest pozwalanie na odrysowanie obu typów ścian.

```
RasterizerState RSCullFront {
    // przednie ściany usuwane
    CullMode = FRONT;
}; DepthStencilState DSSLessEqual {
    // test będą przechodzić piksele o głębokości równej bieżącej
    DepthFunc = LESS_EQUAL;
}; technique10 Skybox {
    pass P0
    {
        SetVertexShader( CompileShader( vs_4_0, VSSkybox() ) );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_4_0, PSSkybox() ) );
        SetRasterizerState(RSCullFront);
        SetDepthStencilState(DSSLessEqual, 0);
    }
}
```

Rysunek 9.3 przedstawia przykład użycia map otoczenia w kontekście nieba sferycznego.

Rysunek 9.3. Przykład renderowania tła na podstawie mapy otoczenia

Wpływać na głębokość pikseli można też za pomocą samej macierzy projekcji. Trzeba wtedy wyznaczyć macierz dla nieskończenie odległej dalekiej płaszczyzny przycinania. Dla projekcji perspektywicznej (roz. 2.6.3):

$$\lim_{z_f \to +\infty} \begin{bmatrix} ctg\frac{\alpha_x}{2} & 0 & 0 & 0 \\ 0 & ctg\frac{\alpha_y}{2} & 0 & 0 \\ 0 & 0 & \frac{z_f}{z_f - z_n} & 1 \\ 0 & 0 & \frac{-z_f z_n}{z_f - z_n} & 0 \end{bmatrix} = \begin{bmatrix} ctg\frac{\alpha_x}{2} & 0 & 0 & 0 \\ 0 & ctg\frac{\alpha_y}{2} & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & -z_n & 0 \end{bmatrix}$$

Wówczas:

$$P_{proj,z} = P_{view,z} - z_n P_{view,w} = P_{view,z} - z_n$$
$$P_{proj,w} = P_{view,z}$$

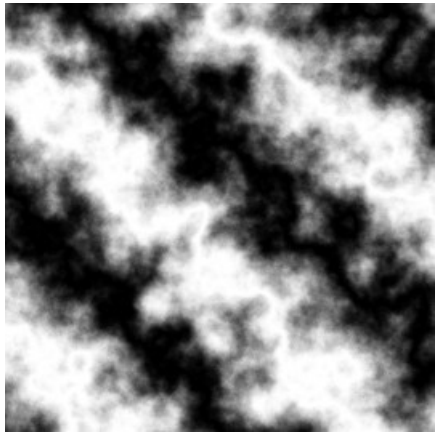
Wobec powyższego wartość głębokości będzie dążyć do jedności wraz ze wzrostem współrzędnej z. Ponieważ ta metoda gwarantuje, że wierzchołki zawsze będą znajdować się przed daleką płaszczyzną przecinania, nadaje się do rysowania siatek w ogromnej skali. Dodatkowo, żeby uniknąć błędów wynikających z braku precyzji liczb zmiennoprzecinkowych, podczas wyznaczania omawianej macierzy uwzględnia się pewną niewielką stałą.

```
void MakeInifniteFarPlane(D3DXMATRIX& projection) {
  const float epsilon = 0.00001f;
  float nearPlane = projection._43 / projection._33;
  projection._33 = 1 - epsilon;
  projection._43 = (epsilon - 1) * nearPlane;
}
```

10

Mapy przemieszczeń

Mapa wysokości (ang. *heightmap*) to rastrowy obraz z jednym kanałem, gdzie każdy piksel zawiera informację o odległości od pewnej powierzchni. Można go przedstawić jako bitmapę w skali szarości, gdzie czarny kolor oznacza minimalną odległość, natomiast biały maksymalną, tak jak przedstawiono na rysunku 10.1. Z pomocą mapy wysokości można więc przekazać informację o położeniu ogromnej liczby nieistniejących jeszcze wierzchołków względem pewnego punktu odniesienia. Tradycyjnie technika ta jest wykorzystywana do szybkiego tworzenia pofałdowanego terenu – generowana jest prostokątna gęsta siatka, której wierzchołki przesuwane są zgodnie z wektorem normalnym płaszczyzny o odczytaną wartość. Obecnie może być również wykorzystywana alternatywnie lub komplementarnie do mapowania normalnych w celu oddania nierówności w ramach płaskiego prymitywu.



Rysunek 10.1. Przykładowa mapa wysokości wygenerowana przez program Blender

10.1 Nanoszenie przemieszczeń

Sama logika nanoszenia przemieszeń jest stosunkowo łatwa w porównaniu z poprzednio omawianymi efektami. Wektor przemieszczenia pozycji wierzchołka określony jest wzorem:

$$\vec{D} = f_D(C_D - 0,5) \cdot \vec{N}$$
$$C_D \in < 0; 1 >$$

gdzie:

$\vec{D}$ - wektor normalny z mapy normalnych
f_D - współczynnik przemieszczenia
C_D - wartość odczytana z mapy przemieszczeń
N - wektor normalny

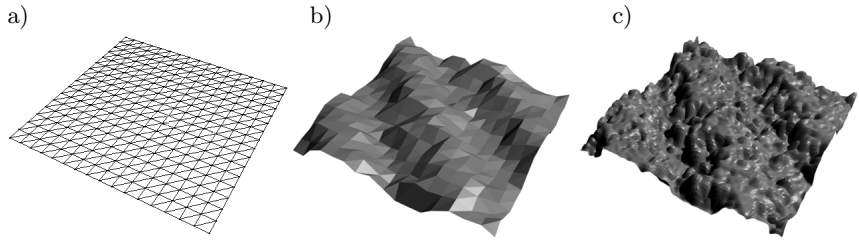
Niestety, przesunięcie przy tej samej wartości współczynnika dla przestrzeni lokalnej i świata może generować różne rezultaty. Dzieje się tak dlatego, gdyż w drugim przypadku nie byłaby uwzględniona składowa skali transformacji. Dlatego też można albo nadać przesunięcie lokalnej pozycji, albo też użyć nieznormalizowanego rezultatu mnożenia wektora normalnego razy macierz świata (długość wektorów jest poddawana skalowaniu). Poniżej, przykładowa implementacja pierwszego rozwiązania w jednostce cieniowania wierzchołków.

```
VertexOutput VSGeneric(VertexInput input) {
  // czy trzeba przemieścić?
  if ( mat_displacementMapEnabled ) {
    // przesunięcie w przestrzeni lokalnej
    float disp = mat_displacementMap.SampleLevel(linearSampler, input.tex,
        0).r-0.5;
    input.pos += input.normal * disp * mat_displacementScale;
  }
  ...
}
```

Szybko się okazuje po obserwacji rezultatów, że nanoszenie przemieszczeń w jednostce wierzchołków ma bardzo ograniczone zastosowanie. Wynika to z prostego faktu: aby uzyskać interesujący efekt, konieczna jest odpowiednio wysoka, w stosunku do rozmiaru siatki na ekranie, częstotliwość próbkowania tekstury. Warunek ten wyczerpuje etap cieniowania pikseli, jednak jedna cecha całkowicie go wyklucza – nie ma możliwości zmiany pozycji piksela. Można więc albo pogodzić się z małą widowiskowością techniki (warto przy tym zmniejszyć rozdzielczość mapy przemieszczeń, aby uniknąć aliasingu), albo dostarczyć potokowi większą ilość wierzchołków. W dalszej części rozdziału omówione będzie drugie rozwiązanie, a przykład jego obrazuje rysunek 10.2.

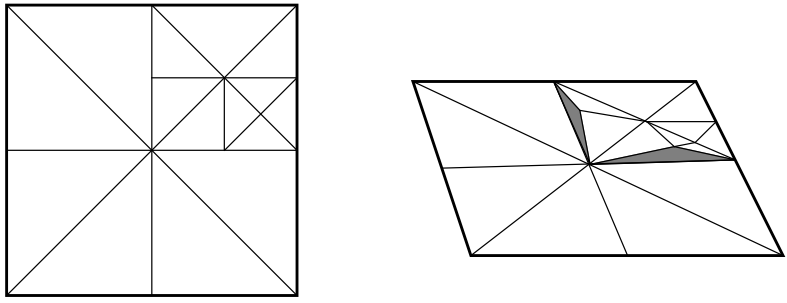
10.2 Algorytmy kafelkowania

Proces kafelkowania (ang. *tessellation*) to podział pewnej powierzchni na rozłączne wielokąty tak, aby wypełniały one w całości jej pole. W grafice komputerowej znajduje zastosowanie w sytuacji, gdy wielokąty, z których zbudowane



Rysunek 10.2. Proces generacji przykładowego terenu przez program Blender z użyciem mapy wysokości z rysunku 10.1:
a) siatka wejściowa
b) siatka po przekształceniach
c) siatka po dodatkowym podziale i przekształceniach

są siatki, nie mogą być wprost odrysowane przez kartę graficzną - dokonuje się wtedy ich podziału na wieloboki o mniejszej ilości krawędzi. Kafelkowanie wykorzystuje się również do podziału trójkątów na mniejsze. W tym przypadku, w odróżnieniu od poprzedniego przykładu, konieczne staje się wprowadzanie nowych wierzchołków. Dzięki temu więcej punktów trafia do odrysowania, co przy odpowiednim wykorzystaniu tego faktu prowadzić może do większej szczegółowości siatek.



Rysunek 10.3. Sąsiadujące trójkąty o różnym stopniu podziału. Nieciągłość powstaje gdyż wierzchołki mniejszych trójkątów nie są współdzielone z większymi [12]

Jednym z kryteriów wyboru algorytmu kafelkowania jest zadecydowanie, czy wszystkie trójkąty siatki mają być dzielone w równym stopniu. W przypadku twierdzącej odpowiedzi problem polega na błędach nieciągłości, które mogą powstać, gdy ta sama krawędź z punktu widzenia dwóch prymitywów ma różne stopnie podziału. Jest to szczególnie dotkliwe w przypadku mapowania przemieszczeń, co obrazuje rysunek 10.3. Wobec powyższego algorytmy można podzielić na trzy grupy:

- jednolite (ang. *uniform*) dzielące każdą krawędź na równą liczbę odcinków,
- niejednolite (ang. *non-uniform*) potencjalnie dzielące każdą krawędź na dowolną liczbę odcinków,
- częściowo jednolite (ang. *semi-uniform*) dzielące w sposób jednolity, ale stosujące algorytmy łączenia połączeń (ang. *transition stiching*).

