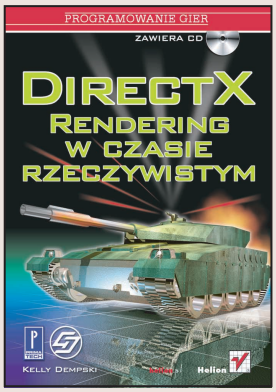
# DirectX. Rendering w czasie rzeczywistym




Autor: Kelly Dempski
Tłumaczenie: Radosław Meryk (wstęp, rozdz. 1 – 25, 37 – 40, dod. A), Krzysztof Jurczyk (rozdz. 26 – 36)
ISBN: 83-7361-120-7
Tytuł oryginału: Real-Time Rendering Tricks and Techniques in DirectX
Format: B5, stron: 688

W ciągu kilku ostatnich lat przetwarzanie grafiki w czasie rzeczywistym osiągnęło stopień najwyższego zaawansowania. Obecnie powszechnie dostępne są procesory przetwarzania grafiki, które dorównują szybkością i możliwościami najdroższym stacjom graficznym sprzed zaledwie kilku lat.

Jeżeli jesteś gotów na pogłębienie swojej wiedzy na temat programowania grafiki, sięgnij po tę książkę. Opisuje ona zaawansowane zagadnienia w taki sposób, że nawet początkujący przyswajają je łatwo. Czytelnicy, którzy będą studiować tę książkę od początku i dokładnie ją przeanalizują, nie powinni mieć problemu z nauczeniem się coraz bardziej skomplikowanych technik. Czytelnicy zaawansowani mogą wykorzystywać tę książkę jako użyteczne źródło informacji, przeskakując od rozdziału do rozdziału, wtedy kiedy mają potrzebę nauczenia się lub dokładniejszego poznania wybranych problemów.

Książka zawiera:

- Powtórzenie wiadomości o wektorach i macierzach
- Omówienie zagadnień związanych z oświetleniem
- Opis potoków graficznych
- Opis konfiguracji środowiska
- Sposoby korzystania z DirectX
- Dokładne omówienie renderingu
- Opis sposobów nakładania tekstur
- Problemy związane z przezroczystością
- Opis technik vertex shader i pixel shader
- Zasady tworzenia cieni przestrzennych
- Sposoby tworzenia animowanych postaci

... i wiele innych zaawansowanych technik, używanych przez profesjonalnych programistów. Towarzyszą jej cenne dodatki na CD (m.in. Microsoft DirectX® 8.1 SDK, wersja demonstracyjna przeglądarki efektów NVidia®, program do zrzutów video – VirtualDub, wersja demonstracyjna programu trueSpace firmy Caligari®)

Niezależnie od Twojego programistycznego doświadczenia, książka ta może posłużyć Ci jako przewodnik pozwalający na osiągnięcie mistrzostwa w wykorzystaniu możliwości programowania grafiki w czasie rzeczywistym.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

# Spis treści

Rozdział 16.

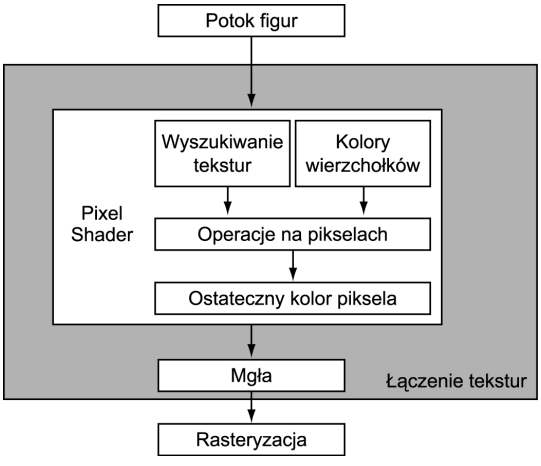
# Mechanizmy pixel shader

Mechanizmy *pixel shader* są analogiczne do mechanizmów vertex shader, z tą różnicą, że operują na pikselach zamiast na wierzchołkach. Po przekształceniach wierzchołków następuje rasteryzacja trójkątów na piksele, które są zapisywane do bufora zapasowego. W poprzednich rozdziałach powiedzieliśmy, w jaki sposób operacje na wierzchołkach, kolorach i teksturach wpływają na kolory uzyskiwane na ekranie. W tym rozdziale przedstawię pojęcia związane z mechanizmami pixel shader. Pojęcia te nabiorą bardziej realnego kształtu w rozdziałach omawiających określone techniki. Ten rozdział służy natomiast jako wstęp do następujących pojęć związanych z mechanizmami pixel shader:

- Różne wersje mechanizmów pixel shader.
- Dane wejściowe i wyjściowe mechanizmów pixel shader.
- Warunkowe odczytywanie tekstur.
- Instrukcje dostępne w mechanizmach pixel shader i łączenie instrukcji w pary.
- Modyfikatory w mechanizmach pixel shader.
- Ograniczenia i ostrzeżenia związane z mechanizmami pixel shader.
- Sprawdzanie obsługi mechanizmów pixel shader.
- Asemblacja i tworzenie mechanizmów pixel shader.
- Prosta aplikacja wykorzystująca mechanizm pixel shader.

## Co to jest pixel shader?

W poprzednich rozdziałach dowiedzieliśmy się, że różne operacje z kolorami mają wpływ na łączenie kolorów tekstury i wierzchołków w czasie rasteryzacji trójkątów. Operacje na kolorach dla fazy tekstury dają programiście niezwykłą kontrolę nad procesem łączenia, ale nie oferują zbyt wielkich możliwości. Mechanizmy pixel shader, podobnie jak mechanizmy vertex shader, pozwalają na o wiele dokładniejszą kontrolę nad sposobem przetwarzania danych przez urządzenie. W przypadku mechanizmów pixel shader dane, o których mowa, to piksele. Shader działa z każdym pikselem, który jest renderowany na ekranie. Zwróćmy uwagę, że nie jest to każdy piksel ekranu, a raczej każdy piksel wchodzący w skład prymitywu renderowanego na ekranie. Na rysunku 16.1 przedstawiono




**Rysunek 16.1.** *Miejsce mechanizmu pixel shader w potoku*

inne spojrzenie na dalsze etapy renderowania pokazane na rysunku 5.1. Jak widzimy, shader ma wpływ na kolorowanie danego prymitywu, ale przetworzony przez shader piksel, zanim trafi na ekran, nadal musi być poddany testowi alfa, głębi i matrycy.

Pixel shader przetwarza każdy piksel renderowanego prymitywu, ale niekoniecznie każdy piksel ekranu czy okna wyjściowego. Oznacza to, że pixel shader ma wpływ na wygląd określonego trójkąta, czy też obiektu. Pixel shader zajmuje miejsce stanów łączenia tekstury i pozwala na dokładniejsze zarządzanie przezroczystością i kolorem dowolnego wykreślanego obiektu. Jak przekonamy się w dalszych rozdziałach, ma to swoje implikacje dotyczące oświetlenia, cieniowania oraz wielu innych operacji na kolorach.

Prawdopodobnie jedną z największych zalet mechanizmów pixel shader jest to, że upraszczają one przedstawianie złożonych operacji łączenia tekstur. Stany faz tekstur wymagają ustawiania operacji na teksturach, argumentów, współczynników łączenia oraz kilku innych stanów. Stany te pozostają „w mocy", dopóki nie zostaną jawnie zmienione. W wyniku tego składnia jest czasami niezgrabna, a czasami trudno kontrolować wszystkie ustawienia. Mechanizmy pixel shader zamieniają wywołania metody `SetTextureStageState` serią stosunkowo prostych instrukcji arytmetycznych z przejrzyście zdefiniowanymi argumentami. Kiedy już zapoznamy się z mechanizmami pixel shader, z pewnością okażą się one bardzo przydatne.

# Wersje mechanizmów pixel shader

Specyfikacja mechanizmów pixel shader zmienia się bardziej gwałtownie oraz w bardziej istotny sposób niż specyfikacja mechanizmów vertex shader. Istnieje wersja 1.0, ale zamiast niej powinniśmy wykorzystywać wersję 1.1. Obecnie prawie wszystkie rodzaje sprzętu obsługujące mechanizm pixel shader obsługują wersję 1.1, ale istnieją implementacje sprzętowe dla wersji 1.2, 1.3 oraz 1.4. Późniejsze wersje w wielu aspektach rozszerzają możliwości. Pomimo to techniki pokazane w dalszych rozdziałach będą ograniczały się do wersji 1.1. W tym rozdziale opiszę właściwości wszystkich wersji i zaznaczę te właściwości, które są dostępne tylko w wersjach późniejszych. Jeżeli nie zaznaczę inaczej, to można założyć, że określony opis dotyczy wszystkich wersji.

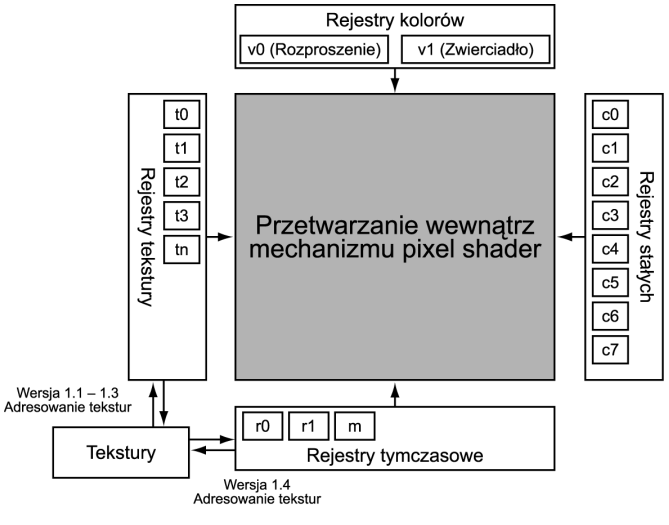
**Mechanizmy pixel shader, a mechanizmy vertex shader**

Wersja 1.1. mechanizmu pixel shader jest dość ograniczona w porównaniu z wersją 1.1 mechanizmu vertex shader. Późniejsze wersje zawierają bardziej zaawansowane funkcje odczytywania tekstur oraz oferujące większe możliwości funkcje porównań. Zakres możliwości mechanizmów pixel shader stopniowo zbliża się do zakresu oferowanego przez mechanizmy vertex shader.

