Rozdział 5
Oświetlenie: cienie własne

Wersja z 2014-04-04

Trudno przecenić rolę światła w grafice 3D. Podobnie jak w kinie, to dzięki oświetleniu nasz umysł daje się nabierać i obraz widziany na płaskim ekranie interpretuje jako trójwymiarowy. Nawet najbardziej wymyślne trójwymiarowe modele bez oświetlenia są po prostu płaskie. W dużym stopniu, wspólnie z teksturowaniem, to oświetlenie odpowiedzialne jest za realizm wygenerowanej sceny. Wreszcie to cienie własne i rzucane są także jednym z tych elementów, które budują nastrój sceny.
W MonoGame zaimplementowany jest rozszerzony model oświetlenia Phonga, w którym oprócz światła otoczenia, światła rozproszonego i rozbłysku uwzględnione zostało również tzw. światło emisyjne. Klasa `BasicEffect`, z której w tym module będziemy nadal korzystać zawiera trzy predefiniowane źródła światła, których większość parametrów można modyfikować. Wśród tych parametrów nie ma jednak położenia, które jest ustalone. Pełną kontrolę nad źródłami światła uzyskamy dopiero implementując model Phonga w tworzonych samodzielnie efektach (moduły 11-13). W tych modułach założenia teoretyczne tego modelu zostaną przedstawione dokładniej. Natomiast w tym module skoncentrujemy się na użyciu gotowego rozwiązania oferowanego przez MonoGame.
Model Phonga nie wyczerpuje jednak zagadnienia oświetlenia w MonoGame. Światło to przecież nie tylko jasność powierzchni (cienie własne) i efekt rozbłysku, a również cienie rzucane na inne przedmioty. Ze światłem jako zjawiskiem fizycznym wiążą się również takie zagadnienia jak przezroczystość i mgła, które realizowane są w grafice 3D za pomocą mechanizmu mieszania kolorów. Kolejnym ważnym zagadnieniem jest inny pomysł zrealizowany przez Phonga, a mianowicie sposób imitowania gładkich zaokrąglonych powierzchni za pomocą niewielkiej ilości werteksów. Mam na myśli cieniowanie Phonga. Te wszystkie zagadnienia zostaną omówione w tym module.

Oświetlenie

Zanim zamontujemy na naszej scenie żarówkę lub reflektor powinniśmy wpierw przygotować wyświetlany na scenie model tak, aby zareagował na oświetlenie. Powinniśmy mianowicie zdefiniować **wektory normalne** – kolejną własność werteksów modelu. Wektor normalny zasadniczo określa kierunek prostopadły do powierzchni. Wektory normalne są wykorzystywane przez silniki graficzne 3D (OpenGL, Direct3D, MonoGame) do obliczania jasności powierzchni. Umożliwiają bowiem wyznaczenie wzajemnego położenia źródła światła i oświetlanej powierzchni. Mimo, że intuicja podpowiada nam, że wektor normalny powinien być własnością powierzchni (prymitywu), przypisany jest do werteksu. Jednak dzięki temu możliwe jest stosowanie technik uśredniania i interpolacji normalnych (o nich niżej), które pozwalają na zmniejszeni ilości werteksów używanych do opisu modelu. Definiowanie normalnych zacznijmy od prostopadłościanu, w przypadku którego ich kierunek jest oczywisty - wyznaczają go wersory kartezjańskiego układu współrzędnych. Potem przejdziemy do czworościanu, w którym tak prosto już nie jest.

Na warsztat weźmy projekt komponentu `Prostopadloscian` z poprzedniego modułu.

Definiowanie własnego formatu werteksów

Do obecnego zestawu własności werteksów, tj. do pozycji i koloru, chcielibyśmy dodać wektory normalne. Zestaw predefiniowanych typów werteksów nie obejmuje takiej możliwości podobnie, jak nie ma typu obejmującego czwórkę własności pozycji, koloru, normalnej i współrzędnej teksturowania. Jest tak ponieważ po włączeniu teksturowania kolor jest rzadko wykorzystywany. My jednak zdefiniujemy odpowiednie struktury, co da nam pretekst do zajrzenia do mechanizmu wiążącego atrybuty werteksu w shaderach z własnościami klasy opisującej werteks w MonoGame. Nowe typy werteksu umieścimy w osobnym pliku (np. *NoweTypyWerteksow.cs*); na razie zdefiniujemy werteks uzupełniający przesyłane dane o wektor normalny, a w rozdziale 7. dodamy jeszcze współrzędne teksturowania:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework;


namespace MojaDrugaGraMonoGame
{
    public struct VertexPositionNormalColor : IVertexType
    {
        public Vector3 Position;
        public Vector3 Normal;
        public Color Color;

        public static readonly VertexDeclaration VertexDeclaration = new VertexDeclaration(
            new VertexElement(0,VertexElementFormat.Vector3,
                              VertexElementUsage.Position, 0),
            new VertexElement(3 * sizeof(float), VertexElementFormat.Vector3,
                              VertexElementUsage.Normal, 0),
            new VertexElement(6 * sizeof(float), VertexElementFormat.Color,
                              VertexElementUsage.Color, 0)
        );