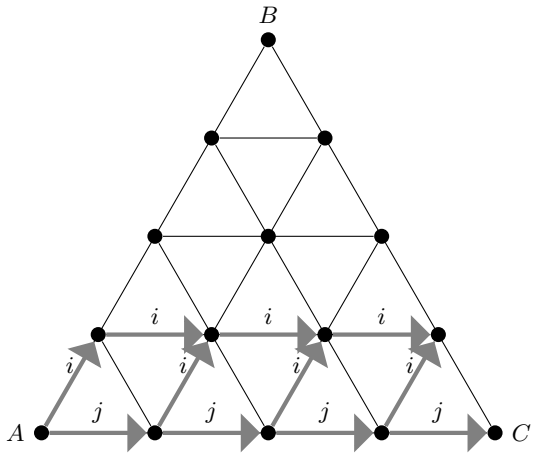
Do zaprezentowania koncepcji oraz przykładowych wyników kafelkowania wystarczający jest wariant jednolity. Zastosowany algorytm dla trójkąta ABC przy stopniu podziału n-tego polega na iteracyjnym dodawaniu wyznaczonych wcześniej wektorów $\vec{i}$ oraz $\vec{j}$ określonych następująco:

$$\vec{i} = \frac{B - A}{n + 1}$$
$$\vec{j} = \frac{C - A}{n + 1}$$

Algorytm sprowadza się do pętli, w której, poczynając od A, wierzchołki mniejszego trójkąta to bieżący wierzchołek oraz jego przekształcenia będące przesunięciami o wektory $\vec{i}$ i $\vec{j}$. Z każdym krokiem punkt odniesienia przesuwany jest o wektor $\vec{j}$, aż do momentu znalezienia się poza trójkątem źródłowym. W tej sytuacji należy wrócić do początku oraz pierwszy wierzchołek odniesienia przesunąć o wektor $\vec{i}$, a następnie powtórzyć całą procedurę. Pseudokod algorytmu przedstawiony jest poniżej, natomiast graficzną interpretację prezentuje rysunek 10.4.

```
n - stopień podziału
A, B, C - wierzchołki trójkąta
i := (B − A)/(n + 1)
j := (C − A)/(n + 1)
T := A
for k := 0 to n + 1 do
    X := T
    for l := 0 to n − k do
        dodajTrójkąt( X, X + i, X + j )
        dodajTrójkąt( X + i, X + j, X + i + j )
        X := X + j
    end for
    dodajTrójkąt( Y, Y + i, Y + j )
    T := T + j
end for
```

Algorytm jest nieoptymalny pod względem obliczeniowym, gdyż znaczna część nowych wierzchołków wyznaczana jest dwukrotnie. Poświęcając optymalność uzyskuje się prostotę implementacji, co okaże się niebagatelną zaletą w dalszej części rozdziału. Warto zauważyć, iż łączna liczba trójkątów k wygenerowa-



Rysunek 10.4. Kafelkowanie regularne przykładowego trójkąta dla stopnia podziału $n = 3$

nych przy stopniu podziału n wynosi:

$$k = (n+1)^2 \tag{10.1}$$

10.3 Kafelkowanie statyczne czy dynamiczne?

Dysponując algorytmem kafelkowania można rozważyć, czy generowana geometria ma mieć charakter statyczny, czy dynamiczny. W uproszczeniu rezultatem działania statycznej generacji geometrii jest zbiór wierzchołków przechowywany w pamięci operacyjnej lub w pamięci karty graficznej. Oczywistą zaletą takiego rozwiązania jest możliwość jednorazowego wykonania funkcji generującej, a następnie posługiwanie się otrzymaną siatką. Jest to optymalna sytuacja, gdzie narzut obliczeniowy nawet bardzo skomplikowanego algorytmu staje się mało istotny wobec jego rzadkiego wykonywania. Niestety, potencjalne źródło zalet może stać się wąskim gardłem aplikacji. Przede wszystkim utrudnione jest uwzględnianie dynamicznie zmieniających się warunków sceny, które mogą mieć wpływ na sposób generacji siatki. Programista musi w tym przypadku rozwiązać nietrywialne problemy związane z zarządzaniem pamięcią oraz takim rozłożeniem obliczeń, żeby uniknąć skokowego spadku liczby ramek na sekundę. W skrajnym przypadku, gdy generacja z jakiegoś powodu musi odbywać się co ramkę, czas wykorzystany na zapis danych jest czasem faktycznie straconym. Na drugim końcu spektrum znajduje się całkowita niezmienność siatki w czasie – w tym przypadku należy rozważyć, czy nie jest lepiej po prostu dostarczyć aplikacji dane już odpowiednio podzielone.

Koncepcja dynamicznej generacji geometrii zakłada natychmiastowe wykorzystanie wygenerowanych danych bez ich zapisywania. Dzięki ulotności pro-

blemy w zarządzaniu pamięcią dotyczące samego procesu tworzenia nowych prymitywów są zmniejszane bądź wręcz nie występują. Dodatkowo zwiększa się elastyczność procesu generacyjnego, gdyż z definicji musi być on przygotowany na działanie „od zera" dla każdych warunków. Ulotność nakłada jednak na algorytm generujący nowe wymaganie – powinien być na tyle szybki, aby mógł być wykonany co każdą ramkę, umożliwiając przy tym aplikacji zachowanie wrażenia płynności animacji.

To, czy ostatecznie bardziej optymalny jest wariant statyczny, czy dynamiczny, zależy w dużej mierze od spodziewanej zmienności rezultatów. W dalszych przykładach przedstawiona zostanie wyłącznie druga opcja – czytelnik na podstawie rozdz. 10.2 powinien sam być w stanie przeprowadzić proces kafelkowania statycznego. Dodatkowo dynamiczny wariant kładzie większy nacisk na programowanie jednostki graficznej.

10.4 Kafelkowanie jednostką cieniowania geometrii

Jednostka cieniowania geometrii, oprócz możliwości operowania na całych prymitywach, jako jedyny etap potoku renderowania jest zdolna do generacji dodatkowych wierzchołków. Co więcej, typ generowanych prymitywów może zupełnie odbiegać od wejściowych. Z programistycznego punktu widzenia danymi wyjściowymi programu cieniowania geometrii jest strumień prymitywów dowolnego typu. Cechy te sprawiają, że jednostka ta może zostać użyta w procesie dynamicznej generacji geometrii, a w szczególności do dynamicznego kafelkowania.

Jak się jednak szybko okazuje, kafelkowanie bazujące na cieniowaniu geometrii napotyka dwa bardzo poważne ograniczenia. Pierwszym jest niski w kontekście kafelkowania limit rozmiaru danych, które mogą być przesłane do rasteryzatora per prymityw wejściowy. Obecnie wynosi on 4kB, wobec czego liczba możliwych do wygenerowania trójkątów wyraża się wzorem:

$$N_t = \lfloor \frac{4096}{S_v * 3} \rfloor$$

gdzie:

N_t - maksymalna liczba wygenerowanych trójkątów
S_v - rozmiar wierzchołka (w bajtach)

Dotychczas typ danych wejściowych dla cieniowania pikseli miał rozmiar 64B. Oznacza to, że liczba wynikowych trójkątów nie może być większa niż 21. Używając odwrotności zależności 10.1 i zakładając, że stopień podziału musi być liczbą całkowitą, możemy wykazać, że zgodnie z zależnością

$$n = \lfloor \sqrt{k} - 1 \rfloor$$

maksymalny stopień podziału w tej sytuacji wynosi 3, co odpowiada 16 trójkątom wyjściowym. Liczby te mogą oczywiście ulec zmianie, jeżeli zmniejszy się

rozmiar danych związanych z wierzchołkiem. Ponieważ wymagałoby to rezygnacji z części już zaimplementowanych efektów, w załączonych przykładach nie zdecydowano się na taki krok.

> Naturalnym sposobem na zwiększenie liczby trójkątów wyjściowych przy tej samej liczbie wierzchołków byłoby zastosowanie topologii paska trójkątów zamiast listy. Niestety, jednostka cieniowania geometrii nie może emitować danych w postaci pasków.

Drugim ograniczeniem, które należy wziąć pod uwagę, to niska wydajność jednostki cieniowania dla dużej liczby danych wyjściowych. Sytuacja taka ma źródło w fakcie, iż karta graficzna została stworzona do przetwarzania równoległego strumienia danych.

10.4.1 Modyfikacja jednostki wierzchołków

Implementację należy rozpocząć od modyfikacji programu cieniowania wierzchołków tak, by nie nanosił przemieszczenia. Żeby nie tracić poprzedniej funkcjonalności, a jednocześnie nie dublować kodu, najlepiej wprowadzić parametr `uniform`, którego wartość ustalana jest na etapie kompilacji jednostek. Dzięki temu wszystkie nieużywane gałęzie będą automatycznie wyeliminowane.

```
VertexOutput VSGeneric(VertexInput input, uniform bool displace) {
  // czy trzeba przemieścić?
  if ( displace && mat_displacementMapEnabled ) {
    float disp = mat_displacementMap.SampleLevel(linearSampler, input.tex,
        0).r-0.5;
    input.pos += input.normal * disp * mat_displacementScale;
  }
  ...
```

Drugim, jeszcze nieoczywistym, dodatkiem do jednostki cieniowania wierzchołków jest obliczanie długości wektora w przestrzeni świata. Wartość ta jest zapisywana do nowego pola `VertexOutput.worldNormalLength`, a jej przeznaczenie zostanie omówione w dalszej części rozdziału.

```
  VertexOutput output;
  ...
  // obrót i~skalowanie wektora normalnego w~przestrzeni świata
  output.worldNormal = mul(input.normal, (float3x3)g_world);
  // długość wektora normalnego (jego skala)
  output.worldNormalLength = length(output.worldNormal);
  ...
  return output;
}
```

10.4.2 Jednostka geometrii

Ponieważ w jednostce cieniowania geometrii następować będzie praktyczna realizacja algorytmu z punktu 10.2, konieczne jest rozpatrzenie, jakie podstawowe operacje muszą zostać zapewnione. W kontekście pseudokodu algorytmu

używane są operacje odejmowania, dodawania oraz mnożenia (jako odwrotności dzielenia) razy skalar. Odpowiednie operatory działają dla wbudowanych typów HLSL, lecz nie dla typu `VertexOutput`. Ponieważ nie ma możliwości przeładowania operatorów jak w C++, dodano odpowiednie funkcje (ponieważ są one trywialne, rozwinięto tylko jedną z nich):

```
//! \return Wetkor, którego komponenty są sumą komponentów lhs i~rhs
VertexOutput VertexAdd(VertexOutput lhs, VertexOutput rhs) {
  VertexOutput res;
  res.clipPosition = lhs.clipPosition + rhs.clipPosition;
  res.texcoords = lhs.texcoords + rhs.texcoords;
  res.worldNormal = lhs.worldNormal + rhs.worldNormal;
  res.worldPosition = lhs.worldPosition + rhs.worldPosition;
  res.worldTangent = lhs.worldTangent + rhs.worldTangent;
  res.worldNormalLength = lhs.worldNormalLength + rhs.worldNormalLength;
  return res;
}

//! \return Wetkor, którego komponenty są róznicą komponentów lhs i~rhs
//!         pomnożoną razy scale
VertexOutput VertexSubAndScale(VertexOutput lhs, VertexOutput rhs,
float scale);
```

Dysponując funkcjami operującymi na wierzchołkach, można przejść do faktycznej implementacji algorytmu. Przed nagłówkiem jednostki należy zdefiniować, za pomocą atrybutu `maxvertexcount` maksymalną liczbę wierzchołków wyjściowych – tutaj wyznaczona jest ona zgodnie z maksymalnym możliwym stopniem podziału.

```
// 16 trójkątów * 3 wierzchołki każdy
[maxvertexcount(3*16)] void GSGeneric( triangle VertexOutput
input[3],
                inout TriangleStream<VertexOutput> output )
{
  int n = mat_tessellationFactor;
  [branch]
  if ( !n ) {

    // trywialny przypadek, pomijany algorytm
    AddTriangle(input[0], input[1], input[2], output);
  } else {

    // wyznaczenie delt
    float factor = 1.0/(n+1);
    VertexOutput delta1 = VertexSubAndScale(input[1], input[0], factor);
    VertexOutput delta2 = VertexSubAndScale(input[2], input[0], factor);

    // zmienne stanu
    VertexOutput T = input[0];
    VertexOutput v0, v1, v2;
    for (int k = 0; k < (n + 1); ++k) {
      v0 = T;
      for (int l = 0; l < (n - k); ++l) {
        v1 = VertexAdd(v0, delta1);
        v2 = VertexAdd(v0, delta2);
        AddTriangle(v0, v1, v2, output);
        v0 = VertexAdd(v1, delta2);
        AddTriangle(v1, v0, v2, output);
        output.RestartStrip();
        v0 = v2;
      }
```

```
      v1 = VertexAdd(v0, delta1);
      v2 = VertexAdd(v0, delta2);
      AddTriangle(v0, v1, v2, output);
      T = VertexAdd(T, delta1);
    }
  }
}
```

W bieżącym przykładzie jednostka cieniowania geometrii pobiera i zwraca wierzchołki tego samego typu, tj. `VertexOutput`. w praktyce tylko typ danych wejściowych musi się zgadzać z rezultatem działania jednostki wierzchołków. Format danych wyjściowych jest dowolny, ale on z kolei musi się zgadzać z typem danych przyjmowanym przez etap pikseli.

