Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

Michał Mazanik

Nowoczesne układy graficzne

Praca Magisterska

Praca wykonana pod kierunkiem
dr Andrzeja Łukaszewskiego

Wrocław 2005

Spis treści

Spis rysunków

Spis tablic

Wstęp

Moment powstania komputerów osobistych w latach osiemdziesiątych wyznaczył początek nowej ery. Przez ponad dwadzieścia lat rozwoju technologicznego moc obliczeniowa komputerów wzrosła o kilka rzędów wielkości. Powstały układy specjalizowane w wąsko określonych zadaniach, między innymi w wyświetlaniu w czasie rzeczywistym skomplikowanych trójwymiarowych scen.

Możliwość interaktywnego oglądania przestrzennych scen otworzyła drogę ośrodkom naukowym do wizualizacji skomplikowanych procesów zachodzących w przyrodzie. Powstały komputerowe maszyny szkoleniowe dla pilotów samolotów, możliwe stało się wirtualne zwiedzanie muzeów oraz branie udziału w podboju kosmosu w grach komputerowych. Wszystko to dzięki małemu układowi scalonemu wykonującemu wszystkie niezbędne obliczenia, układowi graficznemu.

Proces generowania pojedynczej klatki obrazu jest bardzo skomplikowany i składa się z wielu etapów. W każdy z nich są zaangażowane algorytmy przetwarzające ogromne ilości danych. Praca ta ma na celu przybliżenie wszystkich kroków, jakie musi wykonać układ graficzny, aby na ekranie pojawił się pożądany obraz. Szeroko opisane są matematyczne podstawy funkcjonowania algorytmów ukrytych we wnętrzu układu oraz metody kontrolowania całego procesu przez programistę. Oprócz podstawowych technik opisane są różnorodne efekty specjalne, które nadają utworzonemu obrazowi bardziej rzeczywistego charakteru.

Bazą do opracowania tej pracy są karty graficzne szeroko dostępne na rynku konsumenckim. Są to układy bardzo uniwersalne i wykorzystywane w każdej dziedzinie, w której istotny jest czas generowania obrazu. Wiedza teoretyczna ma jednak zastosowanie w każdym rodzaju maszyn wspomagających rysowanie trójwymiarowych scen, dlatego duża część tego opracowania może z powodzeniem stanowić podstawę do poznania tej dziedziny wiedzy.

Rozdział 1

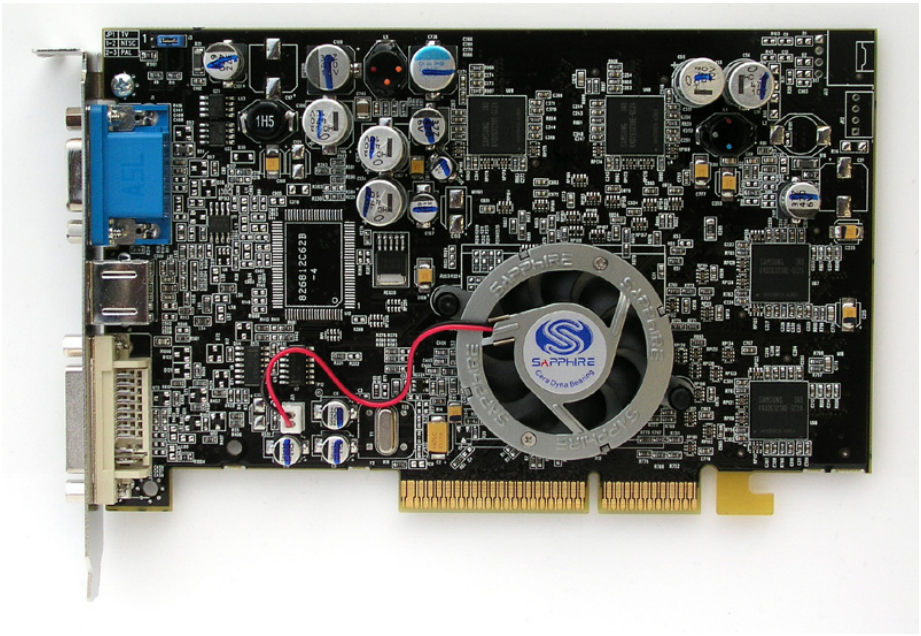
Architektura

1.1. Sprzęt

Architektura komputerów osobistych została zaprojektowana z myślą o jak największej modularności i prostocie rozbudowy. Takie podejście istotnie upraszcza wymianę poszczególnych komponentów systemu, tym samym jego ulepszanie. Rdzeń takiej maszyny stanowi zbiór podstawowych układów kontrolujących pracę, takich jak procesor główny, kontroler pamięci oraz kontroler wejścia-wyjścia. Duża integracja tych układów pozwala na bardzo szybką komunikację pomiędzy nimi, bez niej prędkość wykonywania obliczeń byłaby bardzo mała. Idealnym rozwiązaniem, biorąc pod uwagę wyłącznie szybkość działania systemu, jest umieszczenie wszystkich komponentów na jednej płycie drukowanej. Pozwala to na dużą dowolność w fazie projektowania i uzyskanie maksymalnej mocy obliczeniowej. Niestety, tak zbudowany system jest drogi a rozbudowa niemożliwa, bez wymiany wszystkich komponentów jednocześnie. Rozsądnym rozwiązaniem jest oddzielenie od rdzenia systemu tych układów, które szybciej niż inne ewoluują wraz z postępem nauki, przez co częściej wymagają wymiany. We współczesnych komputerach osobistych bazą jest płyta główna, na której umieszczone są tylko układy kontrolujące i koordynujące pracę pozostałych komponentów. Do układów dostępnych w postaci osobnych modułów należą procesor, pamięć operacyjna oraz wszelkiego rodzaju karty rozszerzające funkcjonalność komputera. Karty rozszerzeń to nic innego jak płytki drukowane z umieszczonymi na nich wyspecjalizowanymi układami scalonymi, które komunikują się ze rdzeniem systemu za pomocą złącz krawędziowych (rys. 1.1).

Podczas ponad 20-letniej historii architektury PC opracowano wiele technologii umożliwiających coraz szybszą transmisję danych przez złącza krawędziowe. Podstawową cechą danego stardardu jest maksymalna prędkość z jaką dane mogą być transportowane przez szynę. Przepustowość magistrali danych można obliczyć w prosty sposób, mnożąc szerokość szyny przez jej taktowanie. W latach 80-tych, ze względu na niskie wymagania, wartości te były bardzo małe (tab. 1.1). Większość komputerów w tamtych czasach komunikowała się z użytkownikiem za pomocą interfejsów tekstowych, nie było potrzeby przesyłać dużej ilości danych do układu graficznego. Wprowadzenie w roku 1990 systemu Windows 3.0 zmieniło tę sytuację radykalnie.

Weźmy następujący przykład: chcemy wyświetlić kolorowy film animowany, o rozdzielczości 640 na 480 pikseli, który został nagrany z częstotliwością 60 klatek na sekundę. Na jeden piksel przypadają 3 bajty, po jednym na każdą składową koloru RGB.



Rysunek 1.1: Karta graficzna z dobrze widocznym złączem krawędziowym

Standard	Rok wprowadzenia	Szerokość szyny (bity)	Taktowanie szyny (MHz)	Prędkość transmisji (MB/s)
ISA-8	1981	8	8	8
ISA-16	1984	16	8	16
EISA	1988	32	8	32
VLB	1993	32	33	128
PCI	1994	32	33	128
AGP	1997	32	od 66 do 66*8	od 256 do 2048

Tablica 1.1: Standardy magistrali danych

Jedna klatka zajmuje więc 900 kilobajtów. Film będzie płynnie odtwarzany, jeśli będziemy w stanie przesłać do karty graficznej 54 megabajty danych na sekundę. Dopiero standard PCI jest w stanie podołać temu zadaniu. Zamiast przesyłać tak ogromne ilości danych, rozsądniejszym rozwiązaniem jest zlecenie karcie wykonania jak największej ilości operacji, odciąży to znacznie pracę procesora oraz zmniejszy wymaganą przepustowość magistrali. Film najpierw kompresuje się specjalnym programem zwanym kodekiem video, który zmniejsza ilość danych potrzebnych do zapamiętania informacji o obrazie nawet kilkusetkrotnie. Dopiero tak przygotowany film wysyłamy do karty graficznej. We współczesnych kartach powszechne są sprzętowe dekodery video, które operują bezpośrednio na skompresowanych danych. Zysk płynący z takiego postępowania jest oczywisty, przesyłamy znacznie mniej danych a procesor główny może się zająć w tym czasie wykonywaniem innych zadań, zamiast ciągle przygotowywać ogromne ilości danych do transmisji.

W graficznych systemach operacyjnych z rodziny Windows wszystkie informacje są

prezentowane w tzw. oknach. Okno jest to prostokątny obszar otoczony ramką i wypełniony treścią. Gdy poruszamy takim oknem po ekranie, system musi przerysować nie tylko całą jego zawartość, ale również wszystkie okna, które znajdowały się pod nim lub na jego drodze. Operacja ta wymaga między innymi wypełniania, za każdym razem na nowo, całego obszaru okienek kolorem ich tła. Podobnie jak w operacji opisanej w poprzednim akapicie, wymaga to przesłania do karty graficznej bardzo dużej ilości danych. Dla jednego okienka o rozmiarze 400 na 400 pikseli, będzie to ponad 160 tysięcy pikseli o tym samym kolorze. Jest to oczywiste marnowanie czasu, więc producenci sprzętu postanowili dodać do kart graficznych specjalne układy, które miały zajmować się wypełnianiem prostokątnych obszarów oraz rysowaniem linii. Ten krok kolosalnie przyspieszył przerysowywanie się okienek w Windows a karty graficzne zyskały miano akceleratorów graficznych. Wraz z upływem lat, zakres możliwości tych małych układów scalonych zwiększyły się do tego stopnia, że potrafią animować w czasie rzeczywistym skomplikowane trójwymiarowe sceny.

W obecnych czasach, jako podstawowy standard przesyłu informacji do kart rozszerzeń służy szyna PCI (ang. Peripheral Component Interconnect). Na każdej płycie głównej komputera jest umieszczonych kilka złącz tego typu, w których mogą być umieszczone między innymi modemy, karty dźwiękowe, karty sieciowe, tunery telewizyjne i wszelkiego rodzaju zewnętrzne kontrolery wejścia-wyjścia. Magistrala PCI ma niestety istotną wadę, pasmo przenoszenia wynoszące 128 MB/s jest dzielone na wszystkie urządzenia w systemie. W praktyce oznacza to, że akcelerator graficzny nie będzie miał do dyspozycji pełnej przepustowości magistrali. Z tego powodu opracowano osobne złącze dla karty graficznej noszące nazwę AGP (ang. Accelerated Graphics Port). Ma ono znacznie większe możliwości oraz nie dzieli pasma z innymi urządzeniami. Prędkość dochodząca do 2 GB/s (w wersji AGPx8) pozwala na swobodne przesyłanie ogromnej ilości danych, bez zakłócania pracy całego systemu.

Przy generowaniu bardzo szczegółowego, dobrej jakości obrazu, wykorzystywane są ogromne ilości danych. Najbardziej pamięcio-chłonnymi elementami są prostokątne obrazy zwane teksturami. Opisują one między innymi fakturę obiektów znajdujących się na scenie. Na przykład, w przypadku drewnianego stołu będzie to obrazek przedstawiający brązowe słoje, w przypadku ściany w pokoju może to być zdjęcie tapety. Takich tekstur przy renderowaniu skomplikowanej sceny może być setki a ich objętość liczona jest w dziesiątkach megabajtów. Często na jednej teksturze, przy obliczaniu koloru dla pojedynczego piksela, wykonuje się kilka operacji. Pobieranie tych danych za każdym razem z pamięci operacyjnej przez magistralę byłoby tragiczne w skutkach, prędkość tworzenia obrazu byłaby bardzo mała. Z tego powodu, na płytce drukowanej wraz z procesorem graficznym jest umieszczona szybka pamięć o pojemności od 32 do 256 megabajtów. Pobieranie danych z tej pamięci przez układ graficzny odbywa się z prędkością dochodzącą nawet do 22 GB/s.

Standard AGP, w przeciwieństwie do poprzednich rozwiązań, oferuje dodatkową funkcjonalność w postaci specjalnie zarezerwowanego obszaru pamięci operacyjnej, do którego dostęp ma równocześnie procesor główny oraz karta graficzna. Pozwala to wirtualnie powiększyć ilość pamięci dostępnej dla układu graficznego. Sama karta, co prawda posiada własną pamięć, do której ma bardzo szybki dostęp, jednak jej ilość jest bardzo ograniczona kosztami produkcji. Z reguły jest to około 64 megabajtów. Tam powinny być trzymane dane, które są bardzo często wykorzystywane przy generowa-

niu obrazu. W pamięci AGP można przechowywać pozostałe, rzadziej używane dane. Dzięki takiemu mechanizmowi można też uniknąć tzw. efektu migotania, czyli wielokrotnego ładowania do pamięci tej samej tekstury, która musiała być w międzyczasie wykasowana, z powodu braku miejsca na inne potrzebne dane. Taka sytuacja na pewno miałaby miejsce w przypadku, gdyby zbiór tekstur potrzebnych do wygenerowania jednej klatki obrazu miał większą objętość niż dostępna pamięć. Dzięki bezpośredniemu dostępowi układu graficznego do pamięci operacyjnej, odczyt odbywa się dużo wolniej, ale żadne dane nie muszą być usuwane z pamięci karty.

Procesor znajdujący się na karcie graficznej dawno przestał być tylko konwerterem sygnału cyfrowego na analogowy. Współczesny akcelerator to w pełni programowalna jednostka, posiadająca własną pamięć operacyjną i działająca niezależnie od reszty systemu. Najważniejszą cechą jest właśnie niezależna asynchroniczna praca, która pozwala jednostce głównej w tym samym czasie wykonywać inne zadania. Sam procesor składa się z setek milionów tranzystorów, nierzadko przewyższając tą liczbą procesor główny. Każdy synchroniczny układ scalony przy dużych częstotliwościach taktowania wydziela dużo ciepła. Jest to obecnie duży problem producentów sprzętu, gdyż na kartach rozszerzeń nie ma po prostu miejsca na ogromne radiatory. Okupione jest to wysokim poziomem szumów, wydobywających się z wysokoobrotowych wentylatorów umieszczonych na układzie.

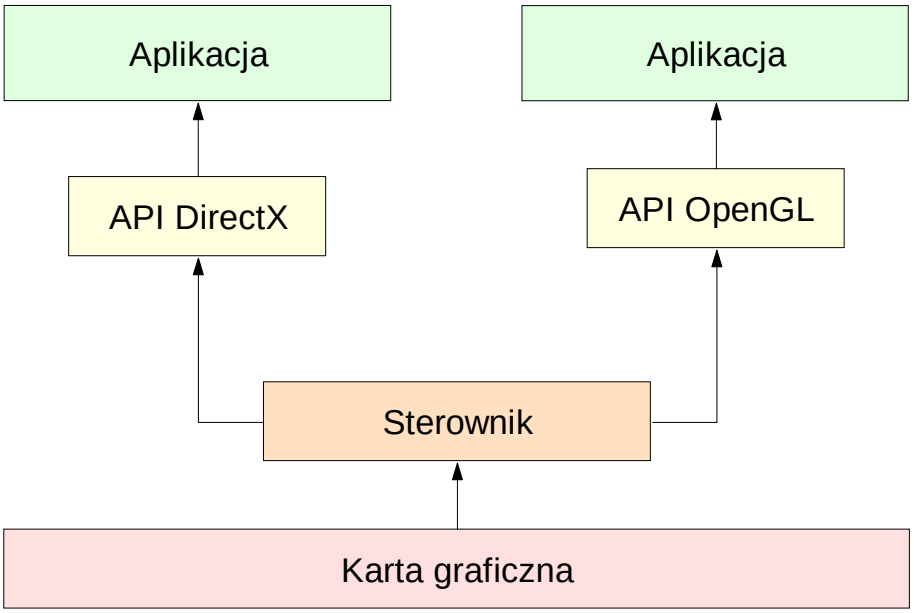
1.2. Sterowniki i biblioteki

W czasie powstania pierwszych akceleratorów, dostęp do ich funkcji miał tylko system operacyjny. Działo się tak, gdyż miały one za zadanie wyłącznie przyspieszać działanie interfejsu graficznego. Układów było stosunkowo mało, więc programowanie każdego z osobna nie sprawiało dużego problemu. Wraz z rozwojem i popularyzacją grafiki trójwymiarowej, możliwości tych układów zwiększały się a ich ilość znacznie wzrosła. Każdy z producentów opracował własne technologie, które niekoniecznie oferowały to samo. Niezbędnym stało się opracowanie wspólnego modelu funkcjonowania tych urządzeń, oraz umożliwienie dostępu do ich funkcji przez aplikacje napisane przez zwykłych użytkowników. Oczywiście zależało na tym również producentom sprzętu. Dzięki standardom dającym gwarancję poprawnego działania w danym środowisku, przekonanie klienta do swojego produktu stało się znacznie łatwiejsze. Programiści aplikacji w końcu mogli skupić się na polepszaniu jakości swoich aplikacji, zamiast marnować czas na pisanie osobnej ścieżki kodu dla każdej karty[1].

Współczesny system operacyjny kontroluje pracę wszystkich urządzeń w systemie za pośrednictwem sterowników, napisanych przez producentów sprzętu. Sterownik jest specjalnym programem, który pośredniczy w wymianie informacji między systemem operacyjnym a danym urządzeniem. Tylko jemu dane jest znać protokoły komunikacyjne, które są niezbędne, aby wywołać jakąkolwiek funkcję karty graficznej. Warstwa, na której operuje sterownik, zwana jest niskopoziomową (rys. 1.2).

Programista ma dostęp do funkcji akceleratora poprzez specjalny komponent systemu, który pełni rolę interfejsu (ang. API - Application Program Interface). W

[1]Niestety, w obecnych czasach bardzo trudno jest napisać uniwersalny kod działający na każdej karcie. Powodem tego nie są jednak problemy opisane w tym akapicie.

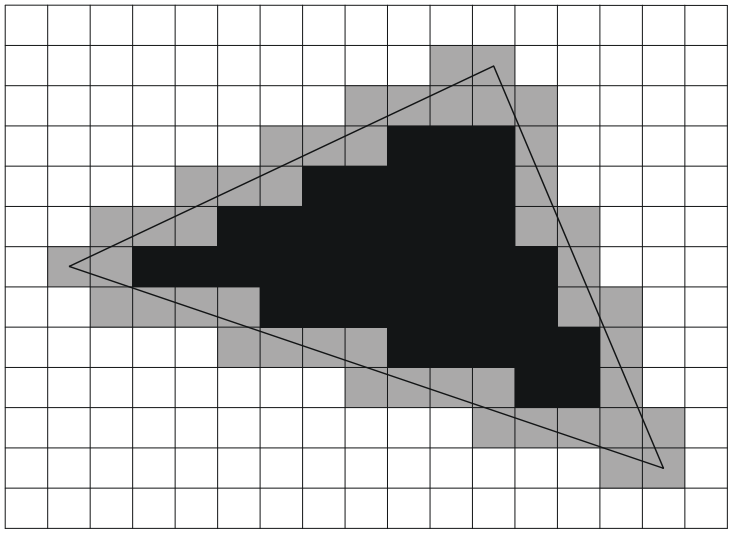


Rysunek 1.2: Dostęp do funkcji układu graficznego z poziomu aplikacji

terminologii języków obiektowych również istnieje pojęcie interfejsu, jako pewnego rodzaju wzorca pozbawionego implementacji. Jest to prawdą również w tym przypadku. Zgodnie z przyjętym modelem, za implementację większości funkcji zawartych w API jest odpowiedzialny sterownik. Główną częścią API są tylko nagłówki procedur wraz z dokładnym opisem ich funkcji oraz kryteriów, które powinny spełniać. Takie rozwiązanie daje dużą swobodę producentom układów w sposobie realizacji określonych zadań. Swoboda ta jest jednak mocno ograniczona, gdyż obraz generowany przez konkretną aplikację powinien wyglądać tak samo na komputerach wyposażonych w różne karty graficzne. Z tego powodu opracowano zbiór zasad, które określają sposób zachowania się w sytuacjach niejednoznacznych. Przykładem tego typu sytuacji jest tzw. problem rasteryzacji. Podstawą pracy akceleratora jest rysowanie trójkątów o zadanych wierzchołkach oraz parametrach, takich jak kolor czy faktura powierzchni. Współrzędne wierzchołków trójkąta są odwzorowywane na ekran, który składa się ze skończonej liczby pikseli. Problem pojawia się, kiedy musimy zadecydować, które piksele należą a które nie należą do wnętrza trójkąta. Dzieje się tak dlatego, że piksel wcale nie jest punktem ale prostokątem, który zajmuje bardzo małą powierzchnię. W przypadku kiedy krawędź trójkąta przecina pojedynczy piksel, nie bardzo wiadomo kiedy nadać mu kolor trójkąta a kiedy pozostawić kolor tła (rys. 1.3). Istnieje kilka sposobów poradzenia sobie z tym problemem. To, który z nich zostanie zastosowany, zależy wyłącznie od przyjętej w konkretnym API strategii. Zauważmy, że same parametry trójkąta, które podajemy w procedurze rysowania, w żaden sposób nie określają jednoznacznego rozwiązania.

Obecnie, na rynku dominują dwa standardy programowania układów graficznych, DirectX stworzony przez koncern Microsoft oraz OpenGL wprowadzony przez firmę

Silicon Graphics. Oba mają swoich zwolenników jak i zagorzałych przeciwników. Podstawową cechą różniącą je od siebie jest przenośność, czyli możliwość ich wykorzystania w różnych systemach operacyjnych. DirectX został stworzony z myślą wyłącznie o Windows i tak też pozostało do dzisiaj. Strategią firmy Microsoft jest w najlepszym przypadku ignorowanie istnienia innych systemów, często również celowe ograniczanie ich rozwoju. Dla wielu ludzi możliwość pracy na przykład w systemie Linux jest niezbędna, chociażby ze względów finansowych. Naturalną koleją rzeczy było powstanie całej rzeszy wrogów wielkiego monopolisty, którzy za wszelką cenę starali się znaleźć wady w oprogramowaniu Microsoftu i podważyć sens korzystania z niego. Powstało wiele mitów dotyczących obu interfejsów, które z reguły są nieprawdziwe a ich geneza jest związana ściśle z brakiem wiedzy ich pomysłodawców.



Rysunek 1.3: Rasteryzacja trójkąta

OpenGL jest następcą standardu IrisGl, powstałym specjalnie dla potrzeb profesjonalnych stacji graficznych. W czasach, kiedy Silicon Graphics produkował swoje komputery, projektanci Microsoftu dopiero zastanawiali się nad możliwością szerszego wykorzystania technologii graficznych. Nie było w tym niczego nienaturalnego, jakiekolwiek wyspecjalizowane układy graficzne były domeną wyłącznie profesjonalnych maszyn. Zmienić miał to Windows 95, który w zamierzeniu twórców miał stać się domową platformą multimedialną. Skończyło się niestety na zupełnie niestabilnej platformie a DirectX został pozbawiony wielu błędów dopiero wiele lat później.

W przeciwieństwie do OpenGL, DirectX był interfejsem niewygodnym w użyciu, chaotycznie zaprojektowanym i słabo znoszącym szybko rozwijający się rynek sprzętu. OpenGL został oparty na systemie rozszerzeń, każdą dodatkową funkcję można było dodać do istniejącej wersji bez jakiejkolwiek ingerencji w specyfikację. Każdy producent mógł bez żadnego problemu wdrożyć nową technologię natychmiast, nie czekając na zatwierdzenie kolejnej wersji API. Sytuacja zmieniła się na początku XXI wieku na korzyść Microsoftu. Paradoksalnie, przyczyną takiego stanu rzeczy nie było pogorszenie

się jakości architektury OpenGL, ale wzrost liczby producentów sprzętu. Każdy z nich, chcąc wylansować swoją technologię wprowadzał swoje wersje rozszerzeń. Spowodowało to sytuację, w której aby wykorzystać de facto tą samą funkcję na różnych kartach, należało napisać osobną ścieżkę kodu dla każdej z nich. Powstała co prawda organizacja OpenGL Architecture Review Board (ARB), która miała kontrolować jego rozwój, ale robiła to zbyt opieszale. Wykorzystał to Microsoft, udoskonalając architekturę DirectX. Wersja numer 9 niczym nie przypomina jej poprzedników, jest prosta w obsłudze i elastyczna na tyle, aby nie ograniczać rozwoju sprzętu[2].

Obecnie, czas tworzenia oprogramowania gra kluczową rolę, dlatego twórcy oprogramowania muszą koncentrować się na bardziej uniwersalnych rozwiązaniach. Pomimo nieco większej wygody pisania programów w OpenGL, pozostaje on bardziej popularny w środowiskach akademickich niż profesjonalnych. Dominujący producenci procesorów graficznych, jak ATI czy NVIDIA, używają standardu OpenGL przede wszystkim do testowania nowych technologii. Wprowadzają nowe rozszerzenia, aby móc szybko zaprezentować nowe możliwości swoich układów. Powstał przez to ogromny chaos a profesjonalni twórcy oprogramowania widzą OpenGL niestety tylko jako pole walki działów marketingowych.

W API Microsoftu nie istnieje pojęcie rozszerzenia, wszystkie możliwe do wykonania funkcje są z góry ustalone. Nie oznacza to jednak żadnego przymusu, karta może, ale nie musi oferować określonych funkcji. Zestaw operacji możliwych do wykonania w danej wersji DirectX, jest przed jego wprowadzeniem szczegółowo analizowany. W takich spotkaniach uczestniczą również producenci sprzętu. Dzięki temu, twórcy oprogramowania wraz z inżynierami mogą ustalić wspólny kierunek rozwoju technologii w sposób, który nie godzi w interesy żadnej ze stron. Takie podejście eliminuje potencjalny chaos, spowodowany różnorodnością strategii rozwoju poszczególnych firm. W momencie wejścia na rynek nowej wersji API, często znajdują się w nim funkcje, które nie mogą być jeszcze wykonane przez żaden dostępny układ graficzny. Dzięki planowaniu posuniętym daleko w przyszłość, nowe wersje specyfikacji można wprowadzać dużo rzadziej.

Bardzo istotnym jest, aby kolejne wersje API stanowiły rozwinięcie poprzedniej architektury. Niestety, zdarzają się sytuacje, w których nie do końca przemyślane rozwiązania muszą zostać usunięte, aby w ich miejsce wprowadzić nowe. Ze względu na przymus kompatybilności nowych wersji ze starymi, nie zawsze jest to możliwe. Prowadzi to do sytuacji, w której najnowsze technologie egzystują obok swoich poprzedników, powodując dodatkowy nieład w dokumentacji.

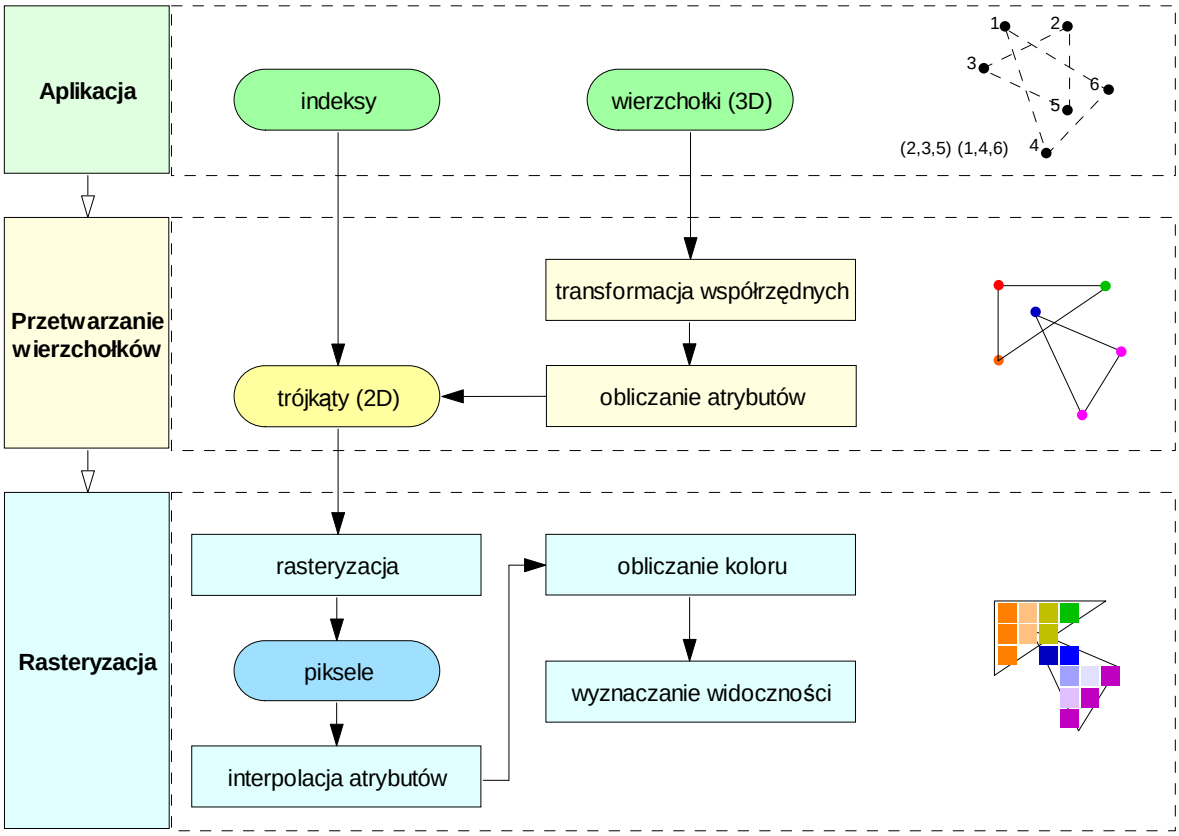
W niniejszej pracy wszystkie opisy funkcji są oparte na API DirectX w wersji 9.0c. Jest to w tej chwili najnowsza wersja specyfikacji. Większość zagadnień omawianych w następnych rozdziałach można odnieść również do innych interfejsów. Jedyną przeszkodą będzie znalezienie odpowiednika danej funkcji w dokumentacji lub nieznaczna zmiana w kodzie programu. Pod względem merytorycznym, wszystkie zagadnienia poruszane w tym opracowaniu pozostaną w pełni wartościowe, niezależnie od wyboru konkretnego środowiska programistycznego.

[2]W czasie, kiedy powstawał ten tekst, organizacja ARB zatwierdziła wersję 2.0 specyfikacji OpenGL. Porządkuje ona większość newralgicznych rozszerzeń, umieszczając ich odpowiedniki w specyfikacji. Na razie nie powstały jeszcze odpowiednie sterowniki, przez co nie wiadomo czy standard zostanie szeroko przyjęty przez środowisko programistów.

1.3. Ogólny schemat pracy układu graficznego

Generowanie obrazu przy użyciu akceleratora graficznego, składa się z trzech etapów (rys. 1.4). Pierwszy z nich wykonuje aplikacja, dwa pozostałe są realizowane przez kartę graficzną. Jedynym zadaniem procesora głównego jest przygotowanie danych w celu przesłania ich do układu.

Rysować można wyłącznie bryły geometryczne składające się z trójkątów. Każdy trójkąt jest opisany przez trzy wierzchołki, które mają określone współrzędne w przestrzeni trójwymiarowej, oraz specjalne dodatkowe atrybuty. Aby zminimalizować ilość informacji przesyłanych do karty, wierzchołki są zapisywane w określonej kolejności. Aby narysować później trójkąt, należy podać tylko indeksy poszczególnych punktów. Ma to istotne znaczenie wtedy, gdy trójkąty mają wspólne wierzchołki. W takim przypadku, zamiast przesyłać te same porcje danych kilkukrotnie, wysyłane są tylko identyfikujące wierzchołki numery.

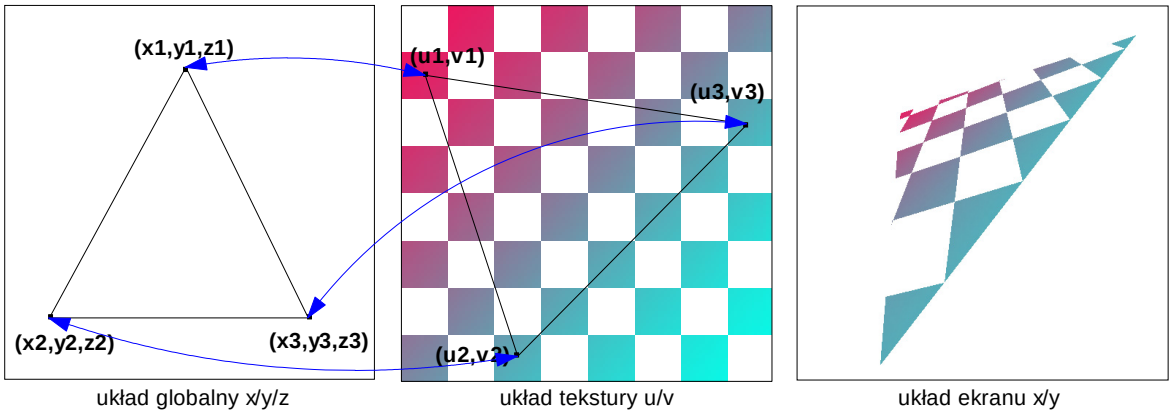


Rysunek 1.4: Schemat procesu generowania obrazu

Zestaw brył geometrycznych, który będzie wyświetlony na monitorze, jest zdefiniowany w przestrzeni trójwymiarowej. Dzięki matematycznemu opisowi sceny, istnieje możliwość dowolnego transformowania tych obiektów. Układ graficzny dysponuje specjalnie wydzieloną jednostką arytmetyczną, która potrafi wykonywać obliczenia na

współrzędnych wierzchołków. Procesem transformacji kieruje specjalny mikroprogram, który jest wykonywany, od początku do końca, dla każdego wierzchołka. Ekran komputera składa się z dwuwymiarowej matrycy pikseli i niezbędna jest odpowiednia transformacja, która dokona rzutu wszystkich elementów na dwuwymiarową przestrzeń wyświetlacza. Do niedawna, wszystkie funkcje związane z tym procesem były wykonywane bez udziału programisty. Miał on do wyboru tylko zestaw predefiniowanych procedur, które potrafiły wykonać tylko podstawowe czynności, jak na przykład przemnożenie wektora przez macierz. W nowoczesnych kartach treść tego mikroprogramu ustala sam programista, ale jego podstawowa funkcja nadal jest z góry narzucona.

Proces przetwarzania wierzchołków nie ogranicza się tylko do transformacji ich współrzędnych. Wraz z informacjami o ich położeniu w przestrzeni, do karty można przesłać dowolne atrybuty. Mogą one stanowić dodatkowe parametry obliczeń lub opisywać pewną własność, jak na przykład kolor. Dodatkowe atrybuty, które są ustalane indywidualnie dla każdego wierzchołka, są w późniejszym etapie generowania obrazu liniowo interpolowane na obszarze całego trójkąta. Nadanie trójkątowi faktury, opisanej dwuwymiarową teksturą, polega na przekazaniu współrzędnych tekstury, które mają być przypisane danemu wierzchołkowi. Jednostka zajmująca się rasteryzacją, interpoluje te współrzędne na całej powierzchni, pokrywając w ten sposób trójkąt dwuwymiarowym obrazkiem (rys. 1.5). Należy zauważyć, że atrybut jest rozłożony liniowo tylko we współrzędnych trójwymiarowych. Przekształcenie perspektywiczne powoduje, że interpolacja wykonywana we współrzędnych ekranowych wcale nie jest liniowa.

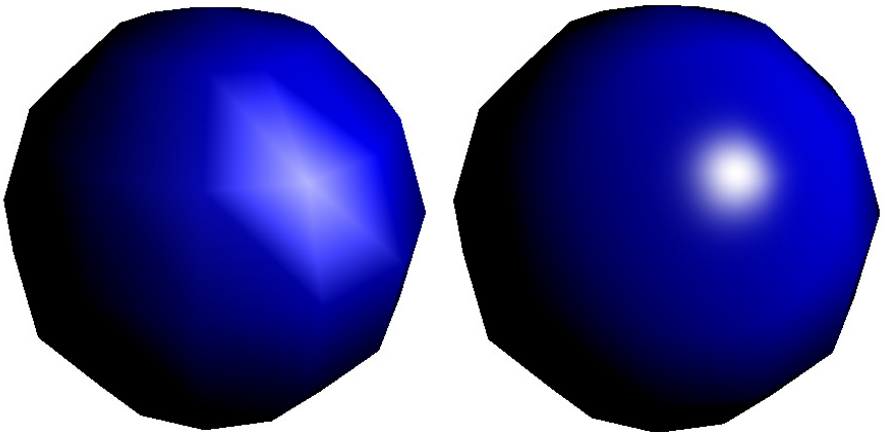


Rysunek 1.5: Proces teksturowania oraz przykład rzutu perspektywicznego.

Nie wszystkie atrybuty koniecznie trzeba przesyłać do procesora graficznego. Mikroprogram może zająć się wygenerowaniem tych danych bezpośrednio na karcie. Przykładem tego typu obliczeń jest pokolorowanie obiektów w zależności od ich odległości od obserwatora. Program wykona wszystkie potrzebne operacje na współrzędnych wierzchołków i nada im odpowiednie barwy tak, aby zasymulować zjawisko mgły. Kolejnym przykładem jest obliczanie natężenia oświetlenia. Mając daną pozycję źródła światła, jego kolor oraz natężenie, można obliczyć intensywność i barwę światła odbitego w kie-

runku obserwatora. Niestety, wykonując tą operację tylko na wierzchołkach obiektu, efekt będzie niezadowalający. Jest to spowodowane liniową interpolacją koloru na obszarze trójkąta. Współczesne układy umożliwiają obliczenia w każdym narysowanym pikselu, w tym w punktach znajdujących się w obrębie trójkąta. Daje to o wiele dokładniejsze i bardziej wiarygodne wyniki (rys. 1.6).

Po obliczeniu współrzędnych i atrybutów dla wierzchołków, przetworzone dane trafiają do jednostki zajmującej się rasteryzacją. Rasteryzacja polega na wypełnieniu odpowiednim kolorem tych pikseli bufora ekranu, które znajdują się w obszarze widocznych po transformacji trójkątów. Udział programisty w tym procesie ogranicza się do napisania drugiego mikroprogramu, który obliczy kolor aktualnie rysowanego piksela. Cała reszta jest już wykonywana automatycznie. Układ sam wyznacza piksele pokrywające obszar trójkąta oraz interpoluje atrybuty z wierzchołków tak, aby można je było później wykorzystać w mikroprogramie. Podobnie jak w jednostce przetwarzania wierzchołków, ten sam program jest uruchamiany dla każdego generowanego piksela.



Rysunek 1.6: Oświetlenie obliczone tylko na wierzchołkach (z lewej), oraz na całej powierzchni trójkątów (z prawej).

Mikroprogram, który wykonuje jednostka odpowiedzialna za rasteryzację, nie wie, który piksel na ekranie jest właśnie obliczany. Jedynymi parametrami, jakimi dysponuje, są już przeinterpolowane atrybuty oraz stałe zmienne. Wśród stałych zmiennych, znajdują się między innymi aktualnie wybrane przez program tekstury. Jedną z podstawowych operacji, którą może wykonać ten mikroprogram, jest pobranie z tekstury wartości koloru, znajdującego się w punkcie o danych współrzędnych. Współrzędne te były albo podane wcześniej przez aplikację, albo zostały wygenerowane przez mikroprogram dla wierzchołków. Przy obliczaniu koloru jednego piksela, można wykorzystać dane aż z 16 tekstur jednocześnie.

