Rozdział 5
Oświetlenie: cienie własne i rzucane. Mieszanie kolorów

Wersja z 2014-03-29

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 XNA 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 XNA.
Model Phonga nie wyczerpuje jednak zagadnienia oświetlenia w XNA. 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.

Podstawy teoretyczne

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, XNA) 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*); będziemy ich używać w kolejnych projektach.

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

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

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

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

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

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

   public struct VertexPositionColorNormalTexture : IVertexType
   {
      public Vector3 Position;
      public Color Color;
      public Vector3 Normal;
      public Vector2 TextureCoordinate;

      public static readonly VertexDeclaration VertexDeclaration = new VertexDeclaration(
         new VertexElement(0,VertexElementFormat.Vector3,VertexElementUsage.Position,0),
         new VertexElement(sizeof(float)*3,VertexElementFormat.Color,
                           VertexElementUsage.Color,0),
         new VertexElement(sizeof(float)*3+sizeof(uint),VertexElementFormat.Vector3,
                           VertexElementUsage.Normal,0),
         new VertexElement(sizeof(float)*6+sizeof(uint),VertexElementFormat.Vector2,
                           VertexElementUsage.TextureCoordinate,0)
      );

      VertexDeclaration IVertexType.VertexDeclaration
```

```
        {
            get
            {
                return VertexDeclaration;
            }
        }

        public VertexPositionColorNormalTexture(Vector3 position, Color color,
                                                Vector3 normal, Vector2 textureCoordinate)
        {
            this.Position = position;
            this.Color = color;
            this.Normal = normal;
            this.TextureCoordinate = textureCoordinate;
        }
    }
}
```

Przyjrzyjmy się pierwszej z powyższych struktur. Wyposażona jest w trzy pola mogące przechowywać własności werteksów: trójelementowy wektor (`Vector3`), w którym przechowywane będzie położenie werteksu, obiekt typu `Color` (z przestrzeni nazw `Microsoft.Xna.Framework.Graphics`, a nie z `System.Drawing`) oraz kolejny trójelementowy wektor przechowujący współrzędne normalnej. Obowiązkowym elementem struktury opisującej werteks jest jej pole lub własność informująca o rozmiarze werteksu po zapisaniu do strumienia werteksów. W naszym przypadku będzie to sześć czterobajtowych liczb typu `float` (współrzędne obu wektorów) oraz liczba typu `uint`, w której zapisywane są cztery bajty kanałów ARGB koloru. Werteks uwzględniający wektor normalny będzie zatem zajmował dwadzieścia bajtów w strumieniu wektorów lub w buforze werteksów. Struktura wyposażona jest w konstruktor, który ułatwia jej inicjację. Najważniejszym elementem jest jednak pole `VertexDeclaration`. Inicjując to pole musimy utworzyć tablicę obiektów typu `VertexElement` opisujących poszczególne atrybuty werteksu. W przypadku struktury `VertexPositionColorNormal` tablica ta zawiera trzy elementy opisujące kolejne własności werteksu przesyłane do strumienia oraz ich interpretację. Dla przykładu trzeci element tej tablicy opisujący normalną zainicjowany poleceniem

```
new VertexElement(sizeof(float)*3+sizeof(uint),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 szesnastego bajtu (każda liczba `float` i `uint` zapisana jest w czterech bajtach), 3) w aplikacji XNA 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).

Druga struktura jest bardzo podobna, poza tym, że oprócz pola przeznaczonego do przechowywania normalnych dodane zostało również pole współrzędnych teksturowania.

Powyższe implementacje różnią się nieco od struktur zdefiniowanych w XNA. 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`,

[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)

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 VertexPositionColorNormal (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 == null) ? Color.Cyan : (Color)kolor;
   Color kolor2 = (kolor == null) ? Color.Magenta : (Color)kolor;
   Color kolor3 = (kolor == null) ? Color.Yellow : (Color)kolor;

   VertexPositionColorNormal[] werteksy = new VertexPositionColorNormal[6 * 4]{
      //przednia sciana
      new VertexPositionColorNormal(punkty[3], kolor1, Vector3.UnitZ),
      new VertexPositionColorNormal(punkty[2], kolor1, Vector3.UnitZ),
      new VertexPositionColorNormal(punkty[0], kolor1, Vector3.UnitZ),
      new VertexPositionColorNormal(punkty[1], kolor1, Vector3.UnitZ),

      //tylnia sciana
      new VertexPositionColorNormal(punkty[7], kolor1, -Vector3.UnitZ),
      new VertexPositionColorNormal(punkty[4], kolor1, -Vector3.UnitZ),
      new VertexPositionColorNormal(punkty[6], kolor1, -Vector3.UnitZ),
      new VertexPositionColorNormal(punkty[5], kolor1, -Vector3.UnitZ),

      //gorna sciana
      new VertexPositionColorNormal(punkty[3], kolor2, Vector3.UnitY),
      new VertexPositionColorNormal(punkty[7], kolor2, Vector3.UnitY),
      new VertexPositionColorNormal(punkty[2], kolor2, Vector3.UnitY),
      new VertexPositionColorNormal(punkty[6], kolor2, Vector3.UnitY),

      //dolna sciana
      new VertexPositionColorNormal(punkty[0], kolor2, -Vector3.UnitY),
      new VertexPositionColorNormal(punkty[1], kolor2, -Vector3.UnitY),
      new VertexPositionColorNormal(punkty[4], kolor2, -Vector3.UnitY),
      new VertexPositionColorNormal(punkty[5], kolor2, -Vector3.UnitY),

      //lewa sciana
      new VertexPositionColorNormal(punkty[3], kolor3, -Vector3.UnitX),
      new VertexPositionColorNormal(punkty[0], kolor3, -Vector3.UnitX),
      new VertexPositionColorNormal(punkty[7], kolor3, -Vector3.UnitX),
```

```
            new VertexPositionColorNormal(punkty[4], kolor3, -Vector3.UnitX),

            //prawa sciana
            new VertexPositionColorNormal(punkty[1], kolor3, Vector3.UnitX),
            new VertexPositionColorNormal(punkty[2], kolor3, Vector3.UnitX),
            new VertexPositionColorNormal(punkty[5], kolor3, Vector3.UnitX),
            new VertexPositionColorNormal(punkty[6], kolor3, Vector3.UnitX)};