Do wyjaśnienia więc pozostała tylko funkcja `AddTriangle`. Jej ogólna postać, bez uwzględnienia kodu odpowiedzialnego za przemieszczenia, wygląda następująco:

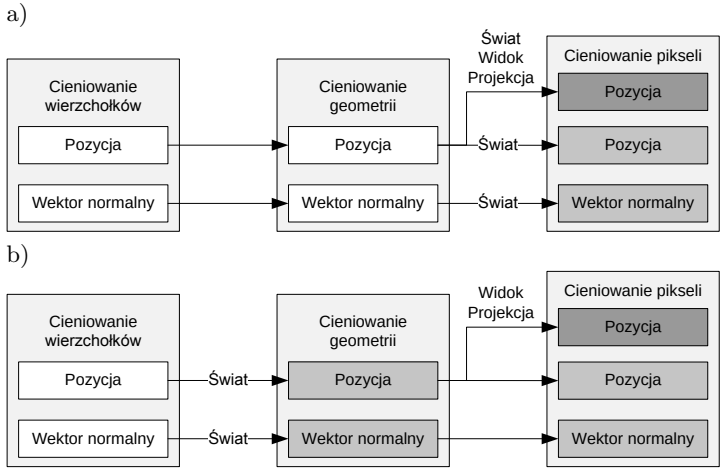
```
void AddTriangle( VertexOutput v0, VertexOutput v1, VertexOutput v2,
                  inout TriangleStream<VertexOutput> output )
{
  // nadanie przemieszczeń wierzchołkom
  if ( mat_displacementMapEnabled ) {
    ...
  }
  // przesłanie trójkąta do rasteryzatora
  output.Append(v0);
  output.Append(v1);
  output.Append(v2);
  output.RestartStrip();
}
```

Przed zaprezentowaniem implementacji przemieszczenia należy przeanalizować sytuację pod kątem uwag z punktu 10.1 oraz rozważyć, z której przestrzeni najefektywniej skorzystać.

Na pozór najlepszym rozwiązaniem byłoby zmodyfikowanie programu wierzchołków tak, aby nie transformowały pozycji ani wektora normalnego z przestrzeni lokalnej do układu odniesienia świata. Dzięki temu obliczenia w jednostce geometrii można by przeprowadzić analogicznie jak w 10.1, a dopiero po wszystkim dokonać transformacji macierzami świata i świata-widoku-projekcji (rys. 10.5a). Problem z tym rozwiązaniem polega na fakcie, że jednostka cieniowania geometrii generuje kolejne wierzchołki w sposób sekwencyjny. Gdy ich liczba jest wysoka, to potok renderowania zostaje wstrzymany do momentu zakończenia tego etapu. Dlatego w przypadku kafelkowania, gdy już samo obciążenie cieniowania geometrii będzie bardzo duże, dokładanie trzech mnożeń macierzowych per wierzchołek wynikowy prowadzi do wyraźnego spadku wydajności.

Wobec powyższego obliczenia związane z transformacją powinno się pozostawić w jednostce cieniowania wierzchołków, a na etapie geometrii operować w przestrzeni świata. Dzięki temu, gdy już nada się właściwe przemieszczenie, przejście do przestrzeni projekcji sprowadza się do mnożenia razy macierz wi-

dok-projekcja, co zaś skutkuje tylko jedną taką operacją per wierzchołek (rysunek 10.5b). Jak zostało zaznaczone w 10.1, algorytm przemieszczający operujący w przestrzeni świata powinien uwzględniać składową skali transformacji model-świat. Informacja o niej dostępna jest za pośrednictwem nienormalizowanego wyniku mnożenia lokalnego wektora normalnego razy macierz świata. Niestety, informacja o skali zniekształcana jest w drodze interpolacji dwuliniowej, która jest pośrednio używana przez algorytm kafelkowania do wyznaczania właściwości nowo powstałych wierzchołków. Istotę zniekształceń przedstawia rysunek 6.9, zaprezentowany podczas implementacji modeli oświetlenia.



Rysunek 10.5. Składanie transformacji dla dynamicznego kafelkowania umożliwiającego nanoszenie przemieszczeń:
a) przemieszczenie w układzie lokalnym; b) przemieszczenie w układzie świata

Rozwiązaniem tego problemu jest przypisanie każdemu źródłowemu wierzchołkowi skalara, który określa długość jego wektora normalnego. Nawet jeżeli wektory normalne w wierzchołkach będą różnej długości, w wyniku skalowania dwuliniowego wielkość ta będzie miała oczekiwaną, płynnie i monotonicznie zmieniającą się wartość. To właśnie w tym celu zostało na etapie cieniowania wierzchołków wprowadzone pole `worldNormalLength`. Dysponując tą informacją, można uzupełnić brakujący fragment funkcji `AddTriangle`.

```
if ( mat_displacementMapEnabled ) {
  float d0 = mat_displacementMap.SampleLevel(linearSampler, v0.texcoords,
      0).r - 0.5;
  float d1 = mat_displacementMap.SampleLevel(linearSampler, v1.texcoords,
      0).r - 0.5;
  float d2 = mat_displacementMap.SampleLevel(linearSampler, v2.texcoords,
      0).r - 0.5;

  float3 n0 = normalize(v0.worldNormal) * v0.worldNormalLength;
  float3 n1 = normalize(v1.worldNormal) * v1.worldNormalLength;
  float3 n2 = normalize(v2.worldNormal) * v2.worldNormalLength;
```

```
  v0.worldPosition += d0 * n0 * mat_displacementScale;
  v1.worldPosition += d1 * n1 * mat_displacementScale;
  v2.worldPosition += d2 * n2 * mat_displacementScale;

  v0.clipPosition = mul(float4(v0.worldPosition,1), g_viewProjection);
  v1.clipPosition = mul(float4(v1.worldPosition,1), g_viewProjection);
  v2.clipPosition = mul(float4(v2.worldPosition,1), g_viewProjection);
}
```

Powyższa implementacja jest bardzo niekorzystna. Biorąc pod uwagę, że zgodnie z rysunkiem 10.4 środkowe wierzchołki będą współdzielone przez aż sześć trójkątów, próbkowanie tekstury, obliczanie przesunięć i normalizacja wektorów normalnych wykonywane są aż sześć razy dla tych samych danych. Kwestie wydajnościowe zostały jednak pominięte na rzecz klarowności logiki.

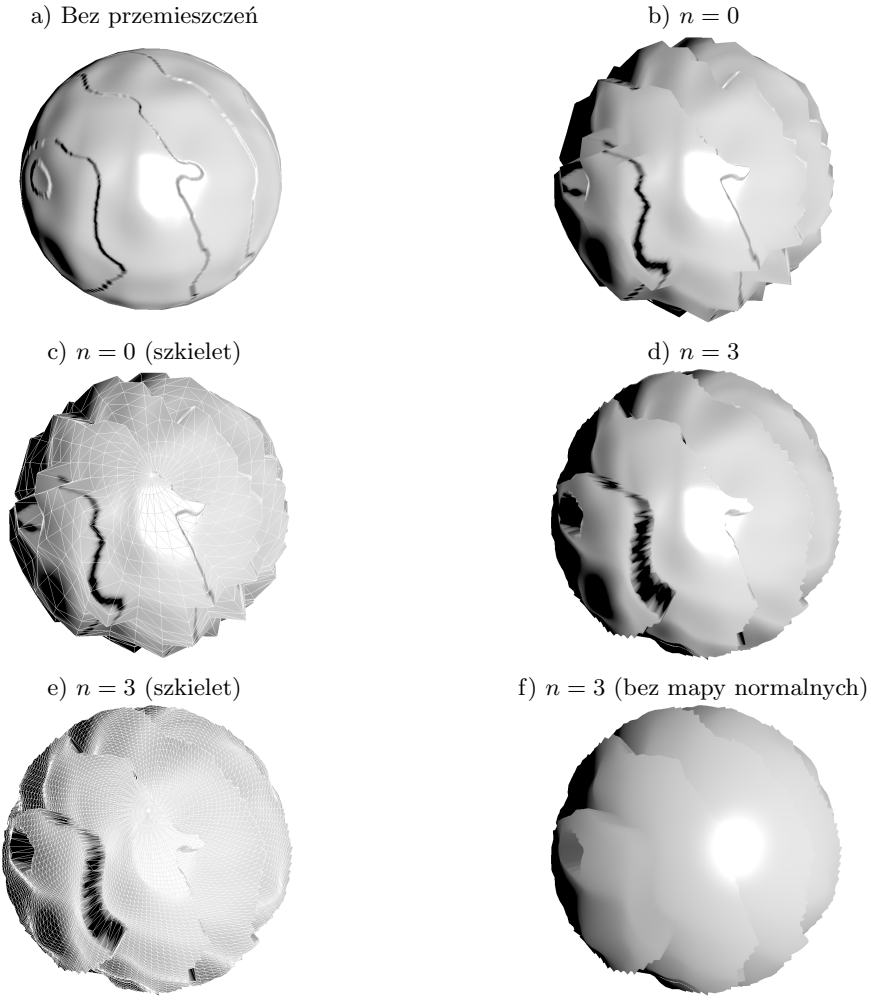
Przykładowe efekty kafelkowania i przemieszczanie pokazuje rysunek 10.6. Warto też zauważyć, że stosowanie mapy przemieszczeń bez mapy normalnych lub algorytmów wprowadzających korekcję wektora normalnego na podstawie gradientu przesunięć mija się z celem, gdyż wówczas obiekt oświetlony jest dokładnie tak, jakby był gładki (rysunek 10.6f).

10.5 Usuwanie niewidocznych ścian

Wymagania obliczeniowe kafelkowania można zmniejszyć, jeżeli nie będzie się przetwarzać niewidzialnych trójkątów. W szczególności nie powinno się dzielić trójkątów, które są zwrócone tyłem do obserwatora lub które znajdują się poza bryłą widoku. Należy tutaj wyjaśnić pewną niejasność: w rozdziale 4.2.1 zostało wspomniane, iż rasteryzator dokonuje analogicznych operacji. Po co w takim razie w ogóle rozważać tę kwestię? Otóż, operacje eliminacji niewidzialnych ścian przeprowadzane są dopiero po etapie cieniowania geometrii. Jeżeli więc niewidzialny trójkąt zostanie podzielony, zazwyczaj będzie to znaczyło, że odbyło się to na marne. Jednostka cieniowania geometrii jest więc zdana sama na siebie.