Ostatnim etapem jest wyznaczenie widoczności danego piksela. Jednym z niejawnych atrybutów obecnych w każdej fazie generowania obrazu, jest odległość punktu od obserwatora. Podczas interpolacji we współrzędnych ekranowych, wartość tej odległości również jest obliczana na bieżąco. Podczas rysowania piksela, wartość ta jest

wpisywana do specjalnego bufora, zwanego buforem-Z. Bufor-Z ma taką samą rozdzielczość jak bufor koloru, w którym składowane są kolory pikseli końcowego obrazu. Jeśli wystąpi sytuacja, w której w tym samym miejscu ma zostać narysowany piksel należący do innego trójkąta, wartość jego odległości jest porównywana z już znajdującą się w tym miejscu wartością w buforze-Z. Jeśli okaże się mniejsza, czyli aktualnie generowany piksel jest bliżej obserwatora, stare wartości w buforach koloru i Z są nadpisywane. W przeciwnym wypadku, aktualnie generowany piksel jest odrzucany.

Do niedawna, akcelerator oferował wyłącznie predefiniowane zestawy funkcji dla wierzchołków oraz pikseli. Dla wierzchołków dostępne były wyłącznie proste przekształcenia macierzowe, a dla pikseli tylko podstawowe operacje na teksturach. Wraz ze wzrostem wymagań co do jakości generowanego obrazu, dodano możliwość programowania układu graficznego. Opracowano specjalne zestawy instrukcji sterujących pracą procesora graficznego. W tej chwili są już dostępne nawet wysokopoziomowe języki programowania, dzięki temu znajomość mikroinstrukcji nie jest już konieczna. Jednak dopiero od niedawna procesor graficzny można nazwać w pełni programowalnym, ponieważ wczesne wersje języków oferowały bardzo ubogi zestaw funkcji wraz z całą masą obostrzeń i zakazów.

W tej pracy jest opisana architektura w wersji 2.0 dla wierzchołków i 2.0 dla pikseli[3]. Stanowią one nierozłączną całość, mimo iż są numerowane osobno. Tę wersję można już nazwać dojrzałym środowiskiem pracy, pozbawionym zasadniczych wad jej poprzedniczek. Programy można pisać swobodnie, bez zaprzątania sobie głowy utrudnieniami spowodowanymi przysłowiowym spadkiem po nieprogramowalnych układach. Poprzednie wersje, od 1.1 do 1.4, były próbą skonstruowania zestawu instrukcji z już istniejących w układzie predefiniowanych funkcji. Od wersji 2.0 większość operacji nie da się już wykonać bez użycia mikroprogramów. Do wielu zalet nowego standardu należy również istotnie zwiększona precyzja wewnętrznych obliczeń, przez co generowany obraz jest już pozbawiony widocznych wad. Dla uzmysłowienia sobie tego faktu warto wspomnieć, że do niedawna format zapisu liczb, na których operowały mikroprogramy miał 8-bitową stałopozycyjną część ułamkową. W wersji 2.0 jest to już 32-bitowy format zmiennopozycyjny.

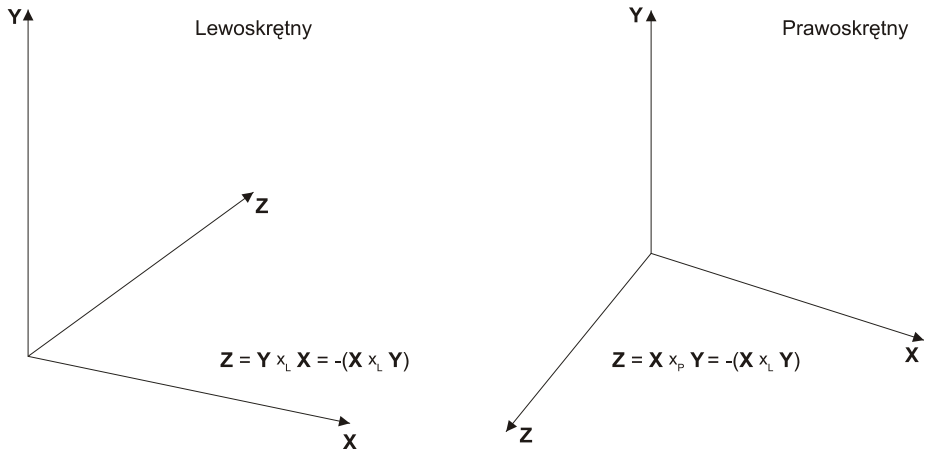
[3]W chwili obecnej, standard 2.0 obsługują układy firmy NVIDIA z serii Geforce FX (5xxx) i Geforce 6 (6xxx) oraz układy firmy ATI oznaczone symbolami Radeon 9550,9600,9700,9800 i wszystkie z serii Radeon X.

Rozdział 2

Przetwarzanie wierzchołków

2.1. Transformacje

Bryły geometryczne, które są przetwarzane przez kartę graficzną, składają się z wierzchołków pogrupowanych w trójkąty. Wierzchołki mają współrzędne zdefiniowane w przestrzeni trójwymiarowej. W przypadku, gdy używane jest API DirectX, układ współrzędnych, w którym są określone wierzchołki musi być lewoskrętny. Różnica między układem prawoskrętnym a lewoskrętnym polega na wzajemnym położeniu osi. W obu układach inaczej jest zdefiniowana operacja iloczynu wektorowego (rys. 2.7).

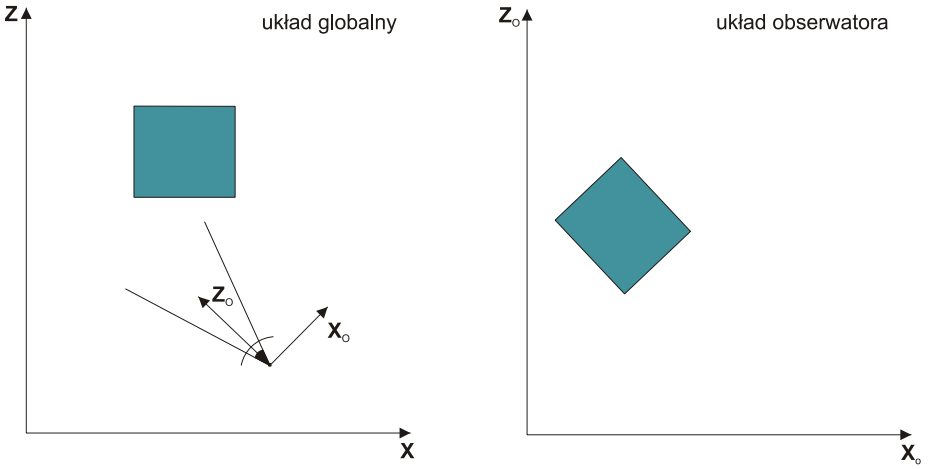


Rysunek 2.7: Dwa rodzaje układów współrzędnych i różnica w definicji iloczynu wektorowego.

Aby wygenerować dwuwymiarowy obraz, w pierwszej kolejności należy zdefiniować punkt, w którym znajduje się obserwator. Obserwatora można porównać do aparatu fotograficznego, a prostokątny obszar pojedynczej klatki filmu do dwuwymiarowej rzutni, która będzie później wyświetlona na ekranie. Z obserwatorem jest związany układ współrzędnych, w którym osie X i Y określają orientację rzuti a oś Z jest kierunkiem patrzenia na scenę. Ze względu na to, że obserwator porusza się po scenie, jego układ jest określony we współrzędnych globalnych, tak jak wszystkie obiekty.

Przed dokonaniem rzutu na dwuwymiarową płaszczyznę, współrzędne wierzchołków

trzeba przedstawić w układzie obserwatora. Dopiero wtedy stosowane jest przekształcenie, odwzorowujące trójwymiarowe bryły w ich dwuwymiarowe obrazy na płaszczyźnie. Takie podejście znacznie upraszcza obliczenia związane z rzutami perspektywicznymi, które dają się łatwo sformułować, jeśli kierunek patrzenia obserwatora pokrywa się z osią Z układu współrzędnych (rys. 2.8).



Rysunek 2.8: Transformacja współrzędnych obiektu z układu globalnego do układu obserwatora. Skala osi nie została zachowana.

Obiekty, a dokładniej wierzchołki je opisujące, można poddawać dowolnym przekształceniom. Zanim punkty będą opisane współrzędnymi globalnymi, często przechodzą szereg transformacji w innych układach. Przykładem może być obracające się koło samochodu. Jeśli koło będzie miało współrzędne zdefiniowane w układzie lokalnym, którego oś Z jest osią obrotu koła, wystarczy zastosować proste przekształcenie obrotu wokół osi Z. Dopiero później współrzędne zdefiniowane lokalnie są zamieniane na globalne.

Wszystkie operacje wykonywane na współrzędnych wierzchołków, wymagają sprawnego aparatu matematycznego, który pozwoli szybko wykonać niezbędne obliczenia. Macierze nadają się do tego doskonale, ze względu na możliwość kumulacji wielu transformacji w jednej macierzy. Niestety, macierze w $\mathbb{R}^3$ spełniają ten warunek tylko w przypadku przekształceń liniowych, które nie obejmują przesunięć. Ominąć to ograniczenie, można wykonując obliczenia w specjalnej przestrzeni, zwaną przestrzenią rzutową.

2.1.1. Przestrzenie rzutowe

Geometria euklidesowa charakteryzuje się dużą przejrzystością oraz jest łatwo przyswajalna. Niektóre własności tej przestrzeni nie pozwalają jednak na pełną swobodę wykonywania pewnych operacji. Piąty aksjomat Euklidesa mówi o tym, że dwie proste mogą się przecinać najwyżej w jednym punkcie. Podczas obliczania punktu przecięcia się dwóch prostych, trzeba bardzo uważać, aby nie doprowadzić do dzielenia przez zero. Dzielenie przez zero pojawi się, gdy proste te są równoległe, a więc wynik tej

operacji jest nieokreślony. Pewnego rodzaju rozszerzeniem przestrzeni euklidesowej jest przestrzeń rzutowa. Podstawową cechą różniącą ją od poprzedniczki, jest gwarancja istnienia dokładnie jednego punktu wspólnego dla każdych dwóch prostych. Ma ona również wiele przydatnych dla grafiki komputerowej własności, które nie mają miejsca w geometrii euklidesowej.

Definicja 1 *Punktami w przestrzeni rzutowej* $P(\mathbb{R}^2)$ *nazywamy klasy abstrakcji relacji* $\tilde{}$, *zdefiniowanej na zbiorze* $\mathbb{R}^3\backslash\{(0,0,0)\}$ *w następujący sposób:*

$$(p,q,r)\tilde{}(x,y,w) \Longleftrightarrow \exists\alpha \in \mathbb{R}\backslash\{0\} : (\alpha p, \alpha q, \alpha r) = (x,y,w).$$

Punkt $[x,y,w]$ *nazywamy właściwym, wtedy i tylko wtedy, gdy* $w \neq 0$. *Punkt* $[x,y,w]$ *nazywamy niewłaściwym, wtedy i tylko wtedy, gdy* $w = 0$.

Relacja $\tilde{}$ jest relacją równoważności, więc klasy abstrakcji tej relacji dzielą zbiór $\mathbb{R}^3\backslash\{(0,0,0)\}$ na rozłączne podzbiory. W związku z tym, każda trójka $(p,q,r) \in [x,y,w]$ jednoznacznie wyznacza cały podzbiór. Konsekwencją tego faktu jest to, że punkt można oznaczać dowolną trójką z tego zbioru, na przykład $[1,1,1]$ i $[2,2,2]$ definiują ten sam punkt w $P(\mathbb{R}^2)$. Liczby x, y i w nazywamy współrzędnymi jednorodnymi. Jeśli $w = 1$, są to znormalizowane współrzędne jednorodne.

Istnieje bezpośredni związek między właściwymi punktami w $P(\mathbb{R}^2)$ i punktami w $\mathbb{R}^2$. Jeśli $w \neq 0$, określamy transformację z $P(\mathbb{R}^2) \to \mathbb{R}^2$ w następujący sposób:

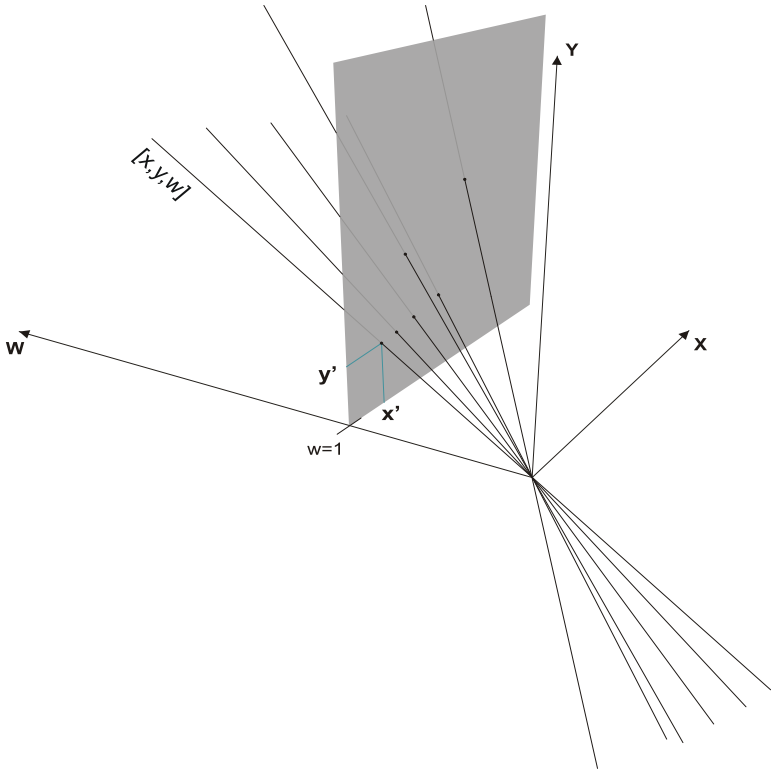
$$(x\prime, y\prime) = (\frac{x}{w}, \frac{y}{w}). \tag{2.1}$$

Odwrotnie, obrazem punktu $(u\prime, v\prime) \in \mathbb{R}^2$ w przestrzeni $P(\mathbb{R}^2)$ jest punkt $[u\prime, v\prime, 1]$, który reprezentuje nieskończony zbiór trójek w postaci $(\alpha u', \alpha v', \alpha)$, gdzie $\alpha \in \mathbb{R}\backslash\{0\}$.

Ponieważ współrzędne punktów $P(\mathbb{R}^2)$ są opisywane przez trzy liczby, można sobie wyobrazić, że opisują one również współrzędne w $\mathbb{R}^3$. Przy takim założeniu, punkt $[x,y,w] \in P(\mathbb{R}^2)$ stanowi w $\mathbb{R}^3$ prostą przechodzącą przez punkt $(0,0,0)$ o wektorze kierunkowym (x,y,w). Jeśli jest to punkt właściwy, para $(x\prime, y\prime)$ jest równa pierwszym dwóm współrzędnym punktu przecięcia tej prostej z płaszczyzną $w = 1$. Obrazem punktu $(x\prime, y\prime)$ po przekształceniu $\mathbb{R}^2 \to P(\mathbb{R}^2)$ w wyimaginowanej przestrzeni $\mathbb{R}^3$, jest prosta przechodzącą przez punkty $(x', y', 1)$ i $(0,0,0)$. Podsumowując, punkty przestrzeni $P(\mathbb{R}^2)$ można sobie wyobrazić w $\mathbb{R}^3$ jako proste przechodzące przez środek układu współrzędnych, a transformację $P(\mathbb{R}^2) \to \mathbb{R}^2$ jako przecięcie tych prostych z płaszczyzną $w = 1$ (rys. 2.9).

Definicja 2 *Prostą* (A,B,C) *w przestrzeni rzutowej* $P(\mathbb{R}^2)$ *nazywamy zbiór punktów spełniających równanie* $Ax+By+Cw = 0$, *gdzie* $(A,B,C) \neq (0,0,0)$. *Prostą* (A,B,C) *nazywamy właściwą, wtedy i tylko wtedy, gdy* $A \neq 0 \vee B \neq 0$. *Prostą* (A,B,C) *nazywamy niewłaściwą, wtedy i tylko wtedy, gdy* $A = B = 0$.

Podobnie jak w przypadku punktów, każda trójka liczb postaci $(\alpha A, \alpha B, \alpha C)$, gdzie $\alpha \in \mathbb{R}\backslash\{0\}$, definiuje tą samą prostą (A,B,C).

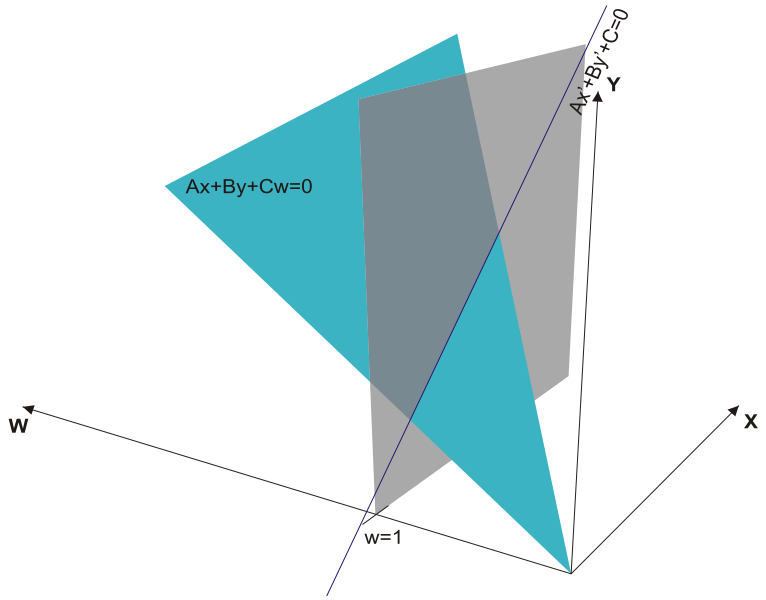


Rysunek 2.9: Punkty właściwe w przestrzeni $P(\mathbf{R}^2)$ przedstawione jako proste w $\mathbf{R}^3$. Punkt $(x\prime, y\prime)$ jest obrazem $[x, y, w]$ po przekształceniu go do $\mathbf{R}^2$.

Właściwy punkt $[x, y, w]$ znajduje się na prostej (A, B, C), jeśli $Ax + By + Cw = 0$. Proste w przestrzeni $P(\mathbb{R}^2)$ możemy sobie wyobrazić w przestrzeni $\mathbb{R}^3$, jako płaszczyzny przechodzące przez punkt $(0, 0, 0)$. Skoro $w \neq 0$, można zastosować wzór (2.1) i podzielić obie strony równania przez w. Otrzymujemy $Ax' + By' + C = 0$. Jest to równanie prostej w przestrzeni $\mathbb{R}^2$. Odwzorowanie $P(\mathbb{R}^2) \to \mathbb{R}^2$ możemy sobie wyobrazić w przestrzeni $\mathbb{R}^3$, jako przecięcie płaszczyzny $Ax + By + Cw = 0$ z płaszczyzną $w = 1$. Wynikiem tego przecięcia jest oczywiście prosta na płaszczyźnie $w = 1$ (rys. 2.10).

Niewłaściwy punkt $[x, y, 0]$ znajduje się na prostej (A, B, C), jeśli $Ax + By = 0$. Wynika z tego, że przez ten punkt w przestrzeni $P(\mathbb{R}^2)$ przechodzi każda właściwa prosta $[A, B, \gamma]$, gdzie $\gamma \in \mathbb{R}$. Jeśli $A \neq 0 \vee B \neq 0$, obrazem prostych $[A, B, \gamma]$ w przestrzeni $\mathbb{R}^2$, jest rodzina prostych równoległych $Ax\prime + By\prime + \gamma = 0$. Wniosek z tego płynący jest bardzo istotny. W przestrzeni $\mathbb{R}^2$ proste równoległe nie mają punktów wspólnych, ale ich odpowiedniki w przestrzeni $P(\mathbb{R}^2)$ zawsze przecinają się w pewnym punkcie niewłaściwym. Idąc dalej tym tropem, można przyjąć, że każdy punkt niewłaściwy $[x, y, 0]$ reprezentuje w przestrzeni $\mathbb{R}^2$ rodzinę prostych równoległych o równaniu $Ax\prime + By\prime + \gamma = 0$.

W przypadku, kiedy $A = B = 0$, punkt $[x, y, w]$ znajduje się na prostej (A, B, C), jeśli $Cw = 0$. Jedynymi punktami spełniającymi tą zależność są punkty o współrzędnych $[x, y, 0]$, czyli wyłącznie punkty niewłaściwe. Prosta $[0, 0, C]$ przecina więc wszystkie



Rysunek 2.10: Właściwa prosta w przestrzeni $P(\mathbf{R}^2)$ przedstawiona jako płaszczyzna w $\mathbf{R}^3$. Prosta $Ax\prime + By\prime + C = 0$ jest obrazem (A, B, C) po przekształceniu jej do $\mathbf{R}^2$.

punkty niewłaściwe przestrzeni $P(\mathbb{R}^2)$.Nie ma ona swojego obrazu w przestrzeni $\mathbb{R}^2$, ale wyobrażając sobie ją w $\mathbb{R}^3$, zobaczymy płaszczyznę o równaniu $w = 0$ (rys. 2.11).

Aby znaleźć prostą $P = (A, B, C)$ przechodzącą przez dwa punkty $X_1 = [x_1, y_1, w_1]$ i $X_2 = [x_2, y_2, w_2]$, należy rozwiązać układ równań:

$$\left\{\begin{matrix} Ax_1 + By_1 + Cw_1 = 0 \\ Ax_2 + By_2 + Cw_2 = 0 \end{matrix}\right\}.$$

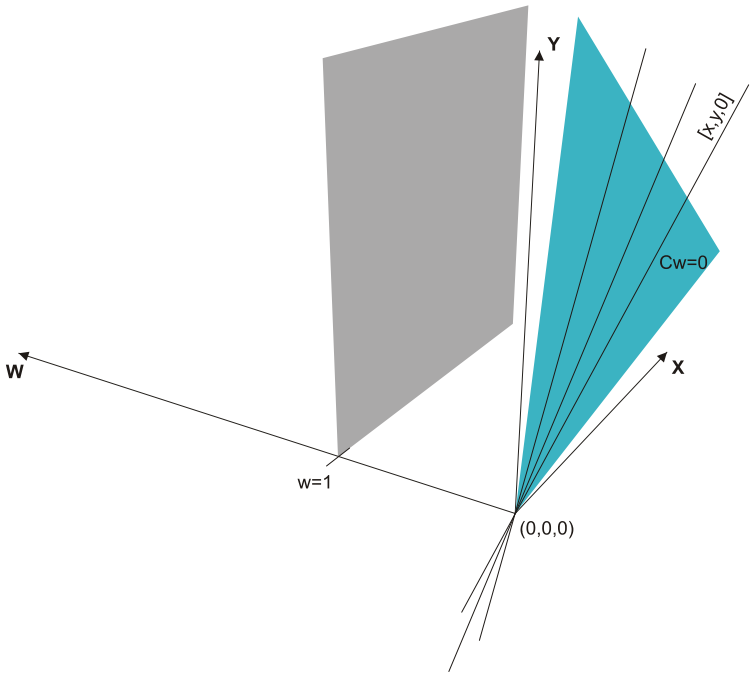
Rozwiązaniem tego układu jest prosta $P = (y_2w_1 - y_1w_2, w_2x_1 - w_1x_2, x_2y_1 - x_1y_2)$. Jeśli potraktować współrzędne prostej P i obu punktów jako wektory w $\mathbb{R}^3$, wynik tej operacji można zapisać za pomocą iloczynu wektorowego:

$$P = X_1 \times X_2.$$

Bardzo podobnie wygląda procedura obliczania punktu X, w którym przecinają się proste A i B:

$$X = A \times B. \tag{2.2}$$

W obu wzorach nie występuję dzielenie, co oznacza, że obie operacje mają zawsze dobrze określony wynik. Przykładowo, dwie proste równoległe $A' = 3x + 2y + 1 = 0$ i $B' = 3x + 2y + 4 = 0$, określone w $\mathbb{R}^2$, mają swoje odpowiedniki w przestrzeni rzutowej $P(\mathbb{R}^2)$ zdefiniowane jako $A = (3, 2, 1)$ oraz $B = (3, 2, 4)$. Podstawiając do wzoru (2.2) otrzymujemy punkt przecięcia się prostych: $X = [-6, 9, 0] = [-2, 3, 0] = [1, -\frac{3}{2}]$.



Rysunek 2.11: Prosta i punkty niewłaściwe w przestrzeni $P(R^2)$ przedstawione w R^3.

X jest punktem niewłaściwym, a jego interpretacja w przestrzeni $\mathbb{R}^2$ to rodzina prostych równoległych o współczynniku kierunkowym $-\frac{3}{2}$, do której należą między innymi A' oraz B'.

Przestrzeń rzutowa $P(\mathbb{R}^3)$, określona na zbiorze $\mathbb{R}^4\backslash\{(0,0,0,0)\}$, jest zdefiniowana na tej samej zasadzie co przestrzeń $P(\mathbb{R}^2)$. Punkt właściwy o współrzędnych $[x, y, z, w]$ można przekształcić w przestrzeń $\mathbb{R}^3$, wykonując rzut na hiperpłaszczyznę $w = 1$ za pomocą wzoru:

$$(x', y', z') = (\frac{x}{w}, \frac{y}{w}, \frac{z}{w}). \tag{2.3}$$

Odpowiednikami prostych $(A, B, C, D) \in P(\mathbb{R}^3)$ są płaszczyzny w przestrzeni trójwymiarowej $\mathbb{R}^3$. Wszystkie proste właściwe (A, B, C, γ), gdzie $\gamma \in \mathbb{R}$, przecinają się w punkcie niewłaściwym $[A, B, C, 0]$. Ten punkt jest odpowiednikiem rodziny płaszczyzn równoległych w $\mathbb{R}^3$, określonych wzorem $Ax + By + Cz + \gamma = 0$.

Przestrzeń $P(\mathbb{R}^3)$ jest najbardziej interesująca z punktu widzenia grafiki komputerowej, gdyż wszystkie funkcje wykonywane na karcie graficznej operują na geometrii zdefiniowanej w trzech wymiarach.

2.1.2. Przekształcenia afiniczne

W przestrzeniach rzutowych nie zachodzą własności klasycznych operacji algebraicznych, nie można również zdefiniować pojęcia odległości ani orientacji. Następujący przykład dobrze ilustruje brak podstawowych reguł arytmetycznych w $P(\mathbb{R}^2)$:

$$[1,0,2]_{1,1} = [2,0,4]_{1,2} \text{ i } [2,0,1]_{2,1} = [6,0,3]_{2,2}, \text{ ale}$$

$$[1,0,2]_{1,1} + [2,0,1]_{2,1} = [3,0,3] \neq [8,0,7] = [2,0,4]_{1,2} + [6,0,3]_{2,2}.$$

Zalety tych przestrzeni można wykorzystać dopiero poprzez ich ścisłe połączenie z klasycznymi odpowiednikami, za pomocą transformacji rzutujących (2.3). Najczęściej wykorzystywaną cechą przestrzeni rzutowych, jest możliwość zapisu dużo szerszej gamy przekształceń w postaci macierzowej. Obejmuje to nie tylko wszystkie przekształcenia afiniczne, ale również rzuty perspektywiczne.

Aby wykorzystać te możliwości, należy postępować zgodnie z następującym schematem:

1. Współrzędne punktów w $\mathbb{R}^3$ przekształcić do przestrzeni $P(\mathbb{R}^3)$, nadając współrzędnej w wartość 1: $(x, y, z) \rightarrow [x, y, z, 1]$.
2. Za pomocą odpowiednio skonstruowanych macierzy $\mathbb{R}^{4x4}$, wykonać wszystkie niezbędne transformacje.
3. Stosując zależność (2.3) przekształcić współrzędne z powrotem do $\mathbb{R}^3$.

Przekształcenie afiniczne w przestrzeni $\mathbb{R}^3$ składa się z macierzy $A \in \mathbb{R}^{3x3}$, reprezentującej część liniową przekształcenia, oraz z wektora przesunięcia $B \in \mathbb{R}^3$:

$$x' = Ax + B. \tag{2.4}$$

Dwie składowe tej transformacji uniemożliwiają składanie kilku przekształceń afinicznych w jedną macierz. Przestrzeń $P(\mathbb{R}^3)$ daje jednak nowe możliwości i konstrukcja takiej zbiorczej macierzy jest możliwa. Operacje w $P(\mathbb{R}^3)$ zachowają własności znane z przestrzeni $\mathbb{R}^3$ pod warunkiem, że przekształcany punkt oraz wynik pomnożenia go przez macierz, będą znajdować się na hiperpłaszczyźnie $w = 1$. W innym przypadku, ze względu na reguły arytmetyczne obowiązujące w $P(\mathbb{R}^3)$, po wykonaniu kroku trzeciego wynikiem nie będzie punkt po przekształceniu afinicznym. Ogólna postać macierzy $\mathbb{R}^{4x4}$ spełniającej ten warunek jest następująca:

$$K = \begin{bmatrix} A & B \\ 0 & 1 \end{bmatrix}, \text{ gdzie } A \in \mathbb{R}^{3x3}, B \in \mathbb{R}^3.$$

Jak się okazuje, macierze A i B będące podmacierzami K, mają identyczną postać jak w zależności (2.4). Konstrukcja macierzy K jest więc bardzo prosta i polega na wstawieniu do niej w nienaruszonej formie, wyrazów odpowiedzialnych za przekształcenie afiniczne w $\mathbb{R}^3$.

Macierze w przestrzeni $P(\mathbb{R}^3)$, odpowiadające najczęściej wykorzystywanym przekształceniom, przedstawione są poniżej.

- Przesunięcie o wektor $P = (P_x, P_y, P_z)$.

$$T = \begin{bmatrix} 1 & 0 & 0 & P_x \\ 0 & 1 & 0 & P_y \\ 0 & 0 & 1 & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Zmiana skali ze współczynnikami S_x, S_y i S_z.

$$S = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Obrót wokół osi $A = (A_x, A_y, A_z)$, przechodzącej przez środek układu współrzędnych, o kąt α. $\|A\| = 1$, $s = \sin(\alpha)$, $c = \cos(\alpha)$.

$$R = \begin{bmatrix} (1-c)A_x^2 + c & cA_xA_y - sA_z & cA_xA_z + sA_y & 0 \\ cA_xA_y + sA_z & (1-c)A_y^2 + c & cA_yA_z - sA_x & 0 \\ cA_xA_z - sA_y & cA_yA_z + sA_x & (1-c)A_z^2 + c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Transformacja współrzędnych z układu L do G. Układ L jest zdefiniowany w układzie G za pomocą czterech wektorów: środka układu $O = (O_x, O_y, O_z)$, osi $X = (X_x, X_y, X_z)$, osi $Y = (Y_x, Y_y, Y_z)$ i osi $Z = (Z_x, Z_y, Z_z)$.

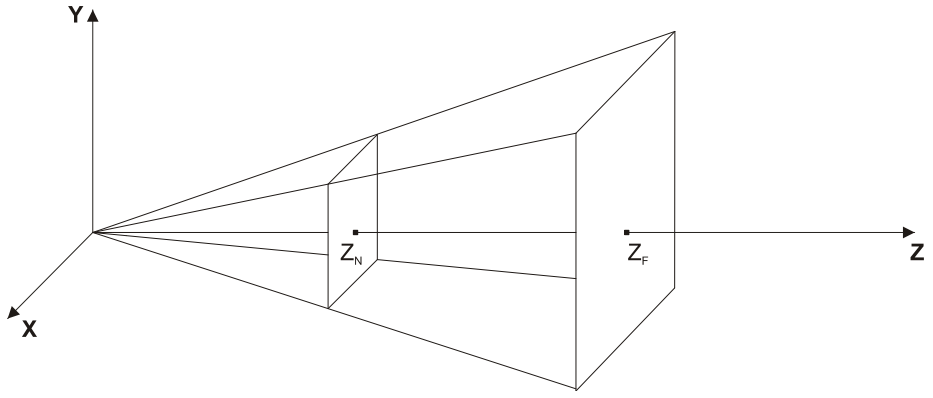
$$T_{L\to G} = \begin{bmatrix} X_x & Y_x & Z_x & O_x \\ X_y & Y_y & Z_y & O_y \\ X_z & Y_z & Z_z & O_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.1.3. Rzut perspektywiczny i równoległy

W celu wyświetlenia generowanego obrazu na ekranie, niezbędna jest transformacja brył trójwymiarowych w ich dwuwymiarowe odpowiedniki. Obiektyw aparatu fotograficznego składa się z szeregu soczewek, które mają za zadanie skupić promienie światła, biegnące z określonego miejsca w przestrzeni, w pewnym punkcie błony fotograficznej. Niestety, sztuczna symulacja takiego układu jest bardzo kosztowna obliczeniowo. W grafice komputerowej najczęściej stosuje się wyidealizowany model obiektywu, który daje obraz ostry niezależnie od odległości obiektu od obserwatora. Efekt głębi ostrości jest w tym przypadku nieosiągalny, ale równocześnie obraz jest pozbawiony wad klasycznego obiektywu, szczególnie jeśli chodzi o zniekształcenia geometrii.

Tak jak w aparacie fotograficznym, w syntetycznym modelu występuje pojęcie kąta widzenia, który określa obszar przestrzeni widoczny na ekranie. Obszar ten jest zbudowany na bazie czterościennego stożka o wierzchołku w środku układu współrzędnych, przez którego środek biegnie oś Z (rys. 2.12). Osie XZ oraz YZ tworzą płaszczyzny symetrii tej bryły. Kąty między przeciwległymi ścianami są ustalane w przedziale od 0 do 180 stopni. Przyjęto, że rozwartość stożka jest wyznaczona przez kąt między jego poziomymi ścianami. Kąt pomiędzy ścianami pionowymi jest wyznaczany tak, aby proporcje obu kątów odzwierciedlały stosunek wysokości do szerokości ekranu.

Ze względu na to, że rzut perspektywiczny ma sens wyłącznie dla punktów o współrzędnej $z > 0$, ustalono dwie płaszczyzny równoległe ograniczające ten stożek. Płaszczyzny te mają równania $z = z_N$ oraz $z = z_F$, gdzie $0 < z_N < z_F$. Płaszczyzna $z = z_N$, zwana rzutnią, jest odpowiednikiem błony filmowej w klasycznym aparacie. Ograniczony sześcioma płaszczyznami fragment przestrzeni, który wyznacza widoczny obszar, jest zwany bryłą widzenia.



Rysunek 2.12: Bryła widzenia.

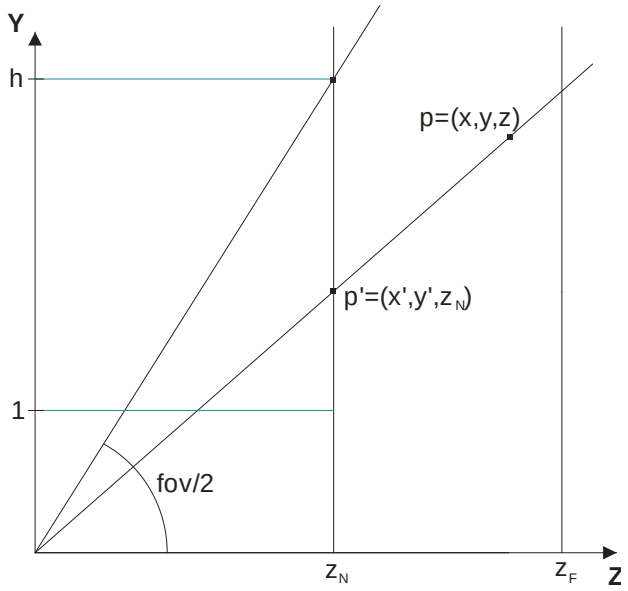
Rzut perspektywiczny punktu p jest zdefiniowany, jako przecięcie prostej przechodzącej przez punkt p oraz środek układu współrzędnych, z płaszczyzną $z = z_N$ (rys. 2.13).

Korzystając z własności $\frac{y}{y'} = \frac{z}{z_N}$ otrzymujemy wzór rzutu perspektywicznego:

$$y' = \frac{yz_N}{z}. \tag{2.5}$$

Na ekranie monitora można przedstawić tylko prostokątny wycinek rzutni. Przyjęto, że wyświetlone na ekranie będą wyłącznie punkty należące do następującego przedziału:

$$(-1 \leq x' \leq 1, -1 \leq y' \leq 1). \tag{2.6}$$



Rysunek 2.13: Rzut perspektywiczny punktu p.

W celu wyświetlenia na ekranie obszaru znajdującego się w bryle widzenia, współrzędne punktów należących do tego obszaru, po wykonaniu rzutu, muszą być zawarte w przedziale (2.6). Aby był spełniony ten warunek, do równania (2.5) należy dodać współczynnik skalujący, który sprowadzi współrzędne do właściwego przedziału. Zgodnie z (rys. 2.13), poprawione równanie ma postać:

$$y' = (\frac{yz_N}{z})/h.$$

Współczynnik h można obliczyć korzystając z zależności $\tan(fov/2) = \frac{h}{z_N}$, gdzie fov jest kątem rozwarcia poziomych boków stożka widzenia. Ostatecznie otrzymujemy:

$$y' = \frac{y \cdot ctg(fov/2)}{z}, \text{ gdzie } fov \in (0, \pi). \tag{2.7}$$

Dla współrzędnej x równanie wygląda niemal identycznie. Rozwartość pionowych boków bryły widzenia może się różnić od rozwartości poziomych, dlatego do równania wprowadza się dodatkowy współczynnik $\lambda = \frac{h}{w}$, gdzie w i h oznaczają odpowiednio szerokość i wysokość rzutni.

$$x' = \frac{x\lambda \cdot ctg(fov/2)}{z} \tag{2.8}$$

Nieliniową zależność zmiennych w równaniach (2.7) i (2.8) da się wyrazić w sposób liniowy w przestrzeni $P(\mathbb{R}^3)$.

Stosując taki sam schemat postępowania jak w przypadku przekształceń afinicznych, można wykorzystać następującą macierz przekształcenia perspektywicznego:

$$P = \begin{bmatrix} \lambda \cdot ctg(fov/2) & 0 & 0 & 0 \\ 0 & ctg(fov/2) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}. \tag{2.9}$$

Każdy punkt o współrzędnych $[x, y, z, 1]$ po przemnożeniu przez macierz (2.9) będzie równy $[\lambda x \cdot ctg(fov/2), y \cdot ctg(fov/2), z, z]$. Dzielenie przez z, będące ostatnim etapem rzutu perspektywicznego, jest realizowane przez przekształcenie $P(\mathbb{R}^3) \rightarrow \mathbb{R}^3$ (2.3), w którym współrzędne dzielone są przez w. Dzięki takiemu rozwiązaniu, wszystkie transformacje włącznie z przekształceniem perspektywicznym można zapisać w postaci jednej zbiorczej macierzy.