# Wejścia, wyjścia oraz operacje realizowane przez mechanizmy pixel shader

Tak jak w przypadku mechanizmów vertex shader, działania wykonywane przez mechanizm pixel shader są uzależnione od zbioru danych wejściowych, zbioru instrukcji oraz rejestrów służących do przechowywania wyników. Architekturę mechanizmu pixel shader pokazano na rysunku 16.2.

**Rysunek 16.2.** *Architektura mechanizmu pixel shader*




Poniżej umieszczono opis każdego z komponentów mechanizmu pixel shader. W tym rozdziale umieszczę tylko zwięzły opis. Pozostałe informacje znajdą się w rozdziałach następnych.

## Rejestry kolorów

Rejestry kolorów — `v0` oraz `v1` to najprostsze rejestry wejściowe. Każdy z nich posiada cztery wartości składowych koloru, które odpowiadają rejestrom wyjściowym `oD0` oraz `oD1` mechanizmu vertex shader. Można je wykorzystać do przekazywania danych o kolorach, współrzędnych tekstury lub nawet wektorach, które mają wpływ na łączenie lub kolorowanie wynikowych pikseli.

## Rejestry tymczasowe i wyjściowe

Podobnie jak w przypadku rejestrów mechanizmu vertex shader, rejestry tymczasowe służą do zapisywania tymczasowych danych wykorzystywanych przez kolejne instrukcje shadera. Istnieją cztery składowe wartości koloru zapisane jako liczby zmiennoprzecinkowe. Inaczej niż w mechanizmach vertex shader, wartość `r0` oznacza ostateczną wartość koloru uzyskaną w wyniku działania shadera. Wartość `r0` możemy wykorzystać także jako rejestr tymczasowy, ale wartość znajdująca się w tym rejestrze po zakończeniu działania shadera będzie przekazana do dalszych etapów potoku. Pamiętajmy też, że wartości rejestrów tymczasowych są wewnątrz shadera interpretowane jako wartości zmiennoprzecinkowe, ale wartość `r0` przed zakończeniem działania shadera jest przeliczana na wartość z zakresu od 0,0 do 1,0.

## Rejestry stałych

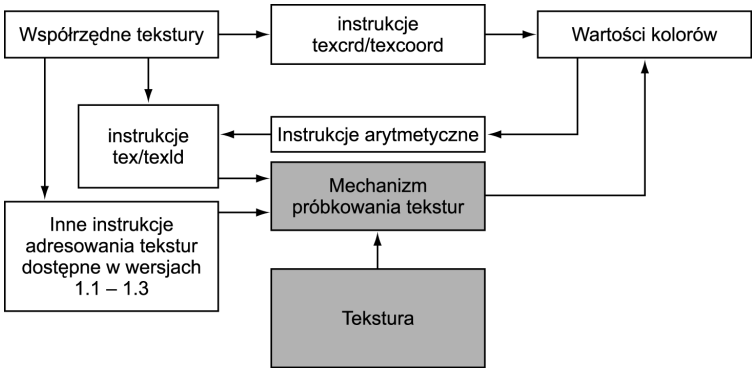
Stałe mechanizmu pixel shader mają dokładnie taką samą postać i działanie jak stałe mechanizmów vertex shader poza tym, że ich wartości powinny mieścić się w zakresie od −1,0 do 1,0. W wielu przypadkach spotkamy się z definiowaniem stałych wewnątrz mechanizmu pixel shader. W większości technik związanych z mechanizmami pixel shader wartości stałe są rzeczywiście stałe, co oznacza, że nigdy się nie zmieniają. Zmodyfikowane wartości można przekazać do mechanizmu pixel shader za pomocą mechanizmu vertex shader. Jeśli jest taka potrzeba, to wartości stałych można także przekazać do mechanizmu pixel shader za pomocą wywołań API.

## Rejestry tekstur

Rejestry tekstur dostarczają do mechanizmu pixel shader informacji o teksturach. Z technicznego punktu widzenia rejestry tekstur zawierają współrzędne tekstur, ale w większości przypadków informacje te są natychmiast zamieniane na dane o kolorach w miarę pobierania danych o teksturze. W shaderach wersji od 1.1 do 1.4 rejestry tekstur mogą być odczytywane jako współrzędne, a następnie zapisywane danymi o kolorach. W wersji 1.4. mogą być jedynie odczytywane i są wykorzystywane jako parametry służące do załadowania danych o kolorach do innych rejestrów. Dokładne przeznaczenie tych rejestrów stanie się bardziej jasne po omówieniu instrukcji dostępnych dla shadera. Liczba rejestrów tekstur zależy od liczby faz tekstur.

# Warunkowe odczytywanie tekstur

W większości przypadków rejestry tekstur wykorzystuje się do próbkowania danych na podstawie współrzędnych tekstury przekazanych z mechanizmu vertex shader. Celem działania niektórych instrukcji pixel shadera jest jednak modyfikowanie współrzędnych tekstury wewnątrz pixel shadera, przed odczytaniem rzeczywistej wartości tekstury. Taką operację nazywa się warunkowym odczytywaniem tekstury, ponieważ współrzędne wykorzystywane do pobierania wartości tekstury zależą od pewnych wcześniejszych operacji wykonywanych przez pixel shader, a nie tylko od czynników zewnętrznych. Graficzną reprezentację operacji warunkowego odczytu tekstury pokazano na rysunku 16.3.




**Rysunek 16.3.** *Zwykłe i warunkowe odczytywanie tekstur*

Operacja warunkowego odczytu tekstur pozwala programiście na tworzenie szeregu faz tekstur, które poszukują wartości dla każdej kolejnej fazy. Przykładowo, istnieją techniki, w których vertex shader dostarcza współrzędnych tekstury, która z kolei zawiera wartości koloru wykorzystywane jako współrzędne tekstury w innej teksturze. Przy prawidłowym wykorzystaniu operacje warunkowego odczytywania tekstur mogą stworzyć podstawę do implementacji funkcji matematycznych, które oferują znacznie większe możliwości wykonywania operacji, niż w przypadku zastosowania stanów faz tekstur. W następnym podrozdziale poświęconym instrukcjom opisano to w sposób bardziej szczegółowy.

# Instrukcje dostępne w mechanizmach pixel shader

Instrukcje pixel shadera działają na wartościach rejestrów wejściowych i tymczasowych w podobny sposób jak instrukcje vertex shadera. Istnieje jednak kilka istotnych różnic. Pierwsza różnica polega na tym, że pixel shadery obsługują znacznie mniejszą liczbę instrukcji. Ogranicza to możliwości shaderów, ale w praktyce jest sensowne, ze względu na częstotliwość zastosowania mechanizmu pixel shader. Pomijając możliwości sprzętu, pisanie długich shaderów, które miałyby przetwarzać miliony pikseli w pojedynczej ramce, wydaje się bezcelowe. Inna różnica polega na zastosowaniu modyfikatorów instrukcji. Modyfikatory instrukcji stanowią cenny dodatek do zestawu instrukcji shadera. Modyfikatory omówimy w następnym podrozdziale.

Niewygodne jest to, że różne wersje mechanizmów pixel shader w pewnym stopniu obsługują różny zestaw instrukcji. W objaśnieniach zamieszczonych poniżej dla każdej instrukcji wymieniam zestaw wersji shaderów, które ją obsługują. W przypadku braku takiej listy będziemy mogli przyjąć, że instrukcja jest obsługiwana we wszystkich dostępnych obecnie wersjach.

Istnieją trzy generalne grupy instrukcji. Nazwałem je instrukcjami konfiguracji, instrukcjami arytmetycznymi oraz instrukcjami adresowania tekstur. Kategoria instrukcji konfiguracyjnych stanowi w pewnym sensie rodzaj kategorii dodatkowej dla trzech instrukcji, które nie pasują do pozostałych kategorii. Instrukcje konfiguracyjne wymieniono w tabeli 16.1.

**Tabela 16.1.** *Instrukcje konfiguracyjne dostępne w pixel shaderach*

| Instrukcja | Opis |
|---|---|
| `ps` | Instrukcja `ps` informuje shader, której wersji shadera używamy. Obecnie najczęściej wykorzystywana wersja to 1.1, ale wkrótce zapewne bardziej popularne będą wersje nowsze. |
| `def` | Instrukcja `def` definiuje wartość stałej jako wektor składający się z czterech składowych. Instrukcja ta przydaje się do ustawiania wartości, które nie zmieniają się w ciągu całego czasu istnienia shadera. |
| `phase` | Ta instrukcja jest unikatowa dla wersji 1.4. Wersja 1.4 pozwala na rozdzielenie pixel shadera na dwie różne fazy działania, co w rezultacie pozwala na podwojenie liczby instrukcji dozwolonych do wykorzystania wewnątrz mechanizmu pixel shader. Wartości kolorów ustawione podczas jednej z faz shadera przechodzą do fazy drugiej, ale wartości kanału alfa nie muszą być takie same. Jeżeli shader nie zawiera instrukcji `phase`, urządzenie uruchamia shader, tak jakby znajdował się w ostatniej fazie (tylko wersja 1.4). |

W tabeli 16.2 zestawiono instrukcje arytmetyczne. Większość spośród tych instrukcji jest przynajmniej częściowo obsługiwana we wszystkich wersjach shaderów. Należy zachować ostrożność, gdyż niektóre z nich albo nie są obsługiwane, albo zużywają więcej niż jeden cykl.

