Rozdział 6
Cienie rzucane. Mieszanie kolorów

Wersja z 2014-04-04

W tym rozdziale omówione są zagadnienia związane „znaczeniowo” z oświetleniem: cienie rzucane, przezroczystość, mgła. Nie maja one jednak nic wspólnego z modelem oświetlenia Phonga.

Rzutowanie cieni

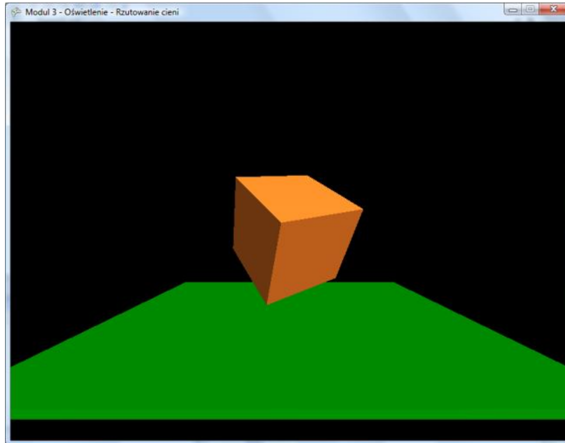
Cienie własne mają kluczowe znaczenie dla postrzegania obiektów jako trójwymiarowych brył. Nie zapominajmy jednak, że cieniom własnym towarzyszą cienie rzucane na podłoże i inne przedmioty. Ich obecność pomaga między innymi lepiej rozpoznawać położenie i orientację obiektów. O ile cienie własne, czyli różna jasność ścian bryły w zależności od ich orientacji względem źródeł światła, są obsługiwane zarówno przez biblioteki MonoGame, 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, 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 pozwalała uzyskać macierz rzutowania równoległego (tzn. odpowiadającego nieskończenie odległemu źródłu światła) na dowolną płaszczyznę korzystając z metody `Matrix.CreateShadows`. Jej dwoma argumentami były wektor wskazujący kierunek do źródła światła oraz płaszczyzna, na którą rzutowanie ma być przeprowadzane. Jeżeli macierz świata została pomnożona przez tą macierz, wszystkie punkty trójwymiarowej bryły znajdowały się na tej płaszczyźnie – tym samym bryła zostanie „spłaszczona”. Wystarczy tylko użyć do jej pomalowania czerni, aby uzyskać cień.
W MonoGame nie ma tej metody, dlatego zmuszeni będziemy napisać ją samodzielnie. Przy okazji uogólnimy ją nieco umożliwiając także uzyskanie rzutowania perspektywicznego pochodzącego od źródła światła, które nie jest nieskończenie odległe. W metodzie zdefiniujemy trzeci dodatkowy argument typu `bool`, który będzie kontrolował współrzędną *w* wektora określającego położenie

źródła światła. Jeżeli będzie równy prawdziwości, współrzędnej *w* nadamy zerową wartość, co oznacza jej przesunięcie do nieskończoności.

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 jak wspomniałem w MonoGame 3 nadal takiej metody nie znajdziemy! Ale nie szkodzi, samodzielnie napiszemy analogiczną metodę w klasie `Game1` rozszerzając ją nawet o możliwość rzutowania perspektywicznego:

```
static Matrix CreateShadowMatrix(Vector3 lightPosition, Plane plane,
                                 bool nieskończenieOdległeŹródłoŚwiatła = true)
{
    Vector4 N = new Vector4(plane.Normal, plane.D);
    Vector4 L = new Vector4(lightPosition, nieskończenieOdległeŹródłoŚwiatła ? 0 : 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;
}
```

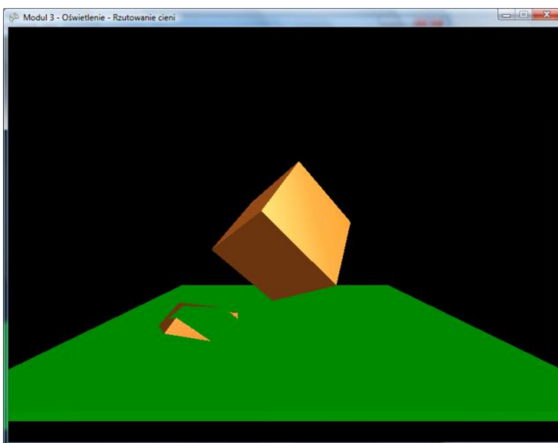
4. Ponieważ oświetlenie jest nieruchome, możemy zdefiniować stałą macierz rzutowania jako pole klasy `Game1` (stała `d` opisuje odległość od środka układu współrzędnych do górnej płaszczyzny prostopadłościanu pełniącego rolę podłoża):