Logika kryjąca się za obiema operacjami jest stosunkowo prosta, ale wymaga dobrego zrozumienia przekształcenia do przestrzeni przycinania. Przestrzeń ta nazywa się dlatego w ten sposób, gdyż znacznie upraszczają się w niej operacje redukcji zbędnej geometrii. W dalszym ciągu rozdziału rozwijana będzie funkcja o następującej sygnaturze:

```
//! \param pos0 Pozycja pierwszego wierzcholka.
//! \param pos1 Pozycja drugiego wierzcholka.
//! \param pos2 Pozycja trzeciego wierzcholka.
//! \param tolerance Tolerancja sprawdzania.
bool IsVisible(float4 pos0, float4 pos1, float4 pos2, float
tolerance) {
    ...
}
```



Rysunek 10.6. Przykład oświetlania tej samej powierzchni z różnym modelem oświetlenia

10.5.1 Test bryły widzenia

Trójkąt jest na pewno niewidoczny, gdy wszystkie jego wierzchołki znajdują się pozą tą samą ścianą bryły widoku. Ponieważ w przestrzeni przycinania koordynaty x i y są proporcjonalne do współczynnika perspektywy w, testy na wystawanie poza obręb bryły stają się trywialne [5], co pokazuje tabela 10.1.

Co nie jest trywialne to zapis tych warunków wykonanie rodzaju iloczynu logicznego w języku HLSL. Oczywiście można zapisać sekwencję warunków, ale ideą przyświecającą temu rozdziałowi jest ogólne zmniejszenie wymagań

Tablica 10.1. Warunki wystawania poza bryłę widoku dla punktu P zdefiniowanego w przestrzeni przycinania

Płaszczyzna przycinania	Warunek
lewa	$P_x < -w$
prawa	$P_x > w$
dolna	$P_y < -w$
górna	$P_y > w$
bliska	$P_z < -w$
daleka	$P_z >= w$

obliczeniowych, a temu niestety rozgałęzienia nie służą. Posługując się wyłącznie obliczeniami na liczbach zmiennoprzecinkowych, obliczenia dla trzech wierzchołków, przy pominięciu testowania współrzędnej z, można przedstawić następująco [13]:

```
// wykonanie testów na wykraczanie poza granice płaszczyzn:
// odpowiednio (lewej, dolnej, prawej, górnej); jeżeli punkt bedzie
    wykraczał
// poza daną płaszczyznę odpowiednia współrzędna wektora outside będzie
// niezeorwa
float4 outside0 = saturate(pos0.xyxy * float4(-1,-1, 1, 1) -
pos0.w); float4 outside1 = saturate(pos1.xyxy * float4(-1,-1, 1, 1)
- pos1.w); float4 outside2 = saturate(pos2.xyxy * float4(-1,-1, 1,
1) - pos2.w);
// "iloczyn logiczny", jeżeli którykolwiek komponent != 0 - trójkąt
    niewidoczny
float4 outside = outside0 * outside1 * outside2;
```

10.5.2 Test tylnych ścian

Algorytm usuwania tylnych ścian jest praktycznie identyczny z tym omówionym w rozdz. 3.4.3 – testu dokonuje się wciąż na podstawie iloczynu skalarnego wektora normalnego płaszczyzny i wektora widoku. Różnica polega na tym, że zamiast komponentu z na jego miejsce wybiera się w. Powodem tego jest fakt, że współrzędna z obserwatora w przestrzeni przycinania nie jest równa zeru, o czym można się przekonać, mnożąc wektor [0 0 0 1] razy macierz projekcji.

```
// krawędzie trójkąta
float3 e1 = pos1.xyw - pos0.xyw; float3 e2 = pos2.xyw - pos0.xyw;
// gdy > 0 trójkąt jest niewidoczny
float NdotV = dot(cross(e1, e2), pos0.xyw);
```

Powyższy kod, chociaż w zupełności poprawny nie sprawdza się w warunkach dynamicznego kafelkowania oraz mapowania przemieszczeń zaprezentowanych w przedstawionej formie. Dochodzi bowiem do eliminacji powierzchni niemal równoległych do wektora widoku (iloczyn skalarny jest niewiele większy od zera). Jeżeli z takich trójkątów w wyniku przemieszczenia i kafelkowania wyłoni się stosunkowo wysoki szczegół, wówczas i on byłby niewidoczny, mimo iż powinien wchodzić w pole widzenia. Aby się przed tym uchronić, wprowa-

dzono współczynnik tolerancji. Niestety, wymaga on normalizacji używanych wektorów.

```
// krawędzie trójkąta
float3 e1 = pos1.xyw - pos0.xyw; float3 e2 = pos2.xyw - pos0.xyw;
float3 n = normalize(cross(e1, e2)); float3 n = normalize(pos0.xyw);
float NdotV = dot(n, v);
```

10.5.3 Podsumowanie

Ostateczny rezulat funkcji określany jest wyrażeniem:

```
return !any(outside) && NdotV < tolerance;
```

Poprawiona, uwzględniająca badanie widoczności postać jednostki cieniowania geometrii wygląda następująco:

```
// 16 trójkątów * 3 wierzchołki każdy
[maxvertexcount(3*16)] void GSGeneric( triangle VertexOutput
input[3],
                inout TriangleStream<VertexOutput> output )
{
  [branch]
  if ( isInViewport(input[0].pos, input[1].pos, input[2].pos, 0.5 ) {
    ...
  }
}
```

11

Instancjonowanie geometrii

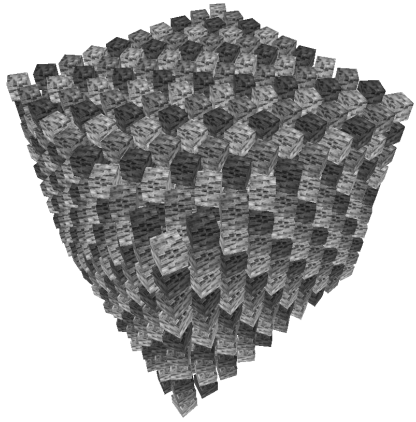
Instancjonowanie geometrii (ang. *geometry instancing*) zostało wprowadzone już w Direct3D 9, aby umożliwić szybkie, wielokrotne odrysowanie tej samej siatki za pomocą jednego wywołania funkcji rysującej. Przydatność takiego mechanizmu, biorąc pod uwagę jeden ustalony stan potoku renderowania oraz ten sam zestaw wierzchołków, byłaby wątpliwa, gdyby nie możliwość przypisywania wystąpieniom siatki (instancjom) pewnych specyficznych im właściwości. Ponieważ instancjonowanie jest zazwyczaj szybsze niż równoważna mu sekwencja wywołań funkcji rysującej, dostępny jest potencjał optymalizacyjny. Dodatkowo z uwagi na fakt, że dane bieżącej instancji są współdzielone przez wszystkie wierzchołki, istnieje sposobność na zastosowanie wzorca projektowego pyłek (ang. *flyweight desgin pattern*) [6].

Z technicznego punktu widzenia dane instancji (ang. *instance data*) to kolejny bufor wierzchołków, który jest jednak inaczej obsługiwany przez etap asemblera wejściowego. Otóż wskaźnik na bieżący element nie jest zwiększany z każdym przetworzonym wierzchołkiem, lecz dopiero po pewnej zadanej ich liczbie. Formalizując, dla liczności instancji n_i oraz n_v wierzchołków par instancja łączna - liczba odrysowanych wierzchołków będzie równa $n_i \times n_v$, a dane wejściowe będzie stanowić każda kombinacja elementów obu źródeł.

11.1 Instancjonowanie obiektów

Klasycznym zastosowaniem instancjonowania jest optymalizacja odrysowań bliźniaczych obiektów. W takich przypadkach dane instancji zawierają macierze transformujące każde wystąpienie siatki do zadanej pozycji, rotacji i skali. Przykład takiego mechanizmu obrazuje rysunek 11.1.

Instancjonowanie obiektów jest bardzo dokładnie opisane w przykładach z DirectX SDK, a przytaczanie sposobu realizacji tego zadania nie wnosiłoby żadnego pierwiastka twórczego. Zamiast tego niniejszy rozdział skupia się na nieco mniej standardowym sposobie wykorzystania tego mechanizmu. To wła-



Rysunek 11.1. Zastosowanie mechanizmu instancji. Siatka sześcianu została odrysowana 1000 razy w różnych pozycjach, o różnych kolorach i kątach obrotu przy użyciu jednego wywołania funkcji rysującej [9]

śnie nieszablonowe postrzeganie możliwości i zastosowań fragmentów API może prowadzić do unikatowych efektów graficznych.

11.2 Szablony kafelkowania

Szablon kafelkowania to zbiór trójkątów powstałych w wyniku kafelkowania barycentrycznego trójkąta równobocznego. Wierzchołki takiej figury pokrywają się z wierzchołkami sympleksu 2-wymiarowego. W celu uproszczenia zapisu przyjęto następujące oznaczenia współrzędnych barycentrycznych:

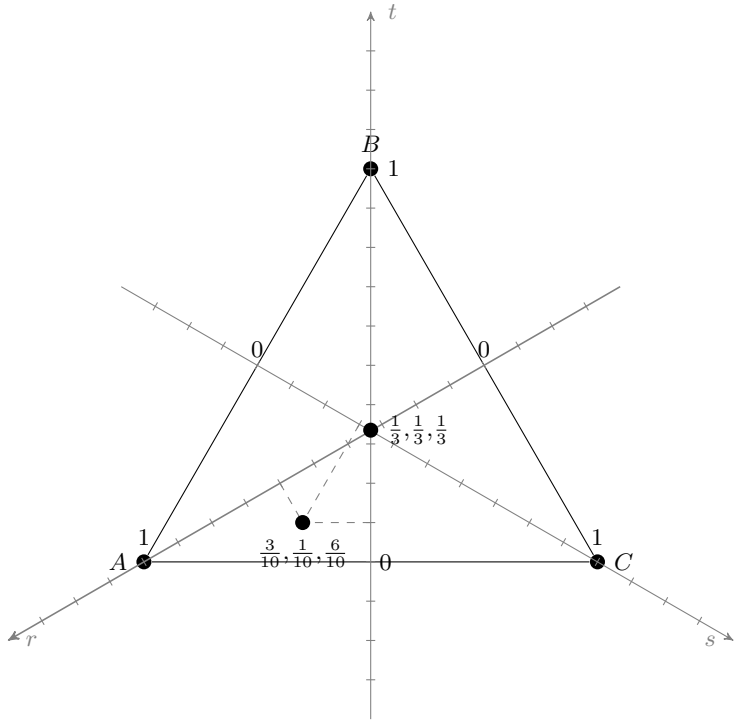
$$a_0 = s$$
$$a_1 = t$$
$$a_2 = r$$

W niniejszej książce wierzchołki równobocznego trójkąta barycentrycznego oznaczono symbolami A, B i C. Posługując się równaniem 2.9, ich współrzędne wyznaczono w następujący sposób:

$$A = (0, 0, 1)$$
$$B = (0, 1, 0)$$
$$C = (1, 0, 0)$$

Taki niekafelkowany jeszcze trójkąt wraz z układem współrzędnych barycentrycznych przedstawiony jest na rysunku 11.2.