Macierz (2.9) powoduje zupełną utratę informacji o odległości punktu od obserwatora, po przejściu do przestrzeni $\mathbb{R}^3$. Wszystkie współrzędne z' są wtedy równe 1. Faza rasteryzacji wymaga, aby punkty były już poddane rzutowi perspektywicznemu. Równocześnie właśnie wtedy potrzebna jest informacja o odległości punktów od obserwatora, która umożliwia wyznaczenie widoczności w buforze-Z. Należy więc tak zmodyfikować macierz (2.9), aby informacje o odległości od obserwatora nie były bezpowrotnie tracone po transformacji z przestrzeni $P(\mathbb{R}^3)$ do $\mathbb{R}^3$.

Rzut perspektywiczny jest określony dla współrzędnych $z \in [a_N, z_F]$ a wartości w buforze-Z muszą być w przedziale $z' \in [0, 1]$. Niezbędne jest takie przekształcenie przedziału $[a_N, z_F]$ w przedział $[0, 1]$, aby uwzględnione było dzielenie przez w, występujące w przekształceniu do przestrzeni R^3. Rozwiązaniem jest następująca zależność:

$$z' = \frac{z\dfrac{z_F}{z_F - z_N} - z_N\dfrac{z_F}{z_F - z_N}}{z}. \tag{2.10}$$

Ostateczna macierz przekształcenia perspektywicznego, po uwzględnieniu poprawek w trzecim wierszu, ma postać:

$$P = \begin{bmatrix} \lambda \cdot ctg(fov/2) & 0 & 0 & 0 \\ 0 & ctg(fov/2) & 0 & 0 \\ 0 & 0 & \dfrac{z_F}{z_F - z_N} & -z_N\dfrac{z_F}{z_F - z_N} \\ 0 & 0 & 1 & 0 \end{bmatrix}. \tag{2.11}$$

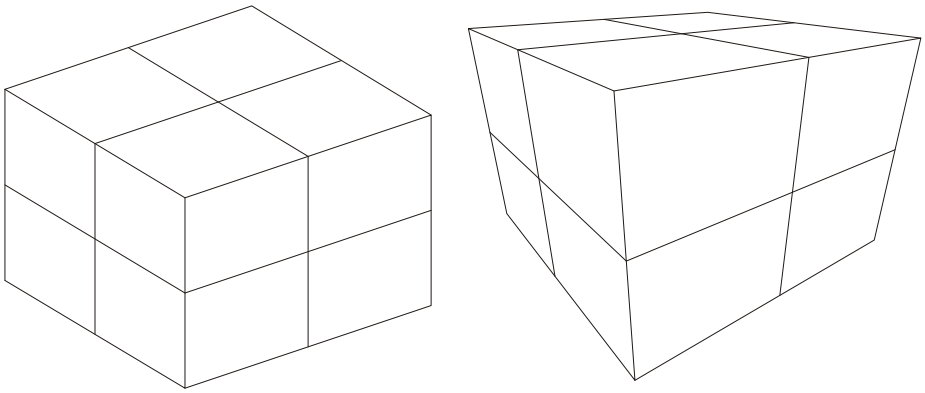
Po zastosowaniu macierzy (2.11) i przejściu do przestrzeni $\mathbb{R}^3$, odległości punktów od obserwatora mogą być w dalszym ciągu porównywane, ale ich proporcje nie są już takie same jak przed rzutem perspektywicznym. Dzieje się tak, ponieważ pochodna funkcji (2.10) nie jest stała. Rozkład odległości po przekształceniu nie jest liniowy i nie można ocenić odległości danego wierzchołka od obserwatora. Funkcja ta jest jednak monotoniczna i dozwolone jest porównywanie wartości ze sobą.

Dużo prostsza w konstrukcji jest macierz rzutu równoległego. Bryła widzenia jest w tym przypadku prostopadłościanem. Zadaniem macierzy jest sprowadzenie współrzędnych punktów w bryle widzenia, do takich samych przedziałów jak w przypadku rzutu

perspektywicznego. Istotną różnicą jest liniowe odwzorowanie odległości od obserwatora w przedziale $z' \in [0, 1]$. Macierz tego przekształcenia jest przedstawiona poniżej, w oraz h oznaczają odpowiednio szerokość i wysokość bryły widzenia.

$$O = \begin{bmatrix} \frac{2}{w} & 0 & 0 & 0 \\ 0 & \frac{2}{h} & 0 & 0 \\ 0 & 0 & \frac{1}{z_F - z_N} & -\frac{z_N}{z_F - z_N} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Układ graficzny wszystkie obliczenia związane z geometrią wykonuje w przestrzeni $P(\mathbb{R}^3)$, następnie zgodnie ze schematem na str. 29 przekształca współrzędne z powrotem do $\mathbb{R}^3$.



Rysunek 2.14: Przykład rzutu równoległego i perspektywicznego.

2.1.4. Obcinanie

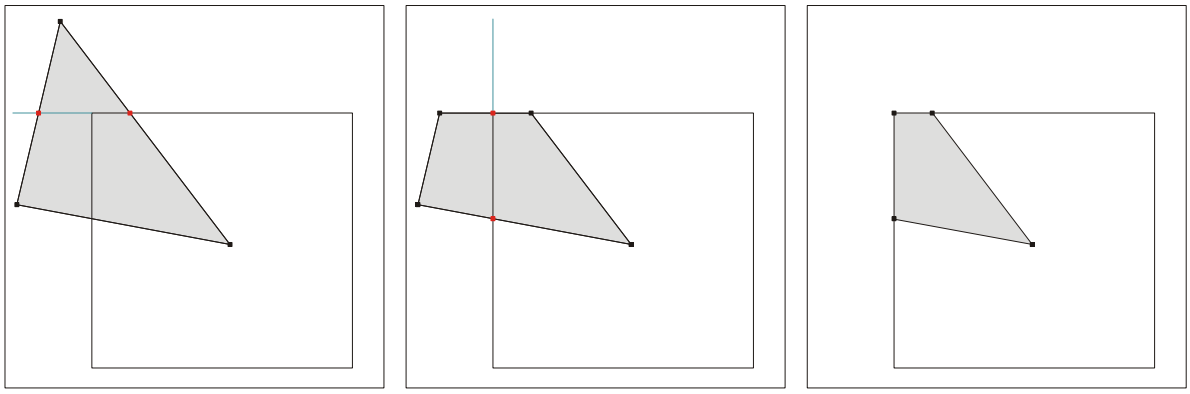
Po przekształceniu punktów do przestrzeni $\mathbb{R}^3$, na ekranie widoczne będą wyłącznie punkty o współrzędnych spełniających układ równań:

$$\left\{ \begin{array}{c} -1 \leq x' \leq 1 \\ -1 \leq y' \leq 1 \\ 0 \leq z' \leq 1 \end{array} \right\}. \tag{2.12}$$

Te trzy przedziały wyznaczają w przestrzeni $\mathbb{R}^3$ prostopadłościan, zwany kanoniczną bryłą widzenia. Konstruując jakiekolwiek przekształcenia rzutujące w $P(\mathbb{R}^3)$, należy tak budować transformacje, aby sprowadzić żądany obszar, który ma być odwzorowany na ekran, do przedziałów (2.12).

Większość przekształceń rzutujących jest dobrze określona tylko na skończonym przedziale przestrzeni $P(\mathbb{R}^3)$, dlatego geometria znajdująca się poza obszarem bryły widzenia jest obcinana i odrzucana jeszcze przed wykonaniem przejścia z $P(\mathbb{R}^3)$ do

$\mathbb{R}^3$, czyli przed podzieleniem współrzędnych przez w. Przyjęto, że obcinane są trójkąty, których wierzchołki przeszły już wszystkie zadane przekształcenia macierzowe, włącznie z transformacjami rzutującymi (np. 2.11). Po operacji obcinania, te części trójkątów, które znajdują się poza obszarem bryły widzenia, zostają odrzucone (rys. 2.15). Wielokąt wypukły, który pozostał w bryle widzenia, jest z powrotem dzielony na trójkąty, które są poddawane dalszym operacjom.



Rysunek 2.15: Proces obcinania trójkąta na przykładzie dwuwymiarowym, z bryłą widzenia w postaci prostokąta.

Sześć płaszczyzn ograniczających obszar (2.12), który będzie narysowany na ekranie, ma swoje odpowiedniki w przestrzeni $P(\mathbb{R}^3)$. Niezależnie od zastosowanej transformacji rzutującej, tuż przed wykonaniem dzielenia przez w, równania tych płaszczyzn są z definicji identyczne (tab. 2.2).

Strona	Równanie w $P(\mathbf{R}^3)$	Równanie w $\mathbf{R}^3$
lewa	$x + w = 0$	$x' = -1$
prawa	$x - w = 0$	$x' = 1$
górna	$y - w = 0$	$y' = 1$
dolna	$y + w = 0$	$y' = -1$
przednia	$z = 0$	$z' = 0$
tylnia	$z - w = 0$	$z' = 1$

Tablica 2.2: Równania płaszczyzn używanych przy obcinaniu.

Podstawowym elementem operacji obcinania jest znalezienie punktu przecięcia pewnej prostej, z jedną z płaszczyzn ograniczających bryłę widzenia. Obcięty wielobok składa się z wierzchołków trójkąta, które znajdują się w bryle widzenia, oraz z punktów nowo utworzonych przez niezbędne przecięcia. Jeśli równanie płaszczyzny ma postać $f(x, y, z, w) = 0$, a prosta przechodząca przez dwa punkty p i q jest dana w postaci parametrycznej $p + t(q - p)$, to zadanie polega na znalezieniu takiej wartości parametru t, aby $f(p + t(q - p)) = 0$. W tym miejscu pojawia się problem. W przypadku, kiedy punkty p i q będą miały współrzędne w o przeciwnych znakach, punkt

przecięcia może w wyniku obliczeń mieć współrzędną $w = 0$. Nie można dopuścić do takiej sytuacji, gdyż takiego punktu nie da się przedstawić w przestrzeni $\mathbb{R}^3$. W związku z tym, w specyfikacjach OpenGL oraz DirectX uznano za dopuszczalne wyłącznie te punkty, które po wszystkich przekształceniach macierzowych w $P(\mathbb{R}^3)$ mają współrzędną $w > 0$. Gwarantuje to zawsze dobrze określony wynik operacji przecięcia.

Założenie dodatniego w jest niezbędne również do wykonania poprawnego rzutu perspektywicznego. Zauważmy, że jeśli punkt $[x, y, z, w]$ ma ujemne współrzędne z oraz w, to wynik przekształcenia $z' = \frac{z}{w}$ ma znak dodatni. Oznacza to, że punkt znajdujący się z tyłu za obserwatorem może się znaleźć w kanonicznej bryle widzenia w $\mathbb{R}^3$. Zawsze dodatni znak współrzędnej w zapobiega całkowicie takiej sytuacji.

Ostatecznie, aby sprawdzić czy dany punkt znajduje się w bryle widzenia w przestrzeni $P(\mathbb{R}^3)$, wystarczy podstawić jego współrzędne do następującego układu nierówności:

$$\begin{Bmatrix} -w \leq x \leq w \\ -w \leq y \leq w \\ 0 \leq z \leq w \end{Bmatrix}.$$

Układ graficzny wykonuje obcinanie całkowicie automatycznie. Podczas obcinania, dla każdego nowo utworzonego wierzchołka są obliczane nowe atrybuty. Atrybuty oryginalnych wierzchołków są liniowo interpolowane na obszarze całego trójkąta, dzięki temu wprowadzanie nowych punktów wielokąta odbywa się bardzo szybko.

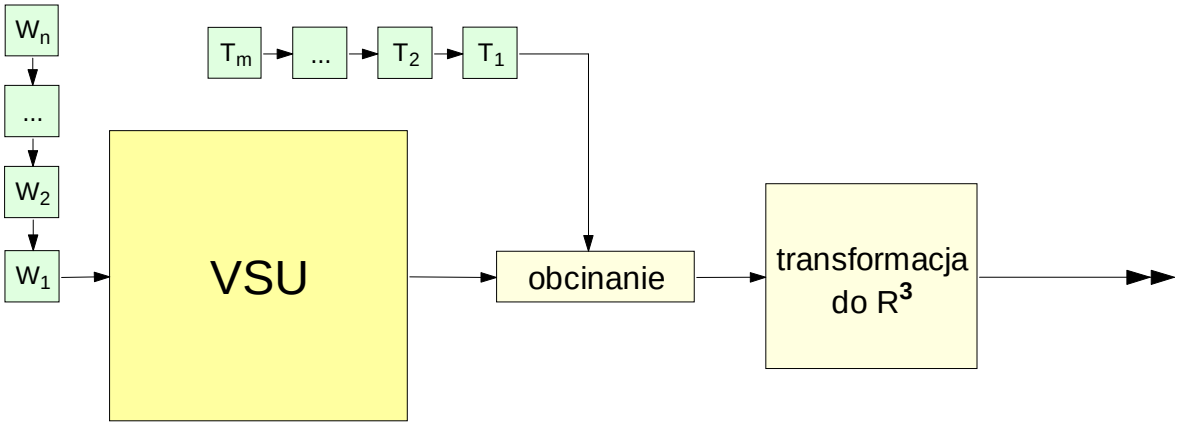
2.2. Programowanie układu

W celu narysowania grupy trójkątów, w pierwszej kolejności należy przygotować dane w pamięci operacyjnej. Dane o wierzchołkach oraz trójkątach są umieszczone w różnych obszarach pamięci i osobno przesyłane do karty. Ponieważ wierzchołki trójkątów są identyfikowane tylko przy pomocy indeksów, informacje o wierzchołkach są przesyłane sekwencyjnie, jeden po drugim. Na każdy wierzchołek składa się od jednego do szesnastu czterowymiarowych wektorów. Semantyczne znaczenie tych wektorów jest zupełnie dowolne, z reguły przynajmniej jeden z nich opisuje pozycję wierzchołka w przestrzeni. Przesłane dane trafiają do jednostki przetwarzającej (ang. Vertex Shader Unit), w której dla każdego wierzchołka jest wykonywany mikroprogram (ang. Vertex Shader). Jego zadaniem jest wygenerowanie przy pomocy danych wejściowych przynajmniej pozycji punktu, opisanej współrzędnymi jednorodnymi. Po wszystkich obliczeniach następuje faza obcinania, w której biorą udział również informacje o trójkątach. Dopiero po obcinaniu współrzędne pozycji punktów są przekształcane do przestrzeni $\mathbb{R}^3$ (rys. 2.16).

Pracą całego układu steruje mikroprogram, który w postaci binarnej jest również przesyłany z pamięci operacyjnej do karty graficznej. W każdym momencie można zmienić aktualnie obowiązujący program, ale czynność ta powoduje duże opóźnienia w pracy układu. Zupełnie nieopłacalne jest wykonywanie osobnego programu dla każdego wierzchołka, dlatego programy są tak konstruowane, aby każdy z nich mógł przetworzyć jak najwięcej wierzchołków sceny. Należy wyraźnie zaznaczyć, że z poziomu programu

są dostępne dane wyłącznie aktualnie przetwarzanego wierzchołka.

Dane wejściowe mogą być dowolne, ale proces na danym wierzchołku musi zakończyć się wygenerowaniem przynajmniej współrzędnej wierzchołka w przestrzeni $P(\mathbb{R}^3)$. Przykładowo, aby narysować falującą powierzchnię jeziora, współrzędne x oraz z mogą być podane bezpośrednio a wysokość danego punktu nad powierzchnią może być generowana proceduralnie wewnątrz mikroprogramu, na przykład przy pomocy funkcji trygonometrycznych. Ogólna strategia w tego typu sytuacjach polega na jak największym odciążeniu procesora głównego, kosztem pracy układu graficznego. Dzięki asynchronicznej pracy obu tych elementów możliwa jest ich równoległa praca.

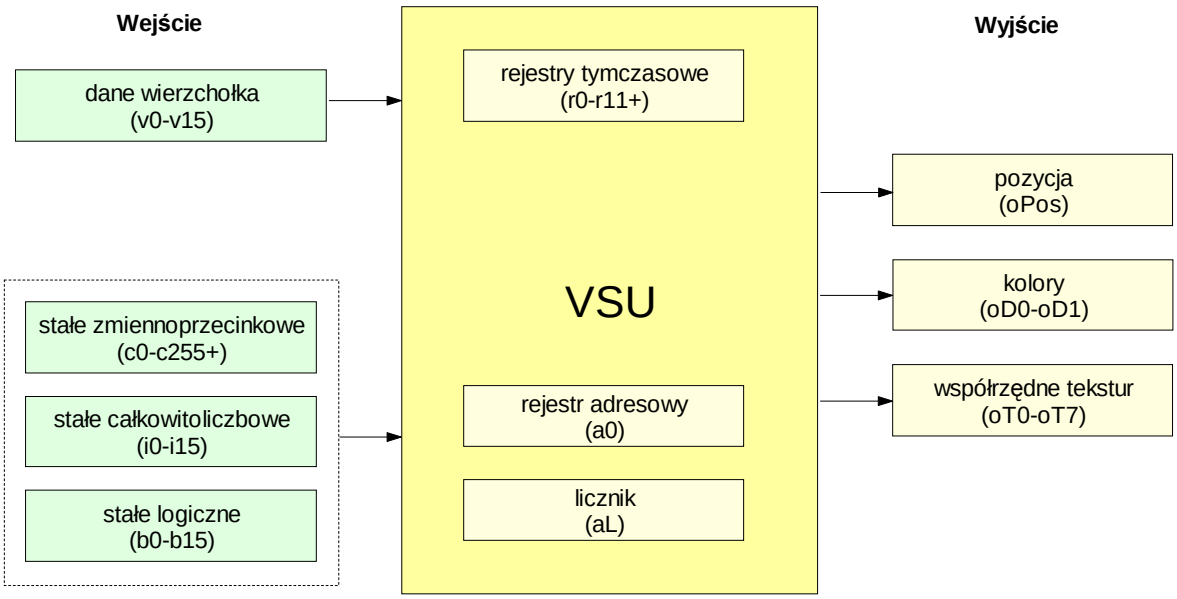


Rysunek 2.16: Proces przetwarzania n wierzchołków przy udziale m trójkątów.

Współczesne procesory graficzne posiadają więcej niż jedną jednostkę przetwarzającą wierzchołki. Dzięki temu, że mikroprogram ma dostęp wyłącznie do informacji o aktualnie przetwarzanym wierzchołku, układ może równolegle wykonywać operacje na kilku wierzchołkach na raz. Obecnie produkowane karty graficzne posiadają od czterech do szesnastu jednostek VSU.

Głównym zadaniem jednostki przetwarzającej wierzchołki jest dostarczenie ostatecznych współrzędnych wierzchołków oraz ich atrybutów. Podczas rasteryzacji, dla każdego piksela są obliczane atrybuty, które są kombinacją liniową atrybutów z trzech wierzchołków. Atrybuty dla wierzchołków są opcjonalnie generowane wcześniej przez mikroprogram i wszystkie mają postać czterech liczb rzeczywistych. Mogą to być dwa kolory ($RGBA$) oraz osiem współrzędnych dla tekstur ($XYZW$). Dane te nie muszą być wykorzystane zgodnie z ich nazwami, jednostka zajmująca się obliczaniem koloru poszczególnych pikseli może je wykorzystać dowolnie, zgodnie z myślą programisty.

Mikroprogram składa się z maksymalnie 256 instrukcji specjalnego mikroasemblera, które operują na zestawie rejestrów układu. Rejestry dzielą się na wejściowe, wyjściowe oraz tymczasowe (rys. 2.17). Przy każdym uruchomieniu programu w 16 rejestrach $v0 - v15$ znajdują się dane wejściowe kolejnego wierzchołka, każda z nich składa się z czterech liczb rzeczywistych ($XYZW$). Oprócz tych informacji programista może dowolnie ustalić zawartość rejestrów reprezentujących stałe, które są niezależne od ko-



Rysunek 2.17: Jednostka przetwarzania wierzchołków.

lejnych uruchomień programu. Do dyspozycji jest co najmniej 256 wektorowych stałych rzeczywistych $c0 - c255$ w postaci $(XYZW)$, 16 wektorowych stałych całkowitoliczbowych $i0 - i15$ w postaci $(XYZW)$, oraz 16 stałych logicznych $b0 - b15$ przyjmujących wartość zero lub jeden. Zawartość tych rejestrów może być zmieniana w dowolnym momencie, ale wprowadza to duże opóźnienia, dlatego nie powinno się zmieniać ich stanu zbyt często. Należy zaznaczyć, że wszystkie rejestry wejściowe służą tylko do odczytu, natomiast rejestry wyjściowe można wyłącznie zapisywać.

Rejestry tymczasowe są dostępne wyłącznie z poziomu mikroprogramu. Należy do nich przynamniej 12 wektorowych rejestrów rzeczywistych $r0-r11$ w postaci $(XYZW)$, czterokomponentowy całkowitoliczbowy rejestr adresowania pośredniego $a0$ w postaci $(XYZW)$ oraz licznik ogólnego przeznaczenia aL.

Głównym zadaniem programu jest wygenerowanie danych w rejestrach wyjściowych. Należy do nich rejestr pozycji $oPos$ w postaci czterech liczb rzeczywistych $(XYZW)$, dwa rejestry koloru $oD0 - oD1$ w postaci czterech liczb rzeczywistych $(RGBA)$ oraz 8 rejestrów dla współrzędnych tekstur $oT0 - oT7$, również w postaci czterech liczb rzeczywistych $(XYZW)$. Każdy program ma obowiązek zapisać przynajmniej rejestr pozycji $oPos$.

2.2.1. Budowa programu

Programowanie w języku maszynowym jest bardzo czasochłonne. Ponieważ rozkazy mają bezpośrednie przełożenie na kod binarny, aby wykonać nawet prostą operację arytmetyczną należy ich użyć przynajmniej kilku. Kod programu staje się mało czytelny i trudny w późniejszej modyfikacji. Wraz ze wzrostem popularności w pełni

programowalnych układów powstały języki wysokiego poziomu, które są interpretowane i tłumaczone na język maszynowy układu graficznego. W tej chwili obowiązują trzy bardzo podobne do siebie języki, CG opracowany przez firmę NVIDIA, ASHLI stworzony przez firmę ATI, oraz HLSL (ang. High Level Shading Language) wprowadzony przez Microsoft. Wszystkie są oparte na konstrukcjach znanych z języka C i bez większych zmian mogą być stosowane zamiennie. Ponieważ język HLSL jest wbudowany w API DirectX, w tej pracy właśnie on będzie stanowił bazę do konstruowania mikroprogramów.

Mikroprogram składa się z zestawu funkcji, w którym jedna jest wyróżniona przy kompilacji jako startowa. Po kompilacji program, który miał wcześniej budowę strukturalną, jest zamieniany na pojedynczy ciąg instrukcji maszynowych.

Język HLSL umożliwia dowolne deklarowanie zmiennych oraz stałych w każdym miejscu programu. Ostateczne rozmieszczenie danych w wewnętrznych rejestrach procesora graficznego jest zależne tylko i wyłącznie od kompilatora. Programista nie ma żadnego wpływu na sposób wykorzystania rejestrów ani ich liczbę. Ze względu na bardzo ograniczone zasoby jednostek VSU, kod programu musi być na tyle prosty, aby ilość niezbędnych rejestrów tymczasowych oraz rejestrów dla stałych mieściła się w określonych limitach. Kompilator automatycznie wykorzystuje rejestry tymczasowe w jak najlepszy sposób, oraz sam rozmieszcza w dostępnej pamięci stałe zadeklarowane w programie.

Stałe mogą być inicjalizowane spoza mikroprogramu, służy do tego wygenerowana przez kompilator tablica stałych, za pomocą której można poprzez nazwę uzyskać dostęp do odpowiednich danych. Rola procesora głównego, czyli programu uruchomionego na komputerze, polega na wysłaniu do karty danych o wierzchołkach i trójkątach, wysłaniu binarnego kodu mikroprogramu oraz na ustaleniu wartości stałych. Resztę wykonuje już procesor graficzny, w tym jednostki VSU.

Jeśli zmienna zadeklarowana w mikroprogramie ma pełnić funkcję rejestru wejścia lub wyjścia, należy dodać do deklaracji dodatkową informację. W przypadku rejestrów wyjściowych będzie to nazwa, która identyfikuje konkretny rejestr: $POSITION$, $COLOR0 - COLOR1$ lub $TEXCOORD0 - TEXCOORD7$. Nazwy te odpowiadają bezpośrednio rejestrom wyjściowym jednostki VSU (rys. 2.17).

Dane wejściowe dla wierzchołka składają się z maksymalnie 16 wektorów. Nie każdy wektor musi być wypełniony w całości, jeśli potrzebna jest tylko pojedyncza wartość to pozostałe trzy nie są przesyłane przez szynę. W programie głównym, wykonywanym przez procesor komputera, trzeba wypełnić tabelę, która będzie zawierać informacje o liczbie oraz formacie danych dla przesyłanych wierzchołków. Każdemu z maksymalnie 16 komponentów jest przyporządkowany typ oraz tekstowy identyfikator. Po przesłaniu danych do karty, poszczególne komponenty wierzchołka są przyporządkowywane zmiennym w mikroprogramie, które jako dodatkową informację posiadają ten sam identyfikator. W ten sposób uniknięto bezpośredniego wiązania danych z konkretnymi rejestrami wejściowymi $v0 - v15$. Kompilator decyduje którą zmienną umieścić w danym rejestrze. Dla programisty ważna jest tylko zgodność identyfikatorów zadeklarowanych w programie z tymi, które są umieszczone przy zmiennych w mikroprogramie dla układu graficznego.

Identyfikatory poszczególnych komponentów nie mogą być dowolne, jest to spowodowane pozostałościami po poprzednich architekturach, w których identyfikatory speł-

niały również funkcje określające zastosowanie konkretnych danych. W przypadku, kiedy programista nie chce w żaden sposób sugerować funkcji określonego komponentu, do identyfikacji należy użyć 16 nazw: $TEXCOORD0$ do $TEXCOORD15$.

Poniższy przykład przedstawia prosty mikroprogram, który przetwarza wierzchołki składające się z pozycji oraz koloru. Dane wejściowe każdego wierzchołka są przesyłane pod identyfikatorami $TEXCOORD0$ (pozycja - 3 liczby) oraz $TEXCOORD1$ (kolor - 3 liczby), następnie w mikroprogramie są przyporządkowane składowym struktury *vin*. Ten program tylko przepisuje dane z rejestrów wejściowych do wyjściowych.

```
struct T_VIN  // dane wierzchołka
{
  float3 pos :  TEXCOORD0; // pozycja
  float3 col : TEXCOORD1; // kolor
};

struct T_VOUT  // dane wyjściowe
{
  float4 pos : POSITION; // pozycja
  float3 col :  COLOR0; // kolor
};

T_VOUT vmain(T_VIN vin)  // mikroprogram
{
  T_VOUT vout;

  (float3)vout.pos=vin.pos;  // przepisanie danych
  vout.pos.w=1;
  vout.color=vin.color;

  return vout;
}
```

(Program 2.1)

Do podstawowych typów danych należą: bool (wartość logiczna true lub false), int (liczba całkowita), float (liczba rzeczywista), half (liczba rzeczywista o zmniejszonej precyzji) oraz double (liczba rzeczywista podwójnej precyzji). Każdy typ danych może występować w postaci pojedynczej, wektorowej lub macierzowej, na przykład int4 oznacza wektor czteroelementowy a float3x4 macierz o 3 wierszach i 4 kolumnach. Zmienne o podstawowych typach danych mogą być grupowane w postaci struktur i tablic.

2.2.2. Zestaw instrukcji

Mikroprogram składa się z zestawu funkcji, w których oprócz klasycznych predefiniowanych instrukcji sterujących mogą występować instrukcje wykonujące specyficzne operacje arytmetyczne. Istnieje pewne podobieństwo tych instrukcji do funkcji bibliotecznych, wchodzących w skład klasycznych dystrybucji języka C, ale ich zestaw jest z góry określony i nie może być poszerzany. Wszystkie funkcje występujące w wersji 2.0 wraz z krótkim opisem ich działania znajdują się w tab. 2.3 oraz w tab. 2.4. Szczegółowe informacje można znaleźć w dokumentacji do języka HLSL [2].

W standardzie 2.0 podczas przetwarzania wierzchołków jest możliwa tylko tzw. statyczna kontrola przepływu w programie. Oznacza to, że instrukcje if, for oraz while nie mogą mieć jako parametrów zmiennych dynamicznych, ale wyłącznie stałe. Nie można

dynamicznie zmieniać ilości powtórzeń pętli ani obliczać kryterium dla instrukcji warunkowej wewnątrz mikroprogramu. Wszystkie te parametry należy ustalić wcześniej w tablicy dla stałych. Powyższe warunki mocno ograniczają zastosowanie instrukcji warunkowych oraz powtórzeniowych, w praktyce służą one wyłącznie do włączania lub wyłączania określonych fragmentów programu oraz do powtarzania fragmentu kodu z góry określoną ilość razy. Dla przykładu, jeśli programista chce napisać program obliczający natężenie oświetlenia w danym wierzchołku, musi wziąć pod uwagę, że światło może padać na ten punkt z kilku źródeł jednocześnie. Statyczna kontrola przepływu pozwala cały kod umieścić w jednym mikroprogramie, który jako stałą przyjmuje ilość świateł mających wpływ na dany wierzchołek. W przypadku braku takiego rozwiązania, programista musiałby napisać osobny program dla każdej możliwej liczby świateł.

W obecnych czasach, powyższy przykład ma znaczenie tylko teoretyczne, ponieważ coraz większe wymagania co do jakości generowanego obrazu wymusiły na producentach stosowanie oświetlenia obliczanego w każdym pikselu, a nie tylko na wierzchołkach. Twórcy układów scalonych nie nadążają za wymaganiami i w mikroprogramach dla pikseli brakuje jakiejkolwiek kontroli przepływu, a kontrola dla wierzchołków stała się w przypadku oświetlenia bezużyteczna. W następnej generacji akceleratorów ten fakt ma się zmienić i dla pikseli ma być już wprowadzona statyczna kontrola przepływu.

Instrukcja	**Opis**
abs(x)	wartość bezwzględna x
acos(x)	arcus cosinus x
all(x)	testuje czy wszystkie komponenty x są $\neq 0$
any(x)	testuje czy którykolwiek komponent x jest $\neq 0$
asin(x)	arcus sinus x
atan(x)	arcus tangens x
atan2(x,y)	arcus tangens wartości $\frac{x}{y}$
ceil(x)	najmniejsza liczba całkowita $\geq$ x
clamp(x,min,max)	obcina x do przedziału [min,max]
cos(x)	cosinus x
cosh(x)	cosinus hiperboliczny x
cross(x,y)	iloczyn wektorowy x $\times$ y
degrees(x)	przelicza radiany na stopnie
determinant(m)	wyznacznik macierzy kwadratowej m
distance(x,y)	odległość między punktami x i y
dot(x,y)	iloczyn skalarny wektorów x i y
exp(x)	e^x
exp2(x)	2^x
faceforward(i,j,k)	oblicza -i·sign(dot(j,k))
floor(x)	największa liczba całkowita $\leq$ x
fmod(x,y)	reszta z dzielenia x przez y

Tablica 2.3: Instrukcje jednostki VSU.

Instrukcja	**Opis**
frac(x)	część ułamkowa x
frexp	zwraca część całkowitą i ułamkową danej liczby
isfinite(x)	true jeśli x jest skończone
isinf(x)	true jeśli x równe -INF lub INF
isnan(x)	true jeśli x równe NAN lub QNAN
ldexp(x,exp)	$x \cdot 2^{exp}$
length(x)	długość wektora x
lerp(x,y,s)	x+s(y-x)
lit(l,h,m)	zwraca wektor [1, l < 0 ? 0 : l, l < 0 || h < 0 ? 0 : h·m]
log(x)	$\log_e(x)$
log2(x)	$\log_2(x)$
log10(x)	$\log_{10}(x)$
max(x,y)	x > y ? x : y
min(x,y)	x < y ? x : y
modf	oblicza dzielenie całkowitoliczbowe
mul(x,y)	mnoży elementy
normalize(x)	$\frac{x}{length(x)}$
pow(x,y)	x^y
radians(x)	przelicza stopnie na radiany
reflect(i,n)	i-2·dot(i,n)·n (wektor odbicia i od płaszczyzny n)
refract	oblicza wektor przejścia między dwoma ośrodkami
round(x)	zaokrągla x do najbliższej liczby całkowitej
rsqrt(x)	$\frac{1}{sqr(x)}$
saturate(x)	clamp(x,0,1)
sign(x)	0 jeśli x=0, 1 jeśli x>0, -1 jeśli x<0
sin(x)	sinus x
sincos	oblicza sinus i cosinus x
sinh(x)	sinus hiperboliczny x
smoothstep	oblicza interpolację Hermite'a
sqrt(x)	pierwiastek kwadratowy x
step(x)	(x ≥ y) ? 1 : 0
tan(x)	tangens x
tanh(x)	tangens hiperboliczny x
transpose(k)	transponuje macierz k

Tablica 2.4: Instrukcje jednostki VSU, ciąg dalszy.

2.2.3. Podstawowe techniki

Mimo iż zestaw instrukcji jednostek VSU jest duży, układ graficzny nie może wykonać wszystkich niezbędnych operacji związanych z wierzchołkami. Podstawowym problemem jest dostępność informacji tylko o pojedynczych wierzchołkach, przez co jakiekolwiek czynności obejmujące topologię obiektu musi przeprowadzać procesor główny. Należy również pamiętać, że niektóre konstrukcje w programie mogą nie zostać skompilowane nawet, jeśli program jest semantycznie poprawny. Wynika to z dużych obostrzeń, przede wszystkim dotyczących dostępnych zasobów rejestrów i pamięci.

Do podstawowych i najważniejszych zastosowań jednostek przetwarzających wierzchołki należą transformacje macierzowe. Odciążenie procesora głównego nawet tylko jednym mnożeniem macierzy na wierzchołek daje ogromne oszczędności czasowe. W typowych aplikacjach występuje wiele obiektów ruchomych, przez co ilość niezbędnych transformacji układów odniesienia oraz innych przekształceń jest na tyle duża, że pomoc układu graficznego jest nieoceniona.

Poniższy program przedstawia najczęściej wykonywaną operację, przemnożenia współrzędnych każdego wierzchołka przez macierz, która odpowiada transformacji współrzędnych z układu obiektu do układu obserwatora. Układ obiektu często jest nazywany układem lokalnym, w przeciwieństwie do układu globalnego, w którym określane są ostateczne pozycje wszystkich elementów sceny. W macierzy mxLocalToView musi być również zawarty rzut perspektywiczny lub równoległy.

```
float4x4 mxLocalToView; // macierz transformacji

struct T_VIN // dane wierzchołka
{
  float3 pos :  TEXCOORD0; // pozycja
};

struct T_VOUT // dane wyjściowe
{
  float4 pos : POSITION; // pozycja
};

T_VOUT vmain(T_VIN vin)
{
  T_VOUT vout;
  float4 wpos;

  (float3)wpos=vin.pos;
  wpos.w=1;

  // transformcja wierzchołka
  vout.pos=mul(mxLocalToView,wpos);

  return vout;
}
```

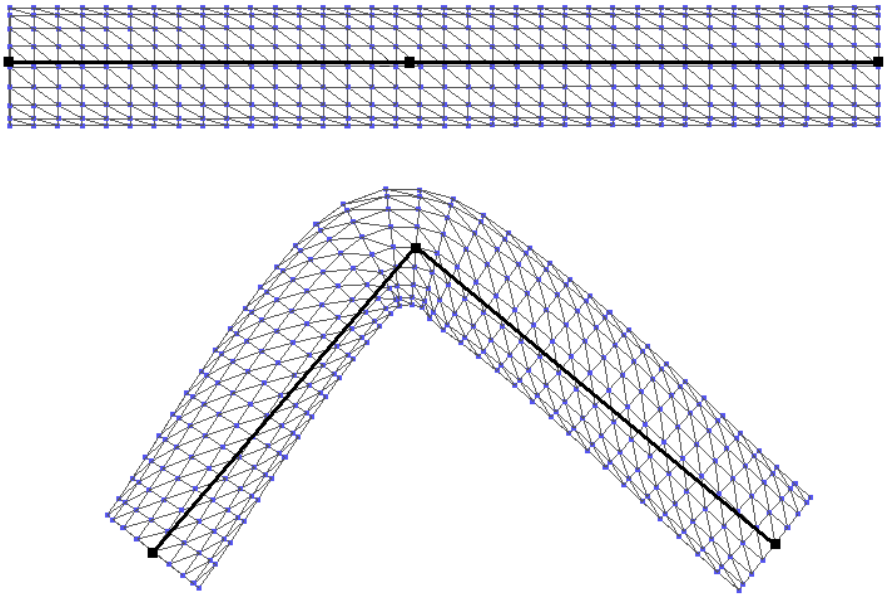
(Program 2.2)

W typowej scenie jest umieszczona pewna ilość obiektów, każdy z nich składa się z określonej ilości trójkątów oraz wierzchołków. Animować obiekty w całości można poprzez zmianę macierzy transformującej go do układu globalnego. Przy takiej technice

animacji, w pamięci procesora graficznego wystarczy umieścić program (2.2) i przy rozpoczynaniu przetwarzania każdego nowego obiektu zmieniać tylko postać macierzy mxLocalToView, przesyłając do karty cały czas te same współrzędne wierzchołków.

W programie (2.2) współrzędne wierzchołków są przesyłane w postaci trzech (XYZ) a nie czterech komponentów (XYZW). Dzieje się tak dlatego, że przede wszystkim należy unikać przesyłania zbędnych informacji przez szynę, w tym przypadku byłaby to współrzędna w, domyślnie równa 1. Konsekwencją takiego postępowania jest konwersja na typ czterokomponenetowy tuż przed przemnożeniem przez macierz. Za współrzędną w podstawia się wtedy 1.

Interesujące efekty można uzyskać stosując kombinację liniową między wynikami przemnożenia tego samego wierzchołka przez kilka różnych macierzy. W taki właśnie sposób w programach graficznych i grach symuluje się efekt napinania się skóry na stawach postaci (rys. 2.18).



Rysunek 2.18: Kombinacja liniowa dwóch przekształceń macierzowych na wierzchołkach walca.

Pierwszym etapem jest przygotowanie modelu i przyporządkowanie wierzchołkom odpowiednich kości. Czynność ta jest przeprowadzana, gdy model jest w stanie spoczynku (górna część rys. 2.18) a każdy wierzchołek ma swoje współrzędne przedstawione w układzie globalnym G. W środku modelu umieszcza się połączone ze sobą odcinki, które odpowiadają prawdziwym kościom. Każdy odcinek wyznacza pewien układ współrzędnych znajdujący się w G.

Niech macierz M oznacza dowolne przekształcenie afiniczne, które odpowiada transformacji kości. Jeśli współrzędne dowolnego wierzchołka przemnożymy przez macierz M, wierzchołek w stosunku do przekształconej przez M kości będzie dokładnie w tej samej pozycji co wcześniej. W przypadku przykładu z walcem, wierzchołki na końcach

powinny poruszać się razem z kością a wierzchołki w pobliżu stawu muszą zostać uśrednione tak, aby wpływ na nie miały obie przyłączone do stawu kości. Ten efekt można uzyskać poprzez kombinację liniową współrzędnych uzyskanych po przemnożeniu wierzchołka przez macierze M_K^1 i M_K^2, odpowiadające obu kościom. Warunkiem uzyskania poprawnych wizualnie wyników jest sumowanie się współczynników przy kombinacji do 1. Ogólne równanie przekształcenia wierzchołka przy użyciu n kości wygląda następująco:

$$v' = \sum_{i=1}^{n} \alpha_i \cdot M_K^i \cdot v, \text{ gdzie } \sum_{i=1}^{n} \alpha_i = 1 \tag{2.13}$$

Dla dwóch kości, wagi α w pobliżu stawu zbliżają się do $\frac{1}{2}$. W środku kości jeden z nich zbliża się do 1 a drugi do 0. Wagi są przyporządkowywane poszczególnym wierzchołkom na stałe w specjalnym programie graficznym, który wspomaga ten proces.