        VertexDeclaration IVertexType.VertexDeclaration
        {
            get
            {
                return VertexDeclaration;
            }
        }

        public VertexPositionNormalColor(Vector3 position, Vector3 normal, Color color)
        {
            this.Position = position;
            this.Normal = normal;
            this.Color = color;
        }
    }
}
```

Przyjrzyjmy się powyższej strukturze. Wyposażona jest w trzy pola mogące przechowywać własności werteksów: dwa trójelementowe wektory (`Vector3`), w którym przechowywane będzie położenie werteksu i wektor normalny oraz obiekt typu `Color` (z przestrzeni nazw `Microsoft.Xna.Framework.Graphics`, a nie z `System.Drawing`). Obowiązkowym elementem struktury opisującej werteks jest pole lub własność typu `VertexDeclaration`. Jej obecność wymusza implementowany przez strukturę interfejs `IVertexType`. Inicjując to pole musimy utworzyć tablicę obiektów typu `VertexElement` opisujących poszczególne atrybuty werteksu. W przypadku struktury `VertexPositionNormalColor` tablica ta zawiera trzy elementy opisujące kolejne własności werteksu przesyłane do strumienia oraz ich interpretację. Dla przykładu drugi element tej tablicy opisujący normalną zainicjowany poleceniem

```
new VertexElement(3 * sizeof(float), VertexElementFormat.Vector3,
                  VertexElementUsage.Normal, 0),
```

informuje, o tym, że 1) element należy przesłać do zerowego (domyślnego) strumienia, 2) że w „paczce" opisującej jeden werteks liczby opisujące wektor normalny zaczynają się od dwunastego bajtu (każda liczba `float` i `uint` zapisana jest w czterech bajtach), 3) w aplikacji MonoGame własność jest zapisywana w trójelementowym wektorze, co określa automatycznie jej wielkość, 4) wskazuje na domyślną metodę teselatora[1], 5) i wreszcie wskazuje na sens własności (pozycja, kolor, normalna, współrzędna teksturowania, głębokość, styczna itd.)[2] oraz 6) indeks interpretacji (używana przy nadawaniu wielu interpretacji).

Powyższa implementacja struktury werteksu różni się nieco od struktur zdefiniowanych w MonoGame. W tamtych, np. w `VertexPositionColor` (można ją obejrzeć dzięki poleceniu *Go To Definition* z menu kontekstowego edytora), a w strukturach zdefiniowane są w nadpisane metody `Equals`, `GetHashCode` i `ToStrings` oraz w operatory porównania (`==` i `!=`). Można to jednak uznać za różnice czysto kosmetyczne.

Definiowanie wektorów normalnych prostopadłościany

Teraz przejdźmy do klasy `Prostopadloscian` definiującej komponent rysujący prostopadłościan i zamieńmy wszystkie wystąpienia `VertexPositionColor` na `VertexPositionNormalColor` (najlepiej użyć automatycznej zmiany – *Ctrl+H*). Oznacza to zmianę w wielu miejscach konstruktora i w dwóch miejscach metody `Draw` (w deklaracji werteksu i w wywołaniu metody `gd.Vertices[0].SetSource`). W konstruktorze w momencie, w którym zmienimy polecenia inicjujące elementy tablicy werteksów będziemy musieli uzupełnić argumenty konstruktora o wektor normalny. Najprościej użyć stałych `Vector3.UnitX`, `Vector3.UnitY` i `Vector3.UnitZ` odpowiednio dla ścian prawej, górnej i przedniej. Do ścian lewej, dolnej i tylnej powinniśmy użyć tych wersorów ze znakiem minus. Po wszystkich zmianach konstruktor powinien wyglądać jak na poniższym listingu:

```
public Prostopadloscian(
    Game game, BasicEffect efekt,
    float dx, float dy, float dz,
    Color? kolor)
    : base(game)
{
    dx /= 2;
    dy /= 2;
    dz /= 2;

    gd = game.GraphicsDevice;
    this.efekt = (BasicEffect)efekt.Clone();

    Vector3[] punkty = new Vector3[8]
    {
        new Vector3(-dx, -dy, dz),
        new Vector3(dx, -dy, dz),
        new Vector3(dx, dy, dz),
        new Vector3(-dx, dy, dz),
        new Vector3(-dx, -dy, -dz),
        new Vector3(dx, -dy, -dz),
        new Vector3(dx, dy, -dz),
        new Vector3(-dx, dy, -dz)
    };

    Color kolor1 = kolor ?? Color.Cyan;
    Color kolor2 = kolor ?? Color.Magenta;
    Color kolor3 = kolor ?? Color.Yellow;
```

[1] Teselator to moduł karty graficznej odpowiedzialny za teselację (triangularyzację). Jest to proces grupowania przesłanej do karty listy werteksów opisujących płaszczyzny w trójki opisujące trójkąty. W nowoczesnych kartach graficznych teselator, obok vertex i pixel shadera może być programowany.

[2] Porównaj z semantyką w HLSL (moduł 11-13)

```
    VertexPositionNormalColor[] werteksy = new VertexPositionNormalColor[6 * 4]
    {
        //przednia ściana
        new VertexPositionNormalColor(punkty[3], Vector3.UnitZ, kolor1),
        new VertexPositionNormalColor(punkty[2], Vector3.UnitZ, kolor1),
        new VertexPositionNormalColor(punkty[0], Vector3.UnitZ, kolor1),
        new VertexPositionNormalColor(punkty[1], Vector3.UnitZ, kolor1),
        //tylna ściana
        new VertexPositionNormalColor(punkty[7], -Vector3.UnitZ, kolor1),
        new VertexPositionNormalColor(punkty[4], -Vector3.UnitZ, kolor1),
        new VertexPositionNormalColor(punkty[6], -Vector3.UnitZ, kolor1),
        new VertexPositionNormalColor(punkty[5], -Vector3.UnitZ, kolor1),