Dysponując gotowym szablonem można w szybki i nieskomplikowany sposób dokonać podziału dowolnego trójkąta określonego w przestrzeni euklidesowej. Proces ten polega na tłumaczeniu wierzchołków pod-trójkątów szablonu



Rysunek 11.2. Barycentryczny trójkąt równoboczny. Osie współrzędnych s, t i v przecinają się w punkcie $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$. Dla przykładowego punktu $\frac{3}{10}, \frac{1}{10}, \frac{6}{10})$ zaznaczono jego rzuty na osie współrzędnych

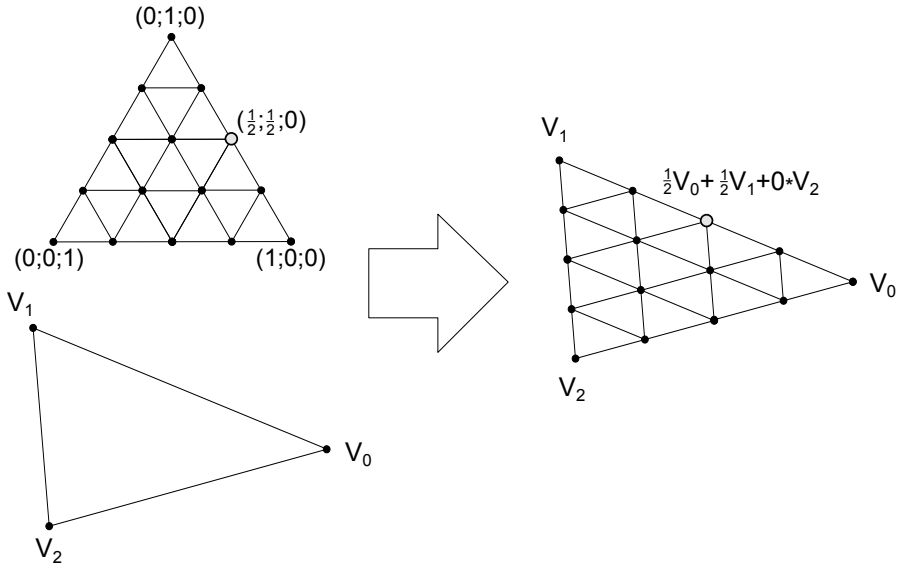
drogą barycentrycznej interpolacji na wierzchołki wynikowe, co obrazuje rysunek 11.3. Dzięki założonej normalizacji oraz równaniu 2.11 można w rozważaniach pominąć ostatnią współrzędną barycentryczną r, gdyż można ją odtworzyć na bazie pary s i t. Wobec tego, biorąc pod uwagę równanie 2.13, przełożenie pod-trójkąta szablonu na pod-trójkąt w przestrzeni euklidesowej przedstawia się następująco:

$$
\begin{aligned}
V_0' &= s_0(V_0 - V_2) + t_0(V_1 - V_2) + V_2 \\
V_1' &= s_1(V_0 - V_2) + t_1(V_1 - V_2) + V_2 \\
V_2' &= s_2(V_0 - V_2) + t_2(V_1 - V_2) + V_2
\end{aligned}
$$

gdzie:

V_0, V_1, V_2	- wierzchołki źródłowego trójkąta
V_0', V_1', V_2'	- wierzchołki wynikowego pod-trójkąta
$(s_0; t_0)$, $(s_1; t_1)$, $(s_2; t_2)$	- współrzędne barycentryczne pod-trójkąta

Pierwszą optymalizacją w stosunku do poprzednio stosowanych metod kafelkowanie jest zredukowanie kosztów obliczeniowych związanych z logiką podziału.



Rysunek 11.3. Odtwarzanie podziału trójkąta $V_0V_1V_2$ na podstawie szablonu barycentrycznego. Obliczenia dla wyróżnionego wierzchołka przeprowadzono zgodne z równaniem 2.12.

Ponieważ algorytm generacyjny wykonywany jest tylko raz, a jego rezultat zapisywany jest w postaci szablonu, może on osiągać dowolne skomplikowanie. Dodatkowo, jak zostanie wykazane w dalszej części rozdziału, zostanie wyeliminowane wąskie gardło wcześniej poznanej metody, czyli jednostka cieniowania geometrii.

W zaproponowanej realizacji typy barycentryczne przedstawiają się następująco:

```
//! Wierzchołek barycentryczny.
struct BarVertex {
  float s;
  float t;
  float r() const
  {
    return 1.0f - s - t;
  }
};
//! Trójkąt barycentryczny.
struct BarTriangle {
  BarVertex v0;
  BarVertex v1;
  BarVertex v2;
};
//! Trójkąty barycentryczne, czyli szablon.
typedef std::vector<BarTriangle> BarTriangles;
```

Funkcja generująca szablon została sporządzona na podstawie algorytmu opisanego w 10.2. Ponieważ operuje ona na współrzędnych barycentrycznych,

musi uwzględniać pewne specyficzne ich cechy, jak na przykład zawsze zerowa współrzędna dla odpowiedniej krawędzi. Warto zauważyć, iż poniższa funkcja nie czyści tablicy docelowej. Ta właściwość zostanie później wykorzystana do umieszczania wszystkich szablonów w jednym buforze.

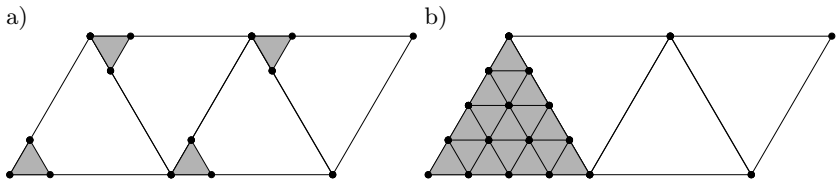
```
//! \param n Stopień podzialu.
//! \param triangles Szablon docelowy.
void GeneratePattern( int n, BarTriangles & triangles ) {
  float delta = 1.0f / (n + 1);
  // pierwszy z~małych trójkątów
  BarTriangle current = { {0.0f, 0.0f}, {delta, 0.0f}, {0.0f, delta} };
  for ( int k = 0; k < n + 1; ++k ) {
    for ( int l = 0; l < n - k; ++l ) {
      triangles.push_back(current);
      std::swap(current.v1, current.v0);
      current.v1.s = current.v2.s + delta;
      current.v1.t = current.v2.t;
      triangles.push_back(current);
      std::swap(current.v1, current.v2);
      current.v1.s = current.v0.s + delta;
      current.v1.t = current.v0.t;
    }
    triangles.push_back(current);
    // przesunięcie w~górę
    current.v0.t += delta;
    current.v1.t += delta;
    current.v2.t += delta;
    // rozpoczęcie od krawędzi
    current.v0.s = 0.0f;
    current.v1.s = delta;
    current.v2.s = 0.0f;
  }
}
```

11.3 Założenia instancjonowanego kafelkowania

Idea kafelkowania z użyciem instancjonowania sprowadza się do takiego połączenia przetwarzania wierzchołków i szablonu, aby w rezultacie osiągnąć pełne wypełnienie źródłowych trójkątów. Kombinacja obu tych źródeł, jest możliwa do osiągnięcia dzięki instancjonowaniu, z definicji wygeneruje potrzebną liczbę wierzchołków. Dzięki temu nie ma konieczności używania emisyjnych możliwości etapu cieniowania geometrii.

Przy projektowaniu rozwiązania należy zadecydować, które dane będą źródłem wierzchołków, a które instancji (rysunek 11.4). Wybór ten całkowicie determinuje sposób realizacji techniki. Przykładowo, jeżeli danymi instancji będą trójkąty szablonu, wówczas interpolacji barycentrycznej można dokonać jedynie podczas etapu geometrii, gdyż tylko wtedy znane są właściwości wszystkich trzech wierzchołków źródłowych. Dla alternatywnego rozwiązania można tego dokonać już podczas etapu wierzchołków, gdyż danymi instancji są same wierzchołki tworzące trójkąt.

W niniejszym rozdziale zaprezentowana jest przykładowa realizacja drugiej z koncepcji, ponieważ wymaga ona użycia dodatkowych, niezaprezentowanych jeszcze konstruktów Direct3D.



Rysunek 11.4. Obraz kafelkowania siatki po przetworzeniu danych dla pierwszej instancji. Na a) źródłem wierzchołków jest siatka, natomiast szablon jest źródłem instancji. Fragment b) przedstawia odwrotną sytuację

11.4 Tworzenie danych wierzchołków

Mechanizm instancjonowania nie zmienia sposobu, w jaki tworzy się dane wierzchołków. Wciąż wymagane jest stworzenie odpowiedniego bufora oraz wypełnienie go danymi. Ponieważ jednak tym razem na dane wierzchołków będą składały się szablony, należy rozważyć kilka dodatkowych okoliczności.

Nie zawsze zachodzi potrzeba silnego kafelkowania siatki, w szczególności gdy jej rzut na płaszczyznę ekranu nie jest sporych rozmiarów. Tym razem nie jest też możliwa kontrola parametrem `mat_tessellationFactor` na poziomie efektu, ponieważ generacja nowych wierzchołków wykonywana jest przez etap asemblera wejściowego poprzez mechanizm instancjonowania. W związku z tym, żeby zapewnić elastyczność, konieczne jest przygotowanie wielu szablonów oraz wybranie z nich tego, który charakteryzuje się zadanym stopniem podziału. Wbrew pozorom nie trzeba tworzyć w tym celu wielu buforów, ponieważ wszystkie dane można zapamiętać w postaci ciągłego obszaru pamięci, następnie wyznaczyć indeks bazowy oraz liczebność wierzchołków dla konkretnego szablonu. W tym przypadku tworzenie bufora wyglądać może następująco:

```
HRESULT CreatePatternsBuffer(ID3D10Device* device, int maxN,
BarTriangles& triangles, ID3D10Buffer *& buffer) {
  HRESULT hr;
  // generacja i~akumulacja szablonów
  triangles.clear();
  for (int i~= 1; i~< maxN; ++i) {
    GeneratePattern(i, triangles);
  }
  // tworzenie zbiorczego bufora wierzchołków
  D3D10_BUFFER_DESC bufferDesc = {
    triangles.size() * 3 * sizeof(BarVertex),
    D3D10_USAGE_IMMUTABLE,
    D3D10_BIND_VERTEX_BUFFER,
    0, 0
  };
  D3D10_SUBRESOURCE_DATA initData = {
    &triangles.front(), 0, 0
  };
  V_RETURN( device->CreateBuffer(&bufferDesc, &initData, buffer) );
  return S_OK;
}
```

Liczebność trójkątów, a więc pośrednio wierzchołków, dla zadanego stopnia podziału oraz wybranego algorytmu jest określona wzorem 10.1. Indeks trójkąta rozpoczynającego szablon, z powodu uporządkowania bufora względem stopnia podziału, można sformalizować jako:

$$i_n = \sum_{i=0}^{n-1}(i+1)^2 = \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

gdzie:

i_n - indeks pierwszego trójkąta szablonu
n - stopień podziału szablonu

Wzór ogólny na sumę kwadratów liczb naturalnych można wyznaczyć na podstawie następującego wyprowadzenia:

$$\sum_{i=0}^{n} i^3 + (n+1)^3 = \sum_{i=0}^{n}(i+1)^3 = \sum_{i=0}^{n} i^3 + 3\sum_{i=0}^{n} i^2 + 3\sum_{i=0}^{n} i + (n+1)$$

Po przyrównaniu skrajnych stron szukany wzór można wyprowadzić następująco:

$$\sum_{i=0}^{n} i^2 = \frac{1}{3}\left((n+1)^3 - 3\sum_{i=0}^{n} i - (n+1)\right) = \frac{n(n+1)(2n+1)}{6}$$

Powyższe zależności zaimplementowano w następujący sposób:

```
//! \return Indeks pierwszego trójkąta z~szablonu dla stopnia n.
inline int GetPatternIdx(int n) {
  return n*(n+1)*(2*n+1)/6;
}
//! Liczba trójkątów dla stopnia n.
inline int GetPatternLength(int n) {
  return (n+1)*(n+1);
}
```

11.5 Tworzenie danych instancji

Z technicznego punktu widzenia bufor instancji to kolejny bufor wierzchołków. Tym razem jednak jeden element bufora musi w całości opisywać trójkąt źródłowy. Wynika to z faktu, że do przeprowadzenia interpolacji barycentrycznej konieczne są informacje o wszystkich wierzchołkach. Jednostka cieniowania geometrii w przypadku nie rozwiązuje tego problemu, gdyż zawsze trzy bieżąco przetwarzane wierzchołki będą miały te same dane instancji.

