Stereoskopia. Implementacja w XNA 4.0

Stereoskopia na dobre zadomowiła się w kinach. Niemal wszystkie przeboje, począwszy od „Avatara" po filmy animowane dla dzieci są teraz wyświetlane w wersji 3D. Także wśród urządzeń kina domowego pojawiają się już urządzenia do prezentacji obrazu z wrażeniem głębi. Mowa zarówno o telewizorach, jak i o akcesoriach komputerowych.

Dowiesz się:

- Artykuł przedstawia prostą implementację animowanego anaglifu. Dzięki niemu zrozumiesz jak tworzone są obrazy dla dwukolorowych okularów i będziesz mógł użyć tej techniki we własnych grach tworzonych w XNA.

Powinieneś wiedzieć:

- Wskazana jest podstawowa znajomość podstaw C#, XNA i HLSL.

Widzenie przestrzenne

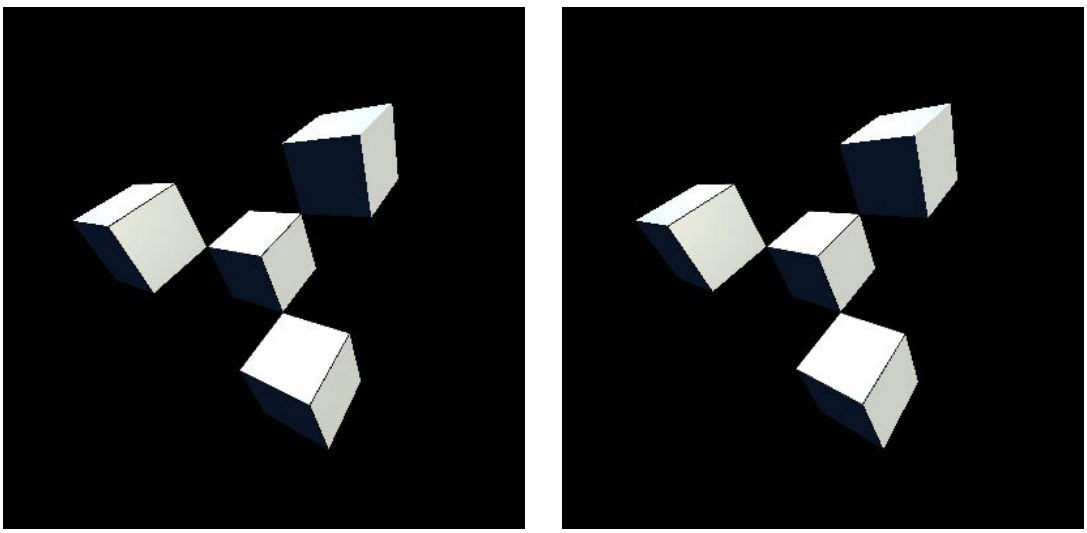
Możemy widzieć obraz przestrzenny. Wbrew temu, co zwykło się pisać w artykułach poświęconych kinu 3D, widzenie stereoskopowe (binokularowe) nie jest tu jedynym, ani nawet najważniejszym mechanizmem. Do oceny odległości przedmiotów korzystamy przede wszystkim z perspektywy – im dalej położony jest przedmiot, tym jest mniejszy. Równie ważne jest oświetlenie. Bez cieni własnych i rzucanych, wszystko staje się płaskie, jak na fotografii z oświetleniem typu *high key*. I odwrotnie, im bardziej „plastyczne" są cienie, tym bardziej sugestywna jest głębia. Z obu mechanizmów co najmniej od czasów Rembrandta korzysta malarstwo, a współcześnie także grafika komputerowa 3D, do przedstawiania głębi w płaskim obrazie.

A jednak nałożenie okularów 3D i oglądanie obrazów stereoskopowych daje nowy rodzaj odczuć, który jest dla widza atrakcyjny. Skąd on się bierze? Odpowiedź może być tylko jedna – z mózgu. To on jest odpowiedzialny za przetwarzanie sygnałów docierających do niego z obu oczu. I to on jest odpowiedzialny za wrażenie głębi. Jedyne co musimy mu do tego zapewnić, to dwa obrazy przedstawiające tą samą scenę widzianą z dwóch, nieco rozsuniętych punktów. Różnica tych dwóch przedstawień sceny nie jest duża (rysunek 1), odzwierciedla odległość między lewym i prawym okiem, która równa jest średnio tylko około 5-7 cm. Mózg łączy te dwa obrazy „kodując" różnice między nimi jako odległość widzianych przedmiotów od obserwatora. Przedmioty nie mogą być jednak zbyt odległe. Ten sposób oceny odległości działa bowiem tylko w najbliższym zakresie odległości, najwyżej do kilku metrów.