        //górna ściana
        new VertexPositionNormalColor(punkty[3], Vector3.UnitY, kolor2),
        new VertexPositionNormalColor(punkty[7], Vector3.UnitY, kolor2),
        new VertexPositionNormalColor(punkty[2], Vector3.UnitY, kolor2),
        new VertexPositionNormalColor(punkty[6], Vector3.UnitY, kolor2),
        //dolna ściana
        new VertexPositionNormalColor(punkty[0], -Vector3.UnitY, kolor2),
        new VertexPositionNormalColor(punkty[1], -Vector3.UnitY, kolor2),
        new VertexPositionNormalColor(punkty[4], -Vector3.UnitY, kolor2),
        new VertexPositionNormalColor(punkty[5], -Vector3.UnitY, kolor2),

        //lewa ściana
        new VertexPositionNormalColor(punkty[3], -Vector3.UnitX, kolor3),
        new VertexPositionNormalColor(punkty[0], -Vector3.UnitX, kolor3),
        new VertexPositionNormalColor(punkty[7], -Vector3.UnitX, kolor3),
        new VertexPositionNormalColor(punkty[4], -Vector3.UnitX, kolor3),
        //prawa ściana
        new VertexPositionNormalColor(punkty[1], Vector3.UnitX, kolor3),
        new VertexPositionNormalColor(punkty[2], Vector3.UnitX, kolor3),
        new VertexPositionNormalColor(punkty[5], Vector3.UnitX, kolor3),
        new VertexPositionNormalColor(punkty[6], Vector3.UnitX, kolor3)
    };

    buforWerteksow = new VertexBuffer(
        gd,
        VertexPositionNormalColor.VertexDeclaration,
        werteksy.Length,
        BufferUsage.WriteOnly);
        buforWerteksow.SetData<VertexPositionNormalColor>(werteksy);
}
```

Zmieniając typ werteksów w całym pliku *Prostopadloscian.cs* zmieniliśmy je także w metodzie `Prostopadloscian.Draw`. Warto jednak upewnić się, czy na pewno wszystko jest w niej w porządku, a więc czy nowy typ werteksu znalazł się w deklaracji i w wywołaniu metody `gd.Vertices[0].SetSource`:

```
public override void Draw(GameTime gameTime)
{
   gd.SetVertexBuffer(buforWerteksów);


   foreach (EffectPass pass in efekt.CurrentTechnique.Passes)
   {
      pass.Apply();

      for (int i = 0; i < 6; ++i)
          gd.DrawPrimitives(PrimitiveType.TriangleStrip, 4 * i, 2);

   }

   base.Draw(gameTime);
}
```

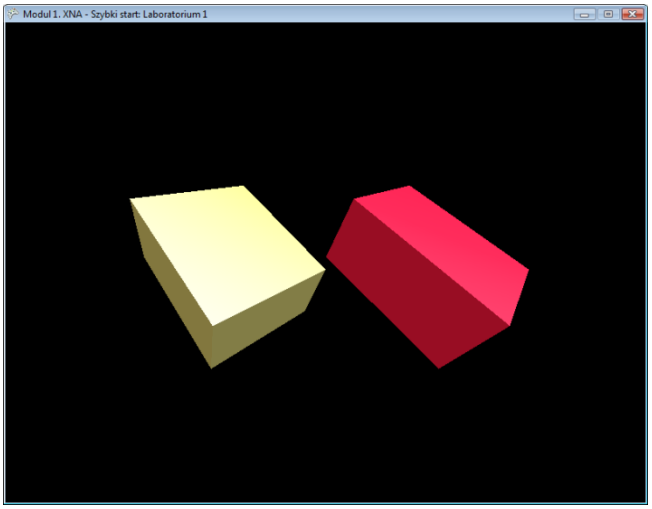
Oświetlenie domyślne

Po przygotowaniu modelu, możemy go oświetlić. Do tego użyjemy oświetlenia zaimplementowanego w klasie efektu `BasicEffect`. Zawiera ona trzy predefiniowane źródła światła, które można nawet w pewnym zakresie kontrolować (nie można jednak zmieniać ich

położenia). Plik efektu zawiera także parametry kontrolujące własności materiału. Należy bowiem zdawać sobie sprawę, że w grafice 3D (OpenGL, Direct3D, MonoGame) stosuje się dość realistyczne podejście do oświetlenia. Na ostateczny kolor modelu, jaki widzimy na ekranie wpływa zarówno kolor jego materiały, jak i kolor źródeł, którymi jest oświetlany. W MonoGame, podobnie jak np. w OpenGL, możemy zatem niezależnie ustalać składowe RGB źródła światła oraz współczynniki absorpcji poszczególnych składowych pochłanianych przez materiał. Współczynniki ustalamy osobno dla każdej składowej poszczególnych typów oświetlenia.
Na razie pozostawimy domyślne ustawienia oświetlenia bez zmian ograniczając się do ich aktywacji i obniżenia współczynnika reakcji materiału na światło otoczenia. To ostatnie jeżeli jest ustawione na maksimum, ukrywa wszystkie cienie (i zmarszczki). Wróćmy zatem do klasy gry `Game1` i do jej metody `Game1.Initialize` dodajmy dwie instrukcje:

```
efekt.EnableDefaultLighting();
efekt.AmbientLightColor = Color.Gray.ToVector3();
```

Należy je oczywiście umieścić po utworzeniu obiektu `efekt`. Aby pogłębić realizm zmieniłem także domyślne radosno-błękitne tło na czarne (argument metody `gd.Clear`, zob. rysunek 1).

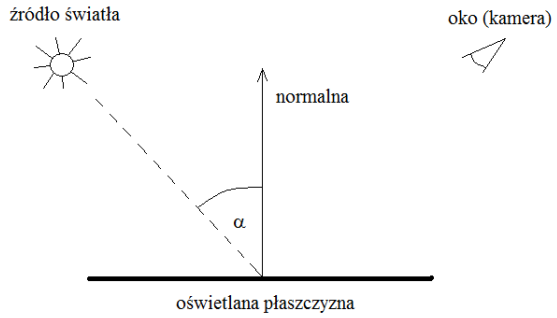


Rysunek 1. Dzięki oświetleniu widać, że obiekty są trójwymiarowe

Typy światła

Oświetlenie domyślne można w pewnym zakresie kontrolować. Możliwości te nie dorównują wprawdzie tym, jakie zyskujemy tworząc własne efekty w języku HLSL, ale wystarczą aby zrozumieć jak działa system oświetlenia w MonoGame.
W MonoGame mamy do czynienia z czterema typami światła: światło otoczenia, światło rozproszone, światło rozbłysku oraz światło emisyjne. Pierwsze trzy pochodzą z modelu Phonga stosowanym także w OpenGL, natomiast czwarte jest znane z Direct3D. W konsekwencji określając własności materiału możemy ustalić jak reaguje on na światło otoczenia, światło rozproszone itd. Natomiast definiując źródła światła określamy jedynie komponent światła rozproszonego i reflektora (światła rozbłysku). Tylko te dwa typy światła mają bowiem ustaloną pozycję (dzięki czemu można w ogóle mówić o ich źródle), a w przypadku reflektora także kierunek.

Światło rozproszone (*diffuse*) – imituje mleczną żarówkę. Posiada pozycję i rozjaśnia powierzchnie skierowane w kierunku źródła. Ich jasność jest proporcjonalna do cosinusa kąta α między ustalanym przy definiowaniu werteksów wektorem normalnym i wektorem wskazującym na źródło (wzór Lamberta). Nie zależy od położenia kamery (rysunek 2).

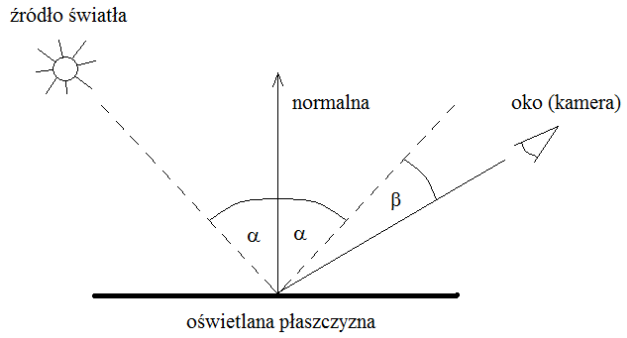


Rysunek 2. Jasność powierzchni w świetle rozproszonym nie zależy od położenia kamery

Powierzchnie nieoświetlone przez światło rozproszone są zupełnie czarne. To niezupełnie odpowiada naszemu codziennemu doświadczeniu. W rzeczywistości bowiem nie tylko żarówki (i gwiazdy) są źródłami światła, ale również wszystkie jasne przedmioty (i księżyc), które odbijają padające na nie światło. Ze względu na możliwości obliczeniowe w OpenGL i MonoGame obiekty oświetlane nie są jednak wtórnymi źródłami światła[3]. Zamiast tego stosuje się światło otoczenia.

Światło otoczenia (*ambient*) w jednakowy sposób rozświetla całą scenę (zob. planszę na następnej stronie). Nie ma źródła, ani kierunku. Zastosowane samodzielnie odpowiada oświetleniu uzyskiwanemu w dobrze oświetlonym pokoju z wieloma wtórnymi źródłami światła – gdy żaden przedmiot nie rzuca wyraźnych cieni.