## Łączenie instrukcji w pary

Pixel shader przetwarza dane opisu koloru oraz kanału alfa w dwóch różnych potokach. Dzięki temu istnieje możliwość zdefiniowania dwóch zupełnie różnych instrukcji dla dwóch potoków, które wykonują się jednocześnie. Przykładowo, możemy zastosować instrukcję `dp3` dla danych RGB, ale dane kanału alfa możemy tylko przesłać z jednego rejestru do drugiego. Można to zrobić w następujący sposób:

```
dp3 r2.rgb, r1, c1
+ mov r0.a, c2
```

Istnieją pewne ograniczenia dotyczące tego, które instrukcje wolno łączyć w pary. Według niepisanej reguły, nie należy grupować instrukcji o największych ograniczeniach. Bez przeszkód można łączyć w pary proste operacje arytmetyczne.

## Instrukcje adresowania tekstur

Instrukcje adresowania tekstur mogą zapewniać większe możliwości niż instrukcje arytmetyczne. Jednocześnie ich użycie może być bardziej kłopotliwe. W tabeli 16.3 zwięźle opisano instrukcje adresowania tekstur. Niektóre z tych instrukcji omówię bardziej szczegółowo w dalszych rozdziałach opisujących techniki.

W przypadku instrukcji adresowania tekstur często trudno zrozumieć, kiedy wykorzystujemy parametr `t(n)` do przechowywania danych opisu współrzędnych tekstury, a kiedy parametr ten zawiera dane o kolorach. Dla wyjaśnienia poniżej podałem kilka przykładów tego, w jaki sposób współrzędne tekstury odpowiadają wartościom opisującym kolory oraz rejestrom wynikowym opisu tekstur.

**Tabela 16.2.** *Instrukcje arytmetyczne w mechanizmach pixel shader*

| Instrukcja | Format | Opis |
|---|---|---|
| `mov` | `mov Wynik, Wejscie0` | Przesyła wartość z jednego rejestru do drugiego. Instrukcja ta przydaje się do przekazywania tymczasowych wartości do rejestru `r0`, który zawiera ostateczny wynik działania shadera. |
| `add` | `add Wynik, Wejscie0, Wejscie1` | Dodaje jedną wartość rejestru do drugiej i umieszcza wynik w rejestrze wynikowym. |
| `sub` | `sub Wynik, Wejscie0, Wejscie1` | Odejmuje jedną wartość rejestru od drugiej i umieszcza wynik w rejestrze wynikowym. Pixel shadery obsługują także negację rejestrów, a zatem można wykorzystać instrukcję `add` z zanegowanym parametrem wejściowym. |
| `mul` | `mul Wynik, Wejscie0, Wejscie1` | Mnoży jedną wartość rejestru przez drugą i umieszcza wynik w rejestrze wynikowym. Podobnie jak vertex shadery, pixel shadery nie obsługują instrukcji dzielenia. Jeżeli chcemy podzielić liczbę, powinniśmy przekazać do shadera jej odwrotność. |
| `mad` | `mad Wynik, Wejscie0, Wejscie1, Wejscie2` | Podobnie jak instrukcja `mad` dostępna dla vertex shaderów, ta instrukcja wykonuje mnożenie i dodawanie w pojedynczej instrukcji. Najpierw wykonywane jest mnożenie parametru `Wejscie0` przez `Wejscie1`, a następnie do obliczonego iloczynu dodawana jest wartość `Wejscie2`. Wynik umieszczany jest w rejestrze wynikowym. |
| `dp3` | `dp3 Wynik, Wejscie0, Wejscie1` | Ta instrukcja wykonuje trójelementową operację iloczynu skalarnego pomiędzy wartościami zapisanymi jako parametry `Wejscie0` i `Wejscie1`. Zakłada się, że wartości te są wektorami, chociaż można wykorzystać tę operację do innego celu. Instrukcję tę wykorzystano w rozdziale 31. |
| `dp4` | `dp4 Wynik, Wejscie0, Wejscie1` | Instrukcja taka jak `dp3`, z tą różnicą, że dotyczy czterech elementów. Instrukcja jest dostępna w wersji 1.2 i w wersjach wyższych. W wersjach 1.2 oraz 1.3. liczy się ją jako dwie instrukcje (1.2, 1.3, 1.4). |
| `cnd` | `cnd Wynik, Wejscie0, Wejscie1, Wejscie2` | Ta instrukcja warunkowa ustawia wartości w rejestrze wynikowym na podstawie tego, czy wartości komponentów są większe niż 0,5. W wersji 1.4 ta instrukcja działa dla każdego komponentu oddzielnie. Jeżeli którykolwiek z komponentów parametru `Wejscie0` jest większy niż 0,5, odpowiadający mu komponent rejestru wynikowego jest ustawiany na wartość komponentu parametru `Wejscie1`; w przeciwnym wypadku wartość jest pobierana z parametru `Wejscie2`. Jest możliwe zatem, że wynik jest swego rodzaju kompozycją wartości parametru `Wejscie1` i `Wejscie2`. We wszystkich wersjach wcześniejszych niż 1.4. wartość porównywana jest ograniczona do jednej wartości — `r0.a` (1.1, 1.2, 1.3, 1.4 z ograniczeniami opisanymi powyżej). |
| `cmp` | `cmp Wynik, Wejscie0, Wejscie1, Wejscie2` | Operacja podobna do `cnd`, ale tym razem komponenty wejściowe są porównywane z wartością 0,0. Jeżeli są większe lub równe 0,0, wybierana jest wartość pochodząca z parametru `Wejscie1`, w innym przypadku pobierana jest wartość z parametru `Wejscie2`. Instrukcja jest dostępna w wersji 1.2 i w wersjach późniejszych, ale w wersjach 1.2 i 1.3 traktuje się ją jako dwie instrukcje (1.2, 1.3, 1.4 z ograniczeniami). |
| `lrp` | `lrp Wynik, Wejscie0, Wejscie1, Wejscie2` | Instrukcja `lrp` wykonuje liniową interpolację wartości zawartych w parametrach `Wejscie1` i `Wejscie2` na podstawie parametru `Wejscie0`. Przykładowo, pojedyncza wartość wyniku jest obliczana jako `Wynik.r = (Wejscie1.r * Wejscie0.r) + (Wejscie2.r * (1.0 - Wejscie0.r))`. |
| `bem` | `bem Wynik.rg, Wejscie0, Wejscie1` | Ta instrukcja jest dostępna tylko w wersji 1.4. Oblicza symulowaną wartość środowiskowego odwzorowania nierówności na podstawie ustawienia stanu macierzy nierówności fazy tekstury. |
| `nop` | `nop` | Ta instrukcja nie wykonuje żadnych działań. |

**Tabela 16.3.** *Instrukcje adresowania tekstur dostępne w mechanizmach pixel shader*

| Instrukcja | Format | Opis |
|---|---|---|
| `tex` | `tex t(n)` | Instrukcja ładuje wartość z wybranej fazy tekstury na podstawie współrzędnych tekstury dla tej fazy. Podstawowym zastosowaniem tej instrukcji jest pobieranie wartości do przetwarzania. Instrukcja `tex` nie jest dostępna w wersji 1.4. W wersji 1.4. shadery wykorzystują instrukcję `texld` (1.1, 1.2, 1.3). |
| `texld` | `texld r(n), t(n)` | Instrukcja ładuje wartość koloru z tekstury do rejestru tymczasowego. W tym przypadku rejestr tekstury `t(n)` zawiera współrzędne tekstury, natomiast rejestr `r(n)` zawiera dane tekstury. Z tego powodu wiersz `texld r2, t0` pobiera współrzędne tekstury z fazy 0 i wykorzystuje je w celu odszukania wartości koloru w teksturze fazy 2. Wartości koloru są zapisywane do rejestru `r2` (tylko 1.4.) |
| `texcrd` | `texcrd Wynik, t(n)` | W wersji 1.4. instrukcja `texcrd` kopiuje dane zawierające współrzędne tekstury z rejestru `t(n)` do rejestru wynikowego, jako dane opisujące kolor. W drugim etapie działania pixel shadera można wykorzystać rejestr wynikowy jako źródło w wywołaniu instrukcji `texld`, ale tylko w etapie 2 (tylko 1.4). |
| `texcoord` | `texcoord t(n)` | `texcoord` jest w pewnym sensie instrukcją towarzyszącą instrukcji `tex` i instrukcją analogiczną do `texcrd`. Jeżeli shader wywoła tę instrukcję zamiast `tex`, dane opisu współrzędnych tekstury będą załadowane zamiast wartości opisującej kolor. Instrukcja może być przydatna w tych technikach, gdzie vertex shader przekazuje dane do pixel shadera poprzez współrzędne tekstury (1.1, 1.2, 1.3). |
| `texreg2ar` | `texreg2ar t(m), t(n)` | Ta instrukcja interpretuje komponenty kanału alfa oraz koloru czerwonego rejestru `t(n)` jako współrzędne tekstury u oraz v. Te współrzędne tekstury są wykorzystywane jako indeks do rejestru `t(m)` w celu pobrania wartości opisu koloru z tej fazy tekstury. Przykładowo, vertex shader może ustawić współrzędne tekstury z fazy 0 na (0,0). Komponenty kanału alfa oraz koloru czerwonego w tym punkcie w teksturze mogą obydwa mieć wartość 0,5. Z tego powodu możemy wykorzystać współrzędne (0,5; 0,5) do odszukania wartości teksela wewnątrz rejestru `t1`. Wewnątrz pixel shadera odpowiedni wiersz kodu przyjąłby postać `texreg2ar t1, t0`. Docelowa faza tekstury musi być większa od źródłowej. (1.1, 1.2, 1.3). |
| `texreg2gb` | `texreg2gb t(m), t(n)` | Instrukcja analogiczna do instrukcji `texreg2ar`, z tą różnicą, że jako współrzędne tekstury wykorzystywane są komponenty koloru zielonego i niebieskiego. (1.1, 1.2, 1.3). |
| `texreg2rgb` | `texreg2rgb t(m), t(n)` | Instrukcja taka sama jak poprzednie, ale wykorzystuje trzy współrzędne tekstury do zastosowania w odwzorowaniach sześciennych lub teksturach 3D (tylko 1.2 i 1.3). |
| `texkill` | `texkill t(n)` | Ta instrukcja „zabija" bieżący piksel, jeżeli dowolna z trzech pierwszych współrzędnych tekstury jest mniejsza niż zero. Instrukcja może być przydatna do implementacji płaszczyzn obcinanych (ang. *clipping planes*), ale może powodować niepożądane wyniki w przypadku stosowania wielopróbkowania (ang. *multisampling*). W wersji 1.4 w etapie 2 rejestr wejściowy może być rejestrem tymczasowym ustawionym w etapie 1. |