Stereoskopia

Tym jak tworzyć dwa odpowiednio przygotowane obrazy, i jak je dostarczyć osobno do lewego i prawego oka zajmuje się stereoskopia. Jest to technika obrazowania, oddająca wrażenie normalnego widzenia przestrzennego tzn. takiego, którego efekt zawiera nie tylko informacje o kształtach i kolorach obiektów, ale także ich wzajemne zależności przestrzenne (za Wikipedią). Ludzie interesowali się stereoskopią już w połowie XIX w. Jedną z najstarszych realizacji tej techniki polegała na łączeniu przygotowanych obrazów w tak zwane stereopary. Jednym z pierwszych i najprostszych, a zarazem najbardziej popularnych stereoskopów był ten skonstruowany przez Homelsa w 1861 r. (rysunek 2). Była to bardzo prosta konstrukcja składająca się z soczewek i uchwytu na obrazy. Mimo to pozwalała na uzyskanie pożądanego efektu. W miarę upływu lat, zainteresowanie tego typu rozrywką rosło i powstawały coraz to nowsze i bardziej skomplikowane urządzenia.

W roku 1866 Alois Polanecky (1826-1911) zbudował urządzenie złożone z 25 stereoskopów wbudowanych w wieloboczną obudowę. Korzystał przy tym z pomocy francuskiego fotografa i optyka Claude-Marie Ferriera.



Rysunek 1. *Obraz tej samej sceny widzianej lewym i prawy okiem.*

Z tym „Salonem Stereoskopowym" występował na jarmarkach. W latach osiemdziesiątych XIX wieku berliński przedsiębiorca August Fuhrmann (1844-1924) udoskonalił wynalazek Polaneckiego. W ten sposób powstały pierwsze fotoplastykony (ponownie cytuję Wikipedię). Pojawiały się one także na terenie obecnej Polski. Pozwalały one 25 osobom na równoczesne oglądanie przezroczy stereoskopowych. Potem zainteresowanie tą technologią zmalało. Na nowo z pełną mocą wróciło dopiero w ostatnich kilku latach pod postacią kina 3D (zob. Ramka).

Okulary 3D

Zadaniem okularów 3D jest dostarczenie do obu oczu dwóch nieco różnych obrazów. Jest kilka metod uzyskania tego efektu: od okularów czynnych, które naprzemiennie zaciemniają jeden z okularów (np. okulary NVIDII), poprzez okulary wykorzystujące zjawisko polaryzacji światła, do okularów dwukolorowych. My zajmiemy się tylko tym ostatnim sposobem, najprostszym i najtańszym w realizacji. Klasyczne okulary dwukolorowe miały szkła czerwone i niebieskie. Obraz, który był przez nie oglądany musiał być wobec tego złożeniem dwóch rysunków: niebieskiego i czerwonego. Lewe oko patrzące przez czerwony okular nie będzie widziało czerwieni, która zleje się z kolorem tła nadanym przez szło całemu lewemu obrazowi. Lewe oko będzie wobec tego postrzegało czerwień jako swoistą „biel". Analogicznie prawe oko nie będzie reagować na niebieski. Pozostałe kolory postrzegane będą przez lewe i prawe oko jako różne odcienie czerwonego i niebieskiego. Ze względu na użyte filtry, kolor zielony postrzegany będzie jako czarny. W efekcie okulary czerwono-niebieskie nie umożliwiają zobaczenia obrazu kolorowego. Aby uniknąć tego problemu obecnie stosuje się raczej szkła czerwone i cyjan (niebieskozielone). Kolory te dopełniają się (ich suma daje biel), a jednocześnie są ortogonalne w przestrzeni kolorów. Jeżeli korzystając z takich okularów spojrzymy na obraz, na którym rysunek przeznaczony dla lewego oka narysowany będzie w czerwieni i nałożony na cyjanowy rysunek dla prawego oka, jak na rysunku 3 (lewy), powinniśmy zobaczyć obraz 3D z wrażeniem głębi. Dla porównania rysunek 3 (środkowy) przedstawia tę samą scenę przygotowaną dla okularów ze szkłami czerwonym i niebieskim.