Zadaniem mikroprogramu jest przeprowadzenie operacji związanych z równaniem (2.13). Do danych przesyłanych do karty dodaje się wagi wierzchołków a macierze M_K wprowadza się jako stałe. Dla każdego wierzchołka może być kilka współczynników wagowych, w zależności od ilości kości mających wpływ na jego pozycję. Ponieważ w mikroprogramach można adresować pośrednio tylko rejestry stałych rzeczywistych, trzeba niestety pisać osobny program dla każdej możliwej ilości kości. Z tego względu, aby nie zmieniać programu zbyt często, wierzchołki wysyła się do karty pogrupowane według ilości kości do nich przyporządkowanych. Poniższy program demonstruje tą technikę dla trzech kości.

```
float3x4 bones[3]; // macierze kości
float4x4 mx_Global2View;

struct T_VIN // dane wierzchołka
{
  float3 pos :  TEXCOORD0; // pozycja
  float w[2] :  TEXCOORD1; // wagi
};

struct T_VOUT // dane wyjściowe
{
  float4 pos :  POSITION;
};

T_VOUT vmain(T_VIN vin) // mikroprogram
{
  T_VOUT vout;
  float3 p1,p2,p3;
  float4 pos2;

  // 3 pozycje wierzchołka
  p1=mul(bones[0],vin.pos);
                          ↓
```

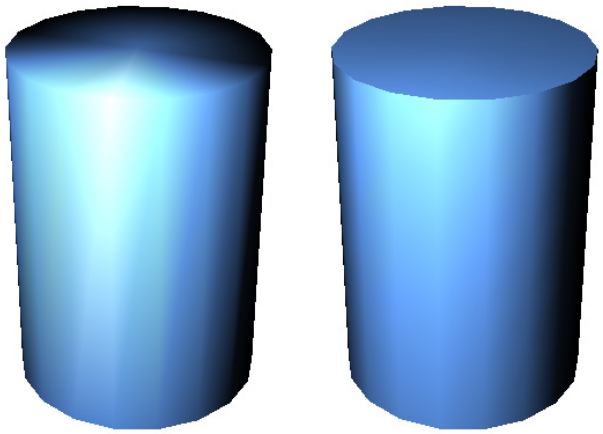
(Program 2.3)

```
                                    ↑
  p2=mul(bones[1],vin.pos);
  p3=mul(bones[2],vin.pos);

  // wagowanie
  (float3)pos2=vin.w[0]*p1+vin.w[1]*p2+(1-vin.w[0]-vin.w[1])*p3;
  pos2.w=1.0f;

  // transformacja
  vout.pos=mul(mx_Global2View,pos2);

  return vout;
}
```

(Program 2.3)

Również w tym przypadku, ze względu na jak najmniejszą ilość przesyłanych informacji, ostatnia waga dla wierzchołka nie jest przesyłana. Można ją w prosty sposób obliczyć, odejmując od jedynki pozostałe wagi.

Ostatnim z podstawowych sposobów wykorzystania jednostki VSU jest obliczanie natężenia oświetlenia padającego na wierzchołki. Każdy wierzchołek jest częścią wspólną kilku trójkątów, które mają wektory normalne do swojej powierzchni. Aby obliczyć oświetlenie, należy określić wektor normalny do powierzchni obiektu w każdym z wierzchołków. Najczęściej wektory z otaczających wierzchołek trójkątów są uśredniane. Dzięki temu oświetlenie będzie płynnie zmieniało intensywność na wierzchołkach mimo tego, że obiekt jest w rzeczywistości kanciasty. Należy pamiętać, że ostatecznie obliczony kolor oświetlenia będzie w obrębie trójkąta interpolowany liniowo i przy małej ilości trójkątów efekt nie będzie zadowalający (lewa strona rys. 1.6).



Rysunek 2.19: Oświetlenie na wierzchołkach. Walec z lewej strony ma wspólne wektory normalne dla wszystkich ścian na krawędzi walca.

Przy obliczaniu wektora normalnego na wierzchołkach są używane również bardziej zaawansowane techniki, uwzględniające na przykład rozmiar poszczególnych trójkątów. Czasami wierzchołek należy do ostrej krawędzi, która nie powinna zostać oświetlona

w sposób gładki. Ponieważ kolor wierzchołka będzie interpolowany na każdą ze ścian, z którą ten wierzchołek jest połączony, należy skopiować jego pozycję i stworzyć nowe wierzchołki w tym samym miejscu. W ten sposób w punkcie, gdzie leżał pierwotny wierzchołek, spotkają się różne kolory (rys. 2.19).

W przypadku obliczania intensywności z wielu źródeł światła, wystarczy napisać tylko jeden mikroprogram, który będzie działał niezależnie od ich liczby. Jest to możliwe, ponieważ dane o światłach są umieszczone w przestrzeni przeznaczonej na stałe, a stałe można indeksować pośrednio. Informacje o wierzchołkach składają się z pozycji, wektora normalnego i koloru powierzchni. Program (2.4) oblicza kolor odbity od wierzchołka w kierunku obserwatora, przy czym liczba świateł jest ograniczona wyłącznie ilością dozwolonych stałych. W programie zastosowano model Phonga [8]:

$$k = k_s \cos(N, L) + k_d \cos^n(R, E), \tag{2.14}$$

gdzie k_s oznacza składową rozproszona światła, k_d - składową zwierciadlaną, N - wektor normalny do powierzchni, L - kierunek padania światła, R to wektor L odbity od powierzchni, E - kierunek patrzenia obserwatora, n - współczynnik gładkości powierzchni.

```
struct T_LIGHT // dane światła
{
  float3 pos; // pozycja światła
  float3 col_diff; // składowa rozproszona
  float3 col_spec; // skł.  zwieciadlana
  float pn; // wsp.  gładkośći
};

int n_lights; // ilość świateł
T_LIGHT lights[8]; // światła (maks.  8)

float4x4 mxLocalToView; // macierz transformacji
float3 eye_pos; // pozycja obserwatora

struct T_VIN // dane wierzchołka
{
  float3 pos :  TEXCOORD0; // pozycja
  float3 n :  TEXCOORD1; // wektor normalny
  float3 col :  TEXCOORD2; // kolor
};

struct T_VOUT // dane wyjściowe
{
  float4 pos :  POSITION; // pozycja
  float3 col :  COLOR0; // kolor
};

T_VOUT vmain(T_VIN vin) // mikroprogram
{
  T_VOUT vout;
  int i;
  float3 eye,eye_rfl,light;
                          ↓
```

(Program 2.4)

```
                          ↑
  float lp;
  float4 pos={0,0,0,1};

  vout.col=(float3)pos;
  eye=normalize(eye_pos-vin.pos);
  eye_rfl=reflect(eye,vin.n);

  // obliczenie natężenia swiatła
  for(i=0;i<n_lights;i++)
  {
    light=normalize(lights[i].pos-vin.pos);

    // składowa rozproszona
    lp=max(0,dot(vin.n,light));
    vout.col+=lights[i].col_diff*lp;

    // składowa zwierciadlana
    lp=max(0,pow(dot(eye_rfl,light),lights[i].pn));
    vout.col+=lights[i].col_spec*lp;
  }

  // kolor powierzchni
  vout.col*=vin.col;
  // pozycja
  (float3)pos=vin.pos;
  vout.pos=mul(mxLocalToView,pos);

  return vout;
}
```

(Program 2.4)

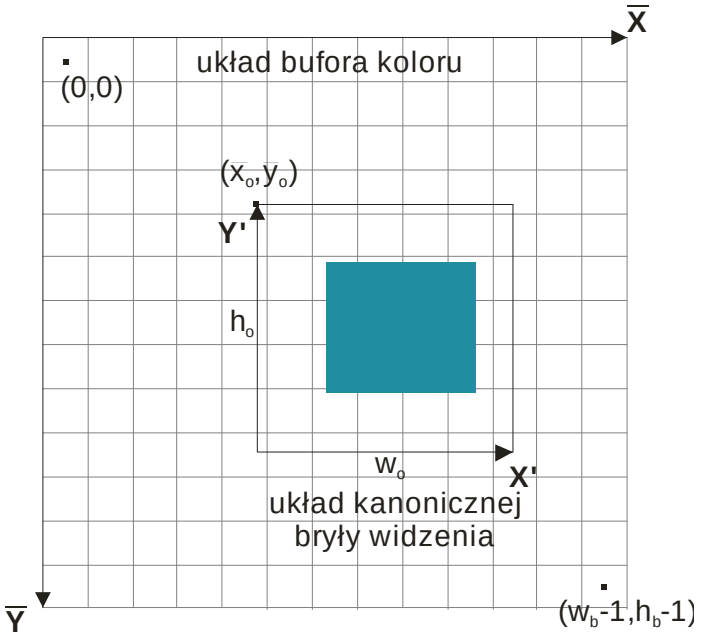
W celu przyspieszenia obliczeń, pozycje świateł oraz obserwatora są podane w tym samym układzie współrzędnych, co wierzchołki obiektu. Również w celach optymalizacyjnych zamiast odbijać od powierzchni wektory kierunku świateł, odbijany jest wektor kierunku patrzenia obserwatora.

Rozdział 3

Generowanie obrazu

Po zakończeniu procesu przetwarzania wierzchołków, informacje o scenie składają się ze współrzędnych wierzchołków w przestrzeni $\mathbb{R}^3$, pogrupowanych w trójkąty. Współrzędne te są obrazem obiektów po przekształceniach rzutujących $(x, y, z, w) \rightarrow (x', y', z')$ i wszystkie należą do tzw. kanonicznej bryły widzenia. Pary $(x', y') \in ([-1, 1], [-1, 1])$ są współrzędnymi na płaszczyźnie rzutni a $z' \in [0, 1]$ jest dodatkową informacją o wzajemnej odległości punktów od obserwatora. Współrzędne x' i y' muszą być przekształcone tak, aby widoczny obszar sceny pokrył żądane prostokątne okno w buforze koloru $(x', y') \rightarrow (\bar{x}, \bar{y})$. Położenie oraz rozmiary okna są dowolne.

Bufor koloru niekoniecznie musi być odwzorowany bezpośrednio na ekran monitora. Może być również teksturą, która zostanie później nałożona na kilka trójkątów na scenie, na przykład w charakterze ekranu telewizora.



Rysunek 3.20: Układ współrzędnych bufora koloru.

Środek lewego górnego piksela ma w układzie bufora koloru współrzędną $(0, 0)$ a środek prawego dolnego $(w_b - 1, h_b - 1)$, gdzie w_b i h_b oznaczają odpowiednio szerokość

i wysokość bufora w pikselach (rys. 3.20).

Transformacja, która zamienia współrzędne kanoniczne na współrzędne bufora jest przedstawiona poniżej:

$$\begin{bmatrix} \bar{x} \\ \bar{y} \end{bmatrix} = \begin{bmatrix} \frac{w_o}{2} & 0 & 0 \\ 0 & -\frac{h_o}{2} & 0 \end{bmatrix} \begin{bmatrix} x' \\ y' \end{bmatrix} + \begin{bmatrix} \frac{w_o}{2} + \bar{x}_o \\ \frac{h_o}{2} + \bar{y}_o \end{bmatrix}, \tag{3.15}$$

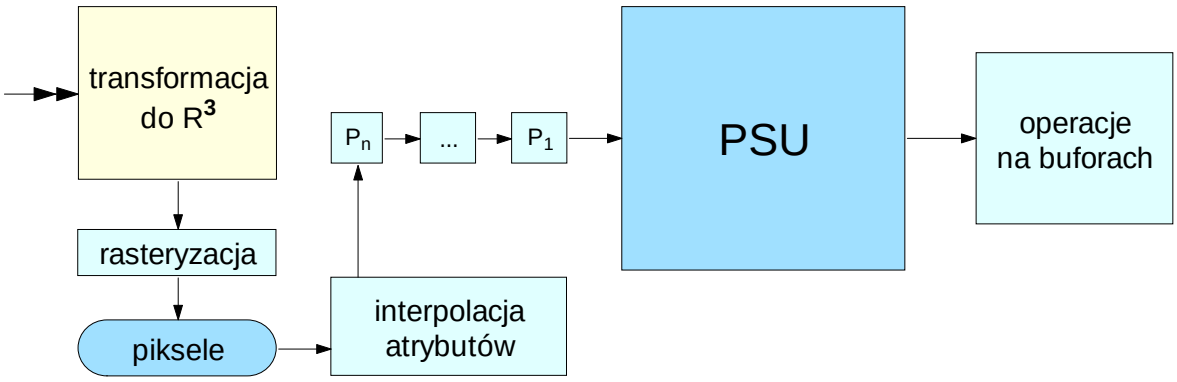
gdzie w_o i h_o określają odpowiednio szerokość i wysokość okna w pikselach a punkt $(\bar{x}_o, \bar{y}_o)$ wyznacza położenie lewego górnego rogu okna. W takiej postaci dane trafiają do części układu, która każdemu trójkątowi przyporządkowuje piksele, które składają się na jego obraz. Proces ten zwany jest rasteryzacją.

W następnej kolejności, dla każdego piksela są obliczane atrybuty, które początkowo mają wartości wyłącznie dla wierzchołków aktualnie przetwarzanego trójkąta. Kolory, współrzędne dla tekstur oraz informacje o odległości od obserwatora są interpolowane w sposób dyskretny na obrazie całego trójkąta. Od tego momentu, każdy piksel jest autonomicznym elementem obrazu, posiadającym własne atrybuty i współrzędne. Pozostałe informacje, między innymi o topologii sceny, nie są już do niczego potrzebne.

Każdy wyodrębniony piksel jest przetwarzany przez specjalną jednostkę układu graficznego (ang. Pixel Shader Unit). Dla każdego piksela jest wykonywany mikroprogram, którego zadaniem jest obliczenie wartości, która będzie umieszczona w buforze koloru. Poza atrybutami, mikroprogram może wykorzystać, poprzez specjalne instrukcje próbkujące, dane pochodzące z maksymalnie kilkunastu tekstur.

Po obliczeniu koloru piksela, układ wykonuje kilka czynności związanych z buforami, w których są zapisane dane o poprzednio generowanych pikselach. Oprócz bufora koloru i bufora-Z, który służy do wyznaczenia widoczności, programista może wykorzystać bufor zliczania (ang. stencil buffer). W typowych zastosowaniach, znajdują się w nim informacje o ilości wygenerowanych pikseli o tych samych współrzędnych.

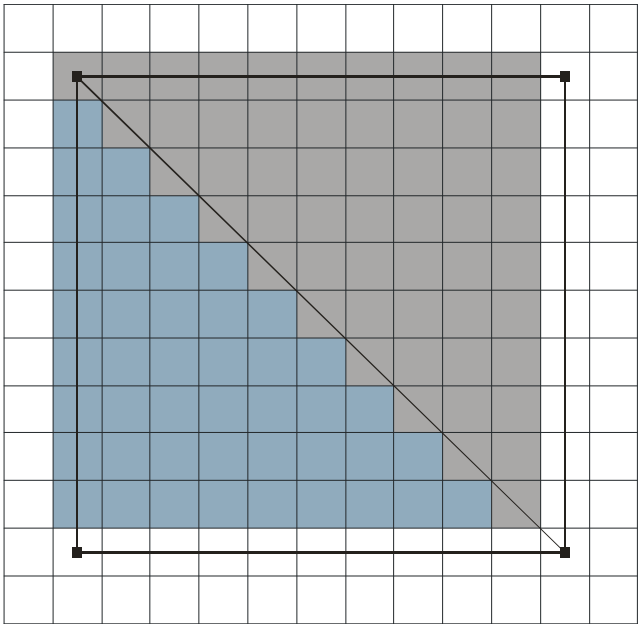
Na buforach można wykonać kilka predefiniowanych funkcji, które jako argumenty przyjmują dane w buforze i informacje o aktualnie przetwarzanym pikselu. Dopiero po wykonaniu tych czynności w buforach są potencjalnie zapisywane nowe wartości.



Rysunek 3.21: Schemat końcowej fazy procesu generowania obrazu.

3.1. Rasteryzacja i interpolacja atrybutów

Reguły dotyczące procesu rasteryzacji są ściśle zdefiniowane, gwarantuje to identycznie wygenerowane obrazy tej samej sceny na różnych akceleratorach. W rozdziale pierwszym przedstawiona jest metoda, która przyporządkowuje do trójkąta każdy piksel, który ma nawet najmniejszą część wspólną z obszarem zajmowanym przez trójkąt (rys. 1.3). W przypadku, kiedy dwa trójkąty mają wspólną krawędź, doprowadzi to do sytuacji, w której grupa pikseli na krawędzi zostanie przyporządkowana do obydwu trójkątów. W konsekwencji niektóre piksele zostaną narysowane dwa razy, co niepotrzebnie zwiększa koszt obliczeń. Przy dużej ilości małych trójkątów, większość miejsca na ekranie będzie zajmowana przez tego typu piksele a zbędne obliczenia mogą zająć blisko połowę czasu poświęconego na generowanie obrazu. Problem ten został skutecznie rozwiązany przez wprowadzenie takich zasad rasteryzacji, aby każde dwa geometrycznie spójne trójkąty nie miały wspólnych pikseli.



Rysunek 3.22: Poprawna rasteryzacja dwóch trójkątów.

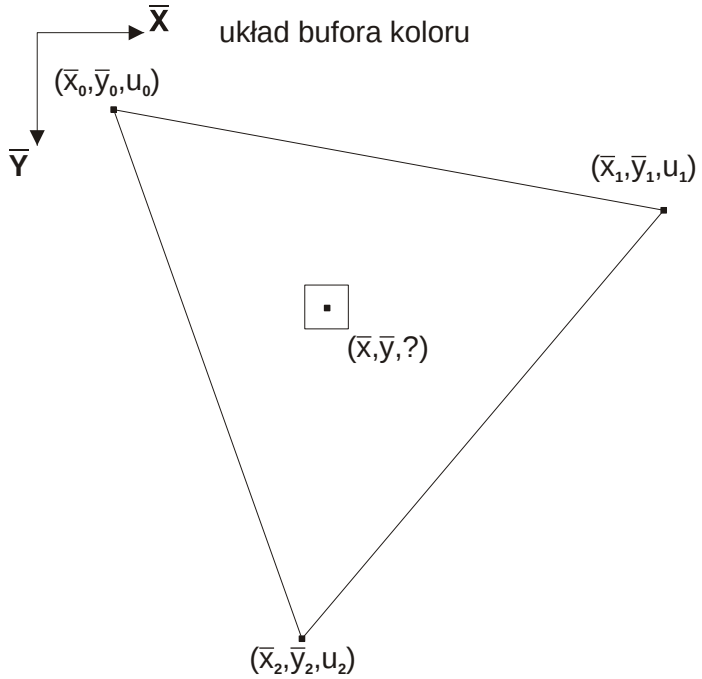
Piksel stanowi część obrazu trójkąta na ekranie tylko wtedy, gdy jego geometryczny środek znajduje się w polu trójkąta. W przypadku, kiedy środek piksela znajduje się na krawędzi, stosowane są specjalne warunki. Krawędzie dzielone są na górne, dolne, lewe oraz prawe. Górne i dolne to krawędzie poziome, lewe oraz prawe to pozostałe krawędzie odpowiednio z lewej oraz z prawej strony trójkąta. Piksel leżący na obrzeżu jest przyporządkowywany trójkątowi tylko wtedy, gdy nie leży na jego dolnej lub prawej krawędzi (rys. 3.22).

Przynależność pikseli wyznacza się za pomocą odpowiednio zmodyfikowanych algorytmów rysowania linii. Najczęściej stosowany jest algorytm Bresenhama [1]. Po podzieleniu krawędzi na grupy, wyznaczane są współrzędne pikseli należących do lewych

i prawych krawędzi. Warunki specjalne są częściowo realizowane podczas generowania linii a częściowo poprzez dodatkowe sprawdzenia. Powierzchnię trójkąta stanowią piksele pomiędzy obliczonymi współrzędnymi. Implementacja tego mechanizmu zależy wyłącznie od projektanta układu graficznego.

3.1.1. Interpolacja liniowa

Po przekształceniach rzutujących, współrzędne wierzchołków każdego trójkąta są przedstawione w dwuwymiarowym układzie bufora. Wartości atrybutów, które są zdefiniowane dla wierzchołków, muszą być w szybki sposób obliczone we wszystkich pikselach składających się na obraz trójkąta (rys. 3.23). Przy założeniu, że wartości atrybutów mają być interpolowane liniowo we współrzędnych bufora, można zastosować bardzo szybką metodę obliczeń, niewymagającą czasochłonnych operacji arytmetycznych.



Rysunek 3.23: Znalezienie wartości u w środku piksela $(\bar{x}, \bar{y})$ wymaga użycia interpolacji danych z wierzchołków trójkąta.

Poszukiwanym rozwiązaniem są współczynniki A, B i C następującej funkcji liniowej:

$$u = f(\bar{x}, \bar{y}) = A \cdot \bar{x} + B \cdot \bar{y} + C. \tag{3.16}$$

Istnieje tylko jedno rozwiązanie tego problemu, ponieważ wartość atrybutu u można wyobrazić sobie jako współrzędną z w przestrzeni $\mathbb{R}^3$. Po liniowej interpolacji tej współrzędnej, wynikiem będą trzy wierzchołki $(\bar{x}_0, \bar{y}_0, z_0), (\bar{x}_1, \bar{y}_1, z_1)$ i $(\bar{x}_2, \bar{y}_2, z_2)$ połączone płaszczyzną w $\mathbb{R}^3$.

Współczynniki funkcji (3.16) można uzyskać poprzez rozwiązanie układu równań liniowych:

$$\begin{bmatrix} \bar{x}_0 & \bar{y}_0 & 1 \\ \bar{x}_1 & \bar{y}_1 & 1 \\ \bar{x}_2 & \bar{y}_2 & 1 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} u_0 \\ u_1 \\ u_2 \end{bmatrix},$$

stąd

$$\left\{ \begin{array}{l} A = \dfrac{du_1 dy_2 - du_2 dy_1}{dx_1 dy_2 - dx_2 dy_1} \\ \\ B = \dfrac{du_2 dx_1 - du_1 dx_2}{dx_1 dy_2 - dx_2 dy_1} \end{array} \right\}, \text{ gdzie } \left\{ \begin{array}{l} dx_1 = \bar{x}_1 - \bar{x}_0,\ dy_1 = \bar{y}_1 - \bar{y}_0,\ du_1 = u_1 - u_0 \\ \\ dx_2 = \bar{x}_2 - \bar{x}_0,\ dy_2 = \bar{y}_2 - \bar{y}_0,\ du_2 = u_2 - u_0 \end{array} \right\}.$$

Współczynniki A i B oblicza się raz na każdy trójkąt. Ponieważ $A = \dfrac{\partial f}{\partial \bar{x}}$ i $B = \dfrac{\partial f}{\partial \bar{y}}$, funkcję f można obliczać w prosty sposób bez użycia parametru C :

$$u = f(\bar{x}, \bar{y}) = A(\bar{x} - \bar{x}_0) + B(\bar{y} - \bar{y}_0) + u_0. \tag{3.17}$$

Interpolacja liniowa pozwala bardzo szybko obliczać wartości atrybutów dla sąsiadujących pikseli. Przesuwając się o jeden piksel w prawo lub w dół, do już obliczonego parametru u wystarczy dodać odpowiednio A lub B. Jeśli scena składa się z dużej ilości trójkątów, wyeliminowanie z obliczeń parametru C znacznie zmniejsza koszt obliczeniowy. Układ graficzny korzysta z sekwencyjnego obliczania wartości atrybutów, więc obliczanie funkcji f w postaci (3.17) jest szybsze niż tej w postaci (3.16).

Dla uzupełnienia, parametr C jest równy:

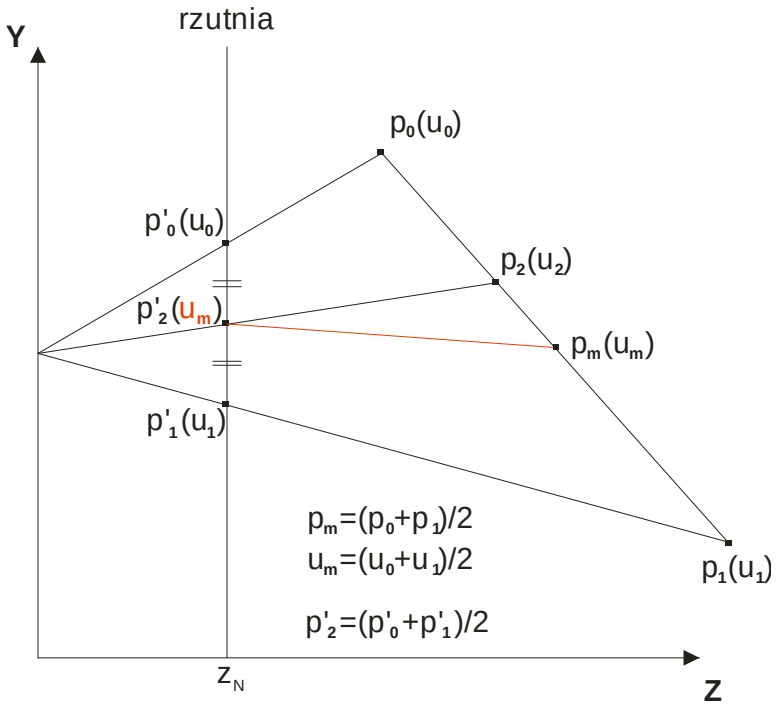
$$C = \frac{\bar{x}_0(du_2 dy_1 - du_1 dy_2) + \bar{y}_0(du_1 dx_2 - du_2 dx_1) + u_0(dx_1 dy_2 - dx_2 dy_1)}{dx_1 dy_2 - dx_2 dy_1}.$$

3.1.2. Interpolacja hiberboliczna

Jeśli rozkład wartości atrybutów pomiędzy wierzchołkami jest liniowy w przestrzeni $P(\mathbb{R}^3)$, to po przejściu do $\mathbb{R}^3$ rozkład może być już nieliniowy. Dla przykładu, jeżeli jeden z atrybutów będzie odpowiadał kolorowi trójkąta, to liniowa interpolacja koloru pomiędzy wierzchołkami wygeneruje błędny obraz sceny, jeśli operacja ta będzie wykonana po rzucie perspektywicznym.

Na rys. 3.24 przedstawiony jest rzut perspektywiczny odcinka o końcach w punktach p_0 i p_1. Do każdego wierzchołka przyporządkowany jest atrybut u, który jest liniowo rozłożony na całej długości odcinka. Punkt p_m leży w połowie długości między punktami p_0 i p_1 a wartość atrybutu wynosi dla niego $u_m = \frac{1}{2}u_0 + \frac{1}{2}u_1$. Punkty p'_0 i p'_1 są obrazami punktów p_0 i p_1 po rzucie perspektywicznym. Oczywiście w punktach p'_0 i p'_1 wartości

atrybutów pozostały niezmienione. Obrazem punktu p_2 na rzutni jest punkt p'_2, który leży dokładnie w połowie odległości między punktami p'_0 i p'_1. Wartość atrybutu w p_2 wynosi u_2 i taka powinna być też w punkcie p'_2. Niestety, przy zastosowaniu liniowej interpolacji atrybutu u pomiędzy punktami p'_0 i p'_1, wartość w punkcie p'_2 wyniesie $\frac{1}{2}u_0 + \frac{1}{2}u_1 = u_m$, co jest różne od u_2.



Rysunek 3.24: Interpolacja liniowa parametru u w układzie rzutni, powoduje błędne obliczenie parametru dla punktu p'_2.

Rozwiązaniem powyższego problemu jest użycie bardziej skomplikowanej metody interpolacji w układzie rzutni. Niech punkty $p_0 = (x_0, y_0, z_0, w_0)$ i $p_1 = (x_1, y_1, z_1, w_1)$ stanowią końce odcinka w przestrzeni $P(\mathbb{R}^3)$. Po wykonaniu rzutu perspektywicznego, wierzchołki będą miały współrzędne $p'_0 = (x'_0, y'_0, z'_0) = (\frac{x_0}{w_0}, \frac{y_0}{w_0}, \frac{z_0}{w_0})$ i $p'_1 = (x'_1, y'_1, z'_1) = (\frac{x_1}{w_1}, \frac{y_1}{w_1}, \frac{z_1}{w_1})$. Interpolacja liniowa współrzędnej y' między punktami p'_0 i p'_1 przed wykonaniem rzutu perspektywicznego jest określona wzorem:

$$y'_t(t) = \frac{y_0(1-t) + y_1 t}{w_0(1-t) + w_1 t}, \text{ gdzie } t \in [0,1]. \tag{3.18}$$

Interpolacja liniowa współrzędnej y' po rzucie perspektywicznym ma postać wzoru:

$$y'_s(s) = \frac{y_0}{w_0}(1-s) + \frac{y_1}{w_1}s, \text{ gdzie } s \in [0,1]. \tag{3.19}$$

Funkcje (3.18) i (3.19) są ciągłe i monotoniczne na przedziale $[0, 1]$ oraz mają takie same dziedziny i przeciwdziedziny. Parametry s i t nie łączy jednak zależność liniowa.

$$y'_t(t) = y'_s(s),$$

czyli

$$\frac{y_0(1-t) + y_1 t}{w_0(1-t) + w_1 t} = \frac{y_0}{w_0}(1-s) + \frac{y_1}{w_1}s. \tag{3.20}$$

Poddając równanie (3.20) serii następujących przekształceń:

$$\frac{y_0(1-t) + y_1 t}{w_0(1-t) + w_1 t} = \frac{y_0 w_1(1-s) + y_1 w_0 s}{w_0 w_1},$$

$$[y_0(1-t) + y_1 t]w_0 w_1 = [w_0(1-t) + w_1 t][y_0 w_1(1-s) + y_1 w_0 s],$$

$$y_0 w_0 w_1 s - y_0 w_0 w_1 st - y_0 w_1^2 t + y_0 w_1^2 st + y_1 w_0 w_1 t - y_1 w_0 w_1 st - y_1 w_0^2 s + y_1 w_0^2 st = 0,$$

$$t = \frac{-y_0 w_0 w_1 s + y_1 w_0^2 s}{-y_0 w_0 w_1 s - y_0 w_1^2 + y_0 w_1^2 s + y_1 w_0 w_1 - y_1 w_0 w_1 s + y_1 w_0^2 s},$$

$$t = \frac{w_0 s(-y_0 w_1 + y_1 w_0)}{[w_0 s + w_1(1-s)](-y_0 w_1 + y_1 w_0)},$$

otrzymujemy

$$t = \frac{w_0 s}{w_0 s + w_1(1-s)}. \tag{3.21}$$

Podstawiając (3.21) do równania $y(t) = y_0(1-t) + y_1 t$ otrzymujemy:

$$y(s) = y_0\left(1 - \frac{w_0 s}{w_0 s + w_1(1-s)}\right) + y_1 \frac{w_0 s}{w_0 s + w_1(1-s)}. \tag{3.22}$$

Po przekształceniu równania (3.22):

$$y(s) = \frac{y_0 w_0 s + y_0 w_1 (1-s) - y_0 w_0 s + y_1 w_0 s}{w_0 s + w_1 (1-s)},$$

$$y(s) = \frac{y_0 w_1 (1-s) + y_1 w_0 s}{w_0 s + w_1 (1-s)},$$

$$y(s) \cdot \frac{\frac{1}{w_0 w_1}}{\frac{1}{w_0 w_1}} = \frac{y_0 w_1 (1-s) + y_1 w_0 s}{w_0 s + w_1 (1-s)} \cdot \frac{\frac{1}{w_0 w_1}}{\frac{1}{w_0 w_1}}$$

otrzymujemy

$$y(s) = \frac{\frac{y_0}{w_0}(1-s) + \frac{y_1}{w_1} s}{\frac{1}{w_0}(1-s) + \frac{1}{w_1} s}. \tag{3.23}$$

Interpolując liniowo parametr s na przedziale $[0,1]$ i równocześnie przebiegając liniowo przedział $[y'_0, y'_1]$ można za pomocą wzoru (3.23) dowiedzieć się, jaką współrzędną y miał dany punkt p' przed rzutem perspektywicznym. Ponieważ parametryzacja (3.21) jest zależna wyłącznie od współrzędnej w, można za jej pomocą poznać nie tylko współrzędną y punktu p' przed rzutem, ale także jakąkolwiek wartość, która miała liniowy rozkład na przedziale $[p_0, p_1]$. Odpowiedni wzór na poprawną interpolację atrybutu u w układzie rzutni, a więc już po rzucie perspektywicznym, ma postać:

$$u(s) = \frac{\frac{u_0}{w_0}(1-s) + \frac{u_1}{w_1} s}{\frac{1}{w_0}(1-s) + \frac{1}{w_1} s}, \text{ dla } s \in [0,1]. \tag{3.24}$$

Aby znaleźć bezpośrednią zależność parametru u od współrzędnej y' wystarczy podstawić za s wyrażenie $\frac{y' - y'_0}{y'_1 - y'_0}$, w ten sposób s będzie równe 0 dla $y' = y'_0$ i równe 1 dla $y' = y'_1$.

W celu poprawnego obliczenia wartości atrybutu u w każdym punkcie należącym do obrazu trójkąta, należy liniowo interpolować po powierzchni obrazu dwie wartości: $\frac{u}{w}$ oraz $\frac{1}{w}$ (licznik i mianownik 3.24). Kiedy wartość u jest potrzeba w konkretnym punkcie, należy wykonać dzielenie $u' = \frac{u}{w} / \frac{1}{w}$ (3.24). Interpolacja we współrzędnych bufora ekranu będzie miała identyczną postać, wystarczy tylko przyporządkować poszczególne atrybuty i współrzędne w odpowiednio przekształconym wierzchołkom (3.15). W ten sposób można poprawnie obliczyć wartość wszystkich atrybutów w każdym pikselu,

należącym do obrazu trójkąta w buforze koloru. Ten rodzaj interpolacji, ze względu na naturę funkcji (3.24), jest nazywany interpolacją hiperboliczną.

Poprawna interpolacja pojedynczej wartości wymaga jednej operacji dzielenia na każdy obliczany piksel obrazu. W porównaniu ze zwykłą liniową interpolacją jest to bardzo duży koszt, ponieważ wykonanie dzielenia jest najwolniej wykonywaną przez procesory operacją arytmetyczną.

Rysunek 3.25: Rzut perspektywiczny prostokąta z nałożoną teksturą. Po lewej współrzędne tekstury interpolowane są liniowo, po prawej hiperbolicznie.

Interpolacja liniowa da zbliżone do poprawnych wyniki tylko wtedy, gdy współrzędne w wierzchołków nie różnią się dużo od siebie. Jeśli $w_0 = w_1$ związek (3.21) zredukuje się do $t = s$. Jeśli scena składa się z bardzo małych trójkątów, wtedy współrzędne w wierzchołków będą do siebie zbliżone i błędy interpolacji nie będą aż tak widoczne jak na rys. 3.25.

Obecnie produkowane układy graficzne nie pozwalają już na włączenie interpolacji liniowej. Wszystkie wartości atrybutów są zawsze interpolowane hiperbolicznie.

3.2. Tekstury

3.2.1. Tekstury dwuwymiarowe

Najczęściej używanym rodzajem tekstur są prostokątne zbiory tekseli, które reprezentują fakturę powierzchni obiektów. Każdy teksel składa się z danych opisujących kolor powierzchni, trzech komponentów w przypadku modelu RGB. Dane są umieszczone w dwuwymiarowej tablicy, którą indeksuje się poprzez współrzędne $u \in \mathbb{R}$ oraz $v \in \mathbb{R}$. Rozmiary tekstury w poziomie i w pionie muszą być równe odpowiednio $m = 2^k$ i $n = 2^l$, gdzie k i l należą do liczb całkowitych nieujemnych a m i n oznaczają odpowiednio szerokość i wysokość tekstury mierzoną w tekselach. Jest to spowodowane możliwością szybszego dostępu do kolejnych wierszy tablicy, gdyż mnożenie przez potęgi dwójki wymaga wyłącznie operacji przesunięć bitów.