Światło rozbłysku (*specular*, w malarstwie odpowiada mu tzw. blink) to odpowiednik „zajączka" oślepiającego kamerę lub oko, gdy promień światła odbitego od powierzchni skierowany jest wprost w obiektyw. W odróżnieniu od światła rozproszonego jasność tej składowej zależy nie tylko od położenia źródła światła i kierunku normalnej, ale także od położenia kamery. Ściśle rzecz biorąc zależy od podniesionego do *n*-tej potęgi kąta β między kierunkiem światła odbitego z punktowego źródła światła, a kierunkiem do kamery (rysunek 3). Wartość potęgi to moc rozbłysku – określa m.in. zakres w jakim efekt jest widoczny. Często modelem, na którym prezentowana jest ta składowa oświetlenia jest sfera, bowiem wartość potęgi *n* wpływa na wielkość widocznej na powierzchni sfery plamki światła (zob. *http://pl.wikipedia.org/wiki/Grafika:Phong_kule.jpg*).
Powierzchnia przejmuje kolor światła rozbłysku tym bardziej im mniejszy jest kąt odchylenia od kierunku światła odbitego.

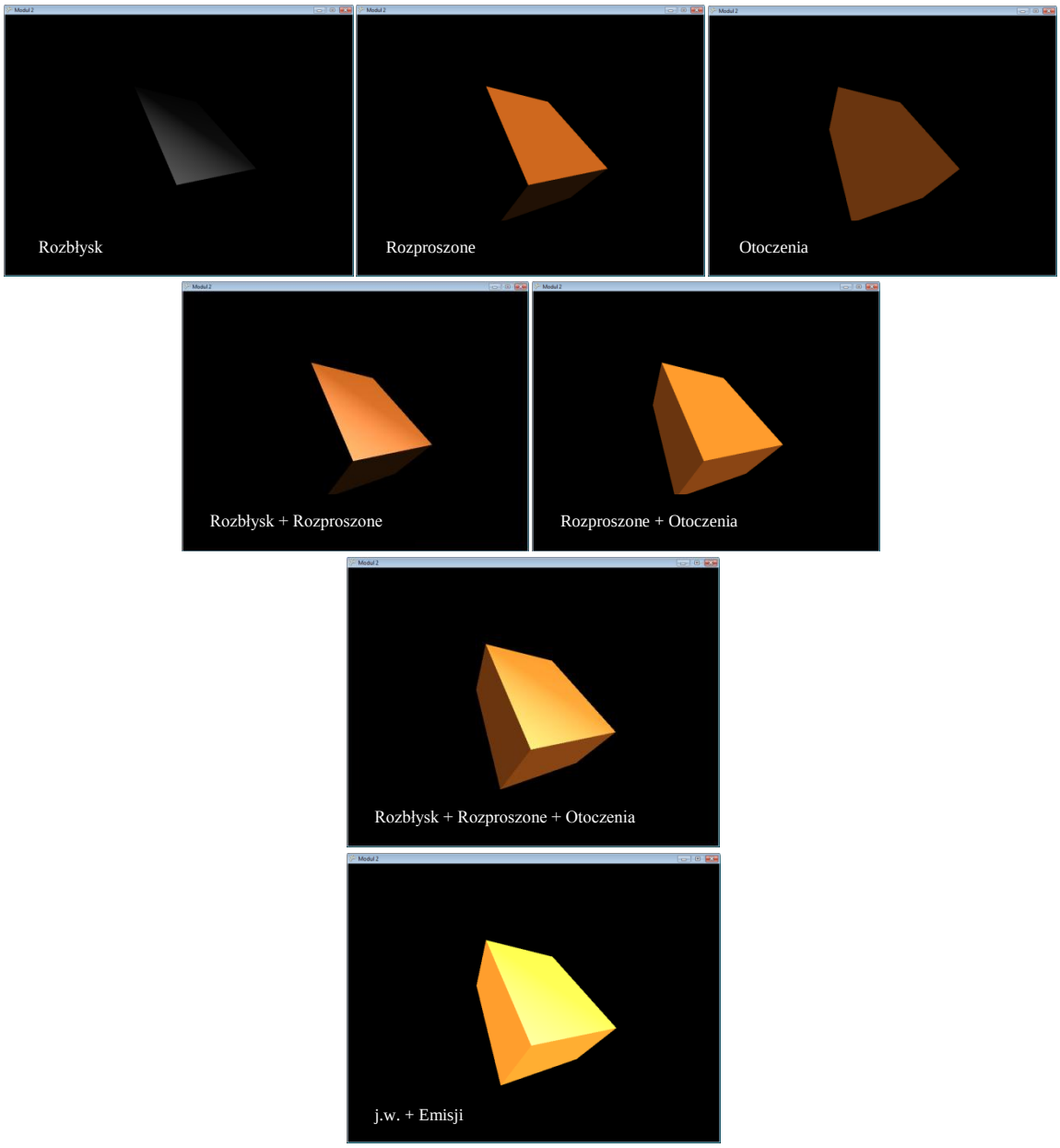


Rysunek 3. Rozbłysk zależy od kąta między promieniem odbitym i kierunkiem do kamery

Światło emisyjne (*emission*) to imitacja światła emitowanego przez przedmiot. Objawia się to jego lśnieniem, tak jakby przedmiot był źródłem światła. Należy jednak pamiętać, że efekt dotyczy tylko samego przedmiotu. Przypisanie bryle komponentu światła emisyjnego nie oznacza, że przedmiot ten oświetla płaszczyzny innych przedmiotów.

[3] Choć mówi się od pewnego czasu o uwzględnieniu elementów technik śledzenia promieni (ang. *ray tracing*) w grafice czasu rzeczywistego, to na razie projekty te nie są zrealizowane.

Bardziej wnikliwy opis modelu oświetlenia Phonga znajduje się w module 13, w którym opisujemy jego własną implementację przygotowaną w HLSL.



Rysunek 4. Plansza prezentująca typy oświetlenia w modelu Phonga

Eksperymenty z własnościami materiału i źródłami światła

W świetle domyślnym klasy `BasicEffect` zdefiniowane są trzy światła. W każdym z nich można zmienić kolor światła rozproszonego oraz kolor i moc światła reflektora odpowiedzialnego za rozbłysk. Niezależnie od tego możemy ustalić własności materiału dla światła otoczenia, światła emisji, światła rozproszonego i rozbłysku.

Przekonajmy się na własne oczy jak to wszystko działa. Zacznijmy od ustawienia wszystkich aborcyjnych własności materiałów na zero, co odpowiada przypisaniu im czarnego koloru. Wyłączmy również źródła światła 1 i 2 pozostawiając jedynie źródło światła o numerze 0, w którym włączamy komponent światła rozproszonego i ustawiamy go na maksimum (kolor biały), a składowe RGB rozbłysku na 50% (kolor szary).