**Tabela 16.3.** *Instrukcje adresowania tekstur dostępne w mechanizmach pixel shader — ciąg dalszy*

| Instrukcja | Format | Opis |
|---|---|---|
| texm3x2pad | texm3x2pad t(m), t(n) | Ta instrukcja wykonuje pierwszą część obliczeń dla macierzy 3×2. Wartości w rejestrze t(n) (z reguły wektory) są mnożone przez pierwsze trzy wartości współrzędnych tekstury fazy m, które w tym przypadku są wykorzystywane jako pierwszy wiersz macierzy 3×2. Instrukcję tę możemy zastosować tylko razem z instrukcją texm3x2tex lub texm3x2depth (opisane dalej). Nie można użyć jej samej. Instrukcję tę należy traktować jako pierwszą część instrukcji dwuczłonowej (1.1, 1.2, 1.3). |
| texm3x2tex | texm3x2tex t(m+1), t(n) | Jest to druga część instrukcji poprzedniej. Rejestr docelowy musi pochodzić z fazy tekstury wyższej niż pozostałe dwie fazy. Instrukcja ta wykonuje drugą część mnożenia macierzy 3×2. Rejestr t(n) jest mnożony przez drugi wiersz macierzy (zapisanej jako współrzędne tekstury dla fazy m+1). Wynik jest następnie wykorzystywany jako parametr wyszukiwania w teksturze fazy m+1. W wyniku działania tej instrukcji rejestr t(m+1) zawiera poszukiwany kolor. Instrukcję tę najlepiej wytłumaczyć na przykładzie. Instrukcje tego typu i podobne wykorzystano w rozdziale 32. (1.1, 1.2, 1.3) |
| texm3x2depth | texm3x2depth t(m+1), t(n) | Jest to druga, alternatywna część instrukcji texm3x2pad. Jeżeli stosujemy tę instrukcję, to do obliczenia wartości z dla tego piksela powinniśmy użyć funkcji texm3x2pad (na podstawie pierwszego wiersza macierzy). Instrukcja ta oblicza współrzędną w, wykorzystując współrzędne z fazy m+1 jako drugi wiersz macierzy. Następnie instrukcja oblicza współrzędne z/w i zaznacza tę wartość do wykorzystania jako alternatywną wartość głębokości dla wybranego piksela (tylko 1.3). |
| texm3x3pad | texm3x3pad t(m), t(n) | Ta instrukcja działa tak samo jak instrukcja texm3x2pad, z tą różnicą, że przetwarza macierz 3×3. Z tego powodu przed dopełnieniem instrukcji za pomocą texm3x3tex, texm3x3spec lub texm3x3vspec instrukcję tę trzeba wywołać dwukrotnie (1.1, 1.2, 1.3). |
| texm3x3tex | texm3x3tex t(m+2), t(n) | Podobnie jak instrukcja texm3x2tex, ta instrukcja jest uzupełnieniem pełnej operacji macierzowej. W tym przypadku jest to mnożenie macierzy 3×3. Zakłada się, że wywołanie instrukcji poprzedzono dwoma wywołaniami instrukcji texm3x3pad oraz że każda kolejna wykorzystana faza była wyższa od poprzedniej. Wynik mnożenia wykorzystywany jest jako współrzędne tekstury w celu pobrania wartości tekstury z t(m+2) (1.1, 1.2, 1.3). |
| texm3x3spec | texm3x3spec t(m+2), t(n), c(n) | Jest to alternatywa dla instrukcji texm3x3tex. Wynik mnożenia macierzy 3×3 jest interpretowany jako wektor normalny wykorzystywany do obliczeń odbicia. Rejestr c(n) przechowuje stałą wektora oka odpowiadającą tej instrukcji dla wynikowego wektora normalnego. Uzyskany w wyniku wektor 3D wykorzystywany jest jako zbiór współrzędnych tekstury dla tekstury w fazie m+1. Instrukcję tę stosuje się do mapowania środowiskowego (1.1, 1.2, 1.3). |
| texm3x3vspec | texm3x3vspec t(m+2), t(n) | Instrukcja podobna do instrukcji texm3x3spec, ale bez wykorzystania stałego wektora oka. W zamian wektor oka jest pobierany z czwartego komponentu wierszy macierzy (1.1, 1.2, 1.3). |
| texm3x3 | texm3x3 t(m+2), t(n) | Ta instrukcja także może uzupełniać ciąg trzech instrukcji. Jej działanie polega na przesłaniu wektora wynikowego do rejestru wyniku bez wykonywania poszukiwania danych opisu tekstury (1.2, 1.3). |

**Tabela 16.3.** *Instrukcje adresowania tekstur dostępne w mechanizmach pixel shader — ciąg dalszy*

| Instrukcja | Format | Opis |
|---|---|---|
| texbem | texbem t(m), t(n) | Ta instrukcja oblicza informacje dotyczące pozornego środowiskowego mapowania nierówności za pomocą macierzy nierówności ustawionej za pomocą wywołania funkcji SetTextureStageState. Wartości koloru w rejestrze t(n) są mnożone przez macierz tekstury, a wynik jest wykorzystywany jako indeks tekstury w fazie m (1.1, 1.2, 1.3). |
| texbem1 | texbem1 t(m), t(n) | Instrukcja stanowi uzupełnienie instrukcji texbem. Wykonuje tę samą funkcję, ale dodatkowo uwzględnia korektę luminancji ze stanów faz tekstur. |
| texdepth | texdepth r(n) | Tej instrukcji można użyć jedynie w drugim etapie w shaderach wersji 1.4. W pierwszym etapie należy wypełnić komponenty r i g rejestru za pomocą wartości z i w. Instrukcja oblicza wartość z/w i zaznacza tę wartość do zastosowania jako wartość głębi dla tego piksela. (1.4. — tylko w drugim etapie). |
| texdp3 | texdp3 t(m), t(n) | Instrukcja oblicza trójelementowy iloczyn skalarny danych zapisanych w rejestrze t(n) oraz współrzędnych tekstury zapisanych w rejestrze t(m). Uzyskana w wyniku wartość skalarna jest kopiowana do wszystkich czterech komponentów rejestru t(m) (1.2, 1.3). |
| texdp3tex | texdp3tex t(m), t(n) | Ta instrukcja oblicza iloczyn skalarny tak jak instrukcja poprzednia, ale uzyskany w wyniku skalar jest używany jako indeks do poszukiwania danych w teksturze fazy m. Uzyskana w wyniku wartość koloru jest zapisywana do rejestru t(m) (1.2, 1.3). |

W poniższym przypadku tekstura w fazie 0 jest próbkowana razem ze współrzędnymi tekstury z fazy 0. Uzyskany w wyniku kolor jest zapisywany do t0. Kolejne instrukcje używające t0 będą wykorzystywały próbkowaną wartość koloru:

```
tex t0
```

W kolejnym przykładzie rejestr t0 jest zapisywany wartością współrzędnych tekstury z fazy 0. Kolejne instrukcje wykorzystujące rejestr t0 będą wykorzystywać dane opisu współrzędnych tekstury interpretowane jako wartość koloru:

```
texcoord t0
```

Kolejny fragment kodu wykorzystuje pierwsze dwie współrzędne tekstury z fazy 0 jako współrzędne tekstury w fazie 1. Tekstura fazy 1. jest próbkowana, a uzyskany w wyniku kolor jest zapisywany do rejestru t1. Kolejne instrukcje korzystające z rejestru t1 będą wykorzystywały wartość koloru z próbki tekstury fazy 1.

```
texreg2ar t1, t0
```

We wszystkich wersjach shaderów wcześniejszych niż 1.4 rejestry t(n) mogą być zarówno odczytywane, jak i zapisywane. W przypadku odczytu uzyskana dana jest interpretowana jako współrzędna tekstury. W przypadku zapisu wartość może zawierać próbkowaną wartość koloru tekstury lub na przykład interpretowaną współrzędną tekstury (w przypadku instrukcji texcoord). W wersji 1.4. rejestry tekstury mogą być tylko odczytywane. W przypadku poniższej instrukcji texld rejestr t0 zawiera współrzędne tekstury, natomiast r0 zawiera wynik zastosowania tych współrzędnych do próbkowania tekstury fazy 0:

```
texld r0, t0
```

W dalszych rozdziałach niektóre z opisanych wcześniej instrukcji zostaną zaprezentowane w praktycznym działaniu. Kiedy zobaczymy, jak się ich używa, nabiorą dla nas większego sensu.