        buforWerteksow = new VertexBuffer(
            gd, VertexPositionColorNormal.VertexDeclaration,
            werteksy.Count() * VertexPositionColorNormal.SizeInBytes,
            BufferUsage.WriteOnly);
        buforWerteksow.SetData<VertexPositionColorNormal>(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);
}
```

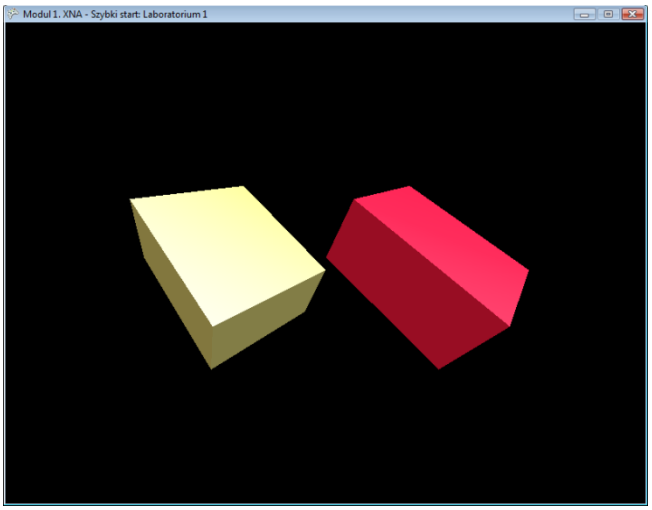
Oświetlenie domyślne

Po przygotowaniu modelu, możemy go oświetlić. Bez umieszczenia na scenie źródeł światła nie zobaczymy, czy zdefiniowane przed chwilą wektory normalne są poprawne. Z tym jest jednak problem – definiowanie własnych źródeł światła możliwe jest wyłącznie na poziomie języka HLSL (język programowania shaderów) i wiąże się z przygotowywaniem pliku efektu. Tym zagadnieniem zajmiemy się dopiero w modułach 9 i 10. Na szczęście twórcy XNA pomyśleli o łagodnych przejściu przygotowując klasę `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, XNA) 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 XNA, 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).



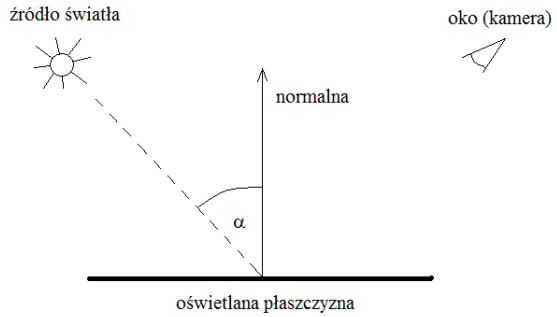
Rysunek 1. Dzięki oświetleniu widać, że obiekty są trójwymiarowe

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 XNA.

W XNA 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).



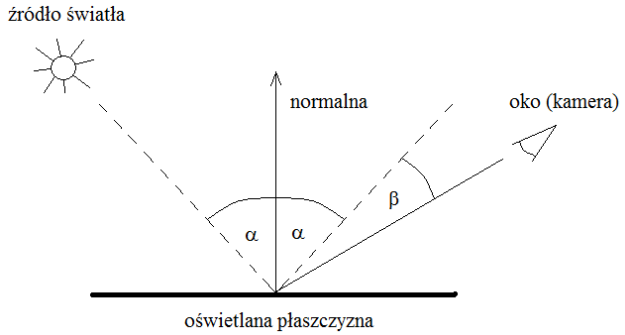
Rysunek 2. Jasność powierzchni w świetle rozproszonym nie zależy od położenia kamery

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 XNA 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.

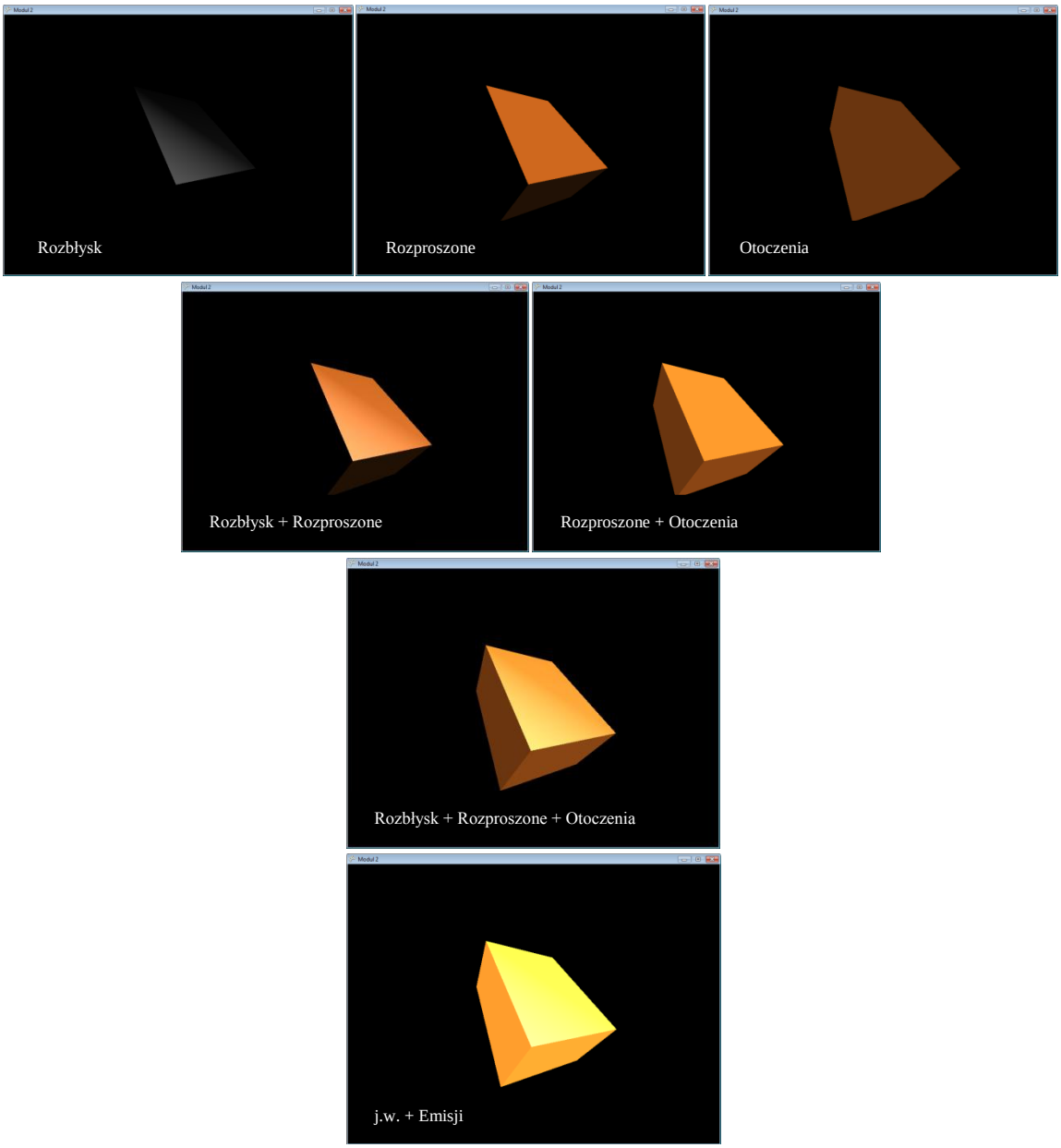


Rysunek 3. Rozbłysk zależy od kąta między promieniem odbitym i kierunkiem do kamery

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.

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

[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.



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):

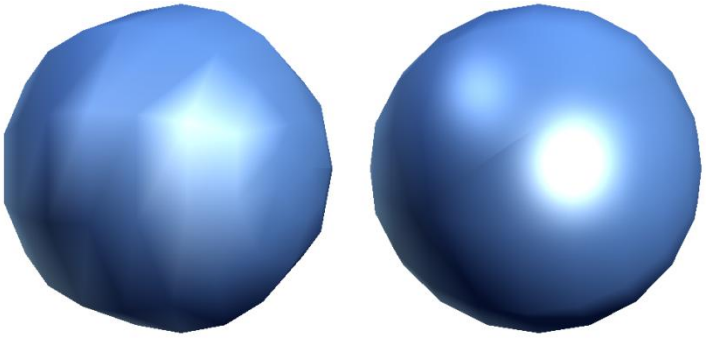
```
efekt.EmissiveColor = Color.White.ToVector3();
```

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?

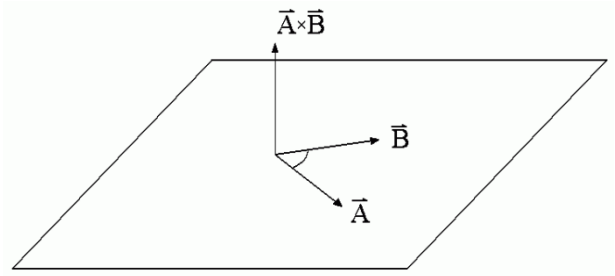
Klasa `BasicEffect` pozwala na ustalenie, czy obliczenia jasności mają być prowadzone w jednostce cieniowania (szaderze) werteksów, czy pikseli. Ta druga możliwość dostępna jest wyłącznie, gdy karta graficzna wspiera Pixer Shader Model w wersji 2.0. Cieniowanie na poziomie werteksów jest szybsze, z tego 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 shader 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

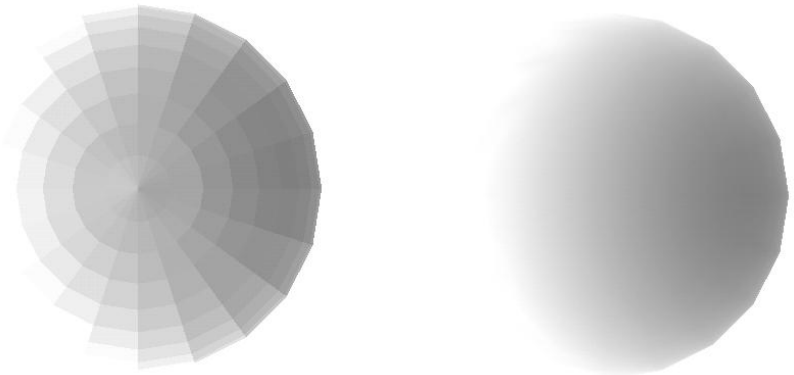
Uśrednianie i interpolacja normalnych

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).

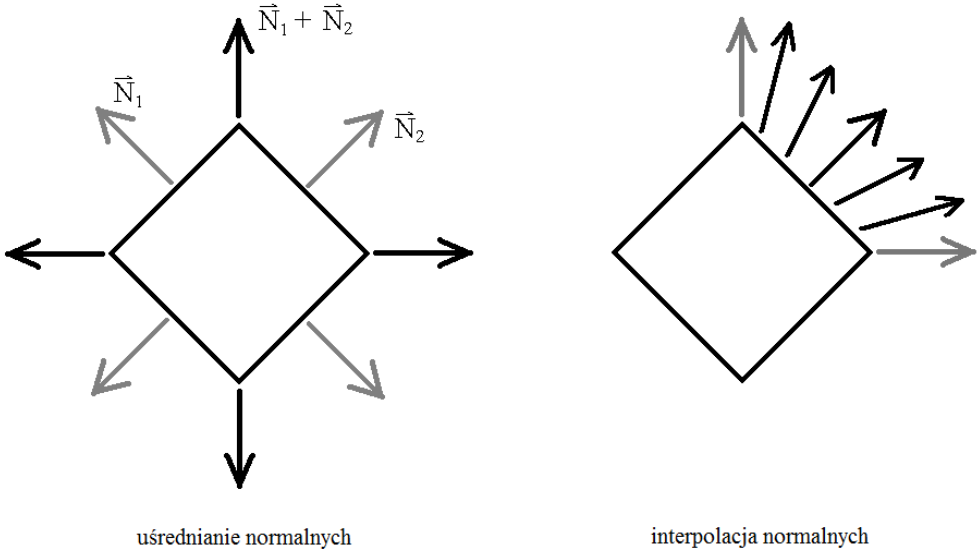
Zsumujmy teraz normalne czterech sąsiadujących kwadratów i ową średnią, po unormowaniu, przypiszmy do tego werteksu, który jest ich wspólnych wierzchołkiem. Po tej operacji każdy kwadrat, z których zbudowana jest sfera wyposażony będzie w cztery różne normalne.

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

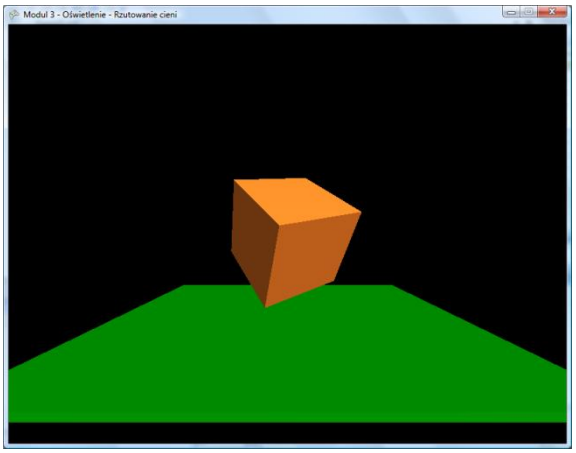
Rzutowanie cieni