```
//cialo doskonale czarne
efekt.AmbientLightColor = Color.Black.ToVector3();
efekt.DiffuseColor = Color.Black.ToVector3();
efekt.SpecularColor = Color.Black.ToVector3();
efekt.EmissiveColor = Color.Black.ToVector3();

//zrodla swiatla
efekt.DirectionalLight0.DiffuseColor = Color.White.ToVector3();
efekt.DirectionalLight0.SpecularColor = Color.Gray.ToVector3();
efekt.DirectionalLight1.DiffuseColor = Color.Black.ToVector3();
efekt.DirectionalLight1.SpecularColor = Color.Black.ToVector3();
efekt.DirectionalLight2.DiffuseColor = Color.Black.ToVector3();
efekt.DirectionalLight2.SpecularColor = Color.Black.ToVector3();
```

Po uruchomieniu aplikacji zobaczymy wielkie nic. Cała scena pogrążona będzie w mroku. Prostokąt nie odbija bowiem żadnego światła choć to jest włączone i znajduje się po prawej stronie sceny. Zwiększmy teraz nieco własności materiału odpowiedzialne za odbicie światła rozbłysku:

```
efekt.SpecularColor = Color.White.ToVector3();
efekt.SpecularPower = 16;
```

Pojawi się sam rozbłysk (lewy rysunek w górnym wierszu na planszy z poprzedniej strony). Tak jakbyśmy mieli do czynienia z zupełnie przezroczystym szkłem (ew. czarnym lśniącym monolitem) oświetlonym reflektorem. Zwiększmy teraz współczynniki odbicia światła rozproszonego:

```
efekt.DiffuseColor = Color.White.ToVector3();
```

Dzięki niemu prostopadłościan zacznie reagować na komponent światła rozproszonego źródła światła i oświetlone zostaną ściany skierowane na prawo (na planszy lewy rysunek w drugim wierszu). Lewa ściana pozostanie jednak zupełnie czarna. To nie odpowiada rzeczywistości, w której zazwyczaj światło dociera ze wszystkich kierunków dzięki wielokrotnym odbiciom i rozproszeniom na wszystkich przedmiotach w otoczeniu. Ten fakt odtwarza światło otoczenia. Jeżeli ustawimy i te parametry materiału, które odpowiadają za reakcję na światło otoczenia zobaczymy cały prostopadłościan z pełnym wrażeniem trójwymiarowości (na planszy lewy rysunek w trzecim wierszu).