W API DirectX związek między współrzędnymi u i v a współrzędną adresowanego

teksela $(t_u, t_v) \in (\{0..m-1\}, \{0..n-1\})$ zależy od wybranego trybu postępowania, gdy współrzędna u lub v wykracza poza przedział $[0, 1)$. Dostępne są trzy podstawowe tryby adresowania:

- Tryb ramki, w którym pobranie teksela o współrzędnych (u, v) leżących poza przedziałem $([0, 1), [0, 1))$ powoduje zwrócenie koloru ramki. Kolor ramki jest podawany jako parametr konkretnej tekstury. Współrzędne leżące w powyższym przedziale wyznaczają teksel o współrzędnych:

$$(t_u, t_v) = (\lfloor um \rfloor, \lfloor vn \rfloor). \tag{3.25}$$

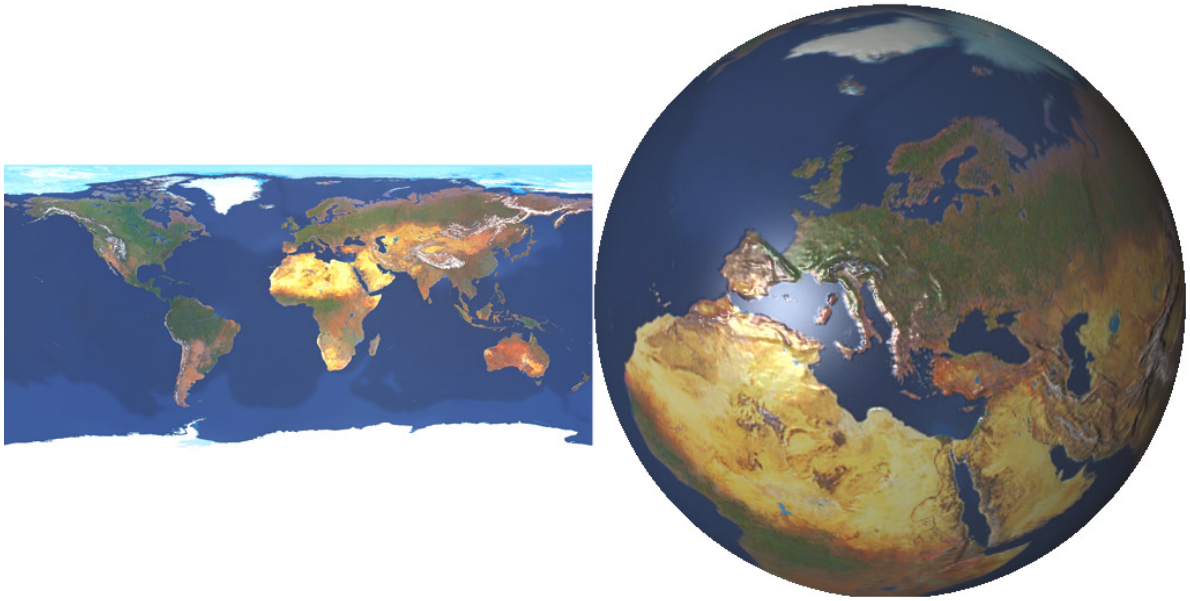
- Tryb powtarzania, w którym współrzędne poza przedziałem są sprowadzane do niego z powrotem za pomocą wzoru:

$$(t_u, t_v) = (\lfloor (u - \lfloor u \rfloor)m \rfloor, \lfloor (v - \lfloor v \rfloor)n \rfloor). \tag{3.26}$$

- Tryb lustrzanego odbicia:

$$t_u = \left\{ \begin{array}{c} \lfloor (u - \lfloor u \rfloor)m \rfloor \text{ jeśli } \lfloor u \rfloor \text{ jest parzyste lub równe } 0 \\ \lfloor (1 - (u - \lfloor u \rfloor))m \rfloor \text{ jeśli } \lfloor u \rfloor \text{ jest nieparzyste} \end{array} \right\}, \tag{3.27}$$

$$t_v = \left\{ \begin{array}{c} \lfloor (v - \lfloor v \rfloor)n \rfloor \text{ jeśli } \lfloor v \rfloor \text{ jest parzyste lub równe } 0 \\ \lfloor (1 - (v - \lfloor v \rfloor))n \rfloor \text{ jeśli } \lfloor v \rfloor \text{ jest nieparzyste} \end{array} \right\}.$$

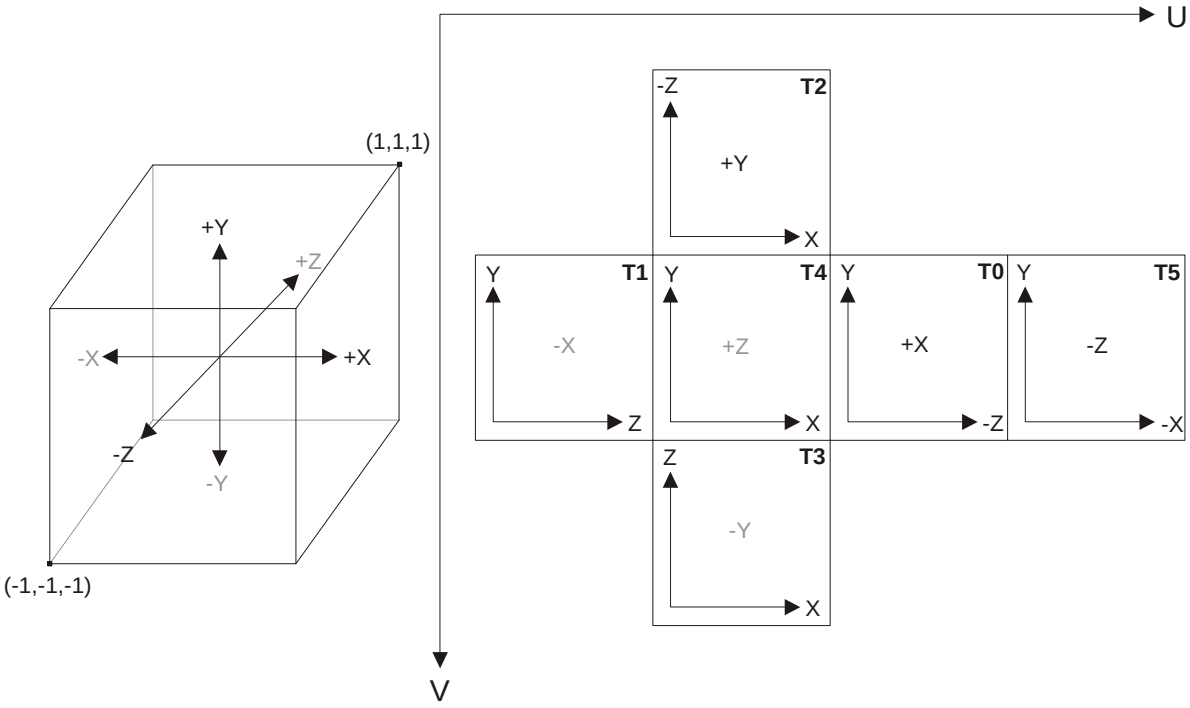


Rysunek 3.26: Dwuwymiarowa tekstura nałożona na powierzchnię kuli. Dodano mapę nierówności i oświetlenie.

Po przyporządkowaniu odpowiednich współrzędnych u i v wierzchołkom modelu, wartości te są interpolowane na obszarze trójkątów. W każdym wygenerowanym pikselu obrazu, przy pomocy współrzędnych u i v, układ pobiera dane konkretnego teksela. Informacje te służą później do obliczenia ostatecznego koloru piksela w buforze koloru (rys. 1.5). Format danych tekseli różni się nie tylko ilością komponentów (od 1 do 4) ale też ilością bajtów przeznaczonych dla pojedynczego komponentu. W zależności od żądanej dokładności może to być liczba całkowita 8 lub 16 bitowa oraz 16 lub 32 bitowa liczba rzeczywista.

3.2.2. Tekstury kubiczne

Tekstura kubiczna składa się z sześciu zwykłych kwadratowych tekstur dwuwymiarowych. Każda z nich reprezentuje jedną ścianę prostopadłościanu rozpiętego pomiędzy punktami $(-1,-1,-1)$ i $(1,1,1)$. Przy pobieraniu danych z tekstury kubicznej używane są trzy współrzędne (x,y,z), które można interpretować jako wektor zaczepiony w punkcie $(0,0,0)$, czyli w środku sześcianu (rys. 3.27).



Rysunek 3.27: Tekstura kubiczna złożona z sześciu kwadratowych tekstur dwuwymiarowych.

Wynikiem operacji pobrania danych o współrzędnych $d = (x,y,z)$ jest teksel przecinany przez półprostą o początku w punkcie $(0,0,0)$ i zwrocie (x,y,z). W pierwszej kolejności układ wybiera najdłuższą składową wektora kierunkowego d. Bezpośrednio wyznacza ona ścianę (teksturę), na której wystąpi przecięcie. Dla przykładu, jeśli najdłuższą składową wektora jest współrzędna z i jest ona dodatnia to wybierana jest

tekstura $T4$. Ponieważ wszystkie ściany są oddalone od środka dokładnie o 1, punkt przecięcia dany jest wzorem:

$$(x_p, y_p) = (\frac{x}{|z|}, \frac{y}{|z|}). \tag{3.28}$$

Do poprawnego odwołania się do tekstury dwuwymiarowej niezbędna jest transformacja współrzędnych (x_p, y_p) do układu tekstury U/V:

$$(u, v) = (\frac{x_p}{2} + \frac{1}{2}, \frac{-y_p}{2} + \frac{1}{2}). \tag{3.29}$$

W przypadku, kiedy najdłuższą współrzędną nie jest z, w równaniu (3.28) współrzędne zamieniają się miejscami a w zależności (3.29) znaki przy współrzędnych dopasowują się tak, aby transformacja do układu U/V była prawidłowa (rys. 3.27).

Tekstury kubiczne mogą służyć jako mapa odwzorowująca w punkcie pewną wartość, która zmienia się wraz z kątem patrzenia z tego punktu. Dobrym przykładem jest intensywność promieniowania padającego na punkt z określonego kierunku. Po obliczeniu tych wartości w określonym miejscu sceny, na przykład metodą raytracingu, wygenerowaną teksturę kubiczną można wykorzystać do oświetlenia małego modelu w czasie rzeczywistym.

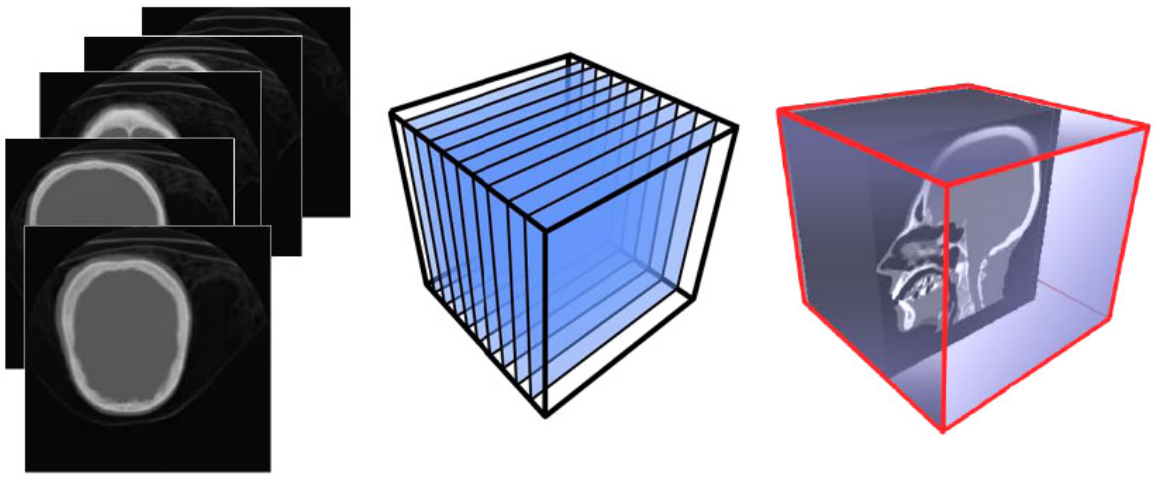
Rysunek 3.28: Kubiczna tekstura odwzorowująca otoczenie (str. lewa górna) wraz z wygenerowaną przy jej pomocy teksturą oświetlenia (str. lewa dolna). Po prawej stronie obiekt oświetlony przy jej użyciu.

Przy założeniu, że model jest mały a środowisko jest znacznie od niego oddalone, pozycje wierzchołków modelu można potraktować jako punkt. W ten sposób kie-

runek wektora normalnego do powierzchni wyznaczy składową rozproszoną światła (rys. 3.28). Wektor normalny do powierzchni zostaje zakodowany na wierzchołkach w trzech współrzędnych dla tekstury i interpolowany przez układ na powierzchni trójkątów. Wszystkie niezbędne obliczenia związane z pobraniem wartości intensywności z tekstury kubicznej wykona mikroprogram dla pikseli.

3.2.3. Tekstury wolumetryczne

Tekstura wolumetryczna składa się z k tekstur dwuwymiarowych $m \times n$ ułożonych równolegle w kształt prostopadłościanu. Taką teksturę można potraktować jako trójwymiarowy zbiór tekseli, który można indeksować za pomocą trzech współrzędnych $(x, y, z) \in \mathbb{R}^3$. Pierwsze dwie są odpowiednikami współrzędnych u i v w przypadku dwuwymiarowym i określają położenie teksela na konkretnej warstwie. Trzecia wyznacza indeks warstwy, która ma być użyta do odczytania danych teksela. W taki sposób zapisywane są dane na przykład z aparatury medycznej, która wykonuje serię zdjęć organu o różnej głębokości penetracji (rys. 3.29).

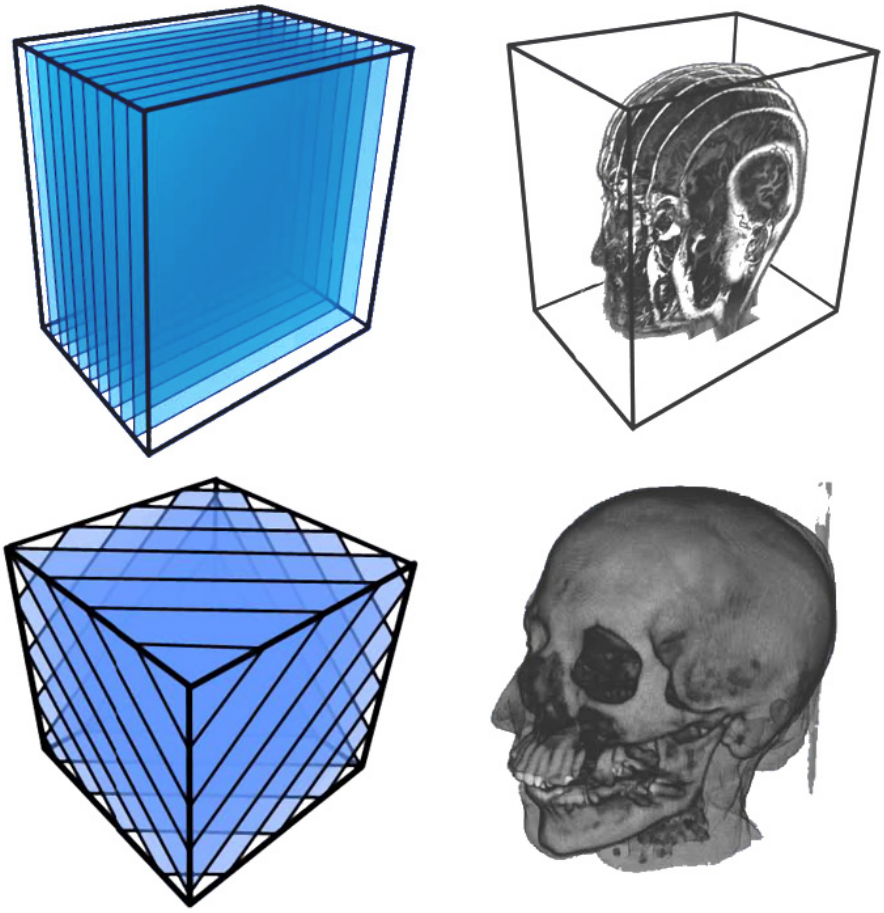


Rysunek 3.29: Tekstura wolumetryczna złożona z kilku warstw.

Przejście ze współrzędnych x i y do indeksu teksela i-tej warstwy (t_x^i, t_y^i) odbywa się identycznie jak w przypadku dwuwymiarowym (równania 3.25 - 3.27). Do wyznaczenia numeru warstwy używane są takie same zależności, ale dla jednej zmiennej: $i = t_z$.

Układ graficzny pozwala na rysowanie wyłącznie dwuwymiarowych elementów (trójkątów), nie jest możliwe narysowanie w pełni trójwymiarowego obiektu. Pierwszym nasuwającym się rozwiązaniem jest narysowanie wszystkich warstw w postaci częściowo przeźroczystej. Niestety na wygenerowanym obrazie będzie widać wyraźne nieciągłości pomiędzy warstwami. Największe przerwy będą widoczne, gdy kierunek patrzenia obserwatora będzie równoległy do którejś z warstw (rys. 3.30).

Używając tekstur wolumetrycznych można rysować warstwy dowolnie zorientowane



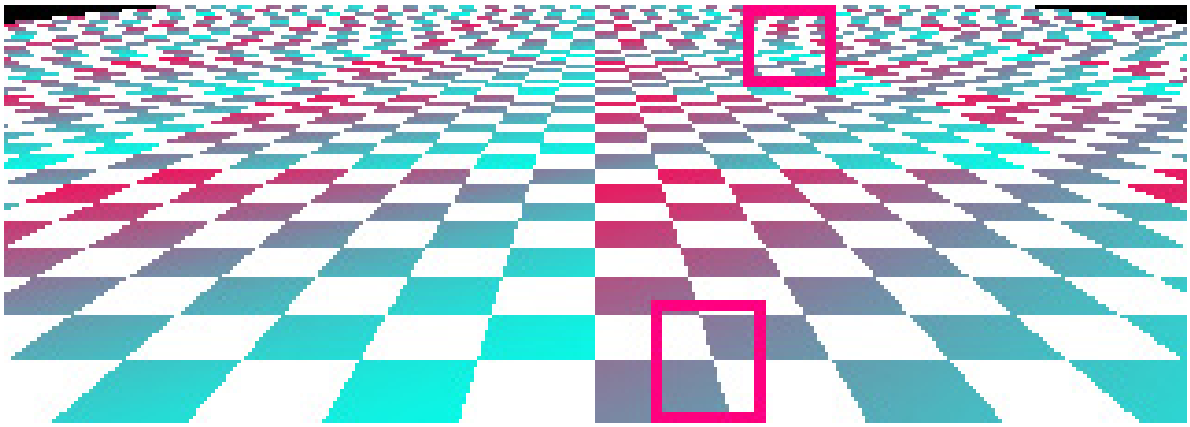
Rysunek 3.30: W przypadku normalnego rysowania warstw widoczne są nieprawidłowości w obrazie (góra). Na dole poprawny obraz wygenerowany przy użyciu tekstury wolumetrycznej. Rysowane wielokąty mogą być ustawione prostopadle do obserwatora.

i o dowolnym kształcie. Każda warstwa składa się z pewnej ilości trójkątów, na których wierzchołkach są przyporządkowane współrzędne punktów w teksturze wolumetrycznej. W czasie rasteryzacji układ interpoluje te współrzędne i dla każdego piksela generowanego obrazu stosuje równania (3.25 - 3.27), które pozwalają obliczyć numer warstwy oraz pozycję teksela na teksturze wolumetrycznej. Jeśli współrzędne tekstury, rysowanych z trójkątów warstw, będą wypełniać prostopadłościan oraz będą zorientowane równolegle do obserwatora, to obraz będzie wypełniony w pełni i praktycznie pozbawiony nieciągłości (rys. 3.30).

3.2.4. Filtrowanie

Bufor koloru posiada określoną rozdzielczość mierzoną w pikselach. Każdy piksel jest bardzo małym prostokątem, który może być zabarwiony w całości pojedynczym kolorem. Każda tekstura również jest złożona ze skończonej liczby prostokątnych tekseli. Wielokąt po wszystkich przekształceniach zostaje zamieniony na zbiór pikseli odwzo-

rowujących go buforze. Jeżeli na wielokąt została nałożona tekstura, to w każdym pikselu zostanie pobrana wartość koloru odpowiedniego teksela, zgodnie z obliczonymi współrzędnymi.



Rysunek 3.31: Obraz wygenerowany bez użycia żadnych filtrów próbkujących teksturę.

Kolor każdego piksela powinien nieść ze sobą informację o wszystkich tekselach, które są widoczne przez małe prostokątne tworzone przez niego okienko. Podane w poprzednim podrozdziale metody pobierania danych z tekstur nie uwzględniają tego faktu. Każdy piksel przyjmie kolor tylko jednego teksela, który jest wyznaczony przez przecięcie półprostej, biegnącej od obserwatora przez środek piksela, z płaszczyzną tekstury nałożonej na wielokąt. Jeśli rozmiar teksela po rzucie perspektywicznym jest mały w stosunku do rozmiaru piksela, pojawią się widoczne artefakty obrazu (górna część rys. 3.31).

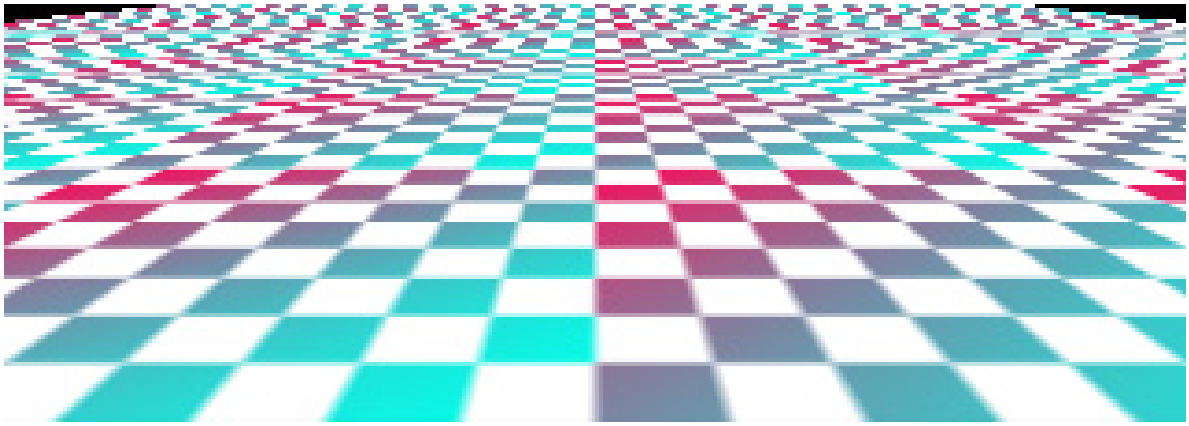
Podobnie jest w przypadku, kiedy rozmiar teksela jest dużo większy w porównaniu do rozmiaru piksela. W takich miejscach pojawią się 'zęby' będące konsekwencją skończonej rozdzielczości obrazu (dolna część rys. 3.31). Jeżeli rozdzielczość zostanie znacznie zwiększona, to krawędzie między tekselami będą gładkie.

Częściowym rozwiązaniem powyższych problemów są specjalne filtry, które biorą udział przy operacji próbkowania danych z tekstury. Rozróżniane są dwa rodzaje filtrów. Pierwszą grupę stanowią filtry przeciwdziałające powstawaniu 'zębów' przy dużych powiększeniach, drugą filtry obliczające kolor piksela przy użyciu więcej niż jednego teksela w przypadku dużych pomniejszeń.

Filtrowanie dwuliniowe

Filtr dwuliniowy należy do pierwszej grupy i jego działanie można porównać do lekkiego rozmazania tekstury na wielokącie (rys. 3.32). Jeśli z równań (3.25 - 3.27) zostanie usunięta ostatnia operacja, czyli sprowadzanie wyniku do dziedziny liczb całkowitych, otrzymane współrzędne (t'_u, t'_v) wyznaczą dokładną pozycję (u, v) na teksturze. (t'_u, t'_v) przebiegają teksturę płynnie wraz ze zmianą współrzędnych u i v. Dla przykładu, jeśli tekstura ma rozmiary 2×2 teksele, to dla $u = 0 \rightarrow t'_u = 0$ (lewa krawędź pierwszej kolumny tekseli) a dla $u = 0.25 \rightarrow t'_u = 0.5$ (środek pierwszej kolumny tekseli). Filtr

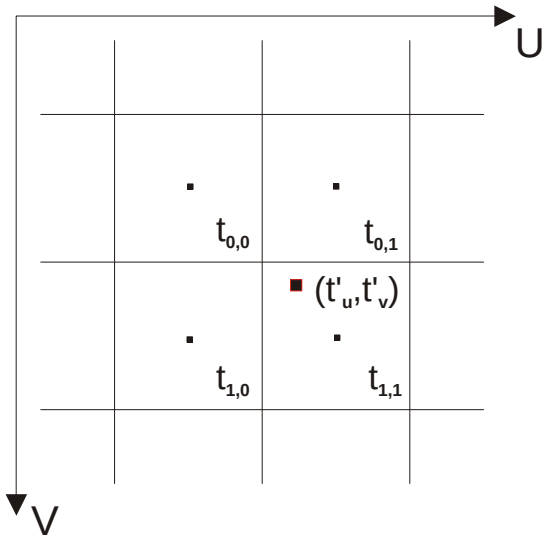
dwuliniowy zamiast pobierać kolor tylko z jednego teksela (t_u, t_v) uśrednia kolory z czterech tekseli najbliższych punktowi (t'_u, t'_v) na teksturze.



Rysunek 3.32: Obraz wygenerowany przy użyciu filtra dwuliniowego.

Niech $\tau(u, v)$ oznacza kolor teksela o współrzędnych (t_u, t_v). Jeśli współrzędne tekstury w środku generowanego piksela są równe u i v, to filtr dwuliniowy uśredni dane z następujących 4 sąsiadujących ze sobą tekseli:

$$\mathbf{t_{0,0}} = \tau(u - \frac{1}{2m}, v - \frac{1}{2n}),\ \mathbf{t_{0,1}} = \tau(u + \frac{1}{2m}, v - \frac{1}{2n}),$$
$$\mathbf{t_{1,0}} = \tau(u - \frac{1}{2m}, v + \frac{1}{2n}),\ \mathbf{t_{1,1}} = \tau(u + \frac{1}{2m}, v + \frac{1}{2n}).$$



Rysunek 3.33: Uśredniane przez filtr dwuliniowy teksele.

Niech:

$$\alpha = um - \frac{1}{2} - \lfloor um - \frac{1}{2} \rfloor,$$

$$\beta = vn - \frac{1}{2} - \lfloor vn - \frac{1}{2} \rfloor.$$

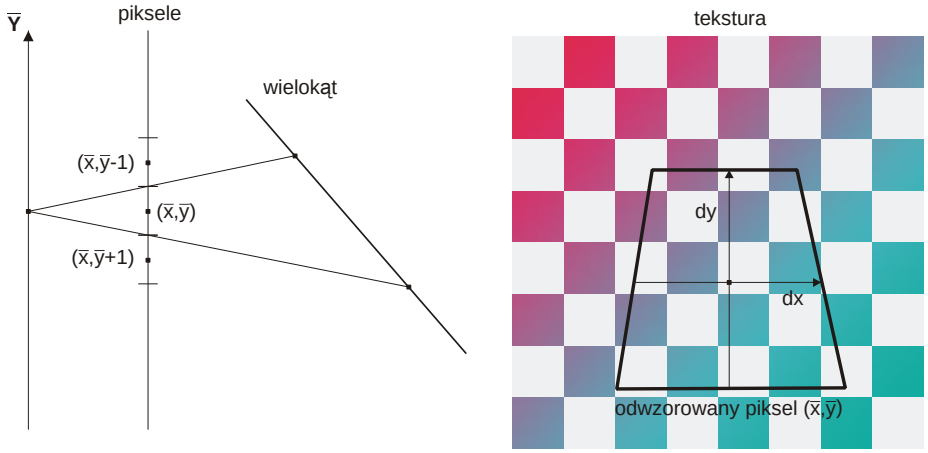
Ostateczny kolor piksela przy użyciu filtra dwuliniowego dany jest wzorem:

$$\boldsymbol{\tau}_{\mathbf{L}}(u, v) = (1 - \alpha)(1 - \beta)t_{0,0} + \alpha(1 - \beta)t_{0,1} + (1 - \alpha)\beta t_{1,0} + \alpha\beta t_{1,1}.$$

Dla tekstur wolumetrycznych metoda jest podobna, ale uśredniane jest 8 najbliższych współrzędnym (t'_x, t'_y, t'_z) tekseli pochodzących z 2 warstw, po 4 na warstwę.

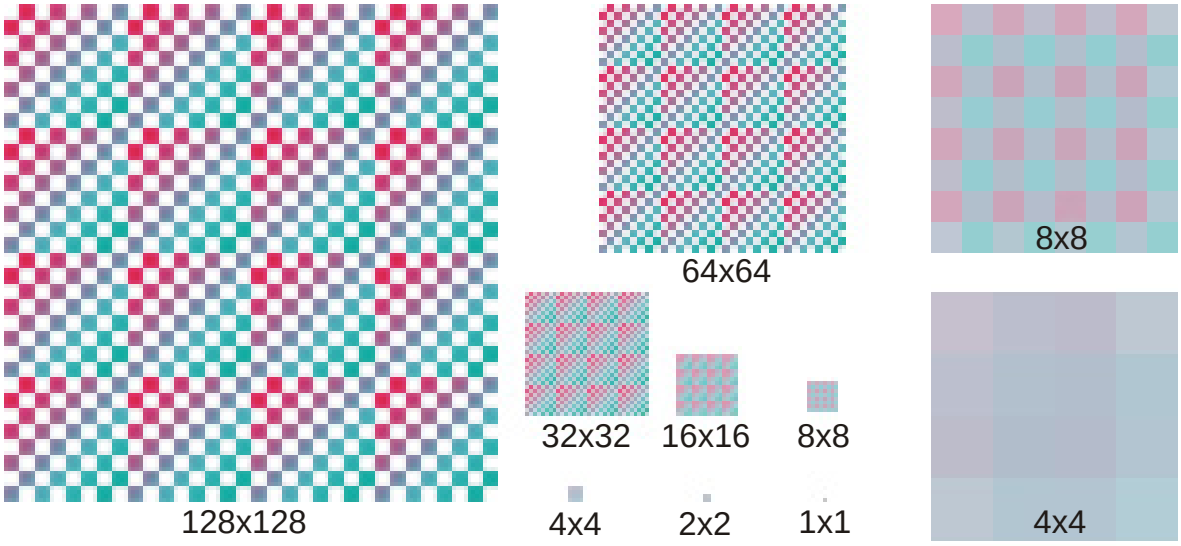
Mip-mapping

Mip-mapping (z łacińskiego: multum in parvo, tłum.: wiele w małym) jest techniką, która należy do drugiej grupy filtrów, poprawiających jakość tekstur wyświetlanych w małych skalach. Czarno-biała szachownica, ustawiona prostopadle do obserwatora w dużej odległości, powinna zamienić się w mały szary prostokąt. Dzieje się tak, ponieważ czarne i białe pola leżą wtedy w bardzo małych odległościach od siebie, uśredniając się w kolor szary. Niestety, dysponując skończoną rozdzielczością bufora ekranu, komputerowa symulacja tego zjawiska nie będzie prawidłowa. Przez małe okno każdego piksela, składającego się na obraz oddalonego prostokąta, będzie widać dużą grupę czarnych i białych tekseli. Podczas operacji pobierania koloru teksela wybrany zostanie kolor biały albo czarny. Tak wygenerowany obraz będzie się składał tylko z białych i czarnych pikseli.



Rysunek 3.34: Obszar tekstury nałożonej na wielokąt, który jest odwzorowany na obszarze piksela $(\bar{x}, \bar{y})$. Bez użycia filtra, dla piksela wybrany będzie kolor biały.

Rozwiązać ten problem można stosując uśrednianie kolorów wszystkich tekseli widocznych przez okienko piksela (rys. 3.34). Takie podejście jest bardzo kosztowne, gdyż na jeden piksel obrazu może przypadać dowolna ilość różnych tekseli. Mip-mapping pozwala na generowanie obrazów przybliżonych do poprawnego. Dla każdej tekstury jest generowany ciąg $\log_2(\min\{m,n\})$ tzw. mip-map, czyli tekstur powstałych przez dwukrotne zmniejszenie każdej poprzedniej. Dla tekstury T_0 o rozmiarze 128×128 wygenerowane będą mip-mapy $T_1..T_7$ o rozmiarach 64×64, 32×32, ..., 1×1 tekseli. Kolor teksela mip-mapy T_k o współrzędnych (p,q), gdzie $\{p,q\} \in \mathbb{Z}$, jest wynikiem uśrednienia 4 tekseli tekstury T_{k-1} o współrzędnych $(2p,2q),(2p+1,2q),(2p,2q+1)$ i $(2p+1,2q+1)$ (rys. 3.35).



Rysunek 3.35: Tekstura wraz z serią mip-map. Z prawej strony dwie mip-mapy pokazane w powiększeniu.

Pobierając kolor tekstury w konkretnym pikselu układ decyduje, której wersji tekstury użyć. Im większy jest rozmiar piksela w stosunku do rozmiaru teksela, tym większy indeks mip-mapy, która będzie użyta do odczytu koloru. W celu obliczenia przybliżonego stosunku rozmiarów używa się następującej zależności:

$$p(\bar{x},\bar{y}) = \max\left\{\sqrt{\left(\frac{\partial u}{\partial \bar{x}}\right)^2 + \left(\frac{\partial v}{\partial \bar{x}}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial \bar{y}}\right)^2 + \left(\frac{\partial v}{\partial \bar{y}}\right)^2}\right\}. \tag{3.30}$$

Wektory $dx = (\frac{\partial u}{\partial \bar{x}}, \frac{\partial v}{\partial \bar{x}})$ i $dy = (\frac{\partial u}{\partial \bar{y}}, \frac{\partial v}{\partial \bar{y}})$ w przybliżeniu rozpinają na sobie obszar tekstury, który jest rzutowany do okienka piksela. Nie jest to wartość dokładna, gdyż wektory zmieniają się na obszarze pojedynczego piksela ze względu na nieliniowy charakter przekształceń rzutujących.

Wartość $p(\bar{x}, \bar{y})$ opisuje szerokość kwadratu (w tekselach), który trzeba uśrednić, aby uzyskać w przybliżeniu poprawny kolor piksela. Do dyspozycji jest kilka różnych wersji tekstury. W każdej są uśrednione bloki tekseli o wielkości od 1 do $\min\{m, n\}$, przy czym blok uśrednionych tekseli w mip-mapie T_k jest dwa razy szerszy i dłuższy od tego w T_{k-1}. Przyjęto, że indeks mip-mapy jest obliczany za pomocą wzoru:

$$k(\bar{x}, \bar{y}, \lambda) = \min\{\ \lfloor\lambda\rfloor,\ \log_2(\min\{m, n\})\ \}, \text{ gdzie } \lambda = \log_2[p(\bar{x}, \bar{y})].$$

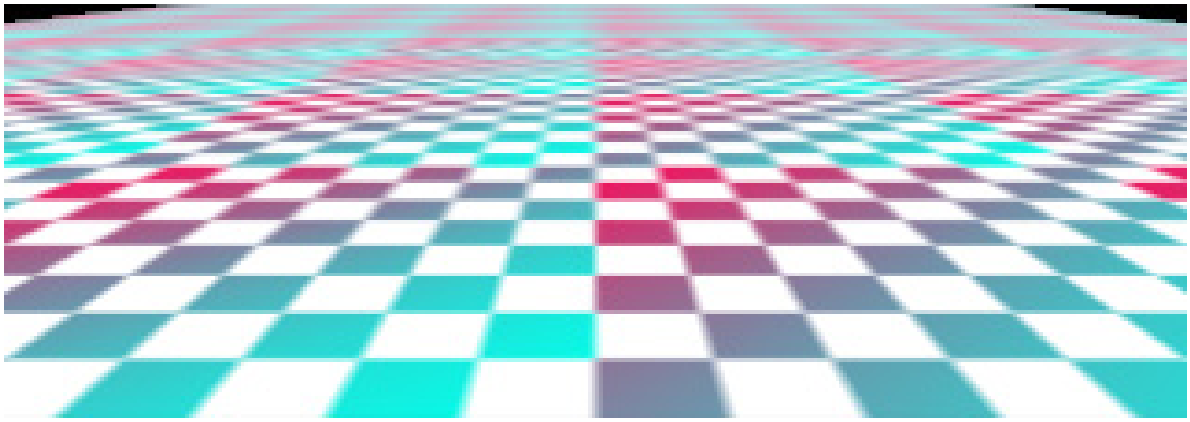
Jeżeli rozmiar teksela będzie nie więcej niż dwukrotnie mniejszy od rozmiaru piksela, to wybrana będzie tekstura T_0. Jeżeli będzie nie więcej niż czterokrotnie mniejszy to będzie wybrana tekstura T_1, itd.. Pobieranie koloru z wybranej już wersji tekstury jest wykonywane przy użyciu filtra dwuliniowego. W celu ukrycia momentu podmiany tekstury w czasie oddalania się od obserwatora, stosuje się podobne do filtra dwuliniowego rozwiązanie. Wybierane są dwie mip-mapy najbliższe wartości λ, kolory pobiera się z obydwu i uśrednia przy pomocy interpolacji dwuliniowej. Ten sposób postępowania nazywany jest interpolacją trójliniową, gdyż przy uśrednianiu bierze udział 8 tekseli, po 4 na mip-mapę (rys. 3.36).

Niech $\tau_L(u, v, k)$ oznacza kolor teksela o współrzędnych (t_u, t_v), pobranego z mip-mapy T_k przy użyciu filtra dwuliniowego. Ostateczny kolor tekstury przy użyciu mip-mappingu dany jest wzorem:

$$\boldsymbol{\tau}_{\mathbf{M}}(u, v) = [1 - \lambda'] \cdot \tau_L(u, v, k_1) + \lambda' \cdot \tau_L(u, v, k_2),$$

gdzie

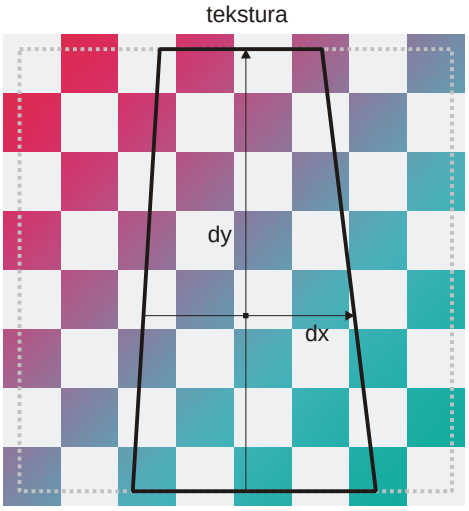
$$\lambda' = \lambda - \lfloor\lambda\rfloor,\ k_1 = k(\bar{x}, \bar{y}, \lambda),\ k_2 = k(\bar{x}, \bar{y}, \lambda + 1).$$



Rysunek 3.36: Obraz wygenerowany przy użyciu mip-mappingu.

Filtrowanie anizotropowe

Długości wektorów składowych równania (3.30): $dx = (\frac{\partial u}{\partial \bar{x}}, \frac{\partial v}{\partial \bar{x}})$ i $dy = (\frac{\partial u}{\partial \bar{y}}, \frac{\partial v}{\partial \bar{y}})$ mogą się od siebie znacznie różnić. Oznacza to, że obraz piksela na teksturze jest wydłużony w kierunku którejś z osi. W tym przypadku układ wyznaczy mip-mapę kierując się dłuższą składową, przez co uśredniony zostanie kwadrat o boku 8×8 tekseli (rys. 3.37). Spowoduje to rozmazanie obrazu (górna część rys. 3.36).

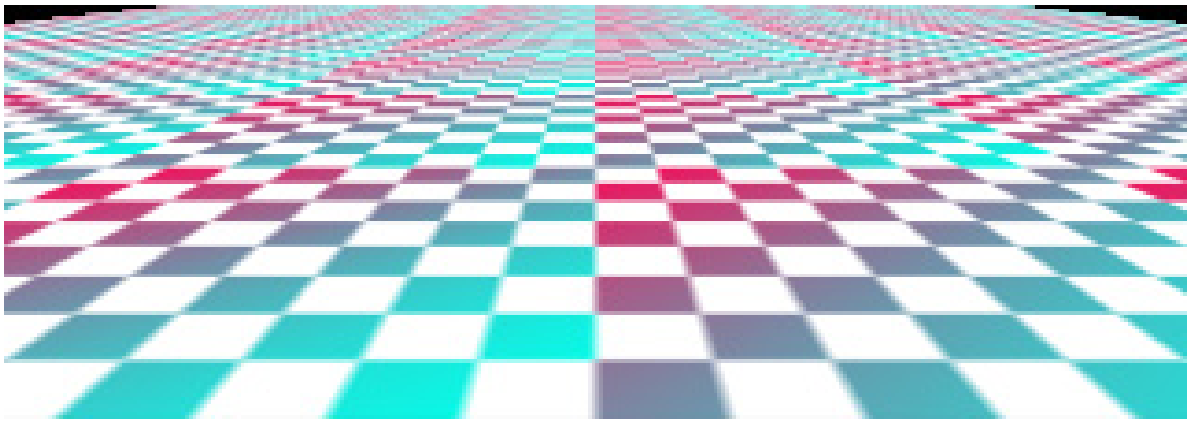


Rysunek 3.37: Obszar tekstury nałożonej na wielokąt, który jest odwzorowany na obszarze piksela. Oś dy jest dwa razy dłuższa od osi dx.