Możliwości przetwarzania nie są ograniczone jedynie do zestawu instrukcji. Podobnie jak w przypadku mechanizmów vertex shader, pixel shadery umożliwiają zastosowanie modyfikatorów, które dają wiele możliwości.

# Modyfikatory dostępne dla mechanizmów pixel shader

W pixel shaderach można stosować niektóre spośród dodatkowych funkcji dostępnych dla vertex shaderów, takich jak negacja czy maski zapisu, ale istnieje także zbiór modyfikatorów instrukcji, które gwarantują dodatkowe możliwości. Dostępne modyfikatory dla rejestrów źródłowych zestawiono w tabeli 16.4.

**Tabela 16.4.** *Modyfikatory rejestrów źródłowych dostępne dla pixel shaderów*

| Modyfikator | Składnia | Opis |
|---|---|---|
| Przesunięcie | `r0_bias` | Odejmuje 0,5 od wszystkich czterech komponentów rejestru. Ten modyfikator można stosować dla dowolnego rejestru źródłowego. |
| Odwrócenie | `1-r0` | Przed wykonaniem instrukcji następuje odjęcie wartości rejestru źródłowego od 1,0. Zawartość rejestru źródłowego nie zmienia się. |
| Negacja | `-r0` | Oblicza negację komponentów przed wykonaniem instrukcji. Znów zawartość rejestru źródłowego nie zmienia się. |
| Podwojenie | `ro_x2` | Ten modyfikator przed wykonaniem instrukcji mnoży komponenty przez 2,0. Jest dostępny tylko w wersji 1.4. |
| Skala ze znakiem | `ro_bx2` | Ten modyfikator odejmuje wartość 0,5, a następnie mnoży uzyskany wynik przez 2,0. Modyfikator szczególnie przydaje się do wykonywania konwersji zakresu koloru od 0,0 do 1,0 na zakres od −1,0 do 1,0. |

W pixel shaderach jest także dostępna ograniczona forma operacji mieszania. Przed wykonaniem instrukcji możemy dokonać replikacji jednego kanału na wszystkie kolory. Podobnie jak w przypadku operacji mieszania dostępnej w vertex shaderach odpowiednie rejestry danych nie ulegają zmianie. Odpowiednie selektory rejestrów źródłowych zestawiono w tabeli 16.5.

Można także stosować maski zapisu. We wszystkich wersjach shaderów można wybrać, czy będziemy zapisywać wszystkie kanały, tylko kanał alfa, czy tylko kolory. W wersji 1.4 do zapisu można wybrać dowolne kanały. Składnia masek zapisu jest taka sama jak w przypadku mechanizmów vertex shader, z tą różnicą, że zamiast etykiet *.xyzw* stosujemy etykiety *.rgba*, co pokazano w tabeli 16.5.

Ostatnią grupą modyfikatorów są modyfikatory instrukcji. O działaniu tych modyfikatorów możemy myśleć jako o operacji wykonywanej przed ustawieniem wartości wyniku. Tego typu modyfikatory opisano w tabeli 16.6. W przykładach posłużono się instrukcją `add`, ale modyfikatorów tych można użyć dla większości instrukcji arytmetycznych.

**Tabela 16.5.** *Selektory rejestrów źródłowych w mechanizmach pixel shader*

| Selektor | Składnia | Opis |
|---|---|---|
| Replikacja kanału alfa | `r0.a` | Ten selektor replikuje wartość kanału alfa do wszystkich komponentów rejestru wejściowego. Selektor jest dostępny we wszystkich wersjach shadera. |
| Replikacja kanału niebieskiego | `r0.b` | Replikuje wartość kanału niebieskiego do wszystkich komponentów rejestru wejściowego. Selektor jest dostępny we wszystkich wersjach shadera w wersji 1.1 i w wersjach późniejszych. |
| Replikacja kanału zielonego | `r0.g` | Replikuje wartość kanału zielonego do wszystkich kanałów. Dostępny jedynie w wersji 1.4. |
| Replikacja kanału czerwonego | `r0.r` | Replikuje wartość kanału czerwonego do wszystkich kanałów. Dostępny jedynie w wersji 1.4. |

**Tabela 16.6.** *Modyfikatory instrukcji w mechanizmach pixel shader*

| Modyfikator | Składnia | Opis |
|---|---|---|
| Mnożenie przez 2 | `add_x2` | Ten modyfikator mnoży wynik instrukcji przez 2. |
| Mnożenie przez 4 | `add_x4` | Mnoży wynik przez 4. |
| Mnożenie przez 8 | `add_x8` | Mnoży wynik przez 8. Dostępny jedynie w wersji 1.4. |
| Dzielenie przez 2 | `add_d2` | Dzieli wynik przez 2. |
| Dzielenie przez 4 | `add_d4` | Dzieli wynik przez 4. Dostępny tylko w wersji 1.4. |
| Dzielenie przez 8 | `add_d8` | Dzieli wynik przez 8. Dostępny tylko w wersji 1.4. |
| Nasycenie | `add_sat` | Konwertuje wynik do zakresu 0,0 – 1,0. Modyfikator nasycenia zapewnia, że wartości mieszczą się w poprawnym zakresie dla kolorów. |

# Ograniczenia i uwagi dotyczące stosowania mechanizmów pixel shader

Podobnie jak w przypadku mechanizmów vertex shader podstawowym ograniczeniem mechanizmów pixel shader jest liczba instrukcji. Pixel shadery podlegają znacznie ostrzejszym ograniczeniom niż vertex shadery. W wersjach 1.1, 1.2 i 1.3 istnieje ograniczenie do 4 instrukcji adresowania tekstur oraz 8 instrukcji arytmetycznych, co daje w sumie 12 instrukcji. Wersja 1.4 dopuszcza 8 instrukcji arytmetycznych oraz sześć instrukcji adresowania tekstur dla każdego z dwóch etapów. W sumie daje to 28 dostępnych instrukcji. Jest to ponad dwukrotna liczba instrukcji w porównaniu z wersjami poprzednimi, ale w dalszym ciągu dużo mniejsza od liczby instrukcji dostępnych w mechanizmach vertex shader.

Chociaż istnieje ograniczenie co do liczby instrukcji adresowania oraz instrukcji arytmetycznych, to nie ma ograniczeń co do liczby instrukcji „konfiguracyjnych". Instrukcje `def`, `phase` oraz `ps` nie zajmują dostępnego limitu liczby instrukcji.

Innym ograniczeniem jest liczba dostępnych rejestrów. W tabeli 16.7 zestawiono ograniczenia liczby rejestrów dla wszystkich wersji mechanizmów pixel shader.

**Tabela 16.7.** *Ograniczenia liczby rejestrów w mechanizmach pixel shader*

| Typ rejestru | Ograniczenie w wersjach 1.1 – 1.3 | Ograniczenie w wersji 1.4 |
|---|---|---|
| Rejestr koloru `v(n)` | 2 | 2 (etap 2) |
| Rejestr tekstury `t(n)` | 4 | 6 |
| Rejestr stałych `c(n)` | 8 | 8 |
| Rejestr tymczasowy `r(n)` | 2 | 6 |

Pixel shadery są ograniczone przez sprzęt, na którym działają. Niektóre karty być może w ogóle nie będą obsługiwały żadnej z wersji pixel shader. Inaczej niż w przypadku mechanizmów vertex shader, dla pixel shaderów nie ma akceptowalnej alternatywy programowej. Jeżeli sprzęt ich nie obsługuje, wydajność jest przeraźliwie niska. Urządzenie referencyjne może służyć do testowania kodu, ale nie do rzeczywistego wykorzystania.

# Sprawdzanie dostępności mechanizmów pixel shader

Podobnie jak w przypadku mechanizmów vertex shader, można sprawdzić, czy nasz sprzęt obsługuje pixel shadery, poprzez wywołanie metody `SetDeviceCaps`. Struktura `Mozl` zawiera pole typu `DWORD` — `PixelShaderVersion`. Wartość zawiera zarówno numer wersji głównej, jak numery wersji pomocniczych. Najlepiej analizować znaczenie tej wartości za pomocą makra `D3DPS_VERSION`:

```
D3DCAPS8 Mozl;
  m_wD3D->GetDeviceCaps(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, &Mozl);
if (Mozl.PixelShaderVersion == D3DPS_VERSION(1,1)
{
        if (FAILED(UtworzProstaAplwOknie(m_uchwyt, D3DDEVTYPE_HAL,
                          D3DCREATE_HARDWARE_VERTEXPROCESSING)))
                return FALSE;
}
```

Uzyskany numer wersji to wersja maksymalna. Urządzenia obsługujące daną wersję shadera powinny także obsługiwać wersje wcześniejsze. Jeżeli urządzenie nie obsługuje mechanizmów pixel shader, być może konieczne będzie opracowanie techniki zastępczej wykorzystującej operacje łączenia tekstur lub wyłączenie niektórych efektów. Jeżeli mechanizmy pixel shader są dostępne, możemy zrobić krok naprzód i przystąpić do utworzenia shadera.