Stereoskopia w kinach

W większości kin w Polsce stosowana jest technologia Dolby 3D Digital Cinema. System ten pozwala uzyskać klarowny obraz 3D ze stosunkowo realistycznym odwzorowaniem barw. Opiera się na podziale pasma barw na części i kierowaniu do lewego i prawego oka pasm o nieco innym odcieniu. System ten nie wymaga specjalnych ekranów kinowych. Co więcej może być zainstalowany wszędzie tam, gdzie znajduje się już projektor cyfrowy spełniający wymagania DCI. Domontowuje się do niego tylko kilka podzespołów. Opiera się na technologii "wavelength triplet" opracowanej przez niemiecką firmę *Infitec*. Filtr krążkowy wielkości płyty CD umieszczony wewnątrz cyfrowego projektora, obracając się z dużą prędkością (jednak w sposób zsynchronizowany z wyświetlaniem klatek - połowa krążka zawsze wyświetla się synchronicznie z klatką przeznaczoną dla lewego oka, druga połowa odpowiednio zsynchronizowana jest z klatką przeznaczoną dla prawego oka) dzieli widzialne pasmo świetlne na "części" o różnym odcieniu barw. Dzięki okularom, które otrzymujemy przed seansem są one kierowane do odpowiednich oczu widza. Filtr może być automatycznie opuszczany i podnoszony, co pozwala w prosty sposób przełączać się między 3D i 2D w zależności od rodzaju projekcji. Ze względy na łatwość instalacji technologia staje się coraz bardziej popularna.
Informacje zaczerpnięte z *http://www.heliosnet.pl/12,Katowice/Kino3D/technologia/*

Najbardziej zaawansowaną technologią wśród okularów dwukolorowych jest opatentowana w 2000 r. technologia ColorCode 3D. Pozwala na oglądanie obrazów trójwymiarowych w stosunkowo pełnym kolorze. Przede wszystkim dotyczy to kolorów czerwonego i zielonego. Okulary tego typu są wyposażone w szkło koloru bursztynowego (amber) dla lewego oka, którego zadaniem jest przekazanie jak największej informacji o kolorze obrazu. Prawy okular ma szło koloru niebieskiego. Tym kanałem przekazywana jest monochromatyczna informacje potrzebna do uzyskania efektu głębi. Obraz przygotowany do tego typu okularów widoczny jest na rysunku 3, prawy.

Stereoskopia a grafika 3D

Oczywiście aby uzyskać wrażenie głębi, musimy znać odległości przedmiotów widocznych na scenie od wirtualnej kamery. Ale to jest informacja, która w grafice 3D jest łatwa do uzyskania. Nawet po przemnożeniu współrzędnych modeli przez macierz rzutowania (po tej operacji położenie przedmiotów wyrażane jest we współrzędnych odpowiadających pikselom na ekranie), informacja o głębi jest nadal przechowywana. Potrzebna jest, aby możliwe było przeprowadzenie testu głębi – ostatniego etapu w potoku renderowania (zob. ramka „Potok renderowania").

Na stronie *http://www.fizyka.umk.pl/~jacek/dydaktyka/3d/stereoskopia.zip* jest do pobrania projekt, w którym na scenie rysowana jest bryła złożona z 27 sześcianów. Z bryły tej możemy wyjąć niektóre sześciany używając klawiszy *F1 – F6*. Projekt przygotowany został w Visual Studio 2010 dla platformy XNA 4.0. Do renderowania korzystam ze standardowego efektu BasicEffect implementującego oświetlenie Phonga i teksturowanie. Macierz rzutowania dodaje do obrazu perspektywę. Widoczna na ekranie bryła obraca się, co możemy wyłączyć naciskając klawisz 0. Wówczas możliwe jest samodzielne obracanie nią za pomocą klawiszy kierunkowych (strzałek).

Tego projektu użyjemy jako punktu wyjście dla przygotowania animowanego anaglifu. Założenie poniższej implementacji jest takie, że chcemy uniknąć ingerencji w sposób rysowania sceny, a tym bardziej w definicje obiektów opisujących przedstawioną bryłę. Naszym celem jest przygotowanie kodu działającego z dowolną wcześniej przygotowaną sceną i modelami, a także niezależnego od użytych do samego renderowania efektów HLSL. Tak, aby można było użyć poniższych rozwiązań do dowolnego projektu przygotowanego przez Czytelnika. To skazuje nas na potrójne renderowanie: raz dla lewego oka, raz dla prawego i wreszcie złożenie uzyskanych w ten sposób obrazów i ich narysowanie na ekranie, niejako w fazie post-processingu. Będziemy się jednocześnie starali, aby przedstawiony kod był jak najprostszy i łatwy do zrozumienia.