Prawidłowo uśrednione powinny być piksele w obrębie prostokąta o długościach boków odpowiadających liczbom $|dx|$ i $|dy|$. Filtr anizotropowy rozwiązuje ten problem wybierając mip-mapę, która odpowiada krótszej osi i uśredniając nie jeden, ale kilka tekseli w obrębie prostokąta (rys. 3.38).

$$p(\bar{x}, \bar{y}) = \min \left\{ \sqrt{\left(\frac{\partial u}{\partial \bar{x}}\right)^2 + \left(\frac{\partial v}{\partial \bar{x}}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial \bar{y}}\right)^2 + \left(\frac{\partial v}{\partial \bar{y}}\right)^2} \right\}$$

Ilość próbek wacha się od 1 do 16 a ich ułożenie jest ściśle zależne od implementacji. Z reguły wybierane są teksele równomiernie rozłożone na odcinku dłuższej z osi. Liczba próbek zmienia się w zależności od stosunku długości obu osi. W sytuacji pokazanej na (rys. 3.37) wybrana będzie mip-mapa T_2 2×2 teksele. Stosunek długości osi wynosi 2 do 1, więc układ uśredni 2 próbki rozłożone równomiernie na osi dy. Filtr anizotropowy jest rozszerzeniem techniki mip-mappingu, ale może być używany i bez niego. W takim przypadku, aby uzyskać zadowalające rezultaty, ilość próbek musi być znacznie większa. W praktyce najbardziej sensowne jest użycie mniejszej ilości próbek z włączonym mip-mappingiem.



Rysunek 3.38: Obraz wygenerowany przy użyciu filtra anizotropowego (16 próbek).

3.3. Programowanie układu

Po zakończeniu procesu interpolacji atrybutów i rasteryzacji jednostka przetwarzania pikseli wykonuje mikroprogram dla każdego piksela składającego się na obraz trójkąta. Treść programu jest ustalana przez programistę i przesyłana do układu przed rozpoczęciem procesu generowania obrazu. Program może być zmieniany w dowolnym momencie, ale jest to bardzo kosztowna czasowo operacja. Dla każdego piksela, przy pomocy danych wejściowych, musi być wygenerowany jeden kolor, który bierze udział w czynnościach związanych z buforami.

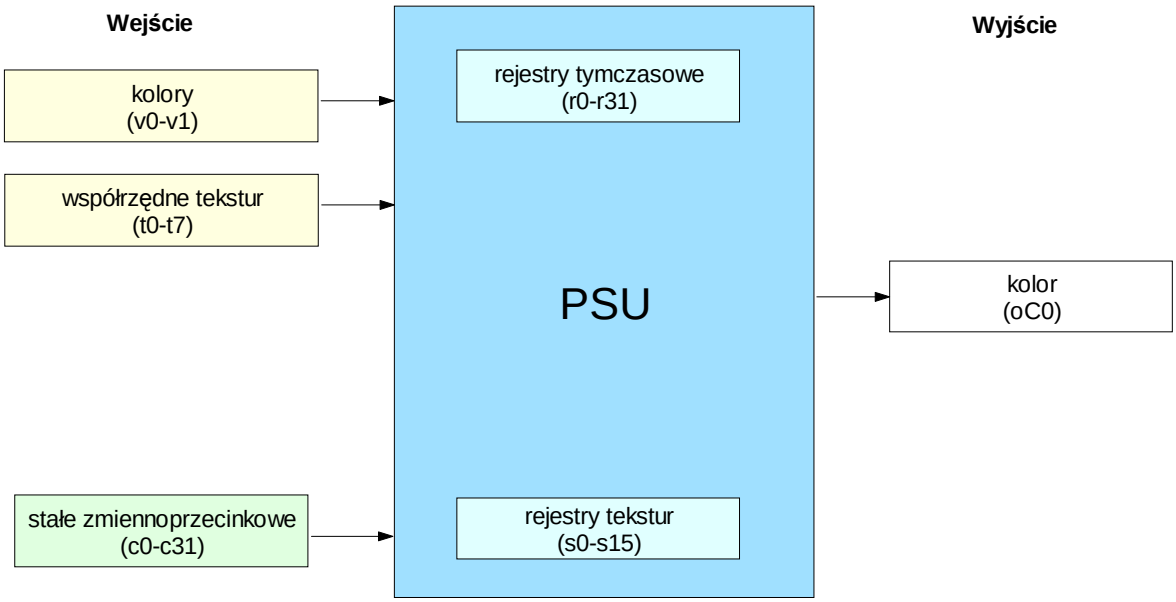
Każdy piksel ma dostęp wyłącznie do własnych danych wejściowych i tymczasowych. Dzięki temu w tym samym czasie może być przetwarzana większa ilość pikseli. Obecnie, układy graficzne posiadają od 1 do 16 jednostek przetwarzana pikseli, zwanych potokami.

Każdy program składa się z maksymalnie 96 instrukcji specjalnego mikroasemblera, w tym maksymalnie 64 instrukcji arytmetycznych i 32 próbkujących tekstury. Rejestry układu dzielą się na wejściowe, wyjściowe, tymczasowe i rejestry tekstur (rys. 3.39). Przy każdym uruchomieniu programu w rejestrach wejściowych znajduje się 8 przeinterpolowanych współrzędnych tekstur ($t0 - t7$) oraz dwa kolory ($v0 - v1$). Każdy rejestr wejściowy składa się z 4 liczb rzeczywistych ($XYZW$). Oprócz tych informacji programista może dowolnie ustalić zawartość rejestrów reprezentujących stałe, które są niezależne od kolejnych uruchomień programu. Do dyspozycji są 32 rejestry wektorowe $c0 - c31$, każdy w postaci czterech liczb rzeczywistych ($XYZW$). Tak jak w przypadku jednostki dla wierzchołków, każda zmiana zawartości rejestrów tymczasowych wprowadza bardzo duże opóźnienia. Należy zaznaczyć, że stałe mogą być zmieniane wyłącznie przez procesor główny spoza poziomu mikroprogramu. Rejestry wejściowe, stałych oraz tekstur można jedynie odczytywać a rejestry wyjściowe zapisywać.

Rejestry tymczasowe są dostępne tylko z poziomu mikroprogramu. Należy do nich od 12 do 32 rejestrów wektorowych $r0 - r31$, każdy w postaci 4 liczb rzeczywistych ($XYZW$).

Zadaniem mikroprogramu jest wygenerowanie koloru w rejestrze wyjściowym $oC0$, który składa się z czterech komponentów należących do liczb rzeczywistych $(XYZW)$.

Rejestrów próbkujących teksturę $s0 - s15$ (ang. texture sampler register) nie można odczytywać ani zapisywać. Służą one jedynie do identyfikacji tekstury, która ma być użyta do pobrania koloru teksela. Konkretne tekstury są przyporządkowywane do tych rejestrów spoza mikroprogramu, przed jego wykonaniem. Każda instrukcja próbkująca wymaga podania przynajmniej jednego rejestru tekstury oraz współrzędnych, które wyznaczają pozycję teksela.



Rysunek 3.39: Jednostka przetwarzania pikseli.

Podczas opisywania właściwości układu producenci często posługują się pojęciami jednostki teksturującej (ang. Texture Mapping Unit) oraz ilości tych jednostek na potok. Jednostka teksturująca jest fizyczną częścią układu scalonego, odpowiedzialną za operacje dostępu oraz pobierania danych z tekstur. Pojęcia te pozwalają na oszacowanie liczby pikseli, które układ jest w stanie obliczyć w tym samym momencie. Niech przykładowy układ dysponuje 8 potokami i 16 jednostkami teksturującymi w tym dwoma na potok. Układ będzie w stanie przetwarzać 8 pikseli na potok tylko pod warunkiem, że liczba tekstur, z których pobierane są dane dla pojedynczego piksela, nie przekracza liczby jednostek teksturujących na potok. Jeśli liczba ta będzie większa, to potok musi 'pożyczyć' jednostki z innego potoku, wyłączając go jednocześnie z procesu przetwarzania. Jeżeli mikroprogram używa danych z 2 tekstur, to układ przeliczy 8 pikseli jednocześnie (tyle ile ma potoków). Jeśli używa 3 lub 4 tekstury, to przeliczy tylko 4 piksele jednocześnie.

3.3.1. Budowa programu

Mikroprogramy dla pikseli można pisać także w języku HLSL. Wszystkie obostrzenia, limity i budowa programu są takie same jak w przypadku programów dla wierzchołków. Inne są liczby rejestrów i ich identyfikatory. Do rejestrów wejściowych $t0 - t7$ program odwołuje się poprzez identyfikatory $TEXCOORD0 - TEXCOORD7$, do $v0 - v1$ poprzez nazwy $COLOR0 - COLOR1$. Rejestr wyjściowy $oC0$ ma identyfikator $COLOR0$. Liczba dostępnych kolorów i współrzędnych dla tekstur zależy od danych wygenerowanych przez jednostkę przetwarzania wierzchołków. Rejestry wyjściowe tejże są bezpośrednio skojarzone z rejestrami wejściowymi dla jednostki PSU.

Rejestry próbkujące tekstury mają swoje odpowiedniki w HLSL w postaci nowych typów danych: $sampler1D$ (tekstura jednowymiarowa), $sampler2D$ (dwuwymiarowa), $sampler3D$ (wolumetryczna) i $samplerCUBE$ (kubiczna). Każda zmienna tego typu zadeklarowana w programie zostanie skojarzona z pewnym rejestrem próbkującym. Nazwa zmiennej jest używana poza mikroprogramem w celu przyporządkowania do niej konkretnej tekstury, określenia typu adresowania i rodzaju filtra. Typ zmiennej jednoznacznie definiuje rodzaj tekstury do niej przyporządkowanej.

Poniższy program kopiuje kolor z rejestru wejściowego do wyjściowego. W ten sposób każdy piksel otrzyma kolor będący wynikiem liniowej interpolacji kolorów pomiędzy trzema wierzchołkami (cieniowanie Gourauda).

```
struct T_PIN // dane interpolowane między wierzchołkami
{
  float4 col :  COLOR0; // kolor wejściowy
};

struct T_POUT // dane wyjściowe
{
  float4 col:  COLOR0; // kolor wynikowy
};

T_POUT PShader(T_PIN pin) // mikroprogram
{
  T_POUT pout;

  pout.col=pin.col; // przepisanie danych

  return pout;
}
```

(Program 3.1)

3.3.2. Zestaw instrukcji

W standardzie 2.0 dla języka HLSL programy dla pikseli nie mogą używać żadnych instrukcji sterujących przepływem. Niedozwolone są instrukcje warunkowe oraz wykonywanie pętli. Zestaw instrukcji składa się z wszystkich rozkazów dla jednostki przetwarzania wierzchołków (tab. 2.3 i tab. 2.4) i dodatkowych próbkujących tekstury (tab. 3.5).

Instrukcja	Opis
tex1D(s,t)	Pobiera kolor o współrzędnej t_x z tekstury jednowymiarowej s .
tex1Dproj(s,t)	Pobiera kolor o współrzędnej $\frac{t_x}{t_w}$ z tekstury jednowymiarowej s.
tex2D(s,t)	Pobiera kolor o współrzędnej (t_x, t_y) z tekstury dwuwymiarowej s.
tex2Dproj(s,t)	Pobiera kolor o współrzędnej $(\frac{t_x}{t_w}, \frac{t_y}{t_w})$ z tekstury dwuwymiarowej s.
tex3D(s,t)	Pobiera kolor o współrzędnej (t_x, t_y, t_z) z tekstury wolumetrycznej s.
tex3Dproj(s,t)	Pobiera kolor o współrzędnej $(\frac{t_x}{t_w}, \frac{t_y}{t_w}, \frac{t_z}{t_w})$ z tekstury wolumetrycznej s.
texCUBE(s,t)	Pobiera kolor o współrzędnej (t_x, t_y, t_z) z tekstury kubicznej s.
texCUBEproj(s,t)	Pobiera kolor o współrzędnej $(\frac{t_x}{t_w}, \frac{t_y}{t_w}, \frac{t_z}{t_w})$ z tekstury kubicznej s.

Tablica 3.5: Dodatkowe instrukcje jednostki PSU

3.3.3. Podstawowe techniki

Programowanie jednostki przetwarzającej piksele sprowadza się do obliczenia jednego wyjściowego koloru. Do dyspozycji są stałe, dane wejściowe pochodzące od wierzchołków oraz tekstury. Przeprowadzanie skomplikowanych obliczeń podczas generowania każdego piksela, pozwala na uzyskanie znacznie bardziej realistycznych efektów niż opisane wcześniej metody. Przede wszystkim chodzi tu o algorytmy symulujące różne właściwości powierzchni, w tym charakterystyki odbijania światła. Wartości liniowo interpolowane między wierzchołkami nigdy nie będą w stanie poprawnie oddać zmian funkcji, które zachodzą we wnętrzu obszaru trójkąta.

Dysponując możliwością przeprowadzenia obliczeń bezpośrednio dla każdego piksela, programy dla wierzchołków tylko przygotowują dane wejściowe. Wektory normalne, kierunek patrzenia obserwatora i inne wartości wektorowe są interpolowane liniowo przez układ. Program dla pikseli pobiera wektory i w zależności od potrzeb normalizuje je, aby zastosowane przekształcenia miały takie same działanie w każdym miejscu trójkąta. Wektory są umieszczane w 8 rejestrach wyjściowych jednostki VSU: $TEXCOORD0$-$TEXCOORD7$, potem interpolowane przez układ i pobierane przez mikroprogram dla pikseli. Oczywiście interpolacja liniowa współrzędnych wektora kierunkowego nie jest poprawną metodą. Interpolowany liniowo powinien być kąt nachylenia wektora a nie jego współrzędne. Błędy te nie są jednak bardzo widoczne a ilość oszczędzonego na obliczeniach czasu jest ogromna. Należy zauważyć, że powyższą metodę stosuje się tylko dla wektorów między którymi kąt jest mniejszy od 180^o.

Poniższy program dla wierzchołków jest przygotowaniem danych do oświetlenia obiektu metodą Phonga (2.14) w każdym generowanym pikselu (Program 3.2). Oblicza on jedynie wektory kierunkowe obserwatora oraz światła w każdym z wierzchołków.

```
float4x4 mxLocalToView; // macierz transformacji
float3 eye_pos; // pozycja obserwatora
float3 light_pos; // pozycja światła

struct T_VIN // dane wejściowe
{
  float3 pos :  TEXCOORD0; // pozycja wierzchołka
  float3 n :  TEXCOORD1; // wektor normalny
};

struct T_VOUT // dane wyjściowe
{
  float4 pos :  POSITION; // pozycja
  float3 eye :  TEXCOORD0; // kierunek patrz.  obserwatora
  float3 light :  TEXCOORD1; // kierunek padania światła
  float3 n :  TEXCOORD2; // wektor normalny
};

T_VOUT VShader(T_VIN vin) // mikroprogram dla wierzchołków
{
  T_VOUT vout;
  float4 pos={0,0,0,1};

  (float3)pos=vin.pos; // pozycja
  vout.pos=mul(mxLocalToView,pos);

  vout.eye=eye_pos-vin.pos; // wektor kierunkowy
  vout.light=light_pos-vin.pos; // wektor kierunkowy
  vout.n=vin.n; // przepisanie

  return vout;
}
```

(Program 3.2)

Po zakończeniu fazy przetwarzania wierzchołków, każdy uruchomiony dla piksela program będzie miał do dyspozycji przeinterpolowany wektor normalny, kierunek z którego patrzy obserwator oraz kierunek z którego pada światło (Program 3.3).

```
// dane światła
float3 light_col_diff; // składowa rozproszona
float3 light_col_spec; // skł.  zwieciadlana
float light_pn; // wsp.  gładkości

float3 scol; // kolor powierzchni obiektu

struct T_PIN // dane wejściowe
{
  float3 eye :  TEXCOORD0; // kierunek patrzenia obserwatora
  float3 light :  TEXCOORD1; // kier.  padania światła
  float3 n :  TEXCOORD2; // wektor normalny |n|!=1
};
                              ↓
```

(Program 3.3)

```
                                ↑
struct T_POUT // dane wyjściowe
{
  float4 col :  COLOR0; // kolor
};

T_POUT PShader(T_PIN pin) // mikroprogram dla pikseli
{
  T_POUT pout;
  float3 eye,eye_rfl,light,n,col;
  float lp;

  // normalizacja wektorów po interpolacji
  eye=normalize(pin.eye);
  light=normalize(pin.light);
  n=normalize(pin.n);

  eye_rfl=reflect(eye,n); // odbicie wektora względem n

  // składowa rozproszona
  lp=max(0,dot(n,light));
  col=light_col_diff*lp;

  // składowa zwierciadlana
  lp=max(0,pow(dot(eye_rfl,light),light_pn));
  col+=light_col_spec*lp;

  col*=scol; // kolor powierzchni

  (float3)pout.col=col;
  pout.col.w=0.0f;

  return pout;
};
```

(Program 3.3)

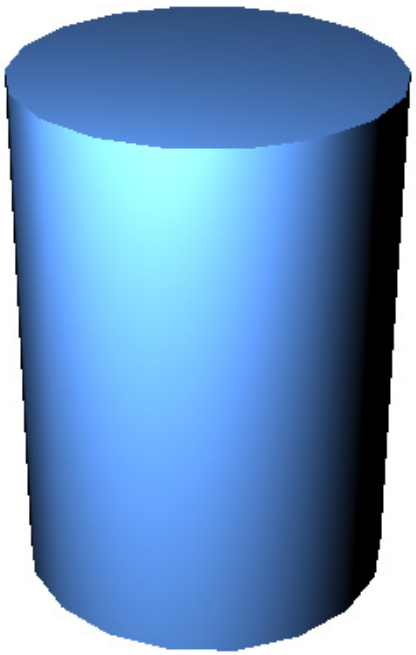
Efekt działania powyższych programów jest przedstawiony na rys. 3.40.

Znormalizowanie 3 wektorów oznacza obliczenie 3 pierwiastków kwadratowych na piksel. Pierwiastek jest operacją arytmetyczną, która zużywa bardzo dużo czasu procesora graficznego. Istnieje metoda, która pozwala przybliżyć wartość pierwiastka przy pomocy odwołania do tekstury kubicznej. Dla każdego teksela tekstury kubicznej o współrzędnych $d = (x, y, z)$, zamiast koloru należy zakodować wartość $\frac{1}{|d|}$. Używając tak przygotowanej tekstury, program dla pikseli może znormalizować wektor d wykonując następującą operację:

$$d_N = texCUBE(s, d) * d.$$

W ten sposób obliczenie pierwiastka kwadratowego zostało zamienione na jedno pobranie wartości z tekstury. Opłacalność tej techniki jest uzależniona od prędkości działania układu. W najnowocześniejszych kartach może się zdarzyć przypadek, że pobranie koloru z tekstury będzie trwało dłużej niż obliczenie pierwiastka. Przyczyną jest pamięć cache, w której procesor umieszcza cały blok tekseli otaczających ten o współrzędnych (x, y, z) zakładając, że będzie się do nich odwoływał w następnej kolejności.

Pamięć cache jest dużo szybsza od podstawowej, ale jest jej bardzo mało. Dlatego załadowanie bloku w celu pobrania tylko jednej wartości może się okazać nieopłacalne.



Rysunek 3.40: Obiekt oświetlony według modelu Phonga w każdym pikselu.

Poniższe dwa mikroprogramy prezentują również technikę oświetlenia modelem Phonga, ale zamiast jednolitego koloru obiektu została nałożona dwuwymiarowa tekstura. W programie zastosowano też normalizację przy użyciu mapy kubicznej.

```
float4x4 mxLocalToView; // macierz transformacji
float3 eye_pos; // pozycja obserwatora
float3 light_pos; // pozycja światła

struct T_VIN // dane wejściowe
{
  float3 pos :  TEXCOORD0; // pozycja wierzchołka
  float3 n :  TEXCOORD1; // wektor normalny
  float2 uv :  TEXCOORD2; // wsp.  tesktury
};

struct T_VOUT // dane wyjściowe
{
  float4 pos :  POSITION; // pozycja
  float3 eye :  TEXCOORD0; // kierunek patrz.  obserwatora
  float3 light :  TEXCOORD1; // kierunek padania światła
  float3 n :  TEXCOORD2; // wektor normalny
  float2 uv :  TEXCOORD3; // wsp.  tekstury
};
                          ↓
```

(Program 3.4)

```
                                ↑
T_VOUT VShader(T_VIN vin) // mikroprogram dla wierzchołków
{
  T_VOUT vout;
  float4 pos={0,0,0,1};

  (float3)pos=vin.pos; // pozycja
  vout.pos=mul(mxLocalToView,pos);

  vout.eye=eye_pos-vin.pos; // wektor kierunkowy
  vout.light=light_pos-vin.pos; // wektor kierunkowy
  vout.n=vin.n; // przepisanie
  vout.uv=vin.uv // przepisanie

  return vout;
}
```

(Program 3.4)

Powyższy mikroprogram jest wykonywany przez jednostkę VSU. Poniżej znajduje się mikroprogram dla jednostki PSU.

```
float3 light_col_diff; // składowa rozproszona
float3 light_col_spec; // skł.  zwieciadlana
float light_pn; // wsp.  gładkości

sampler2D tex; // textura dwuwymiarowa
samplerCUBE nmap; // textura normalizująca

struct T_PIN
{
  float3 eye :  TEXCOORD0; // kier.  patrzenia obserwatora
  float3 light :  TEXCOORD1; // kier.  padania światła
  float3 n :  TEXCOORD2; // wektor normalny |n|!=1
  float2 uv :  TEXCOORD3; // wsp.  tekstury
};

struct T_POUT // dane wyjściowe
{
  float4 col :  COLOR0; // kolor
};

T_POUT PShader(T_PIN pin) // mikroprogram dla pikseli
{
  T_POUT pout;
  float3 eye,eye_rfl,light,n,col,tcol;
  float lp;
                                ↓
```

(Program 3.5)

```
                                    ↑
  // normalizacja wektorów po interpolacji
  eye=texCUBE(nmap,pin.eye)*pin.eye;
  light=texCUBE(nmap,pin.light)*pin.light;
  n=texCUBE(nmap,pin.n)*pin.n;

  eye_rfl=reflect(eye,n); // odbicie wektora względem n

  // składowa rozproszona
  lp=max(0,dot(n,light));
  col=light_col_diff*lp;

  // składowa zwierciadlana
  lp=max(0,pow(dot(eye_rfl,light),light_pn));
  col+=light_col_spec*lp;

  col*=tex2D(tex,pin.uv); // kolor powierzchni

  (float3)pout.col=col;
  pout.col.w=0.0f;

  return pout;
};
```

(Program 3.5)

Rysunek 3.41: Obiekt z nałożoną teksturą oświetlony według modelu Phonga w każdym pikselu.

3.4. Operacje na buforach

3.4.1. Wyznaczanie widoczności

W rozdziale pierwszym została wyjaśniona idea pracy bufora-Z. Po obliczeniu koloru piksela, jego odległość od obserwatora jest porównywana z wartością już znajdującą się w buforze-Z. W przypadku, kiedy test da wynik negatywny, piksel nie jest poddawany dalszym operacjom i kolor nie jest zapisywany do bufora koloru. Jeśli wynik jest pozytywny, to piksel bierze udział w dalszych operacjach a stara wartość w buforze Z jest nadpisywana. Bufor-Z ma zawsze taką samą rozdzielczość jak bufor koloru.

W porównaniach bierze udział współrzędna z', odpowiadająca odległości od obserwatora aktualnie generowanego piksela:

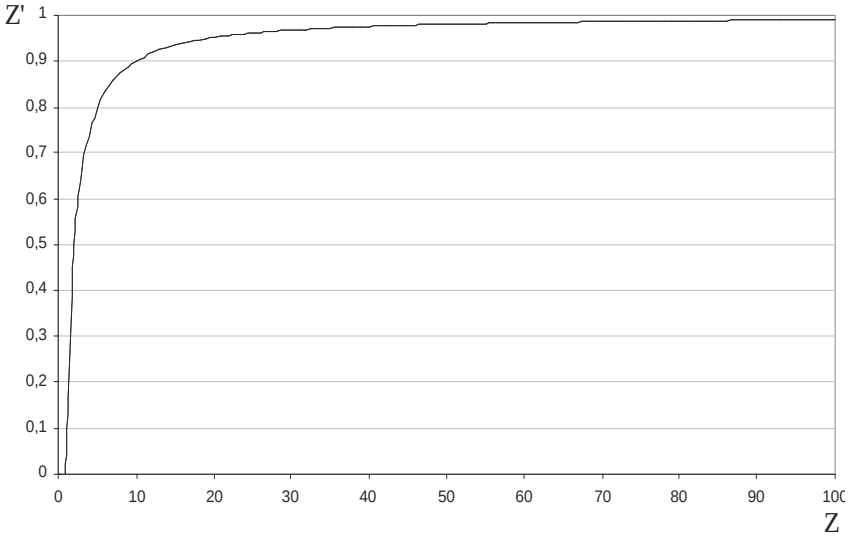
$$z' = \frac{z\frac{z_F}{z_F - z_N} - z_N\frac{z_F}{z_F - z_N}}{z}, \text{ gdzie } z' \in [0,1]. \tag{3.31}$$

Odległość z jest interpolowana na obszarze trójkąta metodą hiperboliczną. Ze względu na to, że z' już jest funkcją hiperboliczną ze względu na z, prawidłowa interpolacja przyjmuje postać liniową $z' = (1-\alpha)z'_1 + \alpha z'_2$.

Załóżmy, że $z_N \ll z_F$, wtedy $\frac{z_F}{z_F - z_N} \approx 1$ a $z_N\frac{z_F}{z_F - z_N} \approx z_N$. Równanie (3.31) przyjmie wówczas postać:

$$z' = 1 - \frac{z_N}{z}.$$

Wykres tej zależności dla $z_N = 1$ wygląda następująco:



Rysunek 3.42: Wykres rozkładu odległości od obserwatora w buforze-Z.

Dla $z = 2$ wartość z' jest równa $\frac{1}{2}$, czyli 50% zakresu bufora-Z jest zużywane na wartości odległości, które leżą bardzo blisko obserwatora. Każda liczba rzeczywista zapisana na komputerze ma skończoną dokładność i połowa dostępnych możliwych liczb jest pochłaniana przez bardzo małą część zakresu zmiennej z. Odbija się to bezpośrednio na dokładności porównań dla odległości tylko trochę dalszych od minimalnej z_N. Stopień zaburzeń będzie tym większy im bliższe zera będzie z_N. Błędy objawiają się w postaci przenikania się trójkątów, które leżą bardzo blisko siebie. Odjęcie od siebie ich odległości daje właściwie wartość przypadkową. Wartość maksymalna z_F nie ma praktycznie żadnego wpływu na dokładność porównań.

Bufor-Z daje satysfakcjonujące wyniki tylko wtedy, gdy wartość z_N jest ustawiona na najwyższą akceptowalną dla obserwatora. Wiąże się z tym brak możliwości podejścia do obiektu bardzo blisko, gdyż trójkąty będą ucinane przez płaszczyznę $z = z_N$.

Liniowy rozkład można uzyskać stosując tzw. bufor-W. Zamiast współrzędnej z', miarą odległości jest wtedy przeskalowana współrzędna w: $w' = \alpha w + \beta$. Interpolacja jest jednak bardzo kosztowna, gdyż interpolowana liniowo musi być wartość $\frac{1}{w'}$. W celu porównania odległości, w każdym pikselu wartość ta musi być odwrócona, co wymusza wykonanie jednego dzielenia na piksel.

Bufor-w nie jest jednak lepszym rozwiązaniem ze względu na rzut perspektywiczny. Obiekty leżące bardzo blisko obserwatora będą porównywane z dużo mniejszą dokładnością niż w przypadku bufora-Z. Tym razem błędy pojawią się nie w dalszych odległościach, ale w bliższych. Decyzja wyboru bufora powinna wiązać się z rodzajem sceny i wymaganiami stawianymi jej wyglądowi.

Problem ten nie występuje, jeśli macierz przekształceń dla wierzchołków nie jest osobliwa. Jeśli dolny wiersz macierzy M jest równy $[0, 0, 0, 1]$, to $w = 1$ i wartości odległości w buforze-Z rozkładają się liniowo.

Test widoczności obejmuje wykonanie porównania nowej wartości głębokości ze starą. Wiele efektów specjalnych korzysta z możliwości zmiany rodzaju porównania. Standardowo test daje wynik pozytywny, jeśli nowa wartość jest mniejsza bądź równa starej. DirectX umożliwia stosowanie dowolnego rodzaju operacji porównania wraz z możliwością wyłączenia zapisywania nowych wartości do bufora.

3.4.2. Bufor zliczania

Bufor zliczania (ang. stencil buffer) jest zawsze takiej samej rozdzielczości, co bufor koloru. Każdemu pikselowi jest przyporządkowana jedna liczba całkowita nieujemna, znajdująca się w buforze zliczania. Najbardziej popularna jest wersja 8-bitowa, mogąca przechowywać liczby od 0 do 255.

Po teście widoczności, każdy piksel może przejść test z buforem zliczania. W pierwszej kolejności, jeszcze przed wysłaniem jakiejkolwiek geometrii do karty, programista ustala wartość jednej stałej całkowitoliczbowej S_c. Podczas generowania obrazu, dla każdego wygenerowanego piksela jest przeprowadzany test, którego wynikiem jest prawda albo fałsz. Piksel nie zostanie narysowany, jeśli test będzie miał wynik negatywny. Test składa się z porównania wartości S_c z wartością, która znajduje się w buforze zliczania pod współrzędnymi aktualnie przetwarzanego piksela. Tak jak w buforze-Z, operacja porównania może być dowolna.

Dla każdego z dwóch możliwych wyników testu, programista ustala jak zmieni się

wartość w buforze zliczania. Może pozostawić ją bez zmian, dodać lub odjąć 1, zastąpić wartością S_c lub wykonać inwersję bitów. Dodatkowo, operacja porównania i reakcji na nie, może być wykonana na argumentach z wyłączeniem żądanych bitów. Maski podaje się przed generowaniem obrazu sceny.

Przykładem użycia bufora zliczania jest testowanie algorytmów wyznaczania widoczności. Funkcja porównania ustawiona jest wtedy na zwracanie zawsze prawdy a wartości w buforze po wykonaniu testu zwiększane są o 1. Po narysowaniu całej sceny, w każdym miejscu bufora zliczania będzie znajdować się liczba narysowanych w tym miejscu pikseli. Bufor zliczana stanowi też niezbędny element jednej z bardzo popularnych technik rysowania cieni, która została opisana w następnym rozdziale.

3.4.3. Alpha-Blending

Alpha-blending jest ostatnim etapem generowania obrazu. Obliczony kolor piksela staje się jednym z argumentów funkcji, której wynikiem jest kolor ostatecznie wpisywany do bufora koloru. Nazwa tej techniki pochodzi od czwartego komponentu wektora koloru, zwanego współczynnikiem alpha. Dla przykładu model RGBA oznacza, że pierwsze trzy składowe opisują kolor a czwarta dodatkowy współczynnik alpha. Współczynnik alpha ma z reguły taki sam format jak pozostałe komponenty, najpopularniejszy jest format 8-8-8-8, czyli 8 bitów na każdą składową.

Funkcja, o której mowa w poprzednim akapicie ma następującą postać:

$$K = F_S K_S + F_D K_D, \tag{3.32}$$

gdzie K oznacza ostateczny kolor wpisywany do bufora, K_S - obliczony przez jednostkę PSU kolor piksela a K_D - stary kolor znajdujący się w buforze koloru. F_S i F_D są współczynnikami, z których każdy może przyjąć jedną z podanych w tab. 3.6 postaci. K_C oznacza wartość koloru, który jest stałym parametrem ustalanym przed generowaniem obrazu.

Nazwa	**Wartość**
ZERO	$(0, 0, 0, 0)$
ONE	$(1, 1, 1, 1)$
SRCCOLOR	$(K_S.r, K_S.g, K_S.b, K_S.a)$
INVSRCCOLOR	$(1 - K_S.r, 1 - K_S.g, 1 - K_S.b, 1 - K_S.a)$
SRCALPHA	$(K_S.a, K_S.a, K_S.a, K_S.a)$
INVSRCALPHA	$(1 - K_S.a, 1 - K_S.a, 1 - K_S.a, 1 - K_S.a)$
DESTALPHA	$(K_D.a, K_D.a, K_D.a, K_D.a)$
INVDESTALPHA	$(1 - K_D.a, 1 - K_D.a, 1 - K_D.a, 1 - K_D.a)$
DESTCOLOR	$(K_D.r, K_D.g, K_D.b, K_D.a)$
INVDESTCOLOR	$(1 - K_D.r, 1 - K_D.g, 1 - K_D.b, 1 - K_D.a)$
SRCALPHASAT	(f, f, f, f), gdzie $f = \min\{K_S.a, 1 - K_D.a)$
BLENDFACTOR	$(K_C.r, K_C.g, K_C.b, K_C.a)$
INVBLENDFACTOR	$(1 - K_C.r, 1 - K_C.g, 1 - K_C.b, 1 - K_C.a)$

Tablica 3.6: Współczynniki używane podczas alpha-blendingu.

Składowe koloru (r, g, b, a) mogą być używane w mikroprogramach zamiennie z oznaczeniami wektora (x, y, z, w).

Podstawowym zastosowaniem alpha-blendingu jest symulacja przeźroczystości powierzchni. W składowej alpha koloru powierzchni umieszcza się współczynnik przeźroczystości i rysuje powierzchnię ze współczynnikami $F_S = SRCALPHA$ i $F_D = INVSRCALPHA$. W ten sposób funkcja (3.32) interpoluje liniowo kolory K_S i K_D ze współczynnikiem $K_S.a$. Przy stopniach przeźroczystości różnych od $\frac{1}{2}$ wszystkie obiekty półprzeźroczyste należy rysować na ekranie od tyłu do przodu. W przeciwnym wypadku obraz wynikowy nie będzie prawidłowy. Niestety są przypadki, kiedy nie uda się posortować trójkątów w całości i trzeba stosować bardziej zaawansowane techniki żeby ten porządek utrzymać.

Drugim przykładem jest tzw. rendering wieloprzebiegowy. Obliczanie w mikroprogramie dla pikseli natężenia z kilku źródeł światła jednocześnie może okazać się niewykonalne. Brakuje kontroli przepływu a ilość rozkazów jest ograniczona. Pozostaje wtedy narysować tą samą geometrię kilka razy, ale umieszczając w buforze koloru inne wartości. Najpierw rysuje się natężenia pierwszego światła. Przy każdym kolejnym, rysuje się ten sam obiekt ze współczynnikami $F_S = ONE$ i $F_D = ONE$. W ten sposób w buforze koloru znajdzie się suma wszystkich natężeń światła. Ostatnim etapem jest narysowanie obiektu kolorem powierzchni z parametrami $F_S = DESTCOLOR$ i $F_D = ZERO$, czyli mnożąc kolor powierzchni przez natężenie światła.

Takie podejście ma jednak poważną wadę. Dokładność liczb umieszczonych w buforze koloru jest dużo mniejsza niż używanych bezpośrednio w mikroprogramach. Powstają ogromne przekłamania w gradientach kolorów. Standardowy bufor koloru ma dokładność 8 bitów na składową, co oznacza, że każda składowa może przyjąć maksymalnie 256 wartości. Rozwiązaniem może być rysowanie do tekstury w formatach zmiennopozycyjnych, ale na nieszczęście współczesne akceleratory nie umożliwiają alpha-blendingu wraz z formatami zmiennopozycyjnymi.

Należy wyraźnie zaznaczyć, że każda składowa koloru obliczonego przez jednostkę PSU jest obcinana do przedziału $[0, 1]$. W buforze koloru odwzorowywanym na ekranie nie mogą się znaleźć wartości spoza niego. Bezpośrednim powodem jest zakres intensywności kolorów, który jest w stanie wyświetlić monitor. Jeśli rysujemy do tekstury, to jest to możliwe, ale tylko przy teksturach o formacie zmiennopozycyjnym.

Rozdział 4

Techniki zaawansowane

Wygląd wygenerowanej sceny zależy od stopnia skomplikowania metod użytych do obliczania koloru poszczególnych pikseli. Proste techniki, które uwzględniają kilka tekstur i lokalny model oświetlenia często nie wystarczają, aby obraz był zadowalającej jakości. W tym rozdziale są przedstawione bardziej zaawansowane sposoby generowania obrazu, które pozwalają na nadanie scenie bardziej realistycznego oblicza. Należy pamiętać, że przedstawione tu metody nie mają na celu jak najlepszego odwzorowania rzeczywistości, ale jedynie wywołanie u obserwatora wrażenia, że obraz jest faktycznie zbliżony do rzeczywistego. Aby osiągnąć ten cel, często używane są metody, które nie mają podłoża fizycznego a jedynie dają optyczne podobieństwo do zjawisk zachodzących w realnym świecie.

Opisane w tym rozdziale techniki są jedynie skromnym wycinkiem możliwości, jakie oferują układy graficzne. Wybrane zostały te efekty, które demonstrują wykorzystanie szerokiego zakresu możliwości układu w możliwie jak najbardziej różnorodny sposób. Do każdego podrozdziału jest dołączony program, który demonstruje działanie opisywanej techniki. Do jego uruchomienia jest potrzebny układ graficzny, który obsługuje w pełni model mikroprogramów w wersji 2.0. W przypadku nie dysponowania tego typu sprzętem, fragment działania każdego z programów został nagrany do pliku AVI.

4.1. Faktura powierzchni i oświetlenie

Kolor każdego piksela jest obliczany wewnątrz mikroprogramu. Danymi wejściowymi są atrybuty pochodzące z wierzchołków oraz ograniczona ilość stałych rzeczywistych. Przy pomocy takiego mechanizmu nie jest możliwa symulacja w czasie rzeczywistym globalnych modeli oświetlenia, gdyż informacje o wyglądzie całej sceny nie zmieszczą się w rejestrach stałych. W typowych zastosowaniach miejsca wystarcza tylko na zapisanie pozycji i atrybutów punktowych źródeł światła.

Globalne modele oświetlenia mogą być zastosowane, ale tylko pod warunkiem, że nie zależą one od pozycji obserwatora na scenie. W takim przypadku, dane przeliczonego wcześniej przez procesor główny oświetlenia są zapisywane w postaci tekstur i później nakładane przez układ statycznie na obiekty. W ten sposób na powierzchni każdego trójkąta będą dostępne dane o intensywności i kolorze rozproszonej składowej oświetlenia. Niestety, ilość miejsca na tekstury również jest ograniczona i ma to zasadniczy wpływ na dokładność tych danych (mała rozdzielczość tekstur).

Do niedawna, nawet symulacja lokalnych modeli oświetlenia pozostawiała wiele do życzenia. Dopiero układy w pełni programowalne i dostatecznie szybkie pozwoliły na

stosowanie bardziej skomplikowanych metod. Wraz z lepszymi układami pojawiły się również możliwości dokładniejszego symulowania optycznych właściwości materiałów, z których składają się poszczególne obiekty.

4.1.1. Symulacja nierówności powierzchni

Obiekty sceny składają się z siatki połączonych ze sobą trójkątów. Bardzo szczegółowe odtworzenie obiektu wymaga ogromnej ilości trójkątów, które opisują jego kształt. W przypadku, kiedy obiekt jest zbudowany z materiału, którego powierzchnia jest nierówna i chropowata, opisanie tych małych odkształceń za pomocą osobnych trójkątów pociąga za sobą bardzo duże koszty obliczeniowe. Ze względu na stosunkowo niedużą moc obliczeniową współczesnych układów graficznych, wszystkie elementy sceny muszą składać się z jak najmniejszej ilości trójkątów.