# Asemblacja, tworzenie i wykorzystywanie mechanizmów pixel shader

Asemblacja pixel shaderów jest działaniem podobnym do asemblacji vertex shaderów. Należy wykorzystać to samo wywołanie funkcji `D3DXAssembleShader` (zobacz poprzedni

rozdział). Jeżeli składnia jest poprawna, to skompilowany shader znajdzie się w buforze shadera. Skompilowany shader może posłużyć do właściwego utworzenia shadera.

Kolejną czynnością jest wywołanie funkcji `CreatePixelShader`. Jest to funkcja podobna do funkcji `CreateVertexShader`, ale niewymagająca deklaracji. Pixel shadery zawsze operują na wartości koloru składającej się z czterech komponentów, niezależnie od formatu tekstury lub bufora:

```
HRESULT IDirect3DDevice8::CreatePixelShader(CONST
                                            DWORD *wSkompilowanyShader,
                                            DWORD *wUchwytShadera);
```

Uzyskany uchwyt może posłużyć do uaktywnienia wybranego shadera:

```
HRESULT IDirect3DDevice8::SetPixelShader(DWORD UchwytShadera);
```

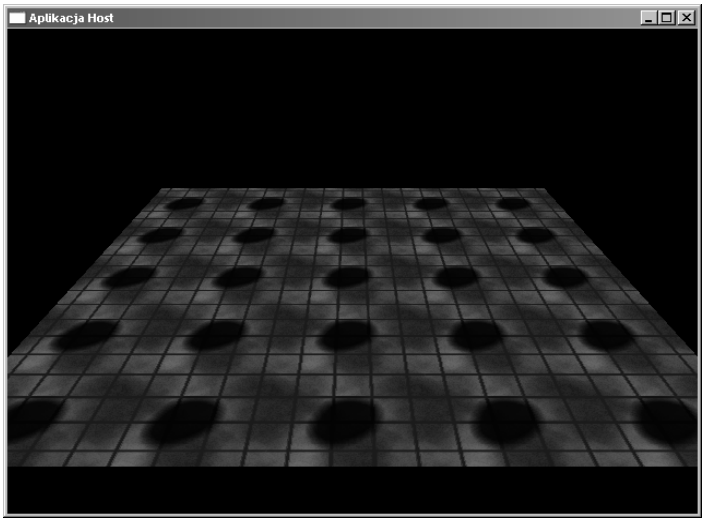
Wywołanie metody `SetPixelShader` ma analogiczne działanie do ustawienia kilku stanów fazy tekstury. Przekazanie wartości `NULL` do metody `SetPixelShader` powoduje dezaktywację pixel shadera. Jeżeli nie ustawiono aktywnego pixel shadera, to można w zwykły sposób korzystać ze stanów faz tekstur. W rzeczywistości w przypadku prostych operacji w dalszym ciągu można wykorzystywać stany tekstury. Przykładowo, domyślnie pierwsza faza tekstury moduluje kolor rozproszenia. Nie ma potrzeby dokonywania implementacji tej operacji wewnątrz shadera, można pozwolić na wykonanie tej czynności w ramach standardowej operacji łączenia.

# Bardzo prosta aplikacja wykorzystująca mechanizm pixel shader

Niestety, zanim przejdę do naprawdę interesujących aplikacji wykorzystujących pixel shadery, najpierw muszę omówić kilka aplikacji wykorzystujących mechanizmy vertex shader. Większość z ciekawszych zastosowań pixel shaderów wymaga użycia mechanizmów vertex shader do dostarczenia danych do pixel shadera. W tym rozdziale skoncentruję się na prostej aplikacji, która przedstawia niektóre podstawowe pojęcia. Do praktycznych zastosowań wykorzystamy pixel shadery, począwszy od rozdziału 29.

W tym rozdziale wykorzystamy prosty vertex shader do obliczenia wartości kierunkowego oświetlenia dla każdego wierzchołka. Współczynniki oświetlenia są interpolowane na powierzchni prostej płaszczyzny. Pixel shader łączy teksturę z wartością oświetlenia, ale wykorzystuje także inną teksturę w celu zdefiniowania obszaru tekstury odbijającego mniej światła. Jest to uproszczona wersja oświetlenia na poziomie pikseli. Oświetlenie na poziomie wierzchołków oblicza vertex shader, ale to pixel shader wykonuje ostatni etap obliczeń, określając wartość oświetlenia dla każdego piksela. Efekt działania aplikacji pokazano na rysunku 16.4.

Aby zrozumieć, jakie dane trafiają do pixel shadera, należy najpierw przeanalizować vertex shader.




**Rysunek 16.4.** *Bardzo prosta aplikacja wykorzystująca pixel shader*

# Proste oświetlenie za pomocą mechanizmu vertex shader

Wykonywanie obliczeń oświetlenia za pomocą vertex shaderów szczegółowo opisano w rozdziale 24., zatem w tym rozdziale nie będę zagłębiał się w teorię. Pokazany tu przykład to niezwykle prosty vertex shader obliczający iloczyn skalarny wektora oświetlenia oraz wektora normalnego wierzchołka. W mojej implementacji vertex shader jest wykorzystany nieco „na siłę", jedynie po to, aby pokazać, w jaki sposób należy przekazywać dane do pixel shadera,. W kolejnych rozdziałach zaprezentuję lepsze przykłady, natomiast przykład pokazany w tym rozdziale ma na celu pokazanie możliwości wzajemnej interakcji pomiędzy shaderami. Poniższy kod shadera pochodzi z pliku *PixelSetup.vsh*:

```
vs.1.1
```

Wyprowadzimy przekształcone pozycje. W żadnym stopniu nie ma to wpływu na pixel shader:

```
dp4 oPos.x, v0, c0
dp4 oPos.y, v0, c1
dp4 oPos.z, v0, c2
dp4 oPos.x, v0, c3
```

Obliczamy iloczyn skalarny normalnej wierzchołka oraz wektora oświetlenia dla światła kierunkowego. Wartość iloczynu skalarnego to kosinus kąta pomiędzy dwoma wektorami. W tym kontekście iloczyn skalarny wykorzystywany jest do określenia stopnia odbicia światła od powierzchni w prostym modelu oświetlenia rozpraszającego. Wektor oświetlenia przekazywany jest do vertex shadera jako `c4`. Negujemy go, aby wykonać przekształcenie na wektor „wierzchołek — oświetlenie". Wynik zapisywany jest do rejestru koloru zwierciadlanego — `oD1`. Temu rejestrowi w mechanizmie pixel shader odpowiada wejściowy rejestr koloru `v1`:

```
dp3 oD1, v3, -c4
```

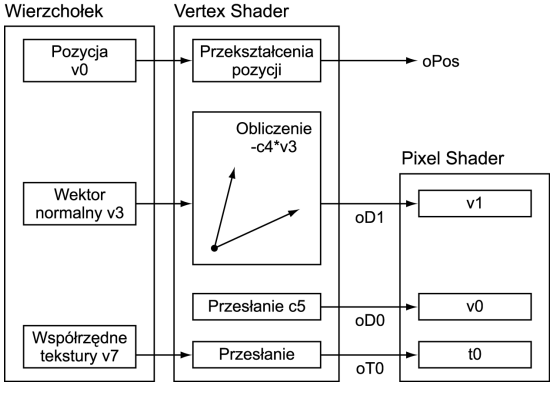
Rejestr stałych — `c5` przechowuje wartość oświetlenia otoczenia. Przesyłamy tę wartość do `oD0` (`v0` to pixel shader). To właśnie w tym miejscu zastosowałem shader trochę „na siłę". Moglibyśmy się sprzeczać, czy oświetlenie kierunkowe i oświetlenie otoczenia można było dodać wewnątrz shadera. Można by również dyskutować, które dane umieścić i z których rejestrów skorzystać. Pamiętajmy, że celem tej aplikacji jest zaprezentowanie możliwości, nie zaś poprawnej techniki oświetlenia:

```
mov oD0, c5
```

Na koniec przesyłamy współrzędne tekstury (`v7`) do wyjściowych rejestrów tekstury. Dzięki temu pixel shader będzie mógł prawidłowo korzystać z tekstury:

```
mov oT0, v7
```

Zatem vertex shader wykonuje czynności konfiguracyjne, przesyłając dwie wartości koloru oraz zbiór współrzędnych tekstury do pixel shadera. To działanie zaprezentowano na rysunku 16.5.



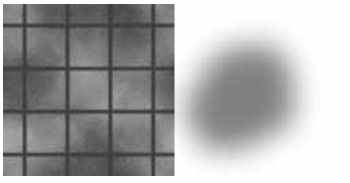

**Rysunek 16.5.** *Konfiguracja parametrów pixel shadera przez vertex shader*

Ostatnia operacja jest przetwarzana wewnątrz mechanizmu pixel shader.

# Proste operacje łączenia wewnątrz pixel shadera

Tekstura fazy 0 zawiera teksturę, która służy dwóm celom. Kanał koloru definiuje kolory obiektu, natomiast kanał alfa decyduje o tym, jak dobrze wybrany piksel odbija światło kierunkowe. Pamiętajmy, że nie jest to najlepszy z możliwych modeli oświetlenia. Kanały koloru oraz kanał alfa tekstury pokazano na rysunku 16.6.