Aby uzyskać dwa obrazy dla lewego i prawego oka, musimy umieścić na scenie dwie kamery (dwie osobne macierze widoku). Odległość, w jakiej powinny znajdować się kamery względem siebie powinna być równa 1/30 odległości od najbliższego obiektu na scenie. Jednak jest to wartość czysto teoretyczna, od której warto zacząć własne eksperymenty. W trakcie implementacji okazało się, że najlepszym sposobem wyznaczenia tej odległości jest metoda prób i błędów. Co więcej odległość kamer dla jednej sceny nie musi sprawdzić

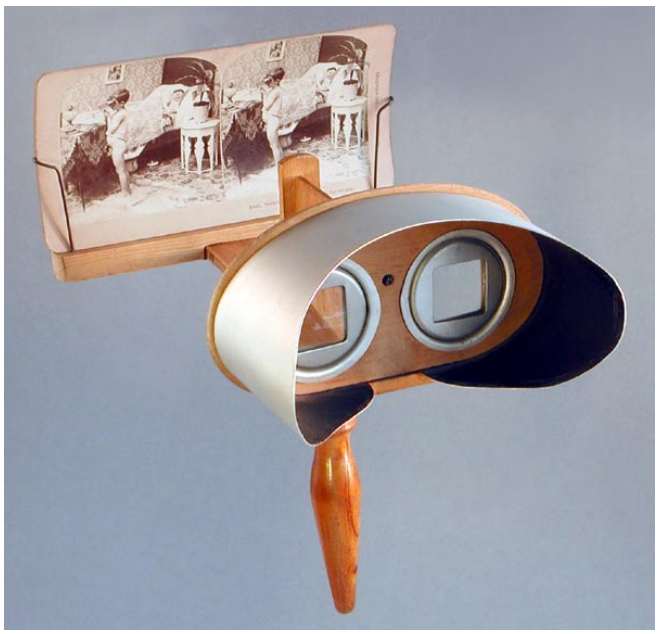
Listing 1. *Pary wielkości charakteryzujących obie kamery*

```
Matrix viewLeft = Matrix.Identity;
Matrix viewRight = Matrix.Identity;

RenderTarget2D renderTargetLeft;
RenderTarget2D renderTargetRight;

Vector3 filterLeft = new Vector3(1, 0, 0);
Vector3 filterRight = new Vector3(0, 1, 1);

bool cameraEnabledLeft = true;
bool cameraEnabledRight = true;
bool showSeparateViews = true;

readonly Vector3 cameraDefaultPositionLeft = new Vector3(0.92f, 1f, 4.5f);
readonly Vector3 cameraDefaultPositionRight = new Vector3(1.08f, 1f, 4.5f);

Vector3 cameraPositionLeft;
Vector3 cameraPositionRight;
```

się, dla inaczej ustawionych aktorów. Pozostałe parametry związane z ruchem kamer, a co za tym idzie ich pozycja oraz kierunek, są ściśle określone. Po pierwsze obie kamery, tak jak nasze oczy, spoglądają z różnych pozycji na ten sam punkt w przestrzeni. Po drugie, jeżeli chodzi o ich ruch musimy je potraktować, jak jeden obiekt – gdy obracamy lub przemieszczamy jedną kamerę, pozycja drugiej również musi zostać odpowiednio zmodyfikowana.

Implementacja CPU

Zacznijmy od implementacji, w której obraz dla lewego i prawego oka zostanie wpierw zapisany do dwóch tekstur. Skorzystamy do tego z dwóch instancji klas RenderTarget, który w XNA 4.0, to nowość, dziedziczy z klasy Texture2D. Następnie tekstury te złożymy piksel po pikselu w jedną teksturę biorąc składową czerwoną pierwszego obrazu oraz zieloną i niebieską drugiego. Tak przygotowany obraz wyświetlimy na ekranie korzystając z obiektu SpriteBatch (obiekt używany w aplikacjach z grafiką 2D). Muszę jednak od razu ostrzec, że składanie dwóch obrazów składających się np. z 800 x 600 pikseli oznacza pętlę z 480 tysiącami iteracji. Nawet superszybki procesor CPU może temu nie podołać w 1/60 sekundy, jaką mamy na przygotowanie obrazu. I to nawet jeżeli pętlę tą zrównoleglimy korzystając z nowej w .NET 4.0 metody Parallel.For (element biblioteki TPL). Mimo to zaczniemy od takiej „naiwnej" implementacji, aby lepiej zrozumieć ideę składania anaglifów.