```
efekt.AmbientLightColor = Color.Gray.ToVector3();
```

Możemy również dodać światło emisyjne imitujące fakt bycia źródłem światła (rysunek w dolnym wierszu):

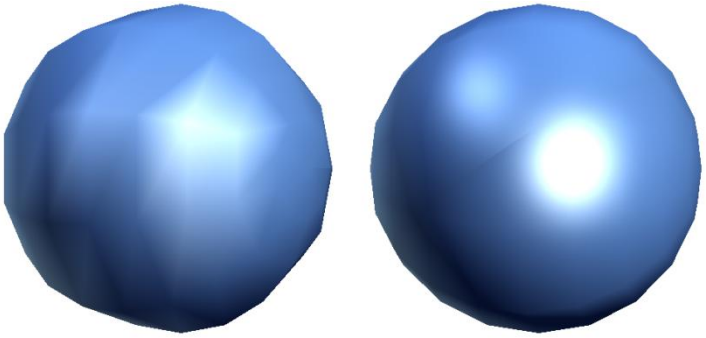
```
efekt.EmissiveColor = Color.White.ToVector3();
```

Uwaga! W świetle domyślnym włączenie samego komponentu materiału odpowiedzialnego za odbijanie światła otoczenia przy wyzerowanych współczynnikach odbijania światła rozproszonego

lub rozbłysku nie daje żadnego efektu. Rysunek z prawej strony górnego wiersza jest więc trochę oszukany – powstał przez wyłączenie komponentów rozproszonych źródeł światła.

Cieniowanie dla werteksów, czy dla pikseli?

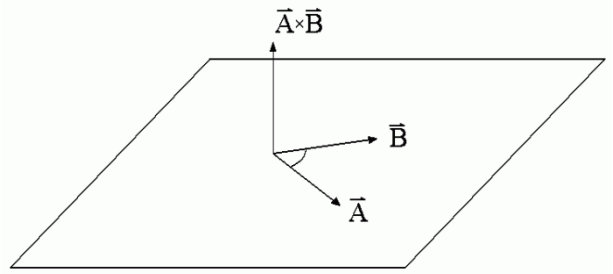
Klasa `BasicEffect` pozwala na ustalenie, czy obliczenia jasności mają być prowadzone w jednostce cieniowania (szaderze) werteksów, czy pikseli. Ta druga możliwość dostępna jest wyłącznie, gdy karta graficzna wspiera Pixer Shader Model w wersji 2.0. Cieniowanie na poziomie werteksów jest szybsze, z tego prostego powodu, że ilość werteksów definiujących aktorów jest mniejsza niż ilość pikseli obrazu. Jednak obliczanie cieniowania w szaderze pikseli daje znacznie lepsze efekty dla tej samej liczby werteksów definiujących gładką figurę (por. rysunek).
Różnica między dwoma możliwościami stanie się jaśniejsza po utworzeniu własnych szaderów w modułach 11-13. W przypadku shadera kryjącego się za klasą `BasicEffect` o tym, gdzie mają być przeprowadzane obliczenia decyduje własność `BasicEffect.PreferPerPixelLighting`. Jeżeli ustawiona jest na `true` i dysponujemy odpowiednim sprzętem obliczenia wykonywane będą w szaderze pikseli (zob. rysunek 5).



Rysunek 5. Oświetlenie zrealizowane w shaderze werteksów (z lewej) i shaderze pikseli (z prawej)

Iloczyn wektorowy

Ustalanie wektorów normalnych w przypadku prostopadłościanów jest łatwe – używamy wersorów kartezjańskiego układu współrzędnych. Załóżmy jednak, że chcemy obliczyć normalną do dowolnego trójkąta zadanego trzema punktami lub dwoma wektorami. Może to być na przykład jedna ze ścian czworościanu foremnego (tetraedru), który będziemy oświetlać w laboratorium podstawowym. Aby ustalić wektor normalny do takiego trójkąta, należy obliczyć unormowany iloczyn wektorowy dwóch wektorów napinających ów trójkąt (rysunek 6). Kolejność wektorów w iloczynie jest ważna – podobnie jak w przypadku nawijania. Normalna do powierzchni powinna być bowiem skierowana na zewnątrz bryły tj. do przodu trójkąta. Zatem obliczając iloczyn wektorowy, można to łatwo zrobić korzystając z metody `Vector3.Cross`, pamiętajmy o regule śruby prawoskrętnej.



Rysunek 6. Iloczyn wektorowy

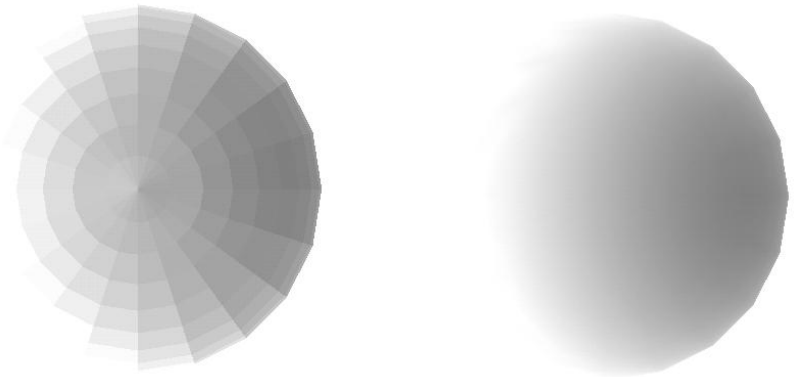
Uśrednianie i interpolacja normalnych

Wprowadzając pojęcie wektora normalnego jako własności werteksu przywołałem termin uśredniania i interpolacji normalnych. Tej techniki dotyczyć będą dwa z zadań laboratorium rozszerzonego.

Wyobraźmy sobie bryłę zbudowaną z kwadratów i przypominającą sferę. Jeżeli werteksy każdego kwadratu owej bryły mają przypisaną jedną wspólną normalną, która jest rzeczywiście prostopadła do płaszczyzny każdego z tych kwadratów, to jasność ustalana jest dla całego kwadratu i przez to są one wyraźnie widoczne. Mamy wówczas do czynienia z **cieniowaniem płaskim** (rysunek poniżej, z lewej strony).