Cienie własne mają kluczowe znaczenie dla postrzegania obiektów jako trójwymiarowych brył. Nie zapominajmy jednak, że cieniom własnym towarzyszą cienie rzucane na podłoże i inne przedmioty. Ich obecność pomaga między innymi lepiej rozpoznawać położenie i orientację obiektów. O ile cienie własne, czyli różna jasność ścian bryły w zależności od ich orientacji względem źródeł światła, są obsługiwane zarówno przez biblioteki XNA, Direct3D czy OpenGL, jak i wspierane sprzętowo, to cienie rzucane pozostawione są całkowicie w gestii programisty. W tym sensie dalsza część rozdziału nie ma nic wspólnego z systemem oświetlenia XNA, o którym mówiliśmy do tej pory.

Jest kilka algorytmów pozwalających na generowanie cieni. Wśród nich najpopularniejsze są trzy: rzutowanie cieni (ang. *projected shadows* lub *planar shadows*), cienie objętościowe (ang. *volumetric shadows*) i odwzorowywanie cieni (ang. *shadow mapping*). Pierwsza technika jest dość prosta, dwie pozostałe można zaliczyć do zaawansowanych. Skupmy się zatem na rzutowaniu cieni. Na czym ono polega?

Cień jest obszarem, do którego nie dochodzi światło ze źródła światła. „Analizując zagadnienie z punktu widzenia geometrii cień jest rzutem obiektu przysłaniającego światło na obiekt zasłaniany. W przypadku punktowego źródła światła położonego w stosunkowo niedużej odległości od obiektu mamy do czynienia z rzutem perspektywicznym. Oddalając jego pozycję aż do nieskończoności (np. w przypadku światła słonecznego) promienie stają się równoległe, a więc rzut staje się równoległy. Oba przypadki można przedstawić w jednolity sposób we współrzędnych jednorodnych.
Idea metody rzutowania opiera się na wykorzystaniu macierzy przekształcenia dokonującej rzutowania obiektu na płaszczyznę przyjmującą cień. Dzięki temu w prosty sposób można wykorzystać te same współrzędne wierzchołków modelu (we współrzędnych modelu) do jego rysowania jak również do wygenerowania jego cienia.” (R. Płoszajczak, praca mgr)
Biblioteka XNA pozwala uzyskać macierz rzutowania równoległego (tzn. odpowiadającego nieskończenie odległemu źródłu światła) na dowolną płaszczyznę korzystając z metody `Matrix.CreateShadows`. Jej dwoma argumentami jest wektor wskazujący kierunek do źródła światła oraz płaszczyzna, na którą rzutowanie ma być przeprowadzane. Jeżeli macierz świata zostanie pomnożona przez tą macierz, wszystkie punkty trójwymiarowej bryły znajdą się na tej płaszczyźnie – tym samym bryła zostanie „spłaszczona”. Wystarczy tylko użyć do jej pomalowania czerni, aby uzyskać cień.

W katalogu ***** jest projekt z umieszczonym na scenie sześcianem oraz powierzchnią imitującą podłoże (rysunek 9). Jest to niemal ten sam projekt, który przygotowaliśmy wcześniej ucząc się ustawiania oświetlenia. Wyłączone jest jednak światło emisji sześcianu.



Rysunek 9. Układ, na którym będziemy ćwiczyć rzutowanie cieni

1. Zacznijmy od narysowania drugiego sześcianu, którego macierz świata zostanie pomnożona przez macierz rzutowania na nasze podłoże. Definiujemy dodatkowe pole – referencję do prostopadłościanu:

```
Prostopadloscian cienSzescianu;
```

2. W metodzie `Game1.Initialize` tworzymy nowy prostopadłościan (zachowujemy na razie jego oryginalny kolor):

```
cieńSześcianu = new Prostopadloscian(this, efekt, 1.5f, 1.5f, 1.5f, Color.DarkGreen);
this.Components.Add(cieńSześcianu);
```

3. Macierz świata sześcianu pełniącego rolę cienia zawierać będzie macierz rzutowania na płaszczyznę. Taka macierz jest w XNA udostępniana przez metodę `Matrix.CreateShadow`. Niestety w MonoGame 3 takiej metody nie znajdziemy! Ale nie szkodzi, samodzielnie napiszemy analogiczną metodę w klasie `Game1`:

```
Matrix CreateShadowMatrix(Vector3 lightDirection, Plane plane)
{
   Vector4 N = new Vector4(plane.Normal, -plane.D);
   Vector4 L = new Vector4(lightDirection, 1);
   float alfa = Vector4.Dot(N, L);
   Matrix m = new Matrix();
   m.M11 = alfa - N.X * L.X;
   m.M22 = alfa - N.Y * L.Y;
   m.M33 = alfa - N.Z * L.Z;
   m.M44 = alfa - N.W * L.W;
   m.M21 = -N.Y * L.X;
   m.M31 = -N.Z * L.X;
   m.M41 = -N.W * L.X;
   m.M12 = -N.X * L.Y;
   m.M32 = -N.Z * L.Y;
   m.M42 = -N.W * L.Y;
   m.M13 = -N.X * L.Z;
   m.M23 = -N.Y * L.Z;
   m.M43 = -N.W * L.Z;
   m.M14 = -N.X * L.W;
   m.M24 = -N.Y * L.W;
   m.M34 = -N.Z * L.W;
   return m;
}
```

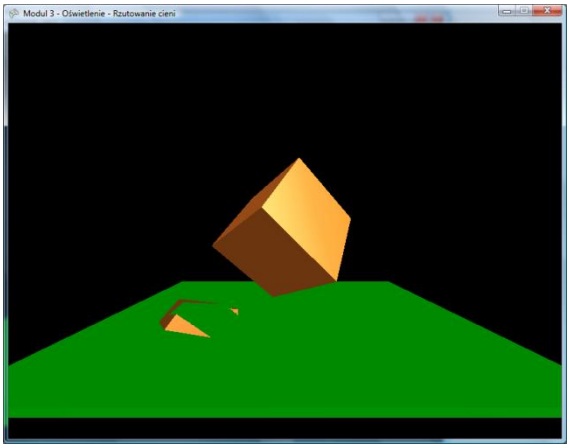
4. Ponieważ oświetlenie jest nieruchome, możemy zdefiniować stałą macierz rzutowania jako pole klasy `Game1` (stała `d` opisuje odległość od środka układu współrzędnych do górnej płaszczyzny prostopadłościanu pełniącego rolę podłoża):

```
const float d = 2 - 0.1f;
static Matrix macierzRzutowaniaNaPodłoże =
   CreateShadowMatrix(new Vector3(3, 5, 3), new Plane(Vector3.Up, d));
```

5. Dysponując tą metodą, w metodzie `Game1.Update` przypisujemy sześcianowi „obracającą się" macierz świata pomnożoną dodatkowo przez macierz rzutowania na podłoże:

```
sześcian.World *=
   Matrix.CreateRotationX(gameTime.ElapsedGameTime.Milliseconds / 1000.0f) *
   Matrix.CreateRotationY(gameTime.ElapsedGameTime.Milliseconds / 1000.0f);
cieńSześcianu.World = sześcian.World * macierzRzutowaniaNaPodłoże;
```

6. Płaszczyzna wskazana w macierzy rzutowania (obiekt typu `Plane`) została utworzona przez wskazanie wektora normalnego (wersor skierowany do góry) oraz odległości płaszczyzny od początku układu współrzędnych. Jeżeli prostopadłościan pełniący rolę podłoża jest obniżony o 1.75, a jego grubość równa jest 0.1, to górna płaszczyzna podłoża jest na wysokości -1.7. I właśnie taką wartość podałem w konstruktorze klasy `Plane`. Efekt widoczny jest na rysunku 10.



Rysunek 10. Drugi sześcian rysowany dokładnie na płaszczyźnie podłoża

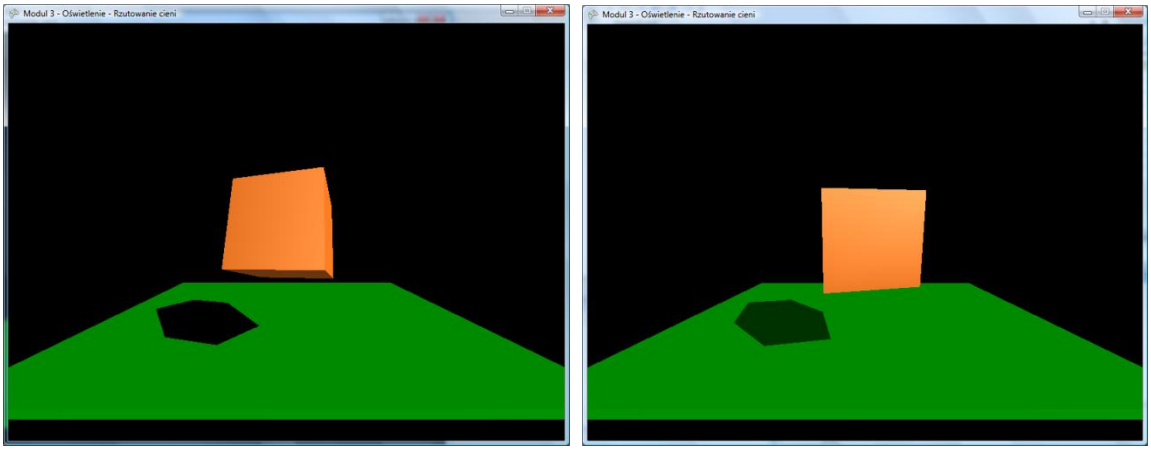
7. Wyraźnie widoczny jest błąd – dwa obiekty rysowane są na tej samej płaszczyźnie i trudno przewidzieć, który będzie widoczny, a który zostanie przesłonięty. Aby tego uniknąć warto wyłączyć bufor głębi na czas rysowania sceny. Jednak aby nie modyfikować komponentu `Prostopadloscian` możemy też zastosować prosty trik – możemy przesunąć płaszczyznę rzutowania minimalnie ponad płaszczyznę podłoża.