Rysunek 2. *Stereoskop Holmesa (źródło Wikipedia)*

Zacznijmy od zdefiniowania nowych pól klasy Game1, które odpowiedzialne będą za przechowanie dwóch osobnych pozycji kamer, kolorów filtrów, czy wreszcie za przechowanie referencji do obiektów-celów renderowania, jakie użyjemy w kodzie. Przedsta-

Listing 2. *Inicjacja pól wymienionych w listingu 1.*

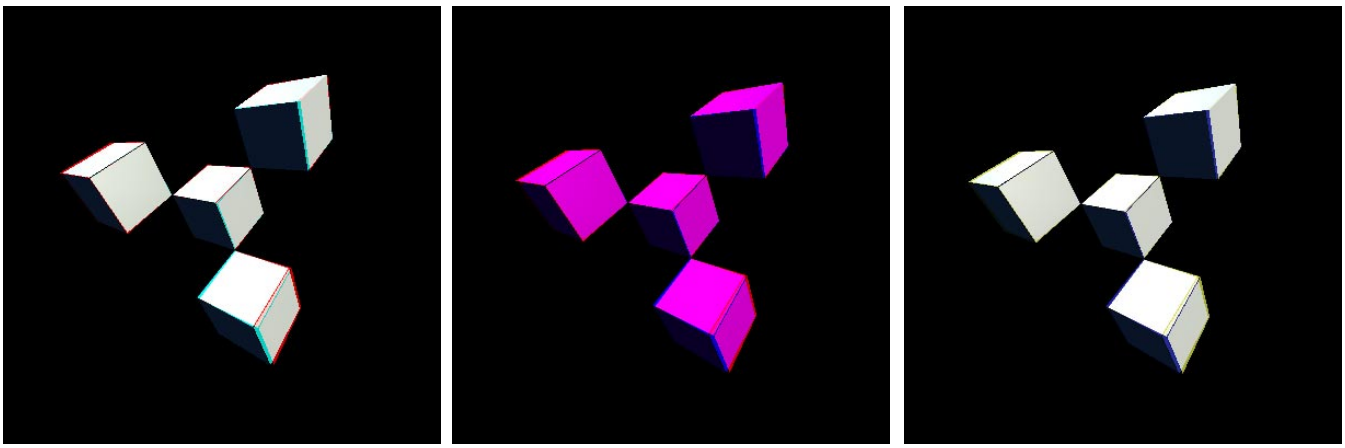
```
protected override void Initialize()
{
    gd = graphics.GraphicsDevice;
    PresentationParameters pp = gd.PresentationParameters;

    renderTargetLeft = new RenderTarget2D(gd, pp.BackBufferWidth, pp.BackBufferHeight, false,
                gd.DisplayMode.Format, DepthFormat.Depth24);
    renderTargetRight = new RenderTarget2D(gd, pp.BackBufferWidth, pp.BackBufferHeight, false,
                gd.DisplayMode.Format, DepthFormat.Depth24);

    effect = new BasicEffect(gd);
    effect.VertexColorEnabled = false;
    effect.Projection = Matrix.CreatePerspective(2.0f * gd.Viewport.AspectRatio, 2.0f, 1.0f, 100.0f);
    effect.EnableDefaultLighting();

    cameraPositionLeft = cameraDefaultPositionLeft;
    cameraPositionRight = cameraDefaultPositionRight;

    viewLeft = Matrix.CreateLookAt(cameraPositionLeft, Vector3.Zero, Vector3.Up);
    viewRight = Matrix.CreateLookAt(cameraPositionRight, Vector3.Zero, Vector3.Up);

    base.Initialize();
}
```

wia je listing 1. Pola te zainicjujemy w metodzie Game1.Initialize (listing 2). Zwróćmy uwagę na definicję dwóch celów renderowania i dwóch macierzy widoku.

W metodzie Update powinny znaleźć się instrukcje obsługujące m.in. klawiaturę, którą możemy sterować położeniem kamer. W tym ich rozsunięciem. Przykład tej metody można obejrzeć w kodzie dostępnym na wspomnianej wcześniej stronie. My natomiast przejdźmy od razu do kluczowej metody Game1.Draw. Po zmianach powinna ona przyjąć postać widoczną na listingu 3.