Gdy siatka nie wykorzystuje indeksów, wówczas jej bufor wierzchołków może być użyty wprost jako dane instancji. Jeżeli jednak używane jest indeksowanie, wówczas możliwe są dwa rozwiązania. Pierwszym jest przetłumaczenie indeksów na wierzchołki oraz zapełnienie tymi danymi nowego bufora. Oczywiście, w przypadku dużych siatek, dla których wierzchołki mają zdefiniowanych wiele właściwości, rozmiar wynikowej struktury może osiągnąć bardzo niepraktyczne rozmiary. Alternatywą jest potraktowanie samych indeksów jako danych instancji oraz ręczna indeksacja. Z uwagi na znacznie mniejszy narzut pamięciowy właśnie to rozwiązanie będzie rozwijane.

Przeszkodą jest fakt, że aby bufor indeksów mógł być użyty jako bufor instancji, konieczne jest podanie flagi `D3D10_BIND_VERTEX_BUFFER` podczas jego tworzenia. Niestety, poznany wcześniej interfejs `ID3DX10Mesh` nie umożliwia wybrania takich niestandardowych ustawień, wobec czego może być konieczne wykonanie kopii bufora. Za tę operację odpowiada wywołanie poniższej funkcji wywołanej z drugim parametrem równym `D3D10_BIND_VERTEX_BUFFER`:

```
HRESULT CloneBuffer( ID3D10Device* device, UINT bindFlags,
ID3D10Buffer* src, ID3D10Buffer *& dst ) {
  HRESULT hr;
  // nadpisanie niektórych ustawień
  D3D10_BUFFER_DESC bufferDesc;
  src->GetDesc(&bufferDesc);
  bufferDesc.Usage = D3D10_USAGE_DEFAULT;
  bufferDesc.BindFlags = bindFlags;
  bufferDesc.CPUAccessFlags = 0;
  bufferDesc.MiscFlags = 0;

  // tworzenie i˜skopiowanie bufora
  hr = device->CreateBuffer(&bufferDesc, NULL, &dst);
  if ( SUCCEEDED(hr) ) {
    device->CopyResource(dst, src);
  }
  return hr;
}
```

11.6 Tworzenie buforów zasobów

Skoro to indeksy będą danymi instancji należy rozwiązać problem dostarczenia faktycznych danych geometrii. Można oczywiście w tym celu stworzyć tablice w buforach stałych, zakładając przy tym pewną maksymalną liczbę wierzchołków możliwych do przetworzenia. Należy jednak pamiętać, że aktualizacja takich struktur jest stosunkowo wolna, gdyż wymaga transferu danych z pamięci operacyjnej do pamięci graficznej. Wówczas, jeżeli kilka obiektów miałoby skorzystać z tego samego efektu kafelkowania, to przesył musiałby następować przed każdym odrysowaniem każdego z nich. Aby móc działać efektywnie, mechanizm musi mieć więc możliwość zapisywania i dostępu do geometrii znajdującej się bezpośrednio w pamięci karty graficznej.

Wymaganie to wyczerpują bufory zasobów cieniowania (ang. *buffer resource*). W odróżnieniu od buforów wierzchołków, których elementy przekazywane są wyłącznie jako parametry jednostki cieniowania wierzchołków, dostęp

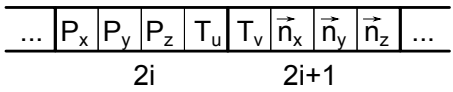
do zasobów może odbywać się bez ograniczeń w każdym programowalnym etapie potoku renderowania. Obiekty te wykazują więc cechy zmiennych globalnych, chociaż, podobnie jak w przypadku tekstur, nie można umieszczać ich w buforach stałych. Z punktu widzenia HLSL sposób użycia buforów zasobów jest, pomimo odmiennej syntaktyki, zbliżony do obsługi tablic o nieznanym rozmiarze z języków wysokiego poziomu. Różnica polega również na fakcie, że zapisu oraz modyfikacji danych można dokonać jedynie z poziomu aplikacji. W ogólnym przypadku omawiany mechanizm może być stosowany w celu przekazania dużej liczby danych do jednostek cieniowania, jeżeli nie mogą być one z jakiegoś powodu w łatwy albo wydajny sposób wprowadzone do wierzchołków.

Silnym ograniczeniem funkcjonalności buforów zasobów jest możliwość przechowywania w nich wyłącznie skalarów oraz wektorów. Gdy zachodzi potrzeba składowania bardziej rozbudowanego typu, takiego jak używany wierzchołek, możliwe są dwa rozwiązania. Pierwszym jest dekompozycja, czyli zapis każdego atrybutu do oddzielnego bufora. Takie podejście pozwala zachować przejrzystość na poziomie efektu, jednak wymaga większego nakładu pracy po stronie aplikacji. Alternatywnie można część atrybutów, których łączny rozmiar jest wielokrotnością rozmiaru typu możliwego do przechowania wprost, zapisać wspólnie i podczas odczytu odpowiednio korygować indeks i wartości. Ponieważ to podejście wiąże się z potencjalnie mniejszą liczbą odwołań do buforów, zastosowano je w omawianym przykładzie.

11.6.1 Definicja w HLSL

Używany na poziomie aplikacji typ `ObjMesh:Vertex` ma rozmiar równy 32B, co odpowiada ośmiu liczbom zmiennoprzecinkowym o pojedynczej precyzji (`float`). Można w tej sytuacji zapisać go na dwóch wektorach czterowymiarowych (rysunek 11.5), gdyż ten typ danych ma rozmiar 16B. Warto zauważyć, że dodanie tangenta (12B) do definicji wierzchołka znacznie skomplikowałoby tę sytuację – jedenastu wynikowych liczb zmiennoprzecinkowych nie da się przedstawić w postaci wielokrotności wektorów.

```
//! spakowane pozycja, współrzędne tekstury oraz wektor normalny
//! łączny rozmiar: 32 bajty
Buffer<float4> g_packedVertexData;
//! tangenty
Buffer<float3> g_tangentData;
```



Rysunek 11.5. Zapis danych wierzchołka w postaci dwóch wektorów czterowymiarowych w buforze zasobów. P - pozycja wierzchołka; T - współrzędne tekstury; $\vec{n}$ - wektor normalny; i - indeks wierzchołka

11.6.2 Dostarczanie danych

Interfejs `ID3D10EffectShaderResourceVariable` – ten sam, co tekstury reprezentuje zmienne, będące buforami zasobów. Podobieństwa na tym się nie kończą, ponieważ w omawianym przypadku również konieczne jest tworzenie widoku `ID3D10ShaderResourceView`. Różnica polega jednak na sposobie tworzenia faktycznego zasobu – w tym przypadku jest to bufor z aktywną flagą `D3D10_BIND_SHADER_RESOURCE`. Funkcja, która klonuje zawartość pewnego bufora oraz na tej podstawie tworzy widok, przedstawia się następująco:

```
HRESULT BindEffectResource(ID3D10Device* device, UINT count,
DXGI_FORMAT format, ID3D10Buffer* src, ID3D10ShaderResourceView*&
dst) {
  HRESULT hr;
  // sklonowanie bufora i~nadanie mu flagi D3D10_BIND_SHADER_RESOURCE
  CComPtr<ID3D10Buffer> buffer;
  V_RETURN(CloneBuffer(device, D3D10_BIND_SHADER_RESOURCE, src, buffer.p));
  // tworzenie widoku
  D3D10_SHADER_RESOURCE_VIEW_DESC srvDesc;
  ZeroMemory(&srvDesc, sizeof(srvDesc));
  srvDesc.Format = format;
  srvDesc.ViewDimension = D3D10_SRV_DIMENSION_BUFFER;
  srvDesc.Buffer.ElementOffset = 0;
  srvDesc.Buffer.ElementWidth = count;
  V_RETURN(device->CreateShaderResourceView(buffer, &srvDesc, &dst));
  return S_OK;
}
```

Sekwencja wywołań powyższej funkcji, prowadząca do odzwierciedlenia deklaracji buforów z punktu 11.6.1, dla siatki z buforami zorganizowanymi zgodnie z przyjętą konwencją, wyglądać może następująco:

```
ID3D10ShaderResourceView* vertexDataView; ID3D10ShaderResourceView*
tangentDataView; ... HRESULT CreateShaderResources( ID3D10Device*
device, ID3DX10Mesh* mesh ) {
  HRESULT hr;
  UINT vertexCount = mesh->GetVertexCount();
  // ponieważ wierzchołki są spakowane na dwóch wektorach, liczba
  // tych drugich musi być dwukrotnie większa!
  CComPtr<ID3D10Buffer> deviceVertexBuffer;
  V_RETURN(mesh->GetDeviceVertexBuffer(0, &deviceVertexBuffer) );
  V_RETURN(BindEffectResource(device, vertexCount * 2,
    DXGI_FORMAT_R32G32B32A32_FLOAT, deviceVertexBuffer, vertexDataView));
  // tworzenie widoku dla tangetna
  deviceVertexBuffer = NULL;
  V_RETURN(mesh->GetDeviceVertexBuffer(1, &deviceVertexBuffer) );
  V_RETURN(BindEffectResource(device, vertexCount,
    DXGI_FORMAT_R32G32B32_FLOAT, deviceVertexBuffer, tangentDataView));
  return S_OK;
}
```

Gwoli ścisłości, poniżej przedstawiono przykładowy sposób pobrania zmiennych efektu reprezentujących bufory zasobów:

```
ID3D10EffectShaderResourceVariable* vertexDataVariable;
ID3D10EffectShaderResourceVariable* tangentDataVariable;
ID3D10Effect* effect; ... vertexDataVariable =
effect->GetVariableByName("g_packedVertexData")->AsShaderResource();
tangentDataVariable =
effect->GetVariableByName("g_tangentData")->AsShaderResource();
```

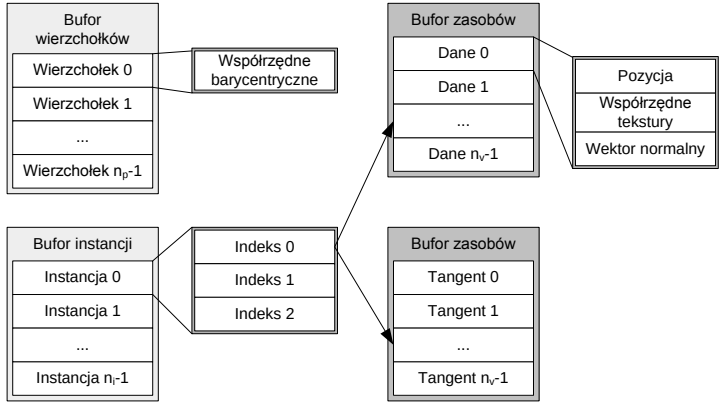
W powyższej funkcji `CreateShaderResources` wykorzystano fakt, że bufory wierzchołków można wprost przełożyć na bufory zasobów. Gdy taka sytuacja nie zachodzi, bo np. trzeba przeprowadzić dekompozycję, konieczne jest stworzenie obiektów od podstaw i inicjalizacja z użyciem wybranej porcji danych. Ekstrakcja pozycji może wyglądać następująco:

```
std::vector<D3DXVECTOR3> positions; UINT vertexCount =
mesh->GetVertexCount();
// pobranie lokalnego bufora, przechowywanego w˜pamięci operacyjnej
CComPtr<ID3DX10MeshBuffer> vertexBuffer; V_RETURN(
mesh->GetVertexBuffer(0, &vertexBuffer) );
// mapowanie wierzchołków
ObjMesh::Vertex* vertex; V_RETURN(
vertexBuffer->Map(reinterpret_cast<void**>(&vertex), NULL));
positions.resize(vertexCount); for (UINT i˜= 0; i˜< vertexCount;
++i) {
  positions[i] = vertex[i].position;
}
// zakończenie mapowania
vertexBuffer->Unmap();
```

Alternatywnie można użyć bufora znajdującego się w urządzeniu, tak jak to miało miejsce w oryginalnym przykładzie. Wówczas funkcja `Map` będzie miała nieco inną sygnaturę. Dodatkowo konieczne jest w tym przypadku sprawdzenie, czy flagi bufora pozwalają na jego odczyt (punkt 4.3.3).