Należy zauważyć, że kanał alfa stanowi wygodne miejsce, gdzie można przechowywać wartości, które nie zawsze są widoczne, ale są potrzebne do obliczeń. Jeżeli jawnie nie wykorzystujemy kanału alfa dla obiektów przezroczystych, możemy wykorzystać go do



**Rysunek 16.6.**
*Kanały koloru i alfa dla tekstury*

innych celów. W naszym przypadku wykorzystuję kanał alfa do przechowywania 8-bitowego współczynnika skalowania dla wartości oświetlenia kierunkowego. Poniższy kod shadera znajduje się w pliku *Simple.psh*. Pierwszy wiersz informuje asembler shadera, którą wersję shadera wykorzystujemy:

```
ps.1.1
```

W pierwszym wierszu ładujemy wartość tekstury do pixel shadera. Kod ten możemy interpretować następująco: wartość współrzędnej tekstury w rejestrze `t0` wykorzystujemy jako daną wejściową. Ten sam rejestr `t0` wykorzystujemy także do umieszczenia w nim wartości koloru wybranego piksela na podstawie współrzędnych tekstury interpolowanych z wierzchołków:

```
tex t0
```

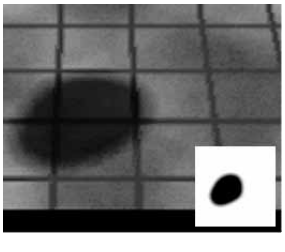
W następnym wierszu przeprowadzamy zasadnicze działania. Instrukcja mad powoduje pomnożenie interpolowanej wartości oświetlenia kierunkowego przez wartość współczynnika skalowania zapisanego w kanale alfa tekstury. Wartość oświetlenia otoczenia jest dodawana do poddanej skalowaniu wartości oświetlenia kierunkowego. Oznacza to, że na powierzchnię równomiernie działa oświetlenie otoczenia, natomiast oświetlenie kierunkowe oddziałuje z różną intensywnością dla różnych pikseli. Jest to dosyć dziwny sposób oświetlania powierzchni, ale pokazuje działanie prostego pixel shadera:

```
mad r0, v1, t0.a, v0
```

Po obliczeniu wartości oświetlenia ostateczna wartość jest modulowana przez wartość koloru tekstury. Rejestr `r0` jest zarówno rejestrem tymczasowym, jak rejestrem wynikowym. W vertex shaderach rejestry wynikowe są tylko do odczytu. Nie można ich wykorzystać jako danych wejściowych instrukcji. W przypadku pixel shaderów tak nie jest. Rejestr `r0` można wykorzystywać wielokrotnie, ale musimy się upewnić, że ostateczna wartość jest tą, którą chcemy przekazać.

```
mul r0, r0, t0
```

Na rysunku 16.7 pokazano powiększenie ostatecznego wyniku działania aplikacji. Zwróćmy uwagę na to, że ciemniejsza plama odpowiada kształtowi zdefiniowanemu dla kanału alfa pokazanemu na rysunku 16.6.



**Rysunek 16.7.**
*Płytki oświetlone światłem kierunkowym o różnym natężeniu*

Shadery są w zasadzie dosyć proste. Pozostaje opracowanie aplikacji, która łączy wszystkie opisane elementy w jedną całość.

# Prosta aplikacja z wykorzystaniem pixel shadera

Większość interesujących działań naszej aplikacji wykonują zaprezentowane wcześniej shadery. Głównym zadaniem aplikacji jest dostarczenie do shaderów danych oraz zapewnienie utworzenia właściwych shaderów i uaktywnienie ich w odpowiednim czasie. W zaprezentowanym niżej kodzie pokazuję tylko nowe funkcje. Pełny kod źródłowy znajduje się na płycie CD (*\Kod\Rozdzial16*).

Najpierw rozbudowuję funkcję `PoZainicjowaniu` o sprawdzenie dostępności pixel shaderów. Jeżeli urządzenie sprzętowe nie obsługuje pixel shaderów, aplikacja sięga do urządzenia referencyjnego. Jest to dobre do testowania, ale w rzeczywistych aplikacjach lepiej wyłączyć technikę niż wykorzystywać urządzenie referencyjne:

```
BOOL KAplikacjiTechniki::PoZainicjowaniu()
{
```

Pobieramy strukturę opisu możliwości i sprawdzamy, czy karta obsługuje mechanizmy vertex shader oraz pixel shader, posługując się makrami opisu wersji. We wszystkich przykładach w tej książce wykorzystujemy shadery w wersji 1.1 w celu zmaksymalizowania obsługi sprzętowej. Jeżeli shadery są dostępne, tworzymy urządzenie sprzętowe za pomocą funkcji pomocniczej. W innym przypadku tworzymy urządzenie referencyjne. Pamiętajmy o tym, że urządzenie referencyjne może być niezwykle wolne, szczególnie w przypadku wolnego procesora głównego. Jeżeli nasz sprzęt nie obsługuje shaderów, możemy czekać nawet kilka sekund na renderowanie ramki. W takim przypadkach należy zachować cierpliwość:

```
D3DCAPS8 Mozl;
m_wD3D->GetDeviceCaps(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, &Mozl);
if (Mozl.VertexShaderVersion == D3DVS_VERSION(1,1) &&
       Mozl.PixelShaderVersion == D3DPS_VERSION(1,1))
{
            if (FAILED(UtworzProstaAplwOknie(m_uchwyt, D3DDEVTYPE_HAL,
                              D3DCREATE_HARDWARE_VERTEXPROCESSING)))
                        return FALSE;
}
else
{
 if (FAILED(UtworzProstaAplwOknie(m_uchwyt, D3DDEVTYPE_REF,
                         D3DCREATE_SOFTWARE_VERTEXPROCESSING)))
    return FALSE;
  }
```

Metody `KonfigurujUrzadzenie` oraz `UtworzBuforPlaszczyzny` wykonują podstawowe działania konfiguracji urządzenia oraz utworzenia bufora dla czterech punktów prostokąta. Funkcji tych nie pokazano w tym rozdziale. Nie dzieje się w nich nic nowego:

```
KonfigurujUrzadzenie();
if (FAILED(UtworzBuforPlaszczyzny()))
     return FALSE;
```

Metoda `UtworzShadery` tworzy zarówno mechanizm vertex shader, jak i pixel shader. Jeżeli wykonanie tej metody nie powiedzie się, będzie to wskazywało na możliwy błąd syntaktyczny w kodzie shadera. Jeżeli shader jest poprawny, wykonanie funkcji zawsze powinno zakończyć się pomyślnie, ze względu na alternatywne wykorzystanie urządzenia referencyjnego. Jeżeli zdezaktywujemy wykorzystanie urządzenia referencyjnego, wywołanie funkcji może się nie powieść w przypadku, gdy sprzęt nie obsługuje shaderów:

```
if (FAILED(UtworzShadery()))
       return FALSE;
```

Poniższa tekstura to ta, którą pokazano na rysunku 16.5. Zarówno kolor, jak i dane dotyczące odbicia zapisano w pojedynczej teksturze:

```
  if (FAILED(D3DXCreateTextureFromFile(m_wUrzadzenieD3D,
                                       "..\\media\\Tile.dds",
                                       &m_wTekstura)))
       return FALSE;

  return TRUE;
}
```

Funkcja `UtworzShadery` tworzy shadery wykorzystywane w czasie renderowania sceny. Poniższy kod jest podobny do kodu prezentowanego w poprzednich rozdziałach:

```
HRESULT KAplikacjiTechniki::UtworzShadery()
{
      ID3DXBuffer* wBuforShadera;
      ID3DXBuffer* wBledyShadera;
```

Tworzymy vertex shader i wykonujemy asemblację zgodnie z opisem z poprzedniego rozdziału. W rzeczywistych implementacjach w przypadku błędów być może należałoby sprawdzić zawartość bufora błędów:

```
if (FAILED(D3DXAssembleShaderFromFile("..\\media\\shaders\\PixelSetup.vsh",
                                      0, NULL, &wBuforShadera, &wBledyShadera)))
       return E_FAIL;

if (FAILED(m_wUrzadzenieD3D->CreateVertexShader(Deklaracja,
                             (DWORD *)wBuforShadera->GetBufferPointer(),
                             &m_ShaderKonfiguracji, 0)))
           return E_FAIL;
```

Zwalniamy bufor shadera, aby można go było wykorzystać ponownie do utworzenia pixel shadera. Zwalnianie bufora błędów nie jest potrzebne, ponieważ jeżeli jesteśmy w tym miejscu, oznacza to, że bufor błędów nie został utworzony:

```
wBuforShadera->Release();
```

Wywołanie wykorzystane do utworzenia pixel shadera jest dokładnie takie same, jak wywołanie do utworzenia vertex shadera. Asembler wykorzystuje pierwszą instrukcję opisującą wersję w celu uzyskania informacji o typie shadera oraz o sposobie jego asemblacji:

```
if (FAILED(D3DXAssembleShaderFromFile("..\\media\\shaders\\Simple.psh",
                                                    0, NULL, &wBuforShadera,
                                                    &wBledyShadera)))
            return E_FAIL;
```

Wywołanie metody `CreatePixelShader` jest podobne do wywołania metody `CerateVertexShader`, ale w tym przypadku deklaracja nie jest potrzebna. Jeżeli operacja powiedzie się, to w wyniku jej działania będziemy mieli poprawny pixel shader do wykorzystania w czasie renderingu:

```
if (FAILED(m_wUrzadzenieD3D->CreatePixelShader(
                            (DWORD *)wBuforShadera->GetBufferPointer(),
                                                &m_ProstyPixelShader)))
                                            return E_FAIL;
          wBuforShadera->Release();

          return S_OK;
}
```

Powyższe wiersze odpowiadały za utworzenie shaderów. Przed zakończeniem aplikacji lub przed jej odtworzeniem powinniśmy zadbać o ich usunięcie. Sposób usunięcia shaderów w funkcji `PrzedOdtworzeniem` pokazano poniżej:

```
BOOL KAplikacjiTechniki::PrzedOdtworzeniem()
{
```