Listing 3. *Metoda odpowiedzialna za przygotowanie anaglifu*

```
protected void Draw(GameTime gameTime)
{
    //Left camera
    gd.SetRenderTarget(renderTargetLeft);
    gd.Clear(Color.Black);
    if (cameraEnabledLeft)
    {
        solid.View = viewLeft;
        solid.Draw();
    }
    gd.SetRenderTarget(null);

    //Right camera
    gd.SetRenderTarget(renderTargetRight);
    gd.Clear(Color.Black);
    if (cameraEnabledRight)
    {
        solid.View = viewRight;
        solid.Draw();
    }
    gd.SetRenderTarget(null);

    //Superimposing
    Color[] imageLeft = new Color[renderTargetLeft.Width * renderTargetLeft.Height];
    renderTargetLeft.GetData<Color>(imageLeft);
    Color[] imageRight = new Color[renderTargetRight.Width * renderTargetRight.Height];
    renderTargetRight.GetData<Color>(imageRight);
    for (int i = 0; i < imageRight.Count(); ++i) imageRight[i] = new Color(imageLeft[i].ToVector3() * filterLeft +
                imageRight[i].ToVector3() * filterRight);
    Texture2D zlozenie3D = new Texture2D(gd, renderTargetRight.Width, renderTargetRight.Height);
    zlozenie3D.SetData<Color>(imageRight);

    spriteBatch.Begin(SpriteSortMode.Immediate, BlendState.Opaque, null, DepthStencilState.Default, null);
    spriteBatch.Draw(zlozenie3D, new Rectangle(0, 0, Window.ClientBounds.Width, Window.ClientBounds.Height),
                Color.White);
    if (showSeparateViews)
    {
        spriteBatch.Draw(renderTargetLeft, new Rectangle(0, 0, Window.ClientBounds.Width / 4, Window.ClientBound
                s.Height / 4), Color.White);
        spriteBatch.Draw(renderTargetRight, new Rectangle(3 * Window.ClientBounds.Width / 4, 0, Window.ClientBou
                nds.Width / 4, Window.ClientBounds.Height / 4), Color.White);
    }
    spriteBatch.End();

    base.Draw(gameTime);
}
```



Rysunek 3. *Anaglify - obrazy przygotowane dla okularów dwukolorowych. Od lewej: czerwony-cyjan, czerwony-niebieski i okularów ColorCode3D (bursztyn-niebieski).*

Metoda ta zbudowana jest z trzech bloków. Dwa pierwsze, odpowiedzialne za renderowanie sceny z pozycji lewego i prawego oka, są bliźniaczo podobne. W każdym ustawiany jest cel renderowania, czyszczony jego bufor, ustawiana macierz widoku zgodnie z pozycją lewej i prawej kamery i wreszcie wywoływana jest metoda odpowiedzialna za rysowanie bryły. W efekcie powstają dwie tekstury. Następnie przechowywane w nich piksele kopiujemy do dwóch jednowymiarowych tabel z elementami typu Color i składamy w jeden obraz z odpowiednio zmieszanymi składowymi RGB. Owo składanie to wyróżniona w kodzie pętla, której wykonanie zajmuje niemal cały używany przez program czas procesora CPU. Wobec tego z pewnością warto ją zrównoleglić, choćby używając metody Parallel.For, nowości w platformie .NET 4.0:

```
Parallel.For(0, imageRight.Count(), i => {
                    imageRight[i] = new Color(imageLeft
                    [i].ToVector3() * filterLeft + image
                    Right[i].ToVector3() * filterRight);
                    });
```

Uzyskany w ten sposób obraz wyświetlamy na całym obszarze okna korzystając z obiektu SpriteBatch. Dodatkowo rysujemy w jego górnych rogach obie pierwotne tekstury z widokami sceny z pozycji lewego i prawego oka.

W powyższym kodzie nie ma tak naprawdę niczego zaskakującego lub nowatorskiego. Co gorsze, jego wykonanie, nawet po zrównolegleniu, zajmuje więcej czasu, niż czas miedzy kolejnymi wywołaniami metod Update i Draw. Przez to własność GameTime.IRunningSlowly jest permanentnie włączona. Mimo to efekt, jaki zobaczymy na ekranie, oczywiście pomijając przestoje związane z problemami z wydajnością, jest naprawdę zadowalający. Zobaczymy na ekranie obraz analogiczny do tego widocznego na rysunku 3 (lewy). Możemy więc nałożyć okulary i sprawdzić czy pojawi się wrażenie głębi.