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

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

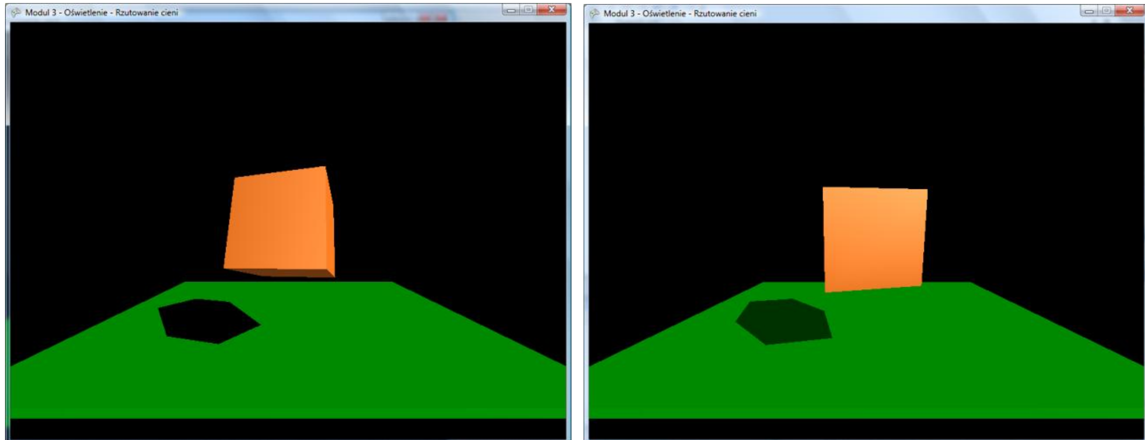
7. Wyraźnie widoczny jest błąd – dwa obiekty rysowane są na tej samej płaszczyźnie i trudno przewidzieć, który będzie widoczny, a który zostanie przesłonięty. Aby tego uniknąć warto wyłączyć bufor głębi na czas rysowania sceny. Jednak aby nie modyfikować komponentu `Prostopadloscian` możemy też zastosować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

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

ZROBIĆ RYSUNKI Z RZUTOWANIEM Z NIESKOŃCZONOŚCI (IZOMETRYCZNYM) I PERSPEKTYWICZNYM
W PRZYPADKU PERSPEKTYWY – MRUGANIE. W PRZYPADKU IZOMETRYCZNEGO – ZMIANA KOLORU

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.

Przezroczystość

Wczytajmy projekt z katalogu ***. W projekcie tym, w metodzie Game1.Initialize zdefiniowane są dwa trójkąty (czerwony i biały), których werteksy znajdują się w następujących punktach:

```
Color kolor1 = Color.Red;
Color kolor2 = Color.White;
float z0 = 0.1f;
VertexPositionColor[] tablicaWerteksów = new VertexPositionColor[]
{
    //trójkąt biały
    new VertexPositionColor(new Vector3(1,-1,-z0), kolor1),
    new VertexPositionColor(new Vector3(-1,-1,-z0), kolor1),
    new VertexPositionColor(new Vector3(0,1,-z0), kolor1),
    //trójkąt czerwony
    new VertexPositionColor(new Vector3(1,-1,z0), kolor2),
```

```
    new VertexPositionColor(new Vector3(-1,-1,z0), kolor2),
    new VertexPositionColor(new Vector3(0,1,z0), kolor2)
};
```

Zdefiniowane jest także 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:

```
GraphicsDevice.RasterizerState = RasterizerState.CullNone;
```

Oba trójkąty rysowane są jednym wywołaniem metody `gd.DrawUserPrimitives`:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Black);

    GraphicsDevice.SetVertexBuffer(buforWerteksów);