```
static Matrix macierzRzutowaniaNaPodłoże =
   CreateShadowMatrix(new Vector3(3, 5, 3), new Plane(Vector3.Up, d - 0.00001f));
```

8. Kolejną rzeczą, którą należy się zająć jest kolor cienia. Może zależeć od koloru podłoża, ale może być po prostu czarny – wszystko zależy od efektu jaki chcemy uzyskać (por. rysunki 11). Warto utworzyć dla obiektu pełniącego rolę cienia osobny efekt, w którym nie włączymy oświetlenia – cień powinien być jednolity. W tym celu do metody `Game1.Initialize` dodajmy następujące instrukcje:

```
BasicEffect efektCienia = (BasicEffect)efekt.Clone();
efektCienia.AmbientLightColor = Color.Black.ToVector3();
efektCienia.DiffuseColor = Color.White.ToVector3();
```

```
efektCienia.SpecularColor = Color.Black.ToVector3();
cieńSześcianu = new Prostopadloscian(this, efektCienia, 1.5f, 1.5f, 1.5f, Color.DarkGreen);
this.Components.Add(cieńSześcianu);
```



Rysunek 11. Całkowicie czarny cień (z lewej) i półcień z kolorem dopasowanym do podłoża

Zaletą generowania cieni za pomocą metody rzutowania jest jej prostota i łatwość implementacji. Wadą – to że nadaje się jedynie do tworzenia cieni na jednej, dużej płaszczyźnie. Cień jest rzeczywistym obiektem rysowanym w przestrzeni sceny (nawet jeżeli spłaszczonym) i trudno myśleć o jego przycinaniu lub zaginaniu, jeżeli podłoże miałoby skomplikowany kształt.

Mieszanie kolorów

Model Phonga imituje zjawiska odbijania światła od obiektów znajdujących się na scenie i rozpraszania go. Są jednak przedmioty, które nie tylko odbijają światło, ale również przepuszczają je. Innymi słowy są częściowo przezroczyste. Z drugiej strony powietrze, którego obecność jest zwykle ignorowana, może nie być doskonale przezroczysta. Przykładem jest na przykład zamglony poranek. Oba te zjawiska: przezroczystości i mgły można w grafice 3D imitować za pomocą techniki nazywanej mieszaniem kolorów.

[---KONIEC---]

Przezroczystość

Wczytajmy projekt z katalogu *3A5-1 MojaDrugaGraXNA (teoria - Mieszanie kolorow)*. Zdefiniowane są w nim dwa trójkąty, których werteksy o następujących punktach:

```
static Color kolor1 = Color.Red;
static Color kolor2 = Color.White;
private VertexPositionColor[] werteksy = new VertexPositionColor[6]{
   //tylni trojkat (czerwony)
   new VertexPositionColor(new Vector3(1, -1, -0.1f), kolor1),
   new VertexPositionColor(new Vector3(-1, -1, -0.1f), kolor1),
   new VertexPositionColor(new Vector3(0, 1, -0.1f), kolor1),
   //przedni trojkat (bialy)
   new VertexPositionColor(new Vector3(1, -1, 0.1f), kolor2),
   new VertexPositionColor(new Vector3(-1, -1, 0.1f), kolor2),
   new VertexPositionColor(new Vector3(0, 1, 0.1f), kolor2)
   };
```

Zdefiniowane jest pole `efekt`, ale bez włączania domyślnego oświetlenia. Oba trójkąty obracane są dzięki mnożeniu w metodzie `Game1.Update` macierzy świata przez macierz obrotu wokół osi OY:

```
efekt.World *= Matrix.CreateRotationY(gameTime.ElapsedGameTime.Milliseconds / 1000.0f);
```

Aby trójkąty były widoczne cały czas podczas obrotu wyłączony jest mechanizm usuwania tylnych powierzchni. Oba trójkąty rysowane są dwoma osobnymi wywołaniami metody `gd.DrawUserPrimitives` – będziemy później zmieniać ich kolejność.

Umieśćmy teraz w metodzie `Game1.Draw` polecenia włączające mechanizm mieszania kolorów (zob. listing poniżej).

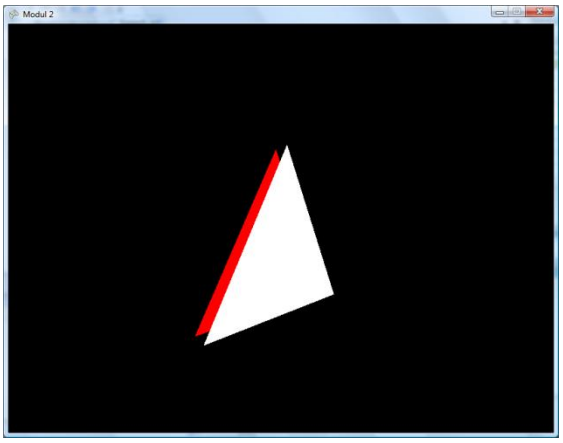
```
protected override void Draw(GameTime gameTime)
{
   GraphicsDevice gd = graphics.GraphicsDevice;
   gd.Clear(Color.Black);
   gd.RenderState.CullMode = CullMode.None;
   gd.VertexDeclaration = new VertexDeclaration(gd, VertexPositionColor.VertexElements);

   //wlaczenie alpha blendingu i ustawienie funkcji mieszania kolorow
   gd.RenderState.AlphaBlendEnable = true;
   gd.RenderState.SourceBlend = Blend.SourceAlpha;
   gd.RenderState.DestinationBlend = Blend.InverseSourceAlpha;
   gd.RenderState.BlendFunction = BlendFunction.Add;