Kolor filtrów użytych do składania obrazu możemy dobrać modyfikując inicjacje pól filterLeft i filterRight. W tej chwili jest to czerwień i cyjan (por. listing 1 i rysunek 4). Dla okularów czerwono-niebieskich kolory filtrów powinny być następujące:

```
Vector3 filterLeft = new Vector3(1, 0, 0);
Vector3 filterRight = new Vector3(0, 0, 1);
```

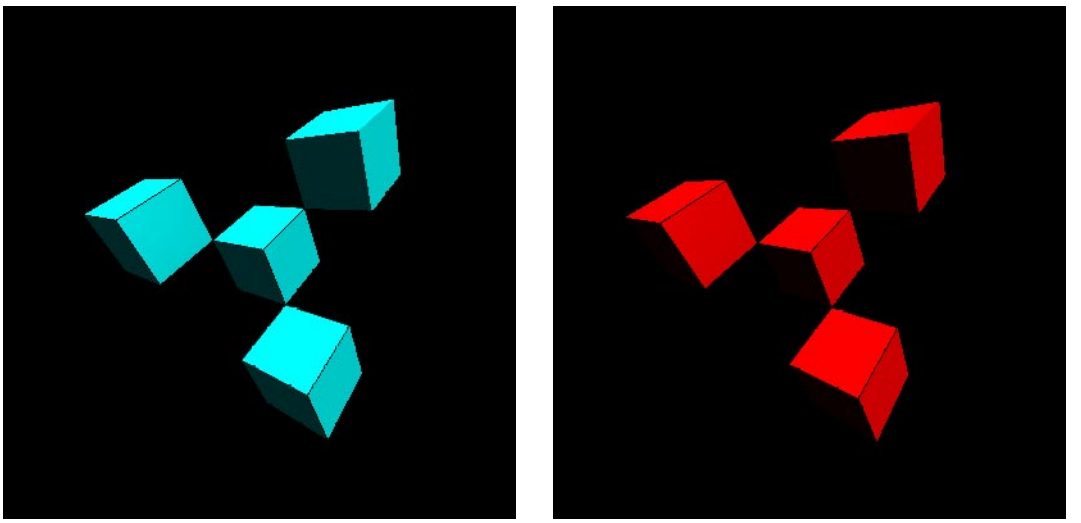
Jak widać z rysunku 3, środkowy filtry czerwony i niebieski stosowane razem nie zapewniają poprawnego odwzorowania kolorów. Kolory tych filtrów nie są dopełniające (ich sumą nie jest biel). W przypadku okularów bursztynowo-granatowych (ColorCode 3D) filtry powinny być natomiast następujące:

```
Vector3 filterLeft = new Vector3(0.85f, 0.85f, 0.3f);
Vector3 filterRight = new Vector3(0.15f, 0.15f, 0.7f);
```

Z CPU do GPU

Mimo swojej prostoty i niezawodności przedstawiona powyżej metoda ma jedną zasadniczą i dyskwalifikującą ją wadę – pętla po tekselach, której zadaniem jest złożenie obu tekstur, jest zabójcza dla wydajności całego programu. Nie pomaga nawet wykorzystanie wszystkich rdzeni CPU. Co więc możemy zrobić? Przenieśmy te obliczenia na GPU. Tam zamiast kilku, będziemy mieli kilkaset rdzeni. To jest w końcu procesor przystosowany do masywnie równoległych obliczeń.

Aby wykorzystać GPU w XNA musimy przygotować efekt – program napisany w języku HLSL. To język blisko spokrewniony z C, ale zawierający dodatkowe wbudowane typy (m.in. wektory dwu-, trój- i czteroelementowe, macierze o takich rozmiarach i funkcje operujące na tych typach), a ponadto mechanizm semantyk określający z jaką własnością werteksu związana



Rysunek 4. *Obraz z lewej i prawej kamery po nałożeniu filtrów czerwonego i cyjanowego.*

jest deklarowana zmienna. W efekcie powinny znaleźć się co najmniej dwie funkcje: shader werteksów zajmujący się obróbką poszczególnych werteksów wysłanych z aplikacji oraz shader pikseli wywoływany dla każdego piksela rysowanej bryły (zob. ramka „Potok renderowania").