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

        GraphicsDevice.DrawPrimitives(
            PrimitiveType.TriangleList,
            0,
            2);
    }

    base.Draw(gameTime);
}
```

Umieśćmy teraz w metodzie `Game1.Initialize` polecenia włączające mechanizm mieszania kolorów:

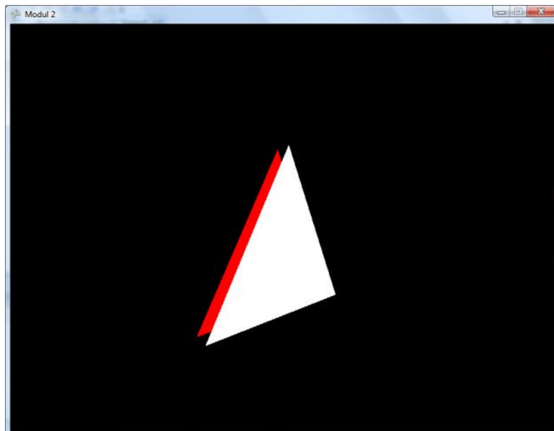
```
BlendState bs = new BlendState();
bs.ColorSourceBlend = Blend.SourceAlpha;
bs.AlphaSourceBlend = Blend.SourceAlpha;
bs.ColorDestinationBlend = Blend.InverseSourceAlpha;
bs.AlphaDestinationBlend = Blend.InverseSourceAlpha;
bs.AlphaBlendFunction = BlendFunction.Add;
GraphicsDevice.BlendState = bs;
```

Ustaliliśmy w ten sposób 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.

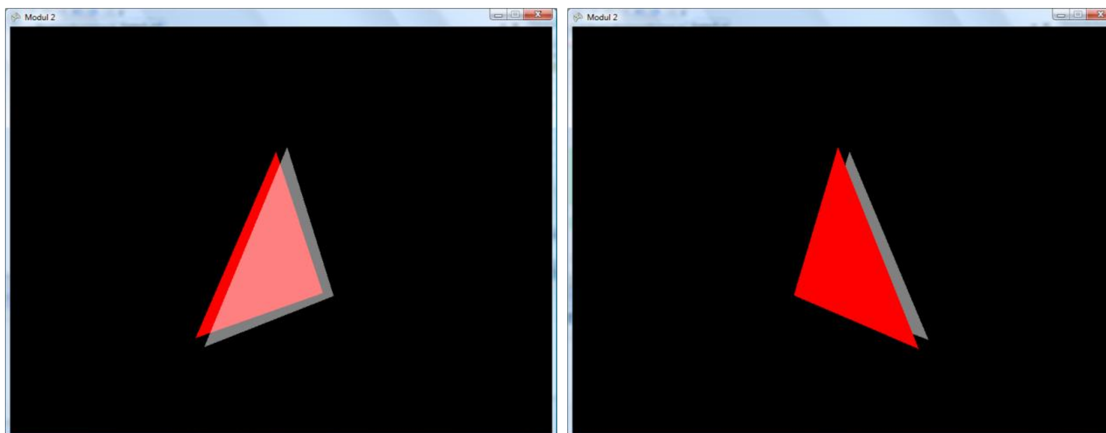
Po skompilowaniu i uruchomieniu gry zobaczymy parę trójkątów (rysunek 12), które nie są przezroczyste, ale też nie powinno to być niespodzianką skoro do ich rysowania użyte są kolory, których współrzędna A (kanał alfa) równa jest 255 (maksymalna nieprzezroczystość).


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);
```