Za pomocą funkcji `DeleteVertexShader` oraz `DeletePixelShader` zapewniamy usunięcie shaderów. Możemy utworzyć bufory wierzchołków, które podlegają automatycznemu odtworzeniu w czasie odtwarzania urządzenia, ale o usunięcie i ponowne utworzenie shaderów musimy zadbać samodzielnie:

```
   m_wUrzadzenieD3D->DeleteVertexShader(m_ShaderKonfiguracji);
   m_wUrzadzenieD3D->DeletePixelShader(m_ProstyPixelShader);
   return TRUE;
}
```

Na koniec, funkcja renderująca wykorzystuje shadery do wykonania interesujących działań. W poniższym kodzie założono, że wcześniej poprawnie utworzono shadery:

```
void KAplikacjiTechniki::Renderuj()
{
```

Teraz trzeba ustawić vertex shader. W naszej prostej aplikacji mogliśmy ustawić vertex shader w czasie tworzenia shadera. W naszym przykładzie jednak umieszczam tę instrukcję wewnątrz funkcji renderującej dla lepszego zilustrowania wykonywanych działań:

```
m_wUrzadzenieD3D->SetVertexShader(m_ShaderKonfiguracji);
```

Poniższy kod wykonuje animację kierunku oświetlenia. W obliczeniach przyjęto, że oświetlenie jest zawsze skierowane w dół, niezależnie od wartości animacji. Po zdefiniowaniu kierunku oświetlenia jest on przekazywany do rejestru `c4` shadera. W rozdziale 24. dowiemy się, że takie przyjmowanie sposobu oddziaływania wektora oświetlenia na przekształcenia obiektu jest naiwne (i zazwyczaj niepoprawne), ale dla naszej prostej aplikacji wystarczy. Jeżeli zdecydujemy się na modyfikację macierzy świata, uzyskamy nieprawidłowe wyniki, ale w tym kontekście nie jest to takie ważne. Moje słowa nabiorą większego sensu w rozdziale 24:

```
float Czas = (float)GetTickCount() / 1000.0f;
D3DXVECTOR4 KierSwiatla = D3DXVECTOR4(sin(Czas), -fabs(cos(Czas)),
                                      0.0f, 0.0f);
m_wUrzadzenieD3D->SetVertexShaderConstant(4, &KierSwiatla, 1);
```

Ustawiamy niewielką składową światła otaczającego. Światło otaczające jest przekazywane do vertex shadera, a następnie do pixel shadera.

```
D3DXVECTOR4 Otoczenie    (0.1,  0.1f, 0.1f, 0.0f);
m_wUrzadzenieD3D->SetVertexShaderConstant(5, &Otoczenie, 1);
```

Do shadera trzeba także przekazać połączoną macierz. Poszczególne macierze ustawiono w funkcji `KonfigurujUrzadzenie`:

```
D3DXMATRIX MacierzShadera = m_MacierzSwiata * m_MacierzWidoku *
                    m_MacierzRzutu;
D3DXMatrixTranspose(&MacierzShadera, &MacierzShadera);
m_wUrzadzenieD3D->SetVertexShaderConstant(0, &MacierzShadera, 4);
```

Ustawiamy teksturę w fazie 0. W tym prostym przypadku teksturę można było ustawić podczas jej ładowania, ale dla jasności ustawiam jej wartość w tym miejscu:

```
m_wUrzadzenieD3D->SetTexture(0, m_wTekstura);
```

Ustawiamy pixel shader. Ten shader wymaga przekazania kilku tekstur wejściowych oraz wartości z vertex shadera. Można by było wykorzystać pixel shader z innym vertex shaderem i teksturą, ale wyniki mogłyby być nieprzewidywalne i prawdopodobnie błędne:

```
m_wUrzadzenieD3D->SetPixelShader(m_ProstyPixelShader);
```

Teraz kiedy wykonaliśmy wszystkie czynności przygotowawcze, możemy wykreślić siatkę. W naszym przypadku jest to płaszczyzna z prostą teksturą. Siatka zawiera współrzędne tekstury, które informują pixel shader, w jaki sposób należy próbkować teksturę. Podobnie jak w przypadku innych pojęć dotyczących tekstur, błędne współrzędne tekstury powodują uzyskiwanie błędnych wyników:

```
m_wUrzadzenieD3D->SetStreamSource(0, m_wPlaszczyznaBuforWierzcholk,
                                  sizeof(WIERZCHOLEK_SIATKI));
m_wUrzadzenieD3D->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
```

Zazwyczaj, jeżeli nie wykorzystujemy pixel shadera, dobrym pomysłem jest jego wyłączenie. W tym przypadku cały czas korzystamy z pixel shadera, ale dodałem poniższą instrukcję po to, aby podkreślić to, jak ważne jest wyłączanie shadera. Podobnie jak stany faz tekstur, pixel shader jest stosowany dla każdego piksela do chwili, kiedy zostanie jawnie zmieniony. Inaczej niż w przypadku stanów faz tekstur, wyłączenie shadera jest znacznie łatwiejsze niż osobne wyłączanie każdego ze stanów tekstury:

```
    m_wUrzadzenieD3D->SetPixelShader(0);
}
```

Wynik działania aplikacji już pokazano na rysunkach 16.4 i 16.6. Ten przykład jest nieco sztuczny, a procedury oświetlenia nie są najwyższej jakości, ale ilustruje podstawy tworzenia mechanizmów pixel shader, zasilania ich danymi przez vertex shadery oraz wykorzystywania do oddziaływania na wynik przebiegu renderingu.

# Podsumowanie

Nie ma w tym nic dziwnego, jeśli czytelnik po lekturze powyższych rozdziału nie czuje się jeszcze zbyt pewnie, zważywszy, że prezentowany materiał jest nowy. Osobiście uważam, że łatwiej zrozumieć pojęcia, kiedy się ich używa. W tym rozdziale moim zamiarem było skrótowe omówienie podstawowych pojęć oraz definicji instrukcji. W dalszych rozdziałach utrwalimy pojęcia za pomocą przykładów.

Kolejnych kilka rozdziałów koncentruje się na vertex shaderach. Nauczymy się składni oraz wielu pojęć na tle interesujących i ciekawych technik. Zaprezentowane techniki pozwolą nam myśleć w kategoriach ograniczeń instrukcji używanych w shaderach. Do czasu, kiedy przeanalizujemy pierwszy pixel shader w rozdziale 29., będziemy dość swobodnie posługiwać się shaderami. Od tego momentu przejście od vertex shaderów do pixel shaderów powinno być łatwiejsze, niż nam się wydaje. Zatem zapamiętajmy z tego rozdziału tyle, ile się da, ale pamiętajmy, że dalsze rozdziały utrwalą wiele spośród pojęć poznanych w tym rozdziale. Zanim przejdziemy dalej, spróbujmy podsumować poznane pojęcia:

- Pixel shadery zamieniają mniej efektywną technologię stanów tekstur, oferując model programowalny podobny do vertex shaderów.
- Istnieje możliwość jednoczesnego wykorzystania w tej samej aplikacji pixel shaderów z technologią łączenia faz tekstur. W zasadzie pixel shadery powinno się wykorzystywać do bardziej skomplikowanych operacji, a prostsze przypadki pozostawić „staremu sposobowi".
- Cztery wersje pixel shaderów są w różnym stopniu obsługiwane przez różne rodzaje sprzętu. Obecnie najnowszą i oferującą największe możliwości, a jednocześnie najmniej rozpowszechnioną wersją jest wersja 1.4. Dowolny sprzęt obsługujący mechanizmy pixel shader obsługuje wersję 1.1.
- Pixel shadery działają w podobny sposób jak vertex shadery. Instrukcje zapisują i odczytują dane z i do rejestrów, a następnie przekazują ostateczne wyniki.
- W większości rejestrów przechowuje się dane o kolorach. Wyjątek stanowią rejestry tekstur. Rejestry tekstur przechowują dane o współrzędnych tekstur. W zależności od instrukcji można ich użyć jako próbek koloru lub jako danych wektorowych.
- Instrukcje `ps`, `def` oraz `phase` wykonują konfigurację shadera i nie zawierają się w ogólnej liczbie dozwolonych instrukcji w shaderze.
- Instrukcji arytmetycznych można użyć w celu wykonania operacji matematycznych dotyczących wartości kolorów. Zazwyczaj można użyć osobnych instrukcji dla wartości opisu kolorów oraz wartości opisu kanału alfa.
- Instrukcje adresowania tekstur oferują największe możliwości. Decydują one o sposobie interpretowania danych wejściowych opisujących tekstury. Instrukcji tych można użyć do załadowania wartości opisu kolorów oraz do wykonywania operacji na macierzach i wektorach na podstawie współrzędnych tekstur.

- Modyfikatory mogą służyć do modyfikowania instrukcji, mieszania rejestrów wejściowych oraz tworzenia masek dla rejestrów wynikowych. Dzięki modyfikatorom możliwe jest wykonywanie większej liczby działań, które są liczone jako mniejsza liczba instrukcji.
- Pixel shadery podlegają ostrzejszym ograniczeniom co do liczby instrukcji niż vertex shadery. Ograniczenia dotyczą nie tylko całkowitej liczby instrukcji, ale także liczby instrukcji określonych typów, a czasami poszczególnych instrukcji i modyfikatorów.
- Dzięki analizie możliwości urządzenia można sprawdzić, czy sprzęt obsługuje mechanizmy pixel shadery.
- Pixel shadery, podobnie jak vertex shadery należy poddać asemblacji, utworzyć i ustawić. Podczas tworzenia pixel shaderów deklaracje nie są potrzebne.
- Dla celów testowania zawsze można skorzystać z urządzenia referencyjnego, ale należy pamiętać, że uzyskana wydajność będzie bardzo niska.