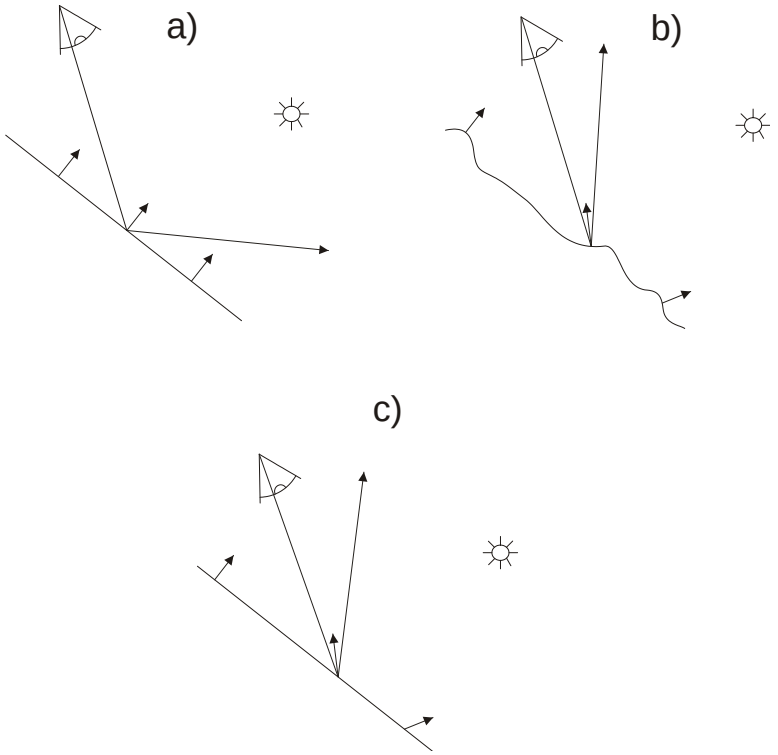
Rozwiązaniem powyższego problemu jest technika symulowania małych nierówności powierzchni za pomocą odpowiednio przygotowanych tekstur (ang. Bump Mapping). Pomysł pochodzi z roku 1978, w którym James Blinn zaprezentował tę technikę po raz pierwszy [12].

Podstawowym komponentem obliczeń związanych z oświetleniem jest wektor prostopadły do powierzchni, zwany wektorem normalnym. Standardowo, wektor ten jest obliczany na wierzchołkach obiektu i interpolowany na obszarze trójkąta. W celu dokładnego odwzorowania nierówności powierzchni, wektor normalny (x, y, z) jest zapisany w teksturze pod trzema składowymi koloru (r, g, b). Tekstura ta pokrywa całą powierzchnię obiektu, więc mikroprogram dla pikseli może pobrać z tekstury wartość wektora normalnego i użyć go zamiast tego interpolowanego pomiędzy wierzchołkami (rys. 4.43).

Jedynym problemem, jaki pojawia się przy zastosowaniu powyższej techniki jest zgodność układów odniesienia, w których są zapisane wektory normalne oraz wektor kierunkowy światła i obserwatora. Do poprawnego obliczenia wartości natężenia światła wszystkie te wektory muszą być określone w tym samym układzie odniesienia. Istnieją dwie metody rozwiązania tego problemu, z których każda ma swoje zalety i wady.

Pierwszą z metod jest wygenerowanie tekstury w taki sposób, aby wektory normalne były zapisane w tym samym układzie współrzędnych, co wierzchołki obiektu (ang. object space bump mapping). Obliczenie natężenia światła nie pociąga za sobą konieczności żadnych dodatkowych obliczeń. W stałych dla mikroprogramu podaje się wówczas współrzędne obserwatora i pozycję światła w układzie obiektu. Następnie mikroprogram dla wierzchołków oblicza wektory kierunkowe obserwatora i światła, które są później interpolowane i normalizowane na powierzchni trójkątów. Mikroprogram dla pikseli pobiera wektor normalny z tekstury za pomocą współrzędnych, które na etapie generowania mapy wektorów normalnych zostały przyporządkowane do poszczególnych wierzchołków. Pozostaje już tylko obliczyć natężenie światła według wybranego modelu.

Istotną wadą tego rozwiązanie jest to, że każdemu punktowi na powierzchni obiektu musi odpowiadać dokładnie jeden punkt na teksturze wektorów normalnych. Dla przykładu, jeśli obiekt ma imitować mur i jego powierzchnia składa się z powtarzającego się obok siebie wzoru cegły, to tekstura dla wektorów normalnych będzie niestety musiała



Rysunek 4.43: Wektor interpolowany na powierzchni obiektu (a) nie oddaje natury chropowatej powierzchni (b). Przygotowana mapa wektorów normalnych (c) pozwala ten efekt uzyskać.

zawierać wzory nierówności dla każdej cegły. Jest to marnowanie miejsca w pamięci karty graficznej, gdyż wystarczy jeden wzór nierówności dla pojedynczej cegły, który będzie powielany na obszarze muru przy pomocy jednego z trybów adresowania tekstury (str. 58).

Jeżeli priorytetem jest ilość pamięci zajmowanej przez tekstury, można zapisać współrzędne wektorów normalnych w układzie tekstury (ang. texture space bump mapping). W odróżnieniu od poprzedniej metody, przygotowanie mapy nierówności nie wymaga znajomości wyglądu obiektu, na który zostanie ona nałożona. Niech G oznacza układ współrzędnych obiektu a T układ współrzędnych tekstury. Wersory układu T pokrywają się z osiami U i V tekstury, czyli są równe odpowiednio $U = (1, 0, 0)$, $V = (0, 1, 0)$ i $W = (0, 0, 1)$. Wersor W jest prostopadły do płaszczyzny tekstury. Jeżeli na powierzchni nie ma być żadnych chropowatości, to wszystkie elementy tekstury będą równe $(0, 0, 1)$, czyli będzie to wektor normalny skierowany prostopadle do płaszczyzny tekstury. Na prostokątnej powierzchni mapy nierówności można odwzorować powierzchnię na przykład pojedynczej cegły.

W następnej kolejności mapa nierówności jest nakładana na powierzchnię obiektu. Każdemu wierzchołkowi zostaje przyporządkowana para współrzędnych $u \in \mathbb{R}$ i $v \in \mathbb{R}$, które określają położenie wierzchołka na mapie nierówności. W ten sposób w każdym punkcie na powierzchni trójkąta są znane współrzędne, pod którymi znajduje się

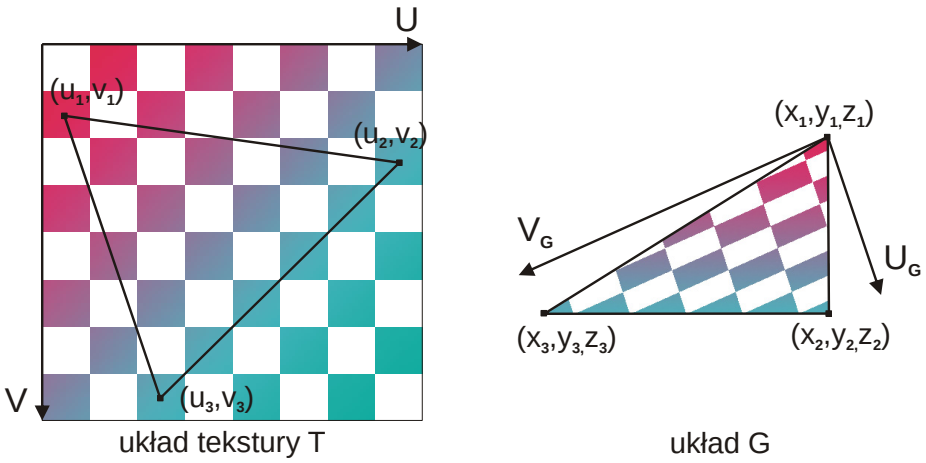
wektor normalny $N_T(u, v)$ zapisany w układzie odniesienia tekstury. Należy zauważyć, że pojedynczy wektor normalny może być przyporządkowany wielu punktom na powierzchni obiektu. Można to osiągnąć stosując adresowanie w trybie powtarzania, wtedy każdy punkt z przyporządkowanymi współrzędnymi $(i+p, j+q)$, gdzie $\{i, j\} \in \mathbb{Z}$ i $\{p, q\} \in \{[0, 1), [0, 1)\}$, będzie miał identyczny wektor normalny (w układzie T).

Oprócz tych współrzędnych, potrzebne są jeszcze informacje o orientacji układu tekstury względem układu odniesienia obiektu G. Macierz $M_{T\to G} \in \mathbb{R}^{3x3}$, która odpowiada transformacji wektora kierunkowego z układu T do G, może być inna dla każdego trójkąta. Z założenia, płaszczyzna U/V układu T pokrywa się z płaszczyzną wyznaczoną przez powierzchnię trójkąta (rys. 4.44). Postać macierzy $M_{T\to G}$ jest przedstawiona poniżej:

$$M_{T\to G} = \begin{pmatrix} U_G & V_G & N_G \end{pmatrix},$$

gdzie U_G i V_G oznaczają wersory układu T o współrzędnych zapisanych w układzie G:

$$U_G = \begin{pmatrix} \frac{\partial x}{\partial u} & \frac{\partial y}{\partial u} & \frac{\partial z}{\partial u} \end{pmatrix}, \quad V_G = \begin{pmatrix} \frac{\partial x}{\partial v} & \frac{\partial y}{\partial v} & \frac{\partial z}{\partial v} \end{pmatrix}, \quad N_G = V_G \times U_G. \tag{4.33}$$



Rysunek 4.44: Wersory U i V układu tekstury T i ich odpowiedniki U_G i V_G w układzie obiektu G.

Ze względu na błędy podczas nakładania tekstury z wektorami normalnymi na obiekt, wektory U_G i V_G mogą mieć długości różne od 1 i mogą też nie być do siebie prostopadłe (rys. 4.44). Podczas transformacji wektorów kierunkowych spowoduje to niepożądane zniekształcenia ich kierunku i długości. Są one niepożądane, ponieważ zasadą obowiązującą podczas nakładania jakiejkolwiek tekstury na obiekt jest zachowanie proporcji i kwadratowego kształtu tekseli na powierzchni trójkątów. Wszelkie odstępstwa od tej reguły są wynikiem błędów i nie powinny mieć wpływu na kierunek

transformowanych wektorów. Wektory składające się na macierz $M_{T\to G}$ są w pierwszej kolejności normalizowane a później ortogonalizowane, na przykład metodą Grahama-Shmidta:

$$\begin{aligned}\bar{U}_G &= U_G - (N_G \cdot U_G)N_G,\\ \bar{V}_G &= V_G - (N_G \cdot V_G)N_G - (\bar{U}_G \cdot V_G)\bar{U}_G,\\ \bar{N}_G &= N_G\end{aligned}$$

która w przypadku zależności (4.33) redukuje się do:

$$\begin{aligned}\bar{U}_G &= U_G,\\ \bar{V}_G &= V_G - (U_G \cdot V_G)U_G,\\ \bar{N}_G &= N_G.\end{aligned}$$

Macierz $M_{G\to T}$, która odpowiada transformacji wektora kierunkowego z układu G do układu tekstury T, ma postać:

$$M_{G\to T} = \begin{pmatrix}\bar{U}_G^T\\ \bar{V}_G^T\\ \bar{N}_G^T\end{pmatrix}.$$

Przy pomocy macierzy $M_{G\to T}$ można dokonać transformacji wektorów kierunkowych światła i obserwatora do układu tekstury T. Po wykonaniu tej operacji i pobraniu z tekstury wektora normalnego $N_T(u,v)$ możliwe jest już obliczenie natężenia światła.

Podobnie jak w przypadku zwykłego oświetlania obiektów, wektory składające się na macierz $M_{G\to T}$ są uśredniane na wierzchołkach, ortogonalizowane i interpolowane na powierzchni trójkątów. Daje to wrażenie wygładzenia oświetlenia na krawędziach pomiędzy trójkątami.

Schemat pracy układu graficznego w przypadku tej metody sprowadza się do interpolacji wektorów $\bar{U}_G$, $\bar{V}_G$ i $\bar{N}_G$ pomiędzy wierzchołkami trójkątów by następnie w mikroprogramie dla pikseli znormalizować je i złożyć w macierz $M_{G\to T}$. Dla każdego generowanego piksela wektory kierunkowe świateł i obserwatora są transformowane do układu T, po czym następuje obliczenie natężenia światła przy pomocy wektora normalnego $N_T(u,v)$. Ponowna ortogonalizacja wektorów macierzy $M_{G\to T}$ w mikroprogramie dla pikseli jest opcjonalna. Przy dobrze nałożonej mapie nierówności i odpowiednio dużej ilości trójkątów składających się na obiekt, zniekształcenia spowodowane interpolacją powinny być niezauważalne.

Interesującym sposobem wykorzystania pierwszej z zaprezentowanych tutaj metod jest szybkie rysowanie bardzo szczegółowych pod względem geometrii modeli. Po zaprojektowaniu modelu, który składa się z kilkudziesięciu lub nawet kilkuset tysięcy trójkątów, generowany jest automatycznie model o znacznie mniejszej ilości trójkątów. Jest to możliwe przy zastosowaniu wielu ogólnie dostępnych algorytmów do redukcji ilości ścian obiektów. Następnie dla modelu o małej ilości ścian jest generowana

mapa nierówności powierzchni, która powstaje dzięki informacji pochodzącej od dokładnej wersji modelu. Algorytmy wykonujące ta operację są dość skomplikowane w konstrukcji, ale efekt uzyskany dzięki nim jest zdumiewający (rys. 4.45).

Rysunek 4.45: Po lewej stronie normalnie oświetlony model (ok. 1000 trójkątów). Po prawej ten sam model oświetlony przy użyciu wcześniej wygenerowanej mapy nierówności powierzchni. Oryginalny model jest zbudowany z 35000 trójkątów.

Do pracy został dołączony program, z którego pochodzi powyższa ilustracja. Znajduje się on w katalogu /programy/4_1_1. Treść mikroprogramów użytych w programie jest dostępna w pliku z rozszerzeniem *.fx w tym samym katalogu.

4.1.2. Tekstury proceduralne

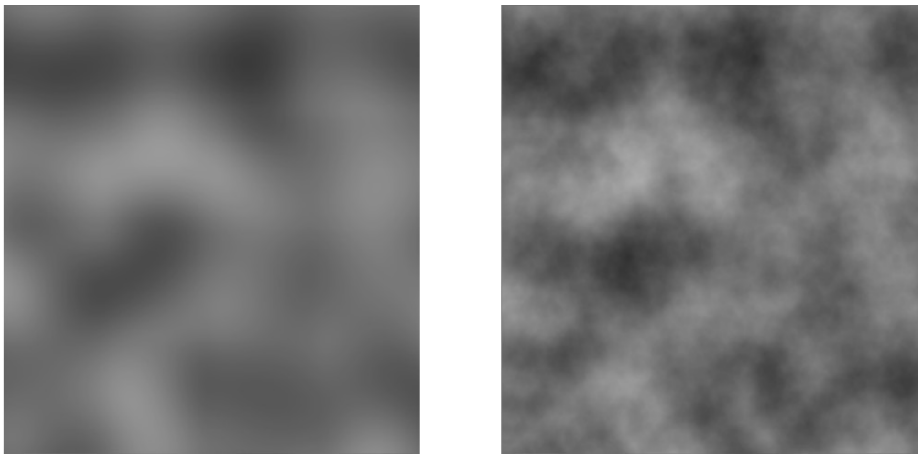
Klasyczne materiały używane podczas generowania obrazów są skonstruowane z dwuwymiarowych tekstur, nakładanych na powierzchnię trójkątów. Budowa obiektu często uniemożliwia nałożenie tekstury w sposób, który da zadowalające wyniki. Często spotykanym efektem ubocznym są nieciągłości kolorów powstałe na styku dwóch trójkątów, na które została nałożona tekstura. Przy dużym skomplikowaniu siatki obiektu takich

miejsc może być bardzo dużo. Drugim problemem jest zachowanie proporcji teksela na powierzchni obiektu. Im mniej nieciągłości, tym bardziej zdeformowana i rozciągnięta może być tekstura. Nie istnieją automatyczne algorytmy, które w każdej sytuacji będą generować poprawne wizualnie wyniki. Co więcej, w wielu przypadkach jest to po prostu niemożliwe.

Częściowym rozwiązaniem powyższego problemu jest stosowanie zupełnie innego rodzaju opisu materiałów. W 1983 roku Ken Perlin wymyślił sposób, który umożliwia opisanie koloru powierzchni za pomocą funkcji o trójwymiarowej dziedzinie [13]. Jako argumenty tej funkcji podawane są współrzędne punktu w $\mathbb{R}^3$, dla którego ma być obliczony kolor. Jeden opis materiału wystarczy do pokrycia nim obiektów o dowolnym kształcie i topologii. Oczywiście bardzo trudno jest opracować tego typu funkcje dla wszystkich materiałów. Metoda ta daje jednak doskonałe wyniki dla tych zbudowanych na bazie fraktali, a więc opartych na samopodobieństwie. Dla przykładu może to być kamień lub drewno.

Metoda Kena Perlina opisuje tylko podstawowe mechanizmy, które stanowią niejako aparat do konstruowania bardziej realistycznych efektów. Ostateczny kształt obliczeń zależy wyłącznie od pomysłowości twórcy i nie ma tu żadnych ściśle obowiązujących reguł.

W pierwszej kolejności należy wygenerować trójwymiarową teksturę szumu. Tekstura ta składa się z wartości wybranych losowo lub pseudolosowo, które są później rozmywane za pomocą filtrów. Losowe liczby należą do przedziału [0,1] i są rozłożone równomiernie na przestrzeni tekstury, niekoniecznie w każdym tekselu jedna. Aby uzyskać szum o mniejszej ziarnistości, losuje się liczby co kilka tekseli i później rozmywa w taki sam sposób (lewa strona rys. 4.46). Odwrotność odległości pomiędzy próbkami jest częstotliwością szumu. Filtrowanie jest przeprowadzane tak, aby naprzeciwległe ściany tekstury wolumetrycznej były identyczne. Przy użyciu trybu powtarzania adresowania tekstury, można indeksować ją współrzędnymi poza przedziałem [0,1) zachowując ciągłość danych.



Rysunek 4.46: Po lewej stronie przekrój tekstury szumów. Po prawej, obraz powstały po odpowiednim zsumowaniu szumów o malejących częstotliwościach.

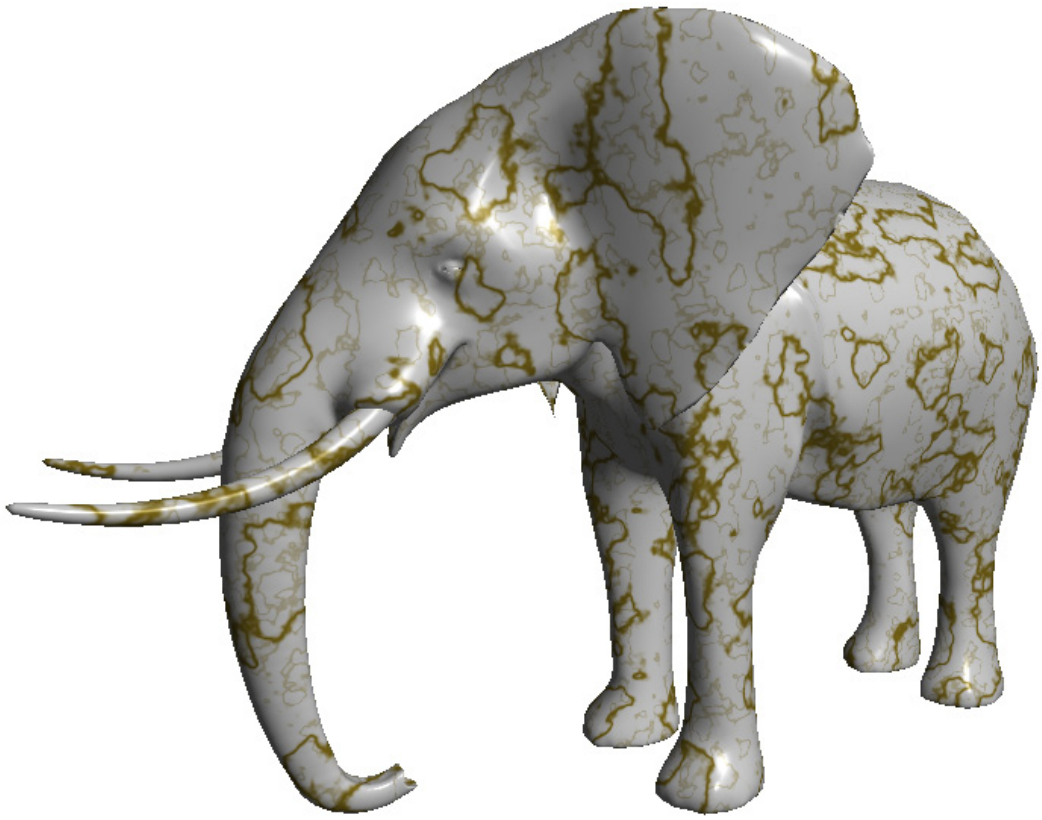
Podstawową operacją jest sumowanie szumów o różnych częstotliwościach. Przykładowo, jeżeli szum dany jest funkcją $N(x, y, z)$, przy obliczaniu koloru można zastosować następującą zależność:

$$c = \sum_{1}^{n} \frac{1}{n} N(nx, ny, nz). \tag{4.34}$$

Funkcje tego typu, ze względu na ich naturę, zwane są sumami fraktalnymi. Efekt działania równania (4.34) można zobaczyć po prawej stronie rys. 4.46.

Przy tworzeniu materiałów proceduralnych dozwolona jest pełna dowolność kształtu obliczeń. Podczas generowania koloru w punkcie, stosuje się dowolne funkcje mieszające dane z tekstury szumu. Oprócz różnego rodzaju sum fraktalnych często używane są filtry dolno i górno przepustowe, funkcje mieszające kolory oraz tekstury pomocnicze.

Przykład materiału proceduralnego jest przedstawiony na rys. 4.47. Funkcja obliczająca kolor pochodzi z biblioteki materiałów profesjonalnego systemu generowania obrazu Renderman firmy Pixar.



Rysunek 4.47: Przykład materiału proceduralnego.

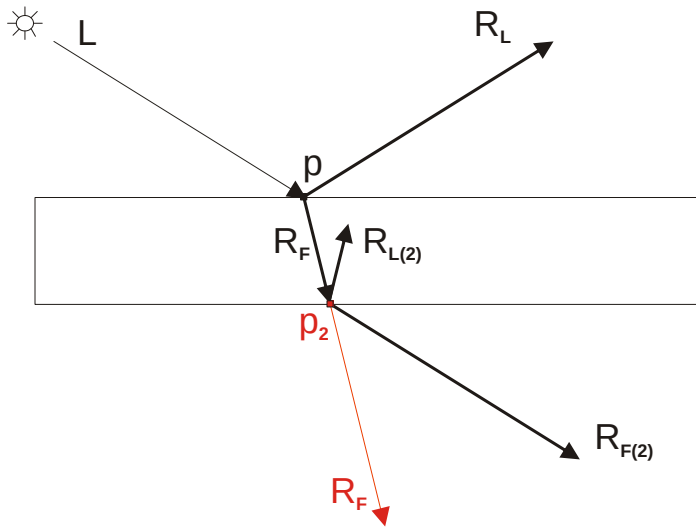
Rola układu graficznego w przypadku materiałów proceduralnych polega na obliczeniu w mikroprogramie dla pikseli koloru, korzystając z wolumetrycznej tekstury szumu. Dane wejściowe stanowią współrzędne wierzchołków, które są interpolowane przez układ na powierzchni całego trójkąta. Pozycja wierzchołka jest zapisywana w rejestrach wyjściowych mikroprogramu dla wierzchołków i pobierana po interpolacji z rejestrów wejściowych jednostki PSU.

Model z rys. 4.47 można obejrzeć interaktywnie, uruchamiając dołączony do pracy program. Znajduję się on w katalogu /programy/4_1_2 a mikroprogramy wykonywane przez kartę graficzną są w pliku z rozszerzeniem *.fx w tym samym katalogu.

4.1.3. Odbicie i załamanie światła

Odbicie i załamanie światła jest naturalnym zjawiskiem podczas rozchodzenia się światła. Powstało wiele algorytmów, które w mniejszym lub większym stopniu oddają naturę tego zjawiska. Jednym z nich jest popularna metoda śledzenia promieni. Niestety, przy użyciu akceleratorów graficznych generowanie obrazu tą techniką jest niemożliwe. Podczas obliczania koloru piksela przez mikroprogram, nie są dostępne informacje o budowie i pozycji obiektów otaczających przetwarzany aktualnie trójkąt. Opracowane zostały jednak metody, które pozwalają na prowizoryczne odtworzenie tych zjawisk.

Zgodnie z zasadami metody śledzenia promieni, promień (L) biegnący od obserwatora rozdziela się w momencie przecięcia z powierzchnią obiektu na dwie części, składową odbitą R_L oraz składową załamaną R_F (rys. 4.48). Po przejściu promienia R_F przez ośrodek, rozdziela się on w punkcie p_2 na kolejne dwie składowe, $R_{L(2)}$ i $R_{F(2)}$. Przy pomocy układu graficznego można zasymulować wyłącznie pierwsze rozdzielenie się promieni, ponieważ mikroprogram obliczając kolor w punkcie p nie posiada informacji o rozmieszczeniu wszystkich obiektów na scenie. Co więcej, nie jest w stanie znaleźć dalszych punktów przecięcia promieni R_L i R_F.



Rysunek 4.48: Metoda śledzenia promieni.

Wektory R_L i R_F można wykorzystać jako współrzędne tekstury kubicznej. Przy odpowiednio spreparowanej teksturze, wartość pobranego koloru będzie odpowiadać kolorowi światła emitowanego z tego kierunku przez dalekie otoczenie. Więcej na ten temat można znaleźć w podrozdziale 3.2.2.

Należy zauważyć, że składowa światła odbitego będzie prawidłowa, ale załamanego już nie. Podczas pobierania koloru światła emitowanego przez dalekie otoczenie powinien być użyty wektor $R_{F(2)}$ a nie R_F (rys. 4.48).

Wektory R_L i R_F obliczane są według następujących zależności [14]:

$$R_L = L - 2(L \cdot N)N, \text{ gdzie } N \text{ to wektor normalny powierzchni,}$$

$$R_F = -kN + \frac{c_2}{c_1}(L - (L \cdot N)N), \text{ gdzie } k = \sqrt{1 - \left(\frac{c_2}{c_1}\right)^2 (1 - (L \cdot N)^2)}.$$

Współczynniki c_1 i c_2 oznaczają odpowiednio indeksy refrakcji ośrodka na zewnątrz i wewnątrz obiektu. Jeśli $\left(\frac{c_2}{c_1}\right)^2 (1-(L\cdot N)^2) > 1$ to światło nie załamuje się. Operacje odbicia i załamania wektora są standardowo wbudowane w język HLSL.

Podczas obliczania ostatecznego koloru piksela, kolory obu składowych muszą być ze sobą połączone. Służy do tego współczynnik Fresnela f [15], który określa ile procent światła docierającego do obserwatora pochodzi ze składowej odbitej od powierzchni obiektu. Im większy kąt, pod którym obserwator patrzy na powierzchnię, tym więcej wkładu do koloru ma składowa odbita a mniej składowa załamana. Obliczenie dokładnego współczynnika Fresnela jest bardzo kosztowne, dlatego trzeba użyć aproksymacji. Jedno z dobrych przybliżeń jest opisane w pracy [16], ma ono następującą postać:

$$f = R + (1 - R)(1 + L \cdot N)^5, \text{ gdzie } R = \frac{(1 - \frac{c_2}{c_1})^2}{(1 + \frac{c_2}{c_1})^2}.$$

Ostateczny kolor piksela dany jest wzorem:

$$K = fK_R + fK_F,$$

gdzie K_R i K_F oznaczają odpowiednio kolory pochodzące ze składowej odbitej i załamanej.

Wykorzystanie tej techniki niewiele różni się od zwykłego oświetlania obiektu. Oprócz wektora odbitego R_L, w mikroprogramie dla pikseli jest obliczany jest również wektor załamania R_F, oba służą jako współrzędne dla tektury kubicznej z mapą oświetlenia. Po uwzględnieniu współczynnika Fresnela kolor wynikowy jest wpisywany do bufora.

W katalogu /programy/4_1_3 znajduje się program wykorzystujący opisaną w tym podrozdziale technikę (rys. 4.49). Użytkownik może przełączyć program na tryb rysowania wyłącznie składowej odbitej, rozproszonej lub obu jednocześnie. Wykorzystane mikroprogramy wraz z komentarzami znajdują się w pliku z rozszerzeniem *.fx.

Rysunek 4.49: Przykład wykorzystania efektu odbicia i załamania światła. Obie składowe połączone współczynnikiem Fresnela przedstawia dolna części rysunku.

4.2. Cienie

Dynamicznie obliczane cienie należą do najtrudniejszych do wykonania, ale równocześnie najbardziej pożądanych efektów. Nadają scenie realistycznego wyglądu i klimatyczności. Architektura układów graficznych pozwala na symulację wyłącznie świateł kierunkowych oraz punktowych, co implikuje ostre krawędzie cieni. Stosowane są metody ich wygładzania, ale nie ma to nic wspólnego z rozmyciem spowodowanym geometrią źródeł światła.

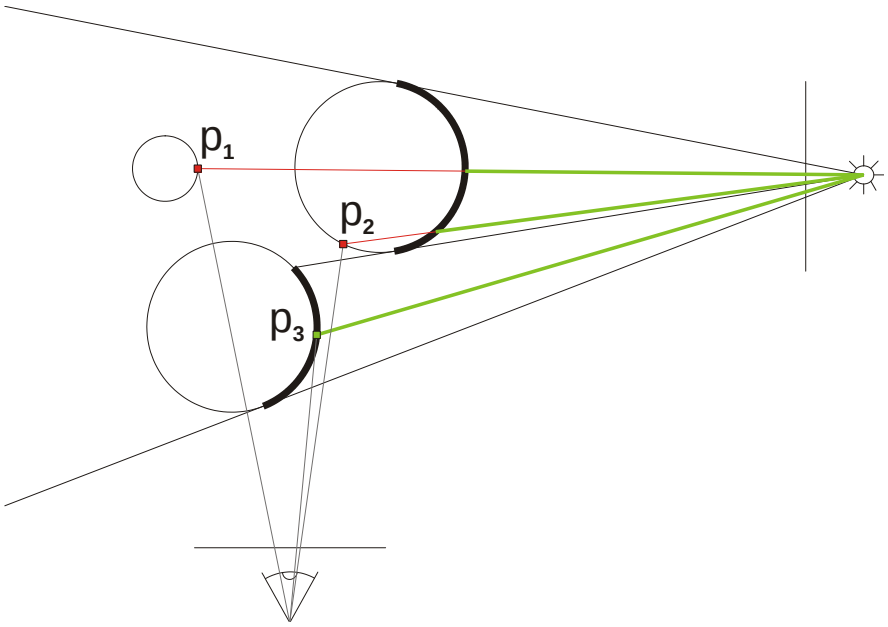
Powszechnie stosowane są dwa sposoby wyznaczania cieni, oba są w pełni realizowane przez akceleratory. Każdy z nich posiada poważne wady i nie nadaje się do wszystkich rodzajów scen. Oba różnią się od siebie konstrukcją oraz wyglądem wygenerowanych cieni. Pierwsza metoda wykorzystuje tzw. mapę cieni i jest nieco szybsza, ale za to mniej dokładna od drugiej, opartej na geometrycznych bryłach cieni.

Wady obu technik wynikają bezpośrednio z ograniczonych zasobów i niedostatecznej mocy obliczeniowej karty. Dla przykładu, mapa cienia bardzo dużej rozdzielczości jest stosowana w profesjonalnych systemach generowania obrazu, takich jak Renderman. Wysoka rozdzielczość gwarantuje bardzo dobrą jakość obrazu, ale tego typu efekt jest na razie nieosiągalny w systemach czasu rzeczywistego. Rozmyte krawędzie cieni można bardzo łatwo uzyskać stosując kilkadziesiąt punktowych źródeł światła, imitujących jedno źródło w kształcie na przykład kuli. Być może przyszłej generacji karty będą w stanie podołać takiemu wyzwaniu.

4.2.1. Mapa cienia

Metoda mapy cienia jest realizowana w dwóch przebiegach. Pierwszy z nich rysuje całą scenę z perspektywy źródła światła, wypełniając bufor koloru wartościami odległości od światła rysowanych punktów. W ten sposób powstaje mapa głębokości, w której są zapisane odległości najbliższych światłu punktów sceny w danym kierunku. Bufor koloru ma kształt prostokąta, więc obszar oświetlany będzie musiał mieć postać stożka lub prostopadłościanu, w zależności od przekształcenia rzutującego użytego podczas rysowania.

W drugim etapie scena jest już rysowana z perspektywy obserwatora. Dla każdego generowanego piksela, współrzędne punktu na trójkącie są tak przekształcane, aby porównać jego odległość od światła. Porównanie jest wykonywane przy pomocy wartości we wcześniej wygenerowanej mapie głębokości (mapy cienia). Jeżeli odległość punktu od światła jest mniejsza lub równa niż ta zapisana w mapie cienia, to punkt nie jest w cieniu. W przeciwnym wypadku punkt leży w cieniu i jest rysowany wyłącznie kolorem globalnego oświetlenia (rys. 4.50).



Rysunek 4.50: Mechanizm działania metody mapy cienia. Punkty p_1 i p_2 leżą w cieniu.

Niech macierz M_O reprezentuje przekształcenie z układu sceny do układu obserwatora, wraz z przekształceniem rzutującym. Macierz M_L również ma podobną postać, ale z punktu widzenia światła. Oś Z układu źródła światła pokrywa się z kierunkiem jego padania na scenę. W pierwszej kolejności, obiekty sceny są rysowane przy użyciu macierzy M_L. Program dla wierzchołków, jako jeden z atrybutów wyjściowych przekazuje współrzędną z oraz w do mikroprogramu dla pikseli. Współrzędne te są wynikiem pomnożenia współrzędnych wierzchołka przez macierz M_L. Po interpolacji obu tych atrybutów na powierzchni trójkąta przez układ, w mikroprogramie dla pikseli wyznaczana jest wartość odległości punktu poprzez podzielenie z przez w: $z' = \frac{z}{w}$. Należy zauważyć, że równocześnie tą samą operację wykonał niejawnie w międzyczasie układ, w celu wyznaczenia widoczności w buforze-Z oraz pozycji we współrzędnych kanonicznych:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x/w \\ y/w \\ z/w \end{pmatrix}.$$

Oprócz tego, układ przekształcił współrzędne kanoniczne do układu bufora koloru, stosując zależność (3.15):

$$\begin{bmatrix} \bar{x} \\ \bar{y} \end{bmatrix} = \begin{bmatrix} \frac{w_o}{2} & 0 & 0 \\ 0 & -\frac{h_o}{2} & 0 \end{bmatrix} \begin{bmatrix} x' \\ y' \end{bmatrix} + \begin{bmatrix} \frac{w_o}{2} + \bar{x}_o \\ \frac{h_o}{2} + \bar{y}_o \end{bmatrix} \tag{4.35}$$

Mikroprogram wpisuje do bufora koloru pod współrzędnymi $(\bar{x}, \bar{y})$ wartość z'. Jest to specjalnie utworzony bufor koloru, który może być później wykorzystany jako tekstura. Jego format jest z reguły jednokomponentowy i o jak największej dokładności, na przykład 32 bitowej liczby rzeczywistej na piksel. Dokładnie tą samą operację przeprowadził układ w buforze-Z. Niestety, w API DirectX nie ma możliwości skopiowania bufora-Z do tekstury w celu późniejszych operacji, więc praktycznie te same dane są wpisywane dwukrotnie. Oczywiście to bufor-Z wykonuje testy widoczności, w mikroprogramie nie są wykonywane żadne porównania. W ten sposób stworzona została mapa cienia.

Następnym krokiem jest narysowanie sceny przy użyciu macierzy M_O z punktu widzenia obserwatora. Przed wykonaniem tego kroku, bufor koloru jest ustawiany na normalny a mapa cienia jako tekstura zostaje podłączona do rejestru próbkującego tekstury.

Mikroprogram dla wierzchołków wykonuje najpierw wszystkie operacje potrzebne do narysowania obiektów, czyli mnoży współrzędne wierzchołka przez macierz M_O. Następnie generuje drugą kopię współrzędnych, ale tym razem przemnażając współrzędne wierzchołka przez macierz M_L. Wszystkie 4 otrzymane liczby (x_L, y_L, z_L, w_L) umieszcza w atrybutach przeznaczonych do interpolacji (na przykład w rejestrze $TEXCOORD0$).

Mikroprogram dla pikseli otrzymuje przeinterpolowane współrzędne i wykonuje dzielenie przez w_L:

$$\begin{pmatrix} x'_L \\ y'_L \\ z'_L \end{pmatrix} = \begin{pmatrix} x_L/w_L \\ y_L/w_L \\ z_L/w_L \end{pmatrix}.$$

Po tej operacji, wartość z'_L jest już gotowa do porównania, ale nie jest jeszcze znana pozycja tego punktu na mapie cienia. Po przeanalizowaniu drogi, jaką przebył punkt rysowany do mapy cienia, jedyną potrzebną jeszcze operacją jest transformacja współrzędnych kanonicznych (x'_L, y'_L) do układu mapy cienia (4.35). Otrzymaną parę $(\bar{x}_L, \bar{y}_L)$ mikroprogram wykorzystuje jako współrzędne do pobrania wartości odległości z tekstury cienia. Po wykonaniu porównania z z'_L mikroprogram uzyskał informację czy punkt ten leży w cieniu czy nie.

Technika ta ma dwie poważne wady. Mapa cienia ma ograniczoną rozdzielczość i dokładność. Objawia się to poważnymi błędami w wygenerowanym obrazie (rys. 4.51). Cienie mają zębate krawędzie (rozdzielczość) oraz nie przechodzą płynnie przez wszystkie trójkąty w czasie ruchu światła (dokładność).

Zwiększyć dokładność wartości odległości można zapisując do mapy cienia nie wartość $z' = \frac{z}{w}$, ale zwykłą odległość punktu od źródła światła. Wartość tą można uzyskać po pomnożeniu współrzędnych wierzchołka przez macierz M_L pozbawioną przekształceń rzutujących. W ten sposób odległość będzie równomiernie wykorzystywała całą przestrzeń liczb zmiennopozycyjnych.

Zęby na krawędziach zostaną częściowo wyeliminowane, jeśli wykonane zostaną cztery porównania odległości z liczbą z'. Po obliczeniu współrzędnych $(\bar{x}_L, \bar{y}_L)$ porównane będą odległości mapy cienia o współrzędnych $(\bar{x}_L, \bar{y}_L)$, $(\bar{x}_L + 1, \bar{y}_L)$, $(\bar{x}_L, \bar{y}_L + 1)$, oraz $(\bar{x}_L + 1, \bar{y}_L + 1)$. Zerojedynkowe wyniki tych porównań są później uśredniane. W ten sposób cień na krawędziach przechodzi płynniej i zęby nie są aż tak ostre.

Rysunek 4.51: Błędy spowodowane małą rozdzielczością i niedokładnością mapy cienia.