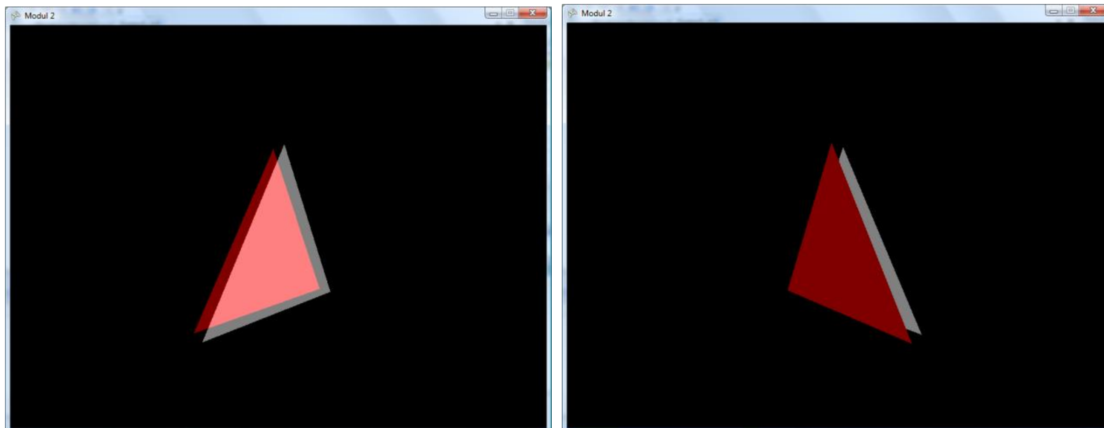
Zgodnie z przewidywaniami biały trójkąt przyjmie rolę firanki i stanie się półprzezroczysty (rysunek 13, lewy). Tam, gdzie jego kolor sumuje się z czarnym tłem przedni trójkąt stanie się szary, a na tle tylniego trójkąta będzie różowy. Nasz mózg, który dąży do „domykania" kształtów będzie widział dwa 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).



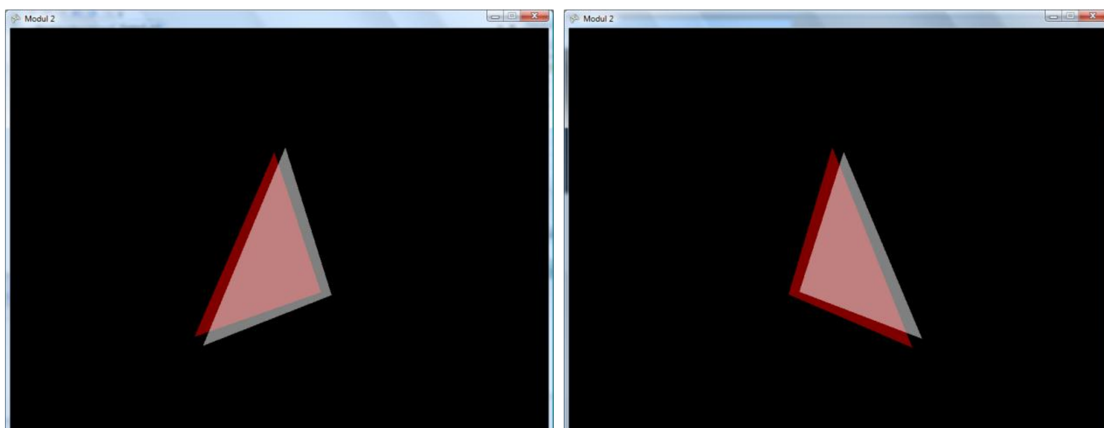
Rysunek 14. Tu trójkąt czarny również powinien być półprzezroczysty

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 poleceniami:

```
DepthStencilState dss = new DepthStencilState();
dss.DepthBufferEnable = false;
GraphicsDevice.DepthStencilState = dss;
```

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*):

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

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

```
bs.ColorDestinationBlend = Blend.One;
bs.AlphaDestinationBlend = 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:

```
float kąt = 0;

...

protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed ||
         Keyboard.GetState().IsKeyDown(Keys.Escape))
        Exit();

    kąt += gameTime.ElapsedGameTime.Milliseconds / 1000.0f;
    if (kąt > MathHelper.TwoPi) kąt = 0;
    efekt.World = Matrix.CreateRotationY(kąt);

    base.Update(gameTime);
}

protected override void Draw(GameTime gameTime)
{
    bool czerwonyPierwszy =
        (kąt > 0 && kąt < MathHelper.PiOver2) ||
        (kąt > 3 * MathHelper.PiOver2 && kąt < MathHelper.TwoPi);

    GraphicsDevice.Clear(Color.Black);
    GraphicsDevice.SetVertexBuffer(buforWerteksów);