11.7 Implementacja

Ogólną organizację danych w kafelkowaniu z użyciem instancjonowania przedstawia rysunek 11.6. Skoro odpowiednie struktury danych istnieją i są gotowe do użycia, można przystąpić do implementacji logiki zdolnej je spożytkować.



Rysunek 11.6. Organizacja danych w kafelkowaniu z użyciem instancjonowania szablonów

11.7.1 Struktura danych wejściowych

Ponieważ całkowicie zmienione są dane wejściowe, konieczne jest stworzenie nowego układu danych wejściowych. Istotną zmianą w stosunku do poprzednich definicji jest przypisanie polu `InputSlotClass` (przedostatnie w strukturze) wartości `D3D10_INPUT_PER_INSTANCE_DATA` – jest to jedyny wyróżnik danych instancji. Kolejne pole, czyli `InstanceDataStepRate` mówi o tym, ile instancji ma być odrysowanych przed wyborem kolejnego elementu z bufora. Typowymi podaje się tutaj zero (ta sama instancja odrysowywana wiele razy) lub 1 (jedna instancja na jedno odrysowanie). Warto dodać, że wartości dla obu tych pól muszą być jednakowe dla wszystkich elementów opisu tego samego bufora.

```
const D3D10_INPUT_ELEMENT_DESC layout[] = {
  { "POSITION", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 0,
      D3D10_INPUT_PER_VERTEX_DATA, 0 },
  { "INDICES", 0, DXGI_FORMAT_R32G32B32_UINT, 1, 0,
      D3D10_INPUT_PER_INSTANCE_DATA, 1 }
};
```

11.7.2 Modyfikacja efektu

Znając układ danych można przystąpić do aktualizacji efektu. Pomimo, iż wymaga to wprowadzenia nowego typu oraz nowej jednostki cieniowania, zakres zmian, dzięki wcześniej przestrzeganej zasadzie uniwersalności, jest stosunkowo niewielki. Szkielet nowej implementacji przedstawia się następująco:

```
struct BarycentricInput {
  float2 position  : POSITION;
  uint3 indices    : INDICES;
};

VertexOutput VSGenericBarycentric (BarycentricInput input, uniform
bool displace) {
  ...
}
```

Warto zwrócić uwagę, iż dzięki zwracaniu typu `VertexOutput` oraz przyjmowaniu parametru `displace` jednostka jest całkowicie zgodna z wyprowadzonymi już programami geometrii i pikseli.

Pierwszym krokiem ku osiągnięciu zamierzonego efektu jest odczyt danych z buforów. Zgodnie z rysunkiem 11.5, w przypadku wierzchołków wymagana jest korekcja indeksu oraz dwukrotny odczyt z bufora.

```
// pobranie danych wierzchołków
float4 v0a = g_packedVertexData.Load(input.indices.x*2); float4 v0b
= g_packedVertexData.Load(input.indices.x*2+1); float4 v1a =
g_packedVertexData.Load(input.indices.y*2); float4 v1b =
g_packedVertexData.Load(input.indices.y*2+1); float4 v2a =
g_packedVertexData.Load(input.indices.z*2); float4 v2b =
g_packedVertexData.Load(input.indices.z*2+1);
// pobranie tangentów
float4 tan0 = g_tangentData.Load(input.indices.x); float4 tan1 =
g_tangentData.Load(input.indices.y); float4 tan2 =
g_tangentData.Load(input.indices.z);
```

Sama interpolacja barycentryczna może być skrótowo zapisana jako mnożenie wektora współczynników razy macierz atrybutów wierzchołka ułożonych wierszami. Wniosek taki nasuwa się po przeanalizowaniu równań 2.3 oraz 2.12. Wobec tego interpolacja parametrów wierzchołka może przybrać następującą postać:

```
VertexInput vertex;
// macierz interpolująca
float3 mult = float3(1.0f - input.position.x - input.position.y,
  input.position.x, input.position.y);
// interpolacja pozycji
vertex.position = mul(mult, float3x3(v0a.xyz, v1a.xyz, v2a.xyz));
// interpolacja współrzędnych tekstury
vertex.texcoords = mul(mult, float3x2(float2(v0a.w, v0b.x),
  float2(v1a.w, v1b.x), float2(v2a.w, v2b.x)));
// interpolacja wektora normalnego
vertex.normal = mul(mult, float3x3(v0b.yzw, v1b.yzw, v2b.yzw));
// interpolacja tangenta
vertex.tangent = mul(mult, float3x4(tan0, tan1, tan2));
```

Zapisanie rezultatów do struktury `VertexInput` nie służy wyłącznie zaznaczeniu zakończenia procesu odtwarzania danych geometrycznych. Dzięki takiemu zabiegowi można, zamiast kodować na nowo obliczenia i przekształcenia wierzchołka, po prostu wywołać wcześniej rozwijaną jednostkę:

```
return VSGeneric(vertex, displace);
```

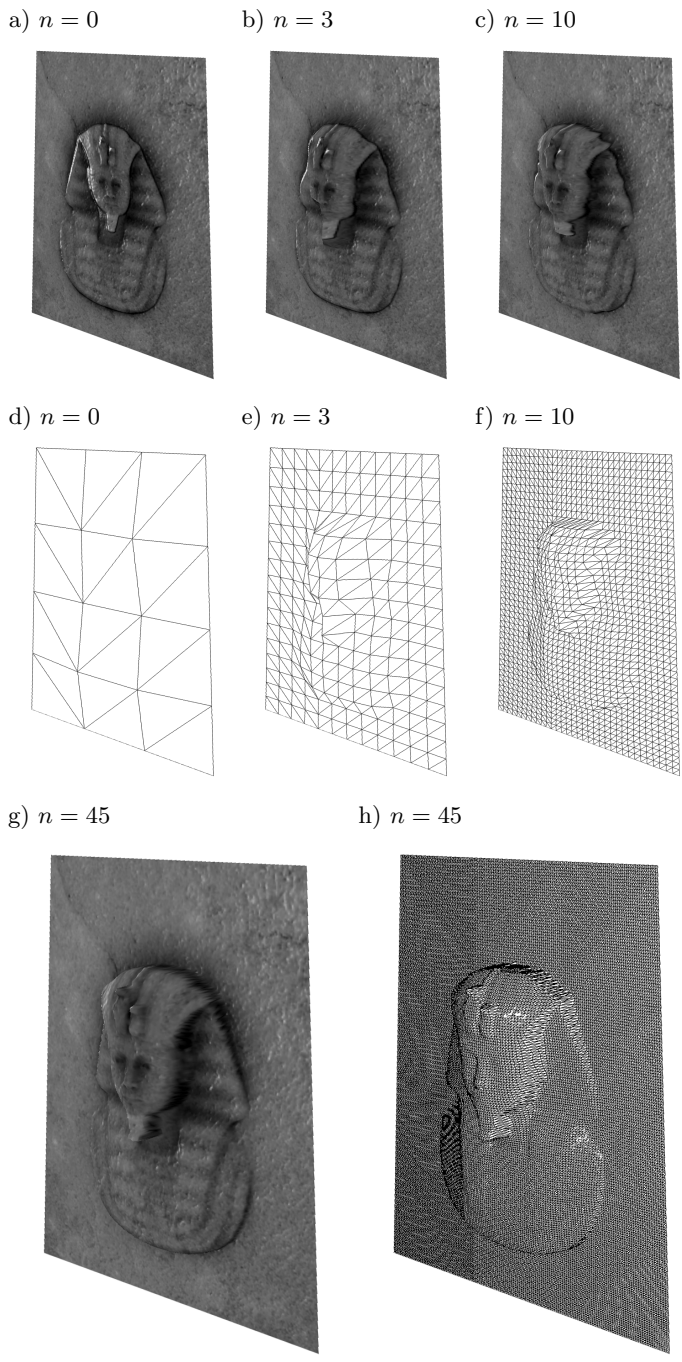
Przy definiowaniu techniki należy uwzględnić fakt, że jednostka cieniowania geometrii nie jest już potrzebna, przynajmniej w postaci stworzonej w poprzednich rozdziałach.

```
VertexShader vsGenericBarycentric = CompileShader( vs_4_0,
VSGenericBarycentric(true) ); ... technique10
BarycentricLightLambert {
  pass Pass0
  {
    SetVertexShader( vsGenericBarycentric );
    SetGeometryShader( NULL );
    SetPixelShader( psGenericLambert );
  }
}
```

11.7.3 Odrysowanie

Poprzednio przedstawiona tu funkcjonalność na poziomie aplikacji zamknięta została w klasie `Tessellator`. Dzięki czemu możliwe było stworzenie metody `Tessellator::Draw`, którą można z powodzeniem zastąpić przez wystąpienia `ID3DX10Mesh::Draw`. Jedynymi operacjami, które powinny towarzyszyć takiej modyfikacji, są wybór odpowiedniej techniki oraz zastosowanie nowego układu danych wejściowych.

Wewnętrznie implementowana metoda bazuje na wywołaniu metody urządzenia `DrawInstanced`. Jej pierwsze dwa parametry to odpowiednio: liczba wierzchołków i liczba instancji do odrysowania. Kolejne dwa wyznaczają indeks pierwszych elementów obu źródeł danych.



Rysunek 11.7. Przykład mapowania przemieszczeń z instancjonowanym kafelkowaniem dla różnych stopni oraz różnych trybów wypełniania

```
HRESULT Tessellator::Draw( ID3D10Device* device, UINT faceStart,
UINT faceCount, UINT tessellationFactor ) {
  // ustawienie danych wejściowych
  ID3D10Buffer* buffers[2] = { s_patternsBuffer, indicesBuffer };
  UINT offset[2] = { 0, 0 };
  UINT stride[2] = { 8, 12  };
  device->IASetIndexBuffer(NULL, DXGI_FORMAT_UNKNOWN, 0);
  device->IASetPrimitiveTopology(D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
  device->IASetVertexBuffers(0, 2, buffers, stride, offset);
  // ustawienie buforów zasobów
  vertexDataVariable->SetResource(vertexDataView);
  tangentDataVariable->SetResource(tangentDataView);
  // odrysowanie
  device->DrawInstanced(GetPatternLength(tessellationFactor)*3,
    faceCount, GetPatternIdx(tessellationFactor)*3, faceStart);
  return S_OK;
}
```

Przykładowe efekty kafelkowania z użyciem instancjonowania przedstawiono na rysunku 11.7.

A

Konfiguracja Visual Studio

Niniejszy poradnik dotyczy wersji Visual Studio 2008, ale uwagi tu zawarte powinny w być łatwy sposób przenoszone do nowszych bądź starszych edycji.