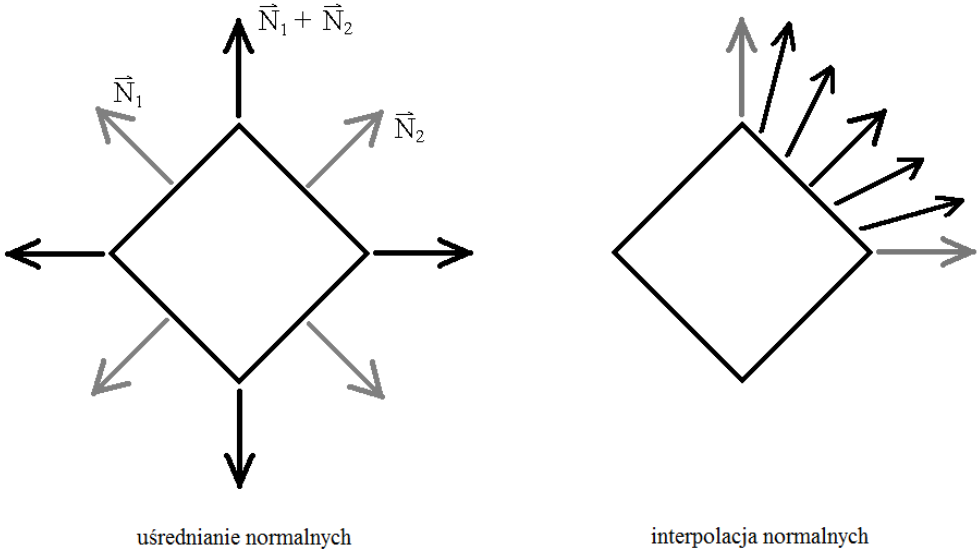
Zsumujmy teraz normalne czterech sąsiadujących kwadratów i ową średnią, po unormowaniu, przypiszmy do tego werteksu, który jest ich wspólnych wierzchołkiem. Po tej operacji każdy kwadrat, z których zbudowana jest sfera wyposażony będzie w cztery różne normalne.

W przypadku sfery jego kierunek można ustalić prościej – wektor normalny jest przedłużeniem promienia.



Rysunek 7. Uśrednianie i interpolacja normalnych

Łatwo to zrozumień na przykładzie figury płaskiej np. kwadratu (rysunek 7, lewy). Proces ten nazywa się **uśrednianiem normalnych**. Dalszy etap to **interpolacja normalnych** (por. rysunek 8). Podobnie, jak w przypadku kolorów, także normalne mogą być bowiem „cieniowane” (rysunek 7, prawy). Powoduje to osobne obliczanie jasności światła rozproszonego i rozbłysku dla każdego punktu prymitywu i stopniową zmianę ich jasności (rysunek powyżej, prawy). W efekcie kanciasta sfera zmienia się w sferę okrągłą bez zwiększenia ilości opisujących ją punktów. Proces ten nazywamy **cieniowaniem Phonga**. Oszczędność jest zresztą podwójna. Po pierwsze dzięki cieniowaniu sfera dobrze wygląda już przy stosunkowo niewielkiej liczbie kwadratów, a po drugie zauważmy, że w każdym wierzchołku nie ma już czterech werteksów z różnymi normalnymi, a jeden, w którym jest wspólna uśredniona normalna.

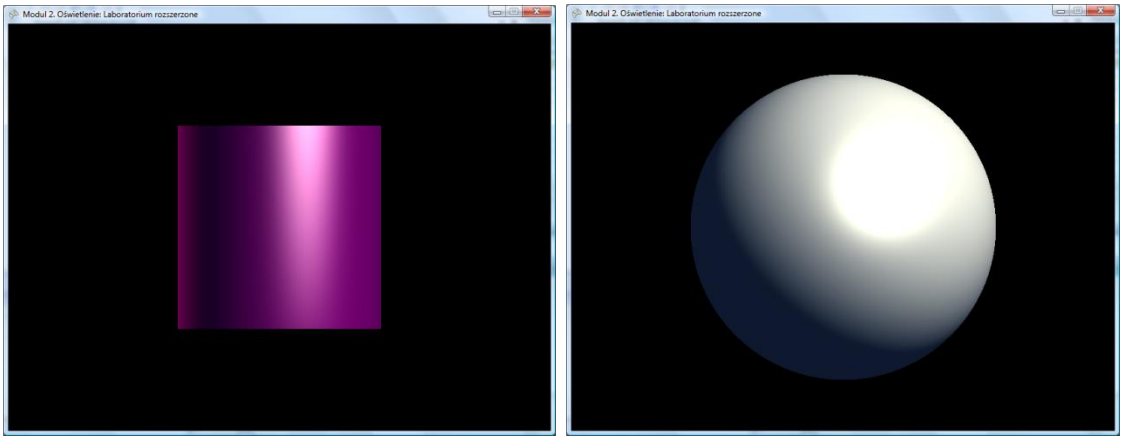


Rysunek 8. Schematyczne przedstawienie uśredniania i interpolacji normalnych

Zadania

Zadanie 1

Po kursie, który przeszedłeś na temat oświetlenia i uśredniania normalnych czujesz się na tyle doświadczony, aby zabrać się za zdefiniowanie normalnych w przypadku kwadryk. Zmodyfikuj istniejącą klasę `Kwadryki` (por. zadanie rozszerzone z modułu pierwszego) tak, aby dla każdej pary werteksów określających kolejny czworokąt (dwa trójkąty) w paśmie trójkątów zdefiniowana została normalna uśredniająca wartości wersorów prostopadłych do tych czworokątów dla dwóch sąsiednich czworokątów.



Podpowiedź: Tak zdefiniowane normalne uwzględniają uśrednianie normalnych w obrębie pasma (`PrimitiveType.TriangleStrip`). Ponieważ większość kwadryk (walec, stożek, dysk) zbudowane są z pojedynczych pasm jest to rozwiązanie w pełni wystarczające. Jednak nie w przypadku sfery, w której należy także dodać uśrednianie po czworokątów z sąsiednich pasm. Można też do sprawy podejść prościej przedefiniowując normalne tak, żeby były równoległe do promienia w punkcie, w którym definiowany jest werteks.

Zadanie 2

W projekcie *2C2-0 Oswietlenie (lab. rozsz.) - Dwudziestoscian* znajduje się komponent rysujący dwudziestościan foremny. Jego ściany są trójkątami foremnymi. Zdefiniuj wektory normalne do każdego werteksu tak, aby był on uśrednieniem pięciu normalnych określonych dla pięciu trójkątów „korzystających” z tego wierzchołka.

Podpowiedź:
Czy musisz obliczać wektory prostopadłe do poszczególnych ścian ikosaedru, żeby policzyć normalne dla werteksów?
Czy można tak zmodyfikować typ werteksu, aby w ogóle uniknąć definiowania normalnych?