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

        if (czerwonyPierwszy)
        {
            GraphicsDevice.DrawPrimitives(
                PrimitiveType.TriangleList,
                0,
                2);
        }
        else
        {
            GraphicsDevice.DrawPrimitives(
                PrimitiveType.TriangleList,
                3,
                1);
            GraphicsDevice.DrawPrimitives(
                PrimitiveType.TriangleList,
                0,
                1);
        }
    }

    base.Draw(gameTime);
}
```

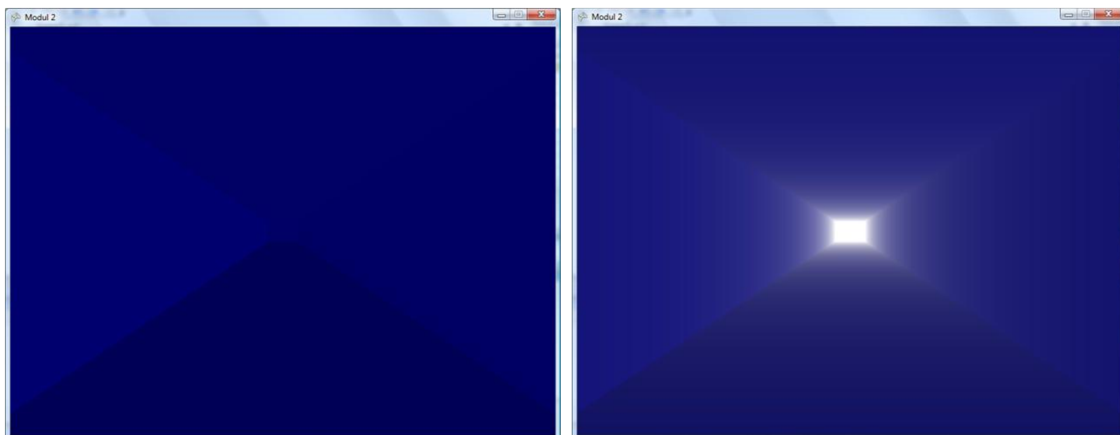
Możemy, a wręcz powinniśmy, teraz przywrócić właściwą wartość drugiej z wag oraz przede wszystkim przywrócić test głębi.

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 tylnich powierzchni, abyśmy mogli zobaczyć wewnętrzną stronę prostopadłościanu zbudowaną z tylnych ścian trójkątów.
Następnie umieśćmy w metodzie `Initialize` (przed utworzeniem prostopadłościanu, a po stworzeniu obiektu efektu) 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 = 10f;
efekt.FogColor = Color.White.ToVector3();
efekt.FogEnabled = true;
```


Rysunek 16. Mgła w tunelu

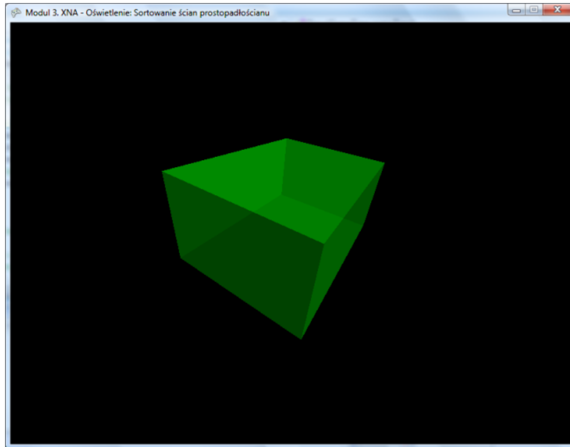
Jeżeli tło będzie czarne, a scena ciemna to można użyć czarnej mgły jeżeli chcemy, żeby nasze przedmiotu oddalając się znikały w mroku. Czerwonej mgły można użyć, gdy bohater jest ranny (por. seria *Call of Duty*).

UWAGA! Każda ściana zbudowana jest tylko z dwóch trójkątów. Dlatego w szerokim zakresie parametrów `FogStart` i `FogEnd`, daleki werteks będzie całkowicie biały, a bliski będzie miał oryginalny kolor. Kolory między nimi zależą już od interpolacji liniowej wykonywanej między shaderem werteksów, a pikseli. W efekcie w przypadku prostopadłościany modyfikacje tych dwóch własności nie przynoszą efektów.

Zadania

Zadanie 1

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 2

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