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

      //czerwony trojkat (tylni)
      gd.DrawUserPrimitives<VertexPositionColor>(PrimitiveType.TriangleList,werteksy,0,1);
      //bialy trojkat (przedni)
      gd.DrawUserPrimitives<VertexPositionColor>(PrimitiveType.TriangleList,werteksy,3,1);

      pass.End();
   }
   efekt.End();

   //wylaczenie przezroczystosci
   gd.RenderState.AlphaBlendEnable = false;

   base.Draw(gameTime);
}
```

Przełączenie własności `gd.RenderState.AlphaBlendEnable` na `true` włącza mechanizm mieszania kolorów. Oprócz tego należy ustalić funkcję jaka jest używania do mieszania. W naszym przypadku kolor każdego piksela nowego trójkąta umieszczanego na tle innych narysowanych wcześniej jest wynikiem sumowania koloru nowego trójkąta (*source*) i tła (*destination*) z wagami odpowiadającymi wartości współrzędnej A koloru nowego trójkąta tj.

$RGB_{\text{na ekranie}} = A * RGB_{\text{nowego trójkąta}} + (1-A) * RGB_{\text{tło}}$.

Oczywiście to sumowanie jest wykonywane dla każdej składowej R, G i B koloru oddzielnie. Należy także pamiętać, że A wskazuje na nieprzezroczystość obiektu, a nie odwrotnie. Zatem im nowy trójkąt jest mniej przezroczysty, tym jego waga w powyższej sumie rośnie, a waga tła maleje.

Po skompilowaniu i uruchomieniu gry zobaczymy parę trójkątów (rysunek 12), które nie są przezroczyste, ale też nie powinno to być niespodzianką skoro do ich rysowania użyte są kolory, których współrzędna A (kanał alfa) równa jest 255 (maksymalna nieprzezroczystość).

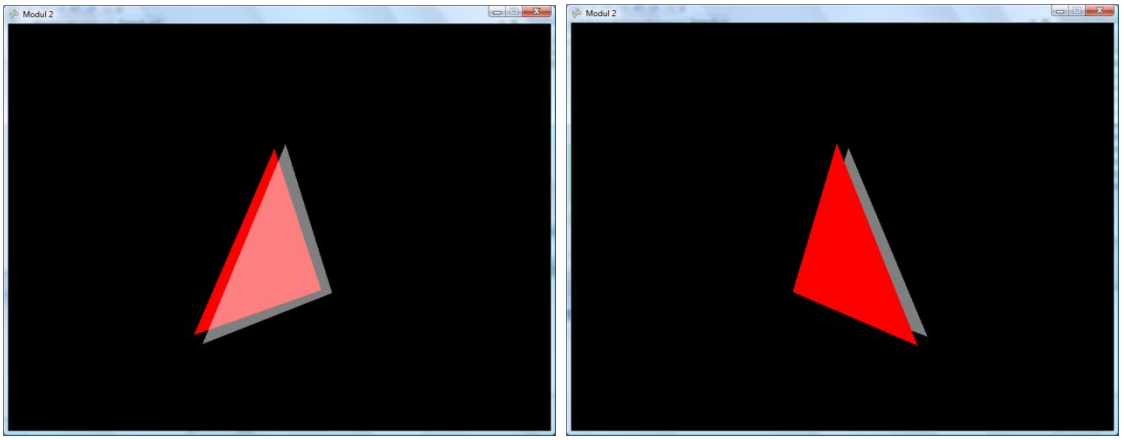


Rysunek 12. Dwa nieprzezroczyste trójkąty

Zmieńmy zatem kolor białego trójkąta tak, aby był półprzezroczysty. Wystarczy zmienić definicję pola `kolor2` w następujący sposób:

```
static Color kolor2 = new Color(Color.White, 128);
```

Zgodnie z przewidywaniami biały trójkąt przyjmie rolę firanki i stanie się półprzezroczysty (rysunek 13, lewy). Tam, gdzie jego kolor sumuje się z czarnym tłem przedni trójkąt stanie się szary, a na tle tylniego trójkąta będzie różowy. Nasz mózg, który dąży do „domykania" kształtów będzie widział dwa trójkąty jeden za drugim. Podczas obrotu, w momencie, w którym nieprzezroczysty czerwony trójkąt będzie z przodu, będzie przesłaniał biały trójkąt (rysunek 13, prawy).

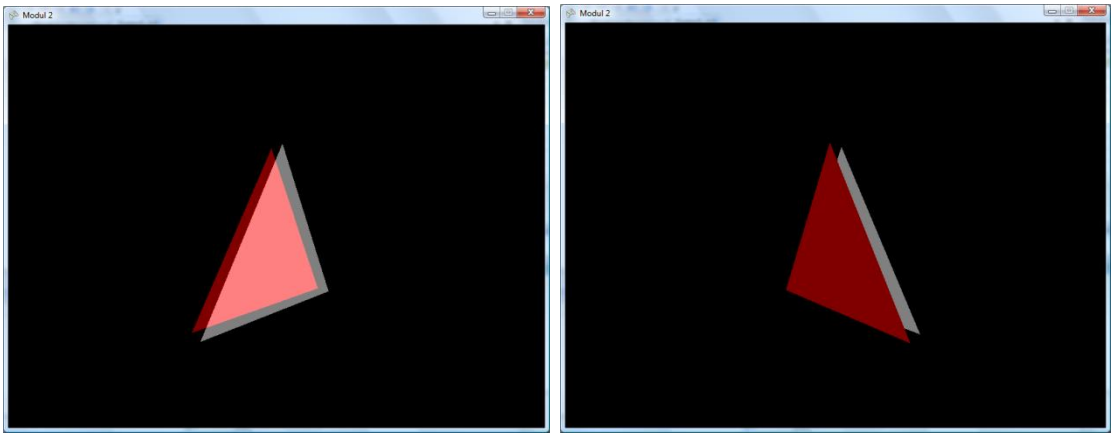


Rysunek 13. Biały trójkąt jest półprzezroczysty

Zachęceni tym sukcesem zmieńmy także kolor czerwonego trójkąta tak, aby jego współrzędna A miała wartość 128:

```
static Color kolor1 = new Color(Color.Red, 128);
```

Kolor czerwony stał się ciemniejszy, bo sumuje się teraz z czarnym kolorem tła (rysunek 14, lewy). W efekcie kolor białego trójkąta także widzi tło - w 25% wpływa ono na jego kolor w tej części, w której przesłania trójkąt czerwony i w 50% w pozostałych punktach. Tak powinno być. Niestety rozczarujemy się, gdy trójkąty obrócą się do nas tyłem. Ciemny trójkąt czerwony nie zostanie rozjaśniony przez znajdujący się teraz za nim trójkąt biały (rysunek 14, prawy).



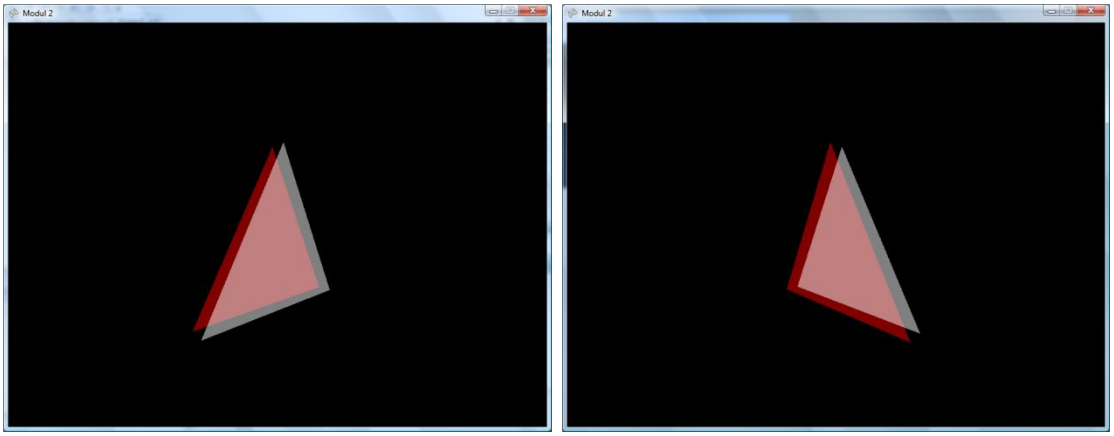
Rysunek 14. Tu trójkąt czarny również powinien być półprzezroczysty

Dzieje się tak gdyż w konflikt wchodzą bufor głębi i mechanizm mieszania kolorów, który sprawia że ważna staje się kolejność rysowania trójkątów. Bufor głębi można sobie wyobrazić jako dwuwymiarową macierz o wielkości odpowiadającej tablicy pikseli renderowanego obrazu sceny przechowującą odległość od kamery obiektów odpowiedzialnych za każdy piksel. W trakcie rysowania nowego obiektu odległość od kamery odpowiadająca nowym pikselom porównywana jest z tymi w buforze głębi i rysowana tylko, jeżeli ten fragment nowej figury jest bliżej. Wówczas aktualizowana jest też wartość w buforze. Załóżmy, że w buforze jest już informacja o czerwonym trójkącie, który jest rysowany jako pierwszy. Jego kolor został wcześniej zmieszany z czarnym tłem. Teraz przystępujemy do rysowania białego trójkąta, który jest dalej od kamery niż czerwony. Mechanizm mieszania kolorów miesza kolor białego i czerwone trójkąta, ale bufor głębi, który jest sprawdzany później wskazuje, że to nie biały, a czerwony trójkąt jest bliżej kamery. W efekcie na ekranie pojawi się kolor ciemnoczerwonego trójkąta, a wymieszany kolor zostanie zignorowany.

Czy jest jakieś rozwiązanie tego problemu? Można oczywiście wyłączyć bufor głębi.

```
gd.RenderState.DepthBufferEnable = false;
```

W najprostszych przypadkach, w których kanał alfa ustawiony jest na 50% i na scenie są jedynie półprzezroczyste przedmioty, może to dać jako takie rezultaty (rysunki 15).



Rysunek 15. Efekt wyłączenia bufora głębi

W przypadku, gdy nieprzezroczystość obrazów jest różna, kolejność ich rysowania wpływa na kolor. Można się z tym uporać stosując mieszanie sumujące (*additive blending*):

$RGB_{\text{na ekranie}} = A * RGB_{\text{nowego trójkąta}} + RGB_{\text{tło}}$.

Wymaga to tylko jednej modyfikacji w naszym kodzie, a mianowicie ustawienia drugiej z wag równej 1:

```
gd.RenderState.DestinationBlend = Blend.One;
```

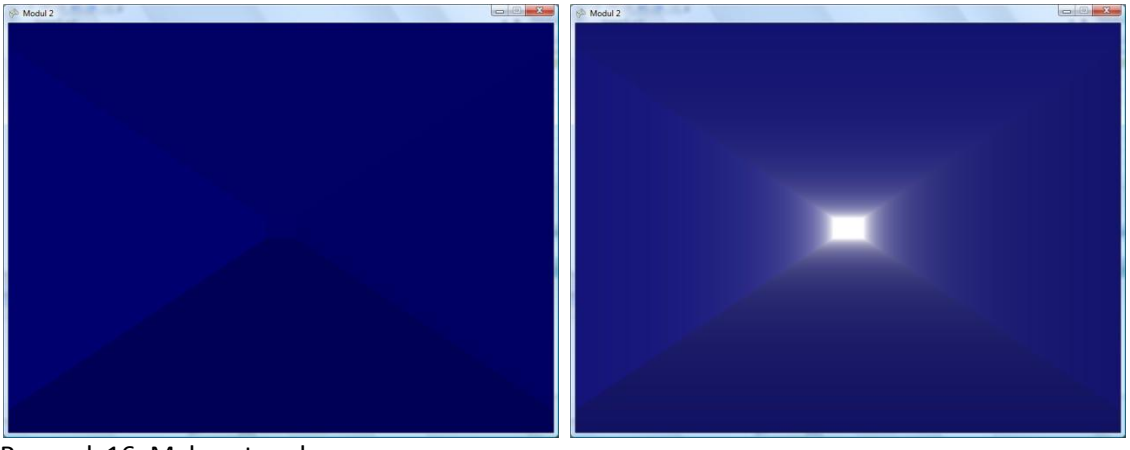
To rozwiązanie jest powszechnie stosowane, ale nawet ono nie gwarantuje w pełni poprawnych rezultatów. Sposób najlepszy, ale zwykle wymagający sporo wysiłku, to sortowanie przezroczystych obiektów. Po pierwsze należy je rysować po wszystkich obiektach nieprzezroczystych. Po drugie należy je rysować w kolejności od najdalszego do najbliższego kamery. W naszym przypadku, w którym na scenie są tylko dwa obiekty, wprowadzenie sortowania jest łatwe (rozwiązanie w kodach źródłowych i na prezentacji).

Mgła

Mieszanie kolorów to nie tylko przezroczystość. Możemy dzięki niemu uzyskać także inne efekty. Jeżeli do koloru obiektu będziemy dodawać kolor biały lub żółty z wagą tym większą im większa jest odległość obiektu od kamery, uzyskamy efekt mgły. Należy jednak pamiętać, że ma on wpływ wyłącznie na kolor obiektów. Skoro mgła nie jest widoczna w tych miejscach, w których nie ma żadnych obiektów, tło powinno być w takim samym kolorze, jak mgła. Nie zobaczymy również smug światła charakterystycznych dla jesiennej mgły – takie efekty wymagają sporo dodatkowego wysiłku.

Umieśćmy na scenie wydłużony prostopadłościan w taki sposób, żeby kamera znajdowała się wewnątrz niego (rysunek 16, lewy). Oczywiście należy wyłączyć usuwanie powierzchni, abyśmy mogli zobaczyć wewnętrzną stronę prostopadłościanu zbudowaną z tylnych ścian trójkątów.
Następnie umieśćmy w metodzie `Initialize` (przed utworzeniem prostopadłościanu) poniższy fragment kodu ustawiający parametry mgły. Efektem będzie pobielenie tych części pudełka, które znajdują się daleko od kamery (rysunek 16, prawy).