A.1 DirectX SDK

Przed rozpoczęciem jakiejkolwiek pracy z Direct3D należy pobrać DirectX SDK (*Software Development Kit*) z witryny *DirectX Developer Center*[1]. Nowe wersje pakietu ukazują się co kilka miesięcy i są wstecznie zgodne (obecnie do Direct3D 9), dlatego też jeżeli nie wystąpią szczególne przesłanki powinno się pobrać najnowszą edycję. Należy dodać, że DirectX SDK to nie tylko zbiór nagłówków i bibliotek, ale również zestaw przydatnych narzędzi, m. in.:

- *Pix for Windows*: program do profilowania i mierzenia wydajności aplikacji korzystających z Direct3D.
- *DirectX Error Lookup*: sprawdzanie kodów błędów
- *DirectX Viewer*: narzędzie do szybkiego podglądu plików .fx oraz .x
- *DirectX Texture Tool*: narzędzie do edycji tekstur w formacie .dds (*Direct-Draw Surface*)

A.2 Biblioteki Direct3D

Każdy projekt korzystający z funkcji Direct3D musi też mieć dostęp do odpowiednich nagłówków oraz bibliotek. Nagłówki znajdują się w katalogu zwanym `%DXSDK_DIR%\Inlcude`, natomiast zbudowane biblioteki umieszczone są w `%DXSDK_DIR%\Lib\x86` lub w `%DXSDK_DIR%\Lib\x64` (odpowiednio dla 32 lub 64-bitowych aplikacji). Procedura ich ustawienia wygląda następująco:

1. Wybrać z menu głównego opcję `Tools/Options...`

[1] `http://msdn.microsoft.com/en-us/directx/default.aspx`

2. Przejść do opcji `Projects and Solutions/VC++ Directiories`
3. Dodać ścieżkę do nagłówków dla listy `Include files`
4. Dodać ścieżkę do bibliotek dla listy `Library files`

Ustawienie podanych ścieżek pozwoli na uruchomienie przykładów. W przypadku tworzeniu projektu „od zera", należy pamiętać, że projektowi trzeba ręcznie wskazać biblioteki do zlinkowania. Absolutnie niezbędną biblioteką jest `d3d10.lib`. Dodatkowo, warto jeszcze dołączyć bibliotekę `d3dx10.lib` (lub `d3dx10d.lib` dla trybu Debug), gdyż dostarcza ona wielu przydatnych komponentów. Dodanie bibliotek odbywa się wg schematu:

1. Wybrać z menu głównego opcję `Project/Properties` lub z menu kontekstowego projektu wybrać opcję `Properties`
2. Przejść do opcji `Configuration Properties/Linker/Input`
3. Do pola `Additional Dependencies` dopisać nazwy nowych bibliotek

B

Kolorowanie składni plików efektów

Domyślnie Visual Studio odczytuje pliki efektów (rozszerzenie .fx) jako zwykłe pliki tekstowe. Z tego powodu programista pozbawiony jest udogodnień typu kolorowanie i uzupełnienie składni. Istnieją jednak sposoby, aby przywrócić chociaż pierwsze usprawnienie.

B.1 Podgląd C++

Można tak skonfigurować Visual Studio, aby pliki efektów były interpretowane jako pliki języka C++ (język HLSL wykazuje sporo podobieństw, więc kolorowanie będzie działało w znacznym stopniu). Aby to zrobić, należy wykonać następującą sekwencję czynności:

1. Wybrać z menu głównego opcję `Tools/Options...`
2. Przejść do opcji `Text Editor/File Extension`
3. W polu tekstowym `Extension:` wpisać `fx`
4. W kombie `Editor:` wybrać `Microsoft Visual C++`
5. Kliknąć `Add`

B.2 Wtyczka zewnętrzna

Poprawniejsze rozpoznawanie składni można uzyskać za pomocą którejś z zewnętrznych wtyczek:

- *NShader*[1]: koloruje składnię HLSL, GLSL oraz GC
- *InteliShade*[2]: koloruje składnię HLSL i oferuje uzupełnianie kodu w zakresie wbudowanych funkcji i typów

[1] `http://nshader.codeplex.com/`
[2] `http://intelishade.net/`

C

Kompilacja plików efektów

W przykładach załączonych z DirectX SDK można zaobserwować, że pliki efektów są kompilowane w trakcie działania programu. Jest to proste rozwiązanie, które jednak niesie z sobą kilka bardzo negatywnych konsekwencji. Przede wszystkim kompilacja następuje przy każdym uruchomieniu – nawet, jeśli plik nie uległ zmianie. Przy nieskomplikowanych efektach narzut czasowy może być niezauważalny, jednak przy większych projektach zaczyna robić się uciążliwy. Drugą konsekwencją jest utrudnione rozwijanie pliku efektu, ponieważ ewentualne błędy wykrywane są dopiero po uruchomieniu programu (nierzadko zaś sama kompilacja poprzedzona jest wczytaniem innych zasobów).

Rozwiązaniem jest umożliwienie Visual Studio kompilację plików efektów. W tym celu użyć należy kompilatora `fxc` z pakietu DirectX SDK i mechanizmu *Custom Build Rules*. Przed wykonaniem poniższych instrukcji należy upewnić się, że w systemie jest zmienna środowiskowa `%DXSDK_DIR%`, która wskazuje miejsce zainstalowania DirectX SDK.

1. Wybrać z menu głównego opcję `Project/Custom Build Rules...` lub z menu kontekstowego projektu wybrać opcję `Custom Build Rules...`
2. Kliknąć w przycisk `New Rule File...`
3. Wypełnić pola `Display Name`, `File Name` i `Directory` według uznania, a następnie kliknąc w `Add Build Rule...`
4. Następująco wypełnić pola:
   - `Name`: wg uznania, np. `FX`
   - `File Extensions`: `*.fx`
   - `Command Line`: `"%DXSDK_DIR%\Utilities\bin\x86\fxc" "/Fo $(OutDir)\$(InputName).fxc" /Tfx_4_0 [inputs]`
   - `Outputs`: `$(InputName).fxc`
5. Po powrocie do okna `Visual C++ Custom Build Rules Files` upewnić się, że przy nazwie dodanego pliku checkbox jest zaznaczony

W tym momencie każdy plik efektu dodany do projektu będzie podlegał kompilacji. Można kompilować cały projekt albo selektywnie (opcją `Compile`

z menu kontekstowego). Jeżeli podczas kompilacji pojawią się błędy, staną się one od razu widoczne w oknie `Error Lists`. Dodatkowo można eksperymentować w opcją `Command Line` - najlepiej sprawdzić opcje kompilatora `fxc` i dostosować polecenie do swoich potrzeb. Przykładowo, pisząc:

```
"%DXSDK_DIR%\Utilities\bin\x86\fxc"
/Fo "$(OutDir)\$(InputName).fxc"
/Fc "$(IntDir)\$(InputName).fxca"
/Tfx_4_0 [inputs]
&& type "$(IntDir)\$(InputName).fxca"
```

do katalogu z pośrednimi artefaktami kompilacji (ang. *intermediate directory*) zostanie dodany listing efektu, czyli informacje o jego skompilowanej postaci. Dodatkowo dzięki wywołaniu programu `type` jego zawartość zostanie wypisana do okna `Output`.

D

Tworzenie tekstur sześciennych

Direct3D wpiera automatyczne (tj. bez konieczności dodatkowego konfigurowania) wczytywanie tekstur sześciennych z obrazów DDS (akronim od DirectDraw Surface). Z tego powodu warto poznać narzędzia, które posłużyć mogą do ich tworzenia.

D.1 DirectX Texture Tool

Narzędzie *DirectX Texture Tool* dostarczane jest razem z DirectX SDK. Aby stworzyć nową teksturę sześcienną należy:

1. Wybrać z menu głównego opcję `File/New Texture...`
2. Zaznaczyć opcję `Cubemap Texture`
3. Kliknąć `OK`

Bieżąco można podglądać tylko jedną część tekstury sześciennej. Zmiany dokonuje się albo opcją `View/Cube Map Face`, albo wypisanymi w niej klawiszami skrótowymi. Załadować obraz dla bieżącej ściany można korzystając z opcji z rodziny `File/Open Onto`.

Niestety, *DirectX Texture Tool* jest bardzo nieergonomiczny. Dodatkowa możliwość ładowania oraz podglądu wyłączenie pojedynczych ścian może rodzić frustrację, gdyż często spotykanym sposobem reprezentacji tekstury sześciennej jest rozłożenie sześcianu w postaci krzyża (ang. *cube cross*). W tej sytuacji można albo ręcznie wyodrębniać składowe, albo skorzystać z innego programu.

D.2 CubeMapGen

Darmowy program *CubeMapGen*[1] umożliwia tworzenie map sześciennych. W odróżnieniu od poprzedniego narzędzia, tekstura nie jest wyświetlana na

[1] `http://developer.amd.com/gpu/cubemapgen/Pages/default.aspx`

płaszczyźnie, lecz zostaje nałożona na kulę (symulacja odbicia) bądź na sferę tła (symulacja otoczenia). Dzięki temu poprawność oraz jakość obrazu może być sprawdzona przed uruchomieniem korzystającej z niej aplikacji.

Program oferuje, poza przetwarzaniem obrazów DDS, odczyt i zapis tekstur sześciennych w postaci krzyża oraz podzielonych na rozdzielonych pliki. To, oraz prostota i intuicyjność obsługi, powoduje, że używa się go dużo przyjemniej, niż narzędzia firmy Microsoft. Ciekawostką jest natomiast fakt, że interfejs narzędzia w wyraźny sposób nawiązuje do DXUT.

Bibliografia

1. Bill Bilodeau and Mike Songy. Real time shadows. *Creativity '99*, 1999.
2. Antoine Bouthors, Fabrice Neyret, Nelson Max, Eric Bruneton, and Cyril Crassin. Interactive multiple anisotropic scattering in clouds. In *ACM Symposium on Interactive 3D Graphics and Games (I3D)*, 2008.
3. Chris Brennan. *Accurate Reflections and Refractions by Adjusting for Object Distance*. Wordware Publishing, Inc, 2002.
4. Kelly Dempski and Emmanuel Viale. *Advanced Lighting and Materials with Shaders*. Wordware Publishing, Inc, 2005.
5. Fletcher Dunn and Ian Parberry. *3D Math Primer for Graphics and Game Developmen*. Wordware Publishing, Inc., 2002.
6. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Wzorce projektowe. Inżynieria oprogramowania*. Wydawnictwa Naukowo-Techniczne, 2008.
7. Janusz Grzyb. *J2ME Tworzenie gier*. Wydawnictwo HELION, Gliwice, 2008.
8. Radosław Grzymkowski. *Matematyka dla studentów wyższych uczelnie technicznych*. Wydawnictwo Pracowni Komputerowej Jacka Skalmierskiego, Gliwice, 2000.
9. Microsoft Corporation. *DirectX SDK*.
10. NVIDIA. *NVIDIA GeForce 8800 GPU Architecture Overview*.
11. Michael Oren and Shree K. Nayar. Generalization of lambert's reflectance model. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 239–246, New York, NY, USA, 1994. ACM.
12. Renato Pajarola. Large scale terrain visualization using the restricted quadtree triangulation, 1998.
13. Emil Persson. Advanced d3d10 rendering. Technical report, AMD, 2007.
14. Fabio Policarpo and Francisco Fonseca. Deferred shading tutorial. Pontifical Catholic University of Rio de Janeiro, 2005.
15. Ken Shoemake. Animating rotation with quaternion curves. *Computer Graphics*, 19(3):245–254, 1985.
16. Sumant Tambe and Wikibooks Community. *More* C++ *Idioms*. Wikibooks, 2010. `http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms`.
17. Gregory J. Ward. Measuring and modeling anisotropic reflection. *SIGGRAPH Comput. Graph.*, 26(2):265–272, 1992.