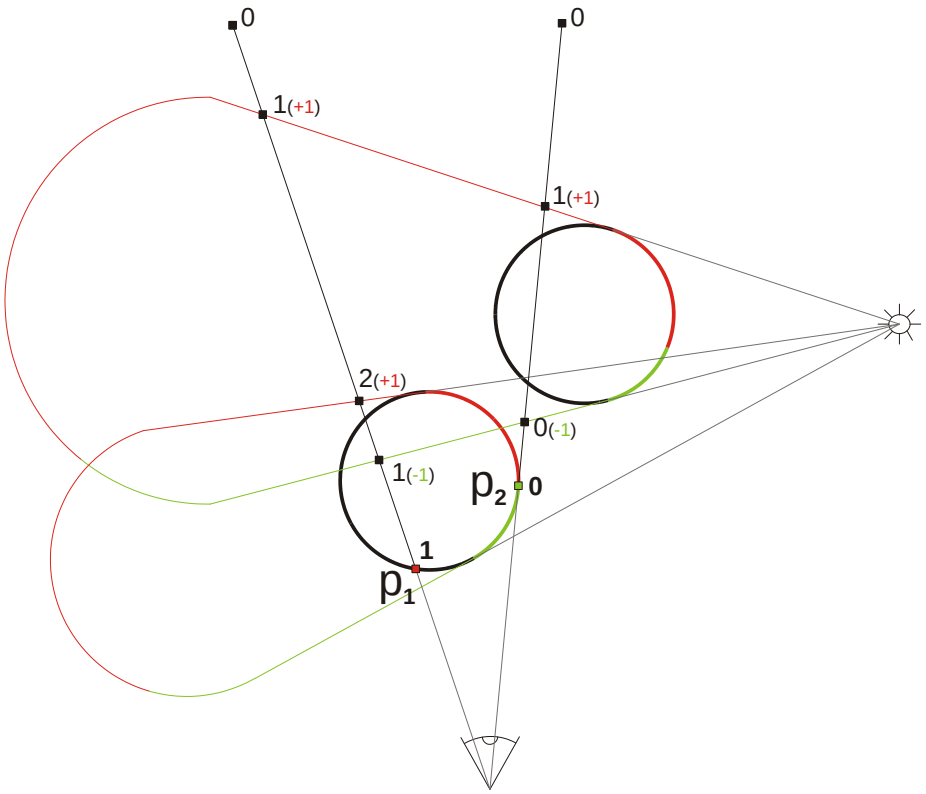
Program znajdujący się w katalogu /programy/4_2_1 demonstruje działanie metody mapy cienia (rys. 4.52). Wraz z nim w pliku z rozszerzeniem *.fx są umieszczone mikroprogramy z komentarzami.

Rysunek 4.52: Po lewej stronie scena wygenerowana przy pomocy mapy cienia. Po prawej, normalnie oświetlona rzeźba.

4.2.2. Cienie wolumetryczne

Metoda cieni wolumetrycznych opiera się na własnościach tzw. brył cienia. Bryła cienia to wycinek bryły podobnej do ściętego stożka. Jego mniejsza podstawa składa się z odwróconych przodem do źródła światła ścian obiektu. Ściany boczne są oparte na wierzchołkach brzegowych podstawy i rozciągnięte w kierunku padania światła (rys. 4.53). Każdy punkt, który znajduje się we wnętrzu bryły cienia znajduje się w cieniu obiektu. Dla uproszczenia algorytmu zakłada się, że każdy obiekt jest bryłą zamkniętą, dlatego środek obiektu można również uznać za obszar znajdujący się w cieniu.

W celu sprawdzenia czy rysowany punkt leży w cieniu, należy poprowadzić półprostą biegnącą od rysowanego punktu do nieskończoności, w kierunku zgodnym z kierunkiem patrzenia obserwatora (rys. 4.53). Zaczynając od części półprostej znajdującej się w nieskończoności, należy poruszać się w kierunku rysowanego aktualnie punktu, obliczając liczbę ścian brył cienia przecinanych przez tą półprostą. Jeśli różnica liczby ścian zwróconych tyłem do obserwatora i zwróconych przodem do niego jest większa od zera, to punkt jest w cieniu.



Rysunek 4.53: Idea działania metody cieni wolumetrycznych. Punkt p_1 jest w cieniu a punkt p_2 nie.

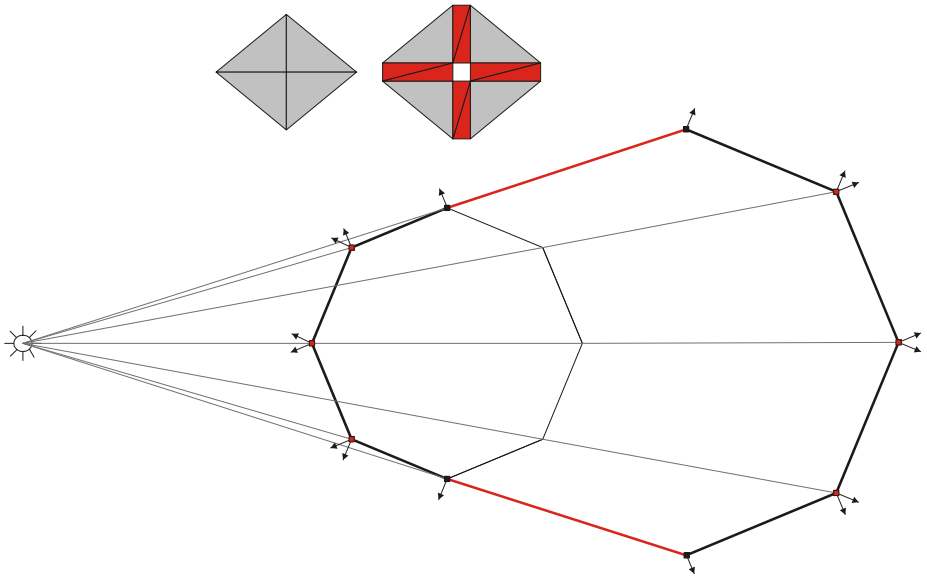
Obiekty muszą być zamknięte, ale nie muszą być wypukłe. W takim przypadku, każdy obiekt będzie posiadał więcej niż jedną stożkowatą bryłę cienia.

Do poprawnego funkcjonowania algorytmu, niezbędne jest zamknięcie bryły cienia z obu stron. Pierwszą stronę zamykają ściany obiektu zwrócone przodem do źródła światła. Drugą może zamykać dowolny kształt, ale najczęściej wykorzystuje się do tego wysunięte na odpowiednio dużą odległość te ściany obiektu, które są zwrócone tyłem do źródła światła.

Na potrzeby sprzętowej realizacji tego pomysłu, została wymyślona sprytna metoda konstrukcji brył cienia. W pierwszej kolejności wszystkie modele są kopiowane. Każda krawędź kopii obiektu jest rozspajana i łączona przez dwa trójkąty o zerowym polu (rys. 4.54). Każdemu wierzchołkowi jest przyporządkowywany wektor normalny oryginalnego trójkąta, którego częścią jest ten wierzchołek. W takiej postaci obie wersje obiektu są zapisywane w pamięci.

W celu obliczenia kształtu bryły cienia dla konkretnej pozycji światła, wierzchołki wszystkich tych ścian, które są zwrócone tyłem do źródła światła, są wysuwane na znaczną odległość. Aby sprawdzić, które to są wierzchołki, należy wykonać iloczyn skalarny wektora kierunkowego światła z wektorem normalnym zapisanym w wierzchołku. W ten sposób zostaną wysunięte tylko niezbędne ściany a obiekt nie zostanie rozspojony. W przypadku, kiedy oba sąsiadujące ze sobą trójkąty zostaną wysunięte lub oba nie zostaną wysunięte, dwa spajające je na krawędzi trójkąty nadal będą miały

zerowe pole. Pozostałe trójkąty spajające będą stanowiły ściany boczne bryły cienia (czerwone trójkąty na rys. 4.54).



Rysunek 4.54: Wierzchołki bryły cienia obrócone tyłem do światła zostają wysunięte.

Podczas generowania cieni przez układ graficzny, intensywnie wykorzystywany jest bufor zliczania. Jest on elementem niezbędnym do narysowania cieni metodą wolumetryczną.

Proces nakładania cieni składa się z trzech etapów. W pierwszym, scena jest rysowana przy użyciu wyłącznie globalnej składowej oświetlenia. Składowa rozproszona oraz zwierciadlana będą narysowane później. Po wykonaniu tego kroku, w buforze koloru znajdują się obiekty w takim kolorze, w jakim byłyby w cieniu. Bufor-Z został wypełniony i zawiera odległości najbliższych obserwatorowi punktów, w każdym z kierunków wyznaczonych przez pozycje pikseli.

W drugim etapie rysowane są wyłącznie bryły cienia. Wysuwaniem wierzchołków zajmuje się całkowicie układ graficzny. W mikroprogramie dla wierzchołków jest przeprowadzany test, który daje liczbę 0 lub 1, w zależności od wartości iloczynu skalarnego wektora normalnego i kierunkowego światła. Każdy wierzchołek jest wysuwany o pewną ustaloną odległość pomnożoną przez wynik testu. W ten sposób wysunięte zostaną wierzchołki tylko tych ścian, które są zwrócone w kierunku padania światła.

Bryły cienia nie są rysowane do bufora koloru, ale wyłącznie do bufora zliczania. Zapisywanie do bufora koloru zostaje wcześniej zabronione. Bufor-Z jest wykorzystywany tylko do odczytu i rysowane są wyłącznie te punkty, które leżą nie bliżej niż te narysowane w poprzednim etapie. Dzieje się tak, gdyż algorytm nie potrzebuje sprawdzać zacienienia punktów leżących przed obiektami sceny, tym samym nie są istotne informacje o ścianach brył cienia znajdujących się przed nimi.

Test kontrolujący bufor zliczania jest ustawiany na zwracanie zawsze prawdy, więc każdy narysowany piksel będzie zmieniał wartości w buforze. Jeżeli ściana bryły cienia

jest zwrócona tyłem do obserwatora, to wartość dla piksela w buforze zliczania jest zwiększana o jeden. Jeśli ściana jest zwrócona przodem, to wartość w buforze jest zmniejszana o jeden. Po wykonaniu tego kroku, w każdym pikselu bufora zliczania znajdzie się wspomniana wcześniej różnica liczb ścian. Jest to dokładne odwzorowanie procesu pokazanego na rys. 4.53.

Przed rozpoczęciem trzeciego etapu, w buforze zliczania każdemu pikselowi jest przyporządkowana liczba, określająca czy znajduje się on w cieniu. Jeżeli wartość ta jest równa zero, to punkt powinien być normalnie oświetlony. Pozostaje tylko narysować całą scenę, dodając do bufora koloru składową rozproszoną oraz zwierciadlaną. Test bufora zliczania jest ustawiany tak, aby zwracać prawdę wyłącznie wtedy, gdy wartość znajdująca się w nim jest równa zero. W przeciwnym razie piksel nie jest rysowany. Tym sposobem do bufora koloru zostaną dodane pozostałe składowe jedynie wtedy, kiedy rysowany punkt nie znajduje się w cieniu.

Opisana powyżej metoda generowania cieni jest pozbawiona wad jej poprzedniczki. Cienie są zawsze doskonałej jakości, ale ich krawędzie mogą wydawać się bardziej ostre. Powodem jest to, że o stopniu zacienienia piksela decyduje tylko jedna próbka, więc jest to wartość zerojedynkowa. Nie jest możliwe jakiekolwiek filtrowanie krawędzi (rys. 4.55).

Często ze względu na duże skomplikowanie sceny, cienie wolumetryczne są dużo wolniejsze od tych wygenerowanych przy pomocy mapy cienia. Bryły cienia są rysowane jedna na drugiej, przez co ich ilość i rozpiętość ma decydujące znaczenie dla prędkości działania programu.

Rysunek 4.55: Metoda cieni wolumetrycznych generuje bardzo ostre krawędzie cieni.

Program demonstrujący technikę cieni wolumetrycznych znajduje się w katalogu /programy/4_2_2. Źródła mikroprogramów dla układu graficznego są w pliku z rozszerzeniem *.fx w tym samym katalogu.

4.3. Przetwarzanie obrazu

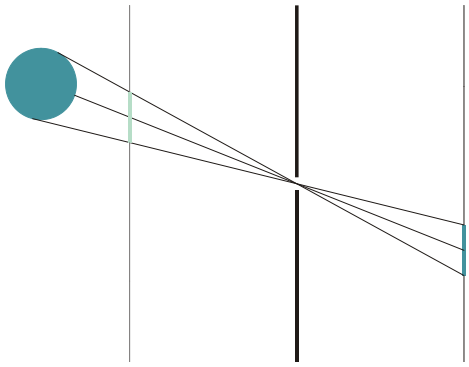
W tym podrozdziale opisane są dwa wybrane procesy przetwarzania obrazu. Operują one na danych znajdujących się w buforach już po wygenerowaniu całej sceny. Do niedawna wszelkiego rodzaju filtry pełnoekranowe były niemożliwe do wykonania, głównie ze względu na bardzo małą prędkość układów graficznych. Nawet najprostszy filtr pełnoekranowy wymaga ogromnej liczby operacji pobierania danych na każdy przetwarzany piksel obrazu.

Największy problem pozostał jednak aktualny do dziś. W czasie obliczania koloru nie ma możliwości korzystania z informacji o sąsiednich pikselach. Jest to spowodowane wielopotokową architekturą, która przyspieszając wielokrotnie działanie układu, równocześnie utrudnia wykonanie nawet najprostszego rozmycia obrazu.

Częściowym rozwiązaniem jest rysowanie sceny do tekstury i wygenerowanie ostatecznego obrazu w dodatkowym przebiegu. Takie podejście pozwala co prawda korzystać z danych o kilku pikselach jednocześnie, ale szybko okazuje się, że układy nie są w stanie przetworzyć takiej ilości danych w rozsądnym czasie. Dopiero akceleratory z najwyższej półki mogą pochwalić się na tyle krótkim czasem generowania jednej klatki obrazu, aby możliwe stało się zaadoptowanie tego typu filtrów w poważnych przedsięwzięciach.

4.3.1. Głębia ostrości

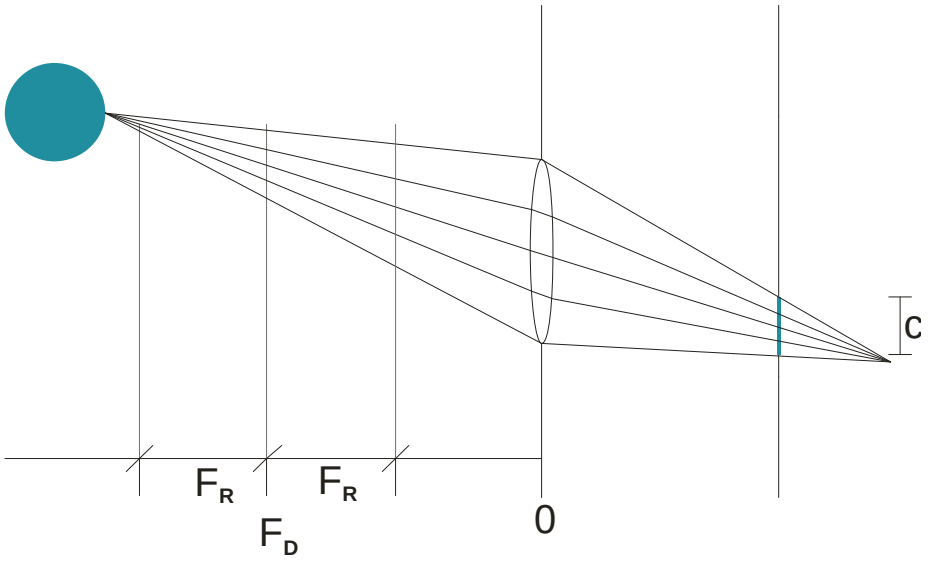
W grafice komputerowej najczęściej stosuje się wyidealizowany model układu optycznego. Promienie światła przechodzą przez nieskończenie mały otwór a odwrócony obraz powstaje na rzutni po drugiej stronie. Dla uproszczenia, na rysunkach rzutnię umieszcza się w tej samej odległości przed sztucznym obiektywem, aby powstały obraz nie był odwrócony. Wszystkie obiekty niezależnie od odległości będą widziane jako ostre, gdyż do każdego punktu na rzutni dochodzi promień światła z dokładnie jednego kierunku (rys. 4.56).



Rysunek 4.56: Wyidealizowany model obiektywu.

W rzeczywistości taki układ optycznie nie istnieje. Rolę otworu stanowią układy soczewek skupiających promienie światła na rzutni (rys. 4.57). Każdy zestaw soczewek jest w stanie odwzorować w akceptowalnej ostrości tylko skończony zakres odległości,

który jest zwany głębią ostrości. Najostrzejszy będzie obraz obiektu znajdującego się w odległości zwanej ogniskową. W klasycznym aparacie fotograficznym ogniskową zmienia się pokrętłem nastawy ostrości. Im dalej od tego punktu znajduje się obiekt, tym mniej ostry będzie jego obraz na rzutni. Powyższe cechy ma każdy występujący w przyrodzie mechanizm, który odwzorowuje trójwymiarowe obiekty na dwuwymiarowej płaszczyźnie.



Rysunek 4.57: Układ optyczny składający się z pojedynczej soczewki skupiającej.

Dokładna symulacja zjawiska głębi ostrości jest bardzo skomplikowana. Wymaga ona śledzenia każdego pojedynczego promienia światła przechodzącego przez układ soczewek, co jest niemożliwe przy użyciu obecnej architektury kart graficznych. Można jednak zastosować bardzo uproszczony model, który generuje obrazy posiadające podstawowe cechy zdjęć wykonywanych prawdziwym obiektywem [19].

Niech F_D oznacza ogniskową (ang. focal distance) a F_R odległość od ogniskowej (ang. focal range), przy której obraz jest już bardzo zamazany. Ze względu na ograniczone możliwości filtrowania przez układ, obiekty znajdujące się dalej będą zamazane w takim samym stopniu jak przy odległości F_R. Stopień rozmycia obrazu w tym modelu jest zależny liniowo od wartości odległości od ogniskowej.

Promienie światła pochodzące z pewnego punktu w przestrzeni, po przejściu przez uproszczony układ optyczny, padają na rzutnię w obszarze koła zwanego kołem rozproszenia. Niech średnica C koła będzie zależna liniowo od odległości punktu od ogniskowej (rys. 4.57):

$$C = sat[|(z - F_D)|/F_R] \cdot C_{\max}, \tag{4.36}$$

gdzie z jest odległością punktu od obserwatora a $C_{\max}$ jest stałą, określającą maksymalne rozmiary koła rozproszenia. Sat jest funkcją, która obcina liczby do zakresu

$[0,1]$ w następujący sposób:

$$sat(x) = \begin{pmatrix} 0, \text{ jeśli } x < 0 \\ 1, \text{ jeśli } x > 1 \\ x \text{ w p.p.} \end{pmatrix} .$$

Rysowanie obrazu z uwzględnieniem głębi ostrości składa się z dwóch etapów. W pierwszym, scena jest rysowana normalnie do bufora koloru, pełniącego również rolę tekstury. Do składowej *alpha* każdego piksela jest wpisywana wartość $sat[|(z-F_D)|/F_R]$. W ten sposób, dla każdego piksela znany jest współczynnik określający stopień rozmazania zależny od odległości od ogniskowej. Dzięki temu, że wartość ta należy do przedziału $[0,1]$, nie trzeba stosować zmiennopozycyjnych formatów tekstury. Niestety 8 bitów na składową nie wystarczy do tego, aby uzyskać zadowalającą dokładność obliczeń. Dla zwiększenia dokładności używa się formatu o 16 bitach na każdą składową koloru.

Drugi etap to rozmazanie obrazu. Bufor koloru użyty w poprzednim kroku jest podłączany jako tekstura. Do normalnego bufora koloru jest rysowany prostokąt, pokrywający w całości jego obszar. Do narożników przyporządkowuje się współrzędne dla tekstury w taki sposób, aby każdy teksel pokrył dokładnie jeden piksel. Jeśli mikroprogram zawierałby wyłącznie pobranie wartości z tekstury i wstawienie jej do rejestru wyjściowego, to ostatecznie wygenerowany obraz wyglądałby identycznie, jak ten z pierwszego etapu. Rysowanie prostokąta pokrywającego cały ekran jest jedyną metodą przeprowadzenia jakiejkolwiek operacji przetwarzania obrazu przy użyciu akceleratora.

Rozmazanie obrazu jest wykonywane poprzez kombinację wagową kolorów pikseli, które leżą w otoczeniu aktualnie przetwarzanego piksela. Otoczenie to jest kołem o promieniu $C_r = \frac{1}{2}C$. Mikroprogram dla pikseli w pierwszej kolejności pobiera z tekstury dane teksela $\tau(u,v)$, którego współrzędne odpowiadają pikselowi obliczanemu przez program. Przy pomocy składowej *alpha* i zależności (4.36) oblicza później średnicę koła rozproszenia w pikselach:

$$C = \tau(u,v)_a \cdot C_{\max}.$$

W rejestrach dla stałych jest umieszczonych n par współrzędnych (du_i, dv_i), które są wartościami wybranymi losowo na obszarze koła o promieniu równym 1. Ostateczny kolor piksela $\tau'(u,v)_k$ dany jest wzorem:

$$\tau'(u,v)_k = \frac{\tau(u,v)_k + \sum_{i=1}^{n} \tau(u + C_r du_i, v + C_r dv_i)_k * \tau(u + C_r du_i, v + C_r dv_i)_a}{1 + \sum_{i=1}^{n} \tau(u + C_r du_i, v + C_r dv_i)_a}. \quad (4.37)$$

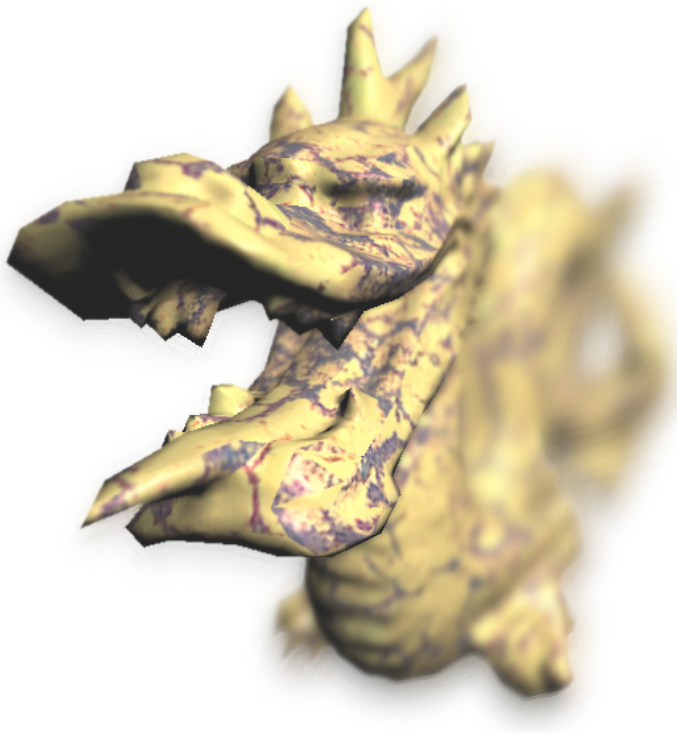
Na prawdziwym zdjęciu wykonanym aparatem fotograficznym część obiektów jest idealnie ostra a pozostałe są rozmazane w taki sposób, że obiekty na ostrym planie nie mają wpływu na kolor rozmazanej części. Powyższa metoda w dużym stopniu spełnia te warunki.

Wzór (4.37) zwróci kolor teksela $\tau(u,v)_k$, jeżeli leży on w pobliżu ogniskowej. Wartość $\tau(u,v)_a$ będzie wtedy bliska zeru, tym samym promień koła rozproszenia będzie bardzo mały. Pozostałe teksele wniosą najwyżej nieznaczne zaburzenia do oryginalnego koloru.

Jeżeli oryginalny teksel leży w znacznej odległości od ogniskowej, współczynnik $\tau(u,v)_a$ będzie bliski liczbie 1. Promień koła rozproszenia będzie duży, ale teksele z ostrego planu i tak nie będą miały wpływu na kolor $\tau'(u,v)_k$, gdyż ich waga $\tau(u + C_r du_i, v + C_r dv_i)_a$ będzie wtedy bliska zeru.

Ze względu na limit liczby instrukcji w mikroprogramie dla pikseli, ilość próbek może sięgnąć najwyżej 12. Na końcowym obrazie będą widoczne niepożądane wzory, które są wynikiem małej liczby próbek uśrednianych w dużym promieniu. Pomóc w tej sytuacji może jedynie większa ilość przebiegów filtra (4.37). Przy każdym powtórzeniu operacji rozmazywania, obrazem źródłowym staje się poprzednio wypełniany bufor. Należy zwrócić uwagę, że wartości wag $\tau(u,v)_a$ w każdym z przebiegów muszą pozostać takie same jak na początku.

Do pracy został dołączony program demonstrujący powyższą metodę (rys. 4.58). Znajduje się on w katalogu /programy/4_3_1 wraz z mikroprogramami (plik z rozszerzeniem *.fx). W programie wykonywane są trzy przebiegi filtra, każdy z innym zestawem 12 współrzędnych dla próbek.

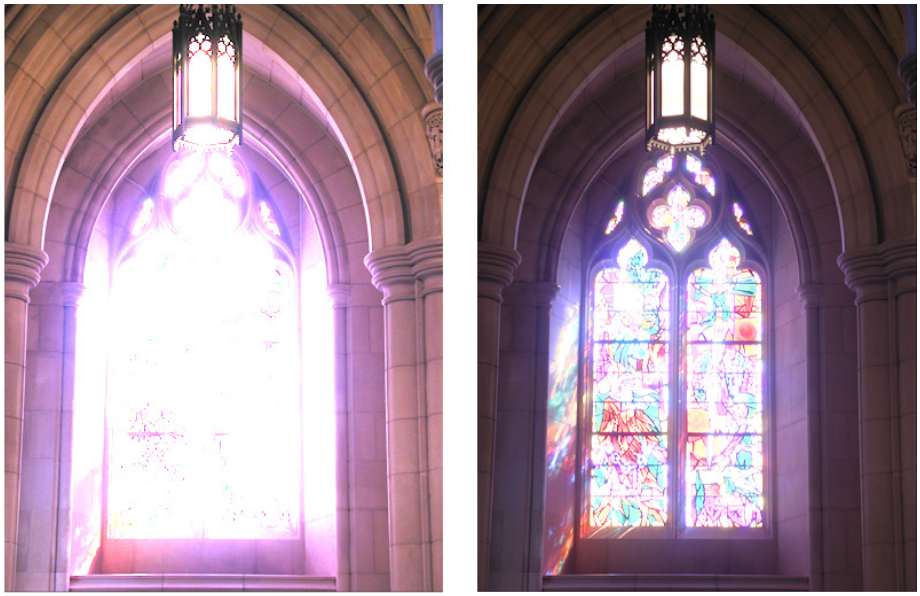


Rysunek 4.58: Obraz z głębią ostrości wygenerowany przy użyciu układu graficznego.

4.3.2. Wyświetlanie obrazów o wysokiej skali jasności

Monitor jest w stanie wyświetlić tylko skończony zakres jasności, które są spotykane w przyrodzie. Wartości składowe koloru RGB odwzorowują jasność kolorów w bardzo małym przedziale. Istnieją jednak formaty danych, które przechowują dane o pełnej skali jasności poszczególnych punktów. Obraz taki można otrzymać poprzez sfotografowanie otoczenia specjalnym aparatem lub złożenie jednego obrazu z kilku zdjęć, wykonanych z różnym czasem naświetlania [20].

Przykładem takiego zdjęcia może być witraż w kościele, na który od zewnątrz padają bezpośrednio promienie słoneczne. Chcąc sfotografować dokładnie witraż, czas ekspozycji będzie musiał być bardzo krótki. W takim przypadku ciemne otoczenie wnętrza kościoła nie będzie widoczne na zdjęciu. Odwrotnie, chcąc sfotografować wnętrze kościoła, witraż będzie tak jasny, że całą jego powierzchnię przykryje biała plama. Powodem tych problemów jest bardzo duża różnica jasności światła docierającego do obserwatora. Światło przenikające przez witraż może być kilkadziesiąt tysięcy razy jaśniejsze niż to występujące we wnętrzu kościoła (rys. 4.59).



Rysunek 4.59: Przykład obrazu o wysokiej skali jasności.

Format obrazu o wysokiej skali jasności (ang. high dynamic range) składa się z danych o barwie punktu i osobnej informacji o jasności. Dysponując takim obrazem nie można go wyświetlić na monitorze tak, aby wszystkie jego szczegóły były widoczne. Potrzebny jest operator, który dokona transformacji całego zakresu jasności występującej na fotografii, do przedziału akceptowalnego przez urządzenia wyświetlające. Istnieje wiele takich operatorów, ale nie wszystkie nadają się do użycia przez układy graficzne, ze względu na prędkość wykonywania przekształceń. W tej pracy został wykorzystany bardzo prosty i szybki operator [21], który daje przyzwoite pod względem jakości wyniki.

Niech L_w oznacza średnią logarytmiczną jasności wszystkich N pikseli obrazu:

$$L_w = \frac{1}{N} \exp(\sum_{\bar{x},\bar{y}} \log[\delta + L(\bar{x},\bar{y})]),$$

gdzie $L(x,y)$ oznacza jasność piksela o współrzędnych $(\bar{x},\bar{y})$ a δ jest bardzo małą liczbą dodatnią, która gwarantuje istnienie logarytmu. W pierwszej kolejności, jasności są liniowo skalowane tak, aby jasność o wartości L_w została sprowadzona do liczby a:

$$L'(x,y) = a\frac{L(x,y)}{L_w}. \tag{4.38}$$

Współczynnik a jest parametrem operatora i kontroluje subiektywną jasność obrazu wynikowego, z reguły jest liczbą mniejszą od 1.

Najczęściej jasności wszystkich punktów sceny mają rozkład nieliniowy, który dzieli punkty na dwie grupy. W pierwszej znajdują się obszary o jasności małej, natomiast w drugiej o jasności nieporównywalnie większej. Operator wykorzystuje ten fakt podczas obliczania ostatecznej jasności pikseli:

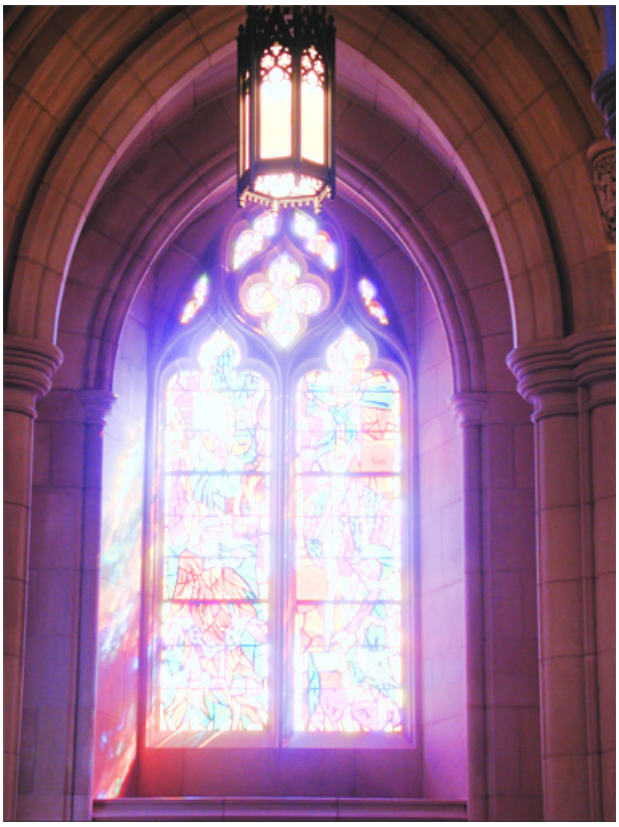
$$\bar{L}(x,y) = \frac{L'(x,y)}{1 + L'(x,y)}. \tag{4.39}$$

Piksele o bardzo wysokiej jasności są znacznie ściemniane a te o małej jasności pozostają praktycznie niezmienione.

Wykorzystanie tego operatora przy użyciu akceleratora nie sprawia większych trudności. W pierwszej kolejności, obraz wejściowy jest zapisywany do tekstury o formacie zmiennopozycyjnym. W drugim przebiegu, bufor koloru jest wypełniany teksturą za pomocą pełnoekranowego prostokąta, tą samą metodą, co w poprzednim podrozdziale. W mikroprogramie stosuje się wyrażenia (4.38) i (4.39) w celu transformacji jasności pikseli.

Model RGB (ang. Red Green Blue) nie nadaje się do zmieniania jasności obrazu. Skalując liniowo każdą ze składowych zwiększy się jasność koloru, ale przy okazji jego barwa może ulegać znacznym zniekształceniom. Z tego powodu lepiej jest używać na przykład modelu HSL (ang. Hue Saturation Lightness). Składowe modelu HSL przechowują osobno informacje o barwie i jasności [22].

Do pracy został dołączony program, który demonstruje użycie opisanego tutaj operatora (rys. 4.60). Wykorzystany został model koloru HSL. Ponieważ obrazy źródłowe są zapisywane w modelu RGB, mikroprogramy dokonują konwersji modeli kolorów w czasie rzeczywistym. Jest to najbardziej czasochłonna część całej operacji. Po wykonaniu skalowania jasności, wartości kolorów są zamieniane z powrotem na model RGB i wyświetlane na ekranie. Program znajduje się w katalogu /programy/4_3_2 wraz z treścią mikroprogramów (plik z rozszerzeniem *.fx).



Rysunek 4.60: Przykład zastosowania operatora skalującego jasności punktów.

Podsumowanie

Głównym motorem rozwoju układów graficznych jest branża rozrywkowa. To ona mobilizuje producentów sprzętu do szybkiego wdrażania nowych technologii. Niektóre z zaprezentowanych w poprzednim rozdziale technik, na przykład głębi ostrości, będą szeroko wykorzystywane w momencie pojawienia się na rynku odpowiednio szybkiego, a zarazem taniego sprzętu. Wszystkie programy dołączone do pracy generują powyżej 30 klatek na sekundę w rozdzielczości 1024 na 768 pikseli, przy użyciu karty wyposażonej w układ ATI Radeon 9800 Pro. Niestety, koszt zakupu takiej karty jest duży i we współczesnych grach komputerowych tego typu efekty są jeszcze rzadko spotykane. Zmieni się to w najbliższej przyszłości.

Mimo znacznego opóźnienia produkcji oprogramowania wykorzystującego bardzo zaawansowane metody generowania obrazu, cały czas opracowywane są nowe techniki, które wymagają jeszcze szybszych akceleratorów. Układy takie są dostępne od razu w sklepach, ale ich cena przekracza zasobność portfela zwykłego konsumenta. Cały ten proces służy do wypromowania określonej technologii i firmy, która ją opracowała.

Opracowywane w tej chwili technologie zmierzają w kierunku coraz szybszego generowania obrazu, przy użyciu obecnie już gotowych algorytmów. Dla przykładu, cienie i wszelkiego rodzaju filtry wymagają ogromnej mocy obliczeniowej jednostek przetwarzających piksele oraz bardzo krótkiego czasu dostępu do pamięci karty. Układ Radeon 9800 Pro posiada 8 jednostek PSU. Zaprezentowane w tym roku na targach Electronic Entertainment Expo układy firmy ATI, mają być wyposażone w aż 48 tego typu jednostek. Ten przykład wyraźnie pokazuje, że obecne technologie są wystarczające do większości zastosowań a producenci skupiają się głównie na mocy obliczeniowej swoich układów.

Architektura mikroprogramów w wersji 2.0 jest w pełni dojrzałą specyfikacją programowania układów graficznych. W czasie powstawania tego opracowania, gotowe były już układy wyposażone w wersję 3.0, w której rozkazy maszynowe wraz z zestawem rejestrów zostały nieco zmodyfikowane. Zmiany te nie są jednak duże, o czym świadczy prawie niezmieniona specyfikacja języka HLSL. Interesującym pomysłem jest wprowadzenie możliwości dostępu do tekstur z poziomu mikroprogramu dla wierzchołków. Pozwoli to na obliczanie pozycji wierzchołków, przy użyciu wcześniej wygenerowanych przez układ tekstur. Nierówności powierzchni symulowane tą techniką, będą widoczne również na konturach obiektów.

Z obecnie dostępnych informacji wynika, że przyszłość układów graficznych leży przede wszystkim w lepszej jakości generowanego obrazu. Rysowanie do buforów w formatach zmiennoprzecinkowych oraz alpha-blending przy ich użyciu, pozwoli na znacznie lepsze odwzorowanie barw i przejść tonalnych. Wraz z coraz szybszymi układami scalonymi powstaną metody generowania miękkich cieni oraz swobodnego filtrowania końcowego obrazu. Wszystkie te elementy stanowią obiecującą wizję przyszłości.

Bibliografia

[1] J.D.Foley, A.van Dam, S.K.Feinerm, J.F.Hughes, R.L.Philips: Wprowadzenie do grafiki komputerowej, Wydawnictwo Naukowo-Techniczne, Warszawa 1995

[2] Microsoft DirectX 9.0 Software Development Kit, Microsoft Corporation, Update (April 2005)

[3] The OpenGL Graphics System: A Specification - Version 2.0, Silicon Graphics, Inc., September 7, 2004

[4] Randy Fernando, Mark Harris, Matthias Wloka, Cyril Zeller: Programming Graphics Hardware, Eurogaphics 2004

[5] Chris Seitz: Evolution of GPUs, Nvidia Corporation, 2004

[6] Projective Geometry, University of Maryland Experimental Geometry Lab, http://www.math.umd.edu/~lidador/Affine/

[7] Jules Bloomenthal, Jon Rokne: Homogeneous Coordinates, Department of Computer Science, The University of Calgary, 1997

[8] Maciej Falski: Przegląd modeli oświetlenia w grafice komputerowej, Uniwersytet Wrocławski, 2004

[9] Real-Time Volume Graphics, Siggraph 2004

[10] ATI Software Development Kit, ATI Technologies Inc., Update (July 2004)

[11] NVIDIA Software Development Kit 8.0, NVIDIA Corporation, 2004

[12] J. F. Blinn: Simulation of Wrinkled Surfaces, Siggraph 1978

[13] Ken Perlin: Making Noise, http://www.noisemachine.com/talk1/index.html, 1999

[14] Ron Goldman: Recursive Ray Tracing, Department of Computer Science, Rice University, 2004

[15] Philip Dutre: Global Illumination Compendium, Cornell University, 2001

[16] Fresnel Reflection Technical Report, NVIDIA Corporation, 2004

[17] Cass Everitt,Mark J. Kilgard: Practical and Robust Stenciled Shadow Volumes

for Hardware-Accelerated Rendering, NVIDIA Corporation, 2002

[18] Mark J. Kilgard: Shadow Mapping with Today's OpenGL Hardware, Game Developers Conference, 2000

[19] Thorsten Scheuermann: Advanced Depth of Field, ATI Research Inc., 2004

[20] Paul E. Debevec, Jitendra Malik: Recovering High Dynamic Range Radiance Maps from Photographs, Siggraph 1997

[21] Erik Reinhard, Michael Stark, Peter Shirley, James Ferwerda: Photographic Tone Reproduction for Digital Images, 2002

[22] Norman Koren: Light and color: an introduction, http://www.normankoren.com