```
efekt.FogStart = 0.2f;
efekt.FogEnd = 20;
efekt.FogColor = Color.White.ToVector3();
efekt.FogEnabled = true;
```



Rysunek 16. Mgła w tunelu

Jeżeli tło będzie czarne, a scena ciemna to można użyć czarnej mgły jeżeli chcemy, żeby nasze przedmiotu oddalając się znikały w mroku. Czerwonej mgły można użyć, gdy bohater jest ranny (por. seria *Call of Duty*).

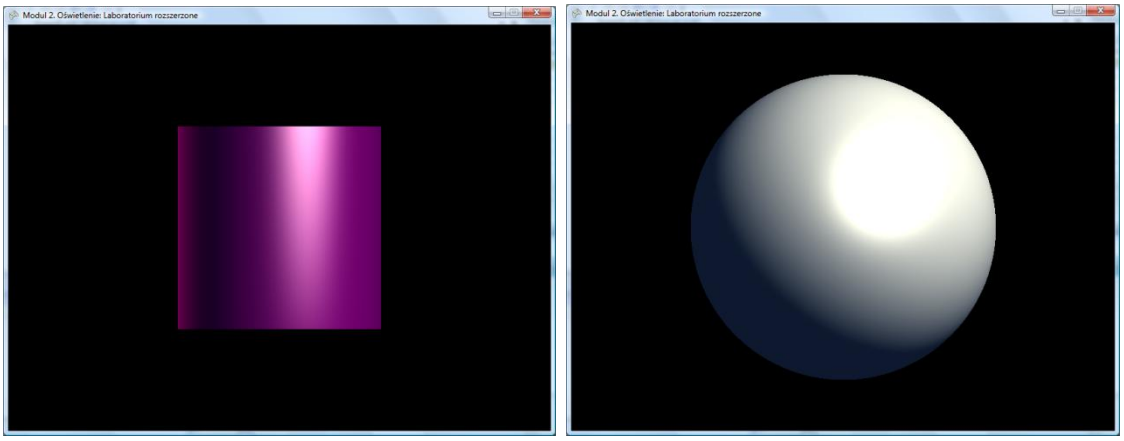
Podsumowanie

Myślę, że zdajesz już sobie sprawę, jak duże znaczenie w grach 3D ma oświetlenie. Prawidłowe jego przygotowanie może zająć sporo czasu, ale to się na pewno opłaci.
Dowiedziałeś się, jakie są typy światła, jak ustalać własności materiału i własności źródeł światła. Wiesz już również jak przygotować obiekt do tego, aby prawidłowo reagował na oświetlenie. Nauczyłeś się również jak, w najprostszy sposób utworzyć cienie rzucane przez Twoich aktorów. Wreszcie poznałeś problemy związane z mieszaniem kolorów.

Zadania

Zadanie 1 (czas realizacji 45 min)

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.

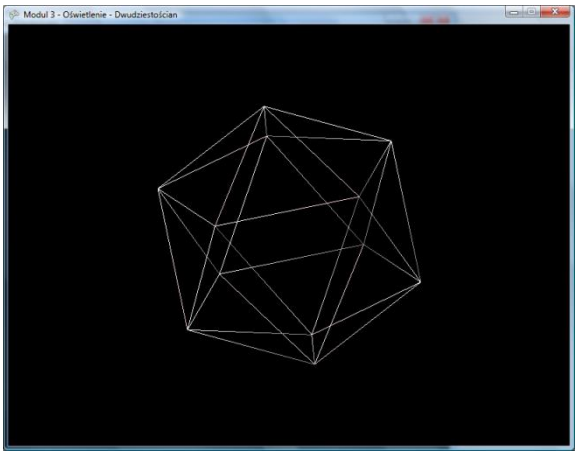


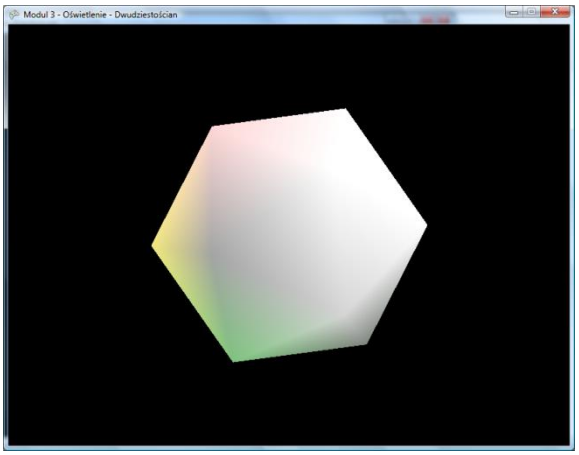
Podpowiedź: Tak zdefiniowane normalne uwzględniają uśrednianie normalnych w obrębie pasma (`PrimitiveType.TriangleStrip`). Ponieważ większość kwadryk (walec, stożek, dysk) zbudowane są z pojedynczych pasm jest to rozwiązanie w pełni wystarczające. Jednak nie w przypadku sfery, w której należy także dodać 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 (czas realizacji 45 min)

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.

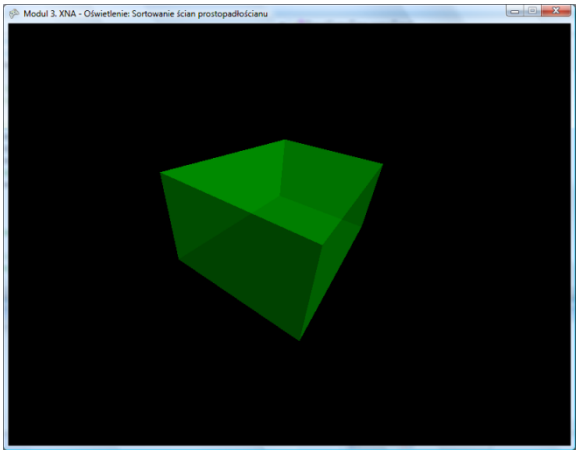
Podpowiedź:
Czy musisz obliczać wektory prostopadłe do poszczególnych ścian ikosaedru, żeby policzyć normalne dla werteksów?
Czy można tak zmodyfikować typ werteksu, aby w ogóle uniknąć definiowania normalnych?





Zadanie 3 (czas realizacji 70 min)

Zmodyfikować klasę `Prostopadloscian` w taki sposób, aby ściany rysowane były w kolejności od najdalszej od kamery do najbliższej. To pozwoli na prawidłowe rysowanie półprzezroczystego prostopadłościanu.



Podpowiedź: przygotuj kolekcję zawierającą aktualne położenia środków ścian oraz ich pozycję w buforze werteksów. Sortowanie tej kolekcji względem położenia środków (po uwzględnieniu macierzy świata i widoku) pozwoli na ustalenie kolejności rysowania ścian.

Zadanie 4 (czas realizacji 20 min)

Przygotować projekt, w którym sfera, walec i stożek rzucają cień na płaskie podłoże.