Kod efektu, który powinniśmy dodać do podprojektu *StereoscopyContent* (plik typu *Effect file* z rozszerzeniem *.fx*) widoczny jest na listingu 4. Jego zasadniczą funkcją jest zastąpienie pętli widocznej na listingu 3. W tym celu musimy do pamięci karty graficznej przesłać obie tekstury uzyskane w dwóch pierwszych przebiegach renderowania. Dlatego efekt zaczyna się od definicji dwóch parametrów – tekstur oraz związanych z nimi próbników. Następnie zdefiniowane są dwa filtry – dla lewego i prawego okularu. Po nich znajduje się struktura opisująca werteks. W zasadzie potrzebna jest nam jedynie współrzędna teksturowania tj. wyrażone w pikselach położenie na teksturze (semantyka TEXCOORD0), ale musimy również dodać współrzędne położenia werteksu (semantyka POSITION0) wyrażone w układzie własnych modelu – tego domagać będzie się kompilator. Nie będziemy ich jednak używać. Naszym prawdziwym „inputem" będą bowiem nie werteksy, a piksele dwóch tekstur.

Przyjrzyjmy się teraz funkcjom VS i PS. To one będą wykonywane przez Vertex i Pixel Shadery. Funkcja VS to w zasadzie nic interesującego – przekazuje jedynie werteksy do interpolatora i pasteryzatora nie modyfikując ich w żaden sposób. Argumentem funkcji PS są zinterpolowane współrzędne. Nas interesują tylko współrzędne teksturowania, które umożliwiają odczytanie właściwych tekselów z obu tekstór. I to właśnie robią dwie pierwsze instrukcje funkcji PS (wywołania funkcji tex2D). Dalsza część kodu odpowiada zawartości pętli z listingu 3. Miesza teksele z obu tekstur z wagami odpowiadającym użytym filtrom. W przypadku okularów czerwono-cyjanowych, mieszanie to pole-

Potok renderowania

Shadery werteksów i shadery pikseli to programowalne jednostki kart graficznych. Tymi samymi nazwami określane są również wykonywane przez nie funkcje (u nas są to odpowiednio VS i PS). Danymi wejściowymi shadera werteksów są własności werteksu przesłane z aplikacji lub odczytane z bufora werteksów. W typowych sytuacjach, ale nie w naszym przykładzie, zadaniem funkcji shadera werteksów jest przeliczenie położenia werteksu ze współrzędnych modelu, poprzez współrzędne sceny do układu odniesienia kamery. Oznacza to ich przemnożenie kolejno przez macierz świata, widoku i rzutowania. Przeliczone położenie trafia do rasteryzatora, który odpowiada za przydzielenie pikseli do trójkątów. To dzięki rasteryzatorowi do shadera pikseli trafiają nie tylko trzy piksele odpowiadające wierzchołkom każdego trójkąta, ale także wszystkie piksele z jego wnętrza. Pozostałe własności werteksu, a więc w naszym przypadku tylko współrzędne teksturowania, trafiają do interpolatora. Dokonuje on interpolacji tych wielkości dla wszystkich pikseli wewnątrz trójkątów. Tak interpolowane dane o werteksie stanowią dane wejściowe shadera pikseli, który jest odpowiedzialny przede wszystkim za ustalenie ostatecznego koloru każdego piksela. W kartach graficznych zgodnych z DirectX 10 podział na jednostki uruchamiające shadery pikseli i werteksów stał się umowny. Jednostki są teraz uniwersalne i mogą być przydzielane dynamicznie do jednego, bądź do drugiego zadania. Ponadto w DirectX 11 pojawił się jeszcze shader geometrii, który pozwala na dodawanie i usuwanie werteksów. Jednak nie jest on jeszcze niestety obsługiwany w XNA. Cały proces renderowania został schematycznie przedstawiony na rysunku 5. Pamiętajmy jednak, że niekoniecznie musi on skończyć wyświetleniem obrazu na ekranie. W naszym programie – zamiast wyświetlać na ekranie, obraz będzie on także zapisywany do tekstur – zastępczych celów renderowania.

ga na użyciu składowej czerwonej pierwszej tekstury i składowych zielonej i niebieskiej drugiej. Wartością funkcji PS jest kolor, który użyty zostanie do rysowania w miejscu określonym przez współrzędne 3D, których wprawdzie jawnie nie używamy, ale które odpowiadają położeniu tekseli w oryginalnych teksturach (obie tekstury mają rozmiar ekranu). Oznacza to, że efektu powinniśmy użyć do rysowania na kwadracie dokładnie wypełniającym ekran. Tak też zrobimy.

Aby użyć powyższego efektu w programie zdefiniujmy pole typu `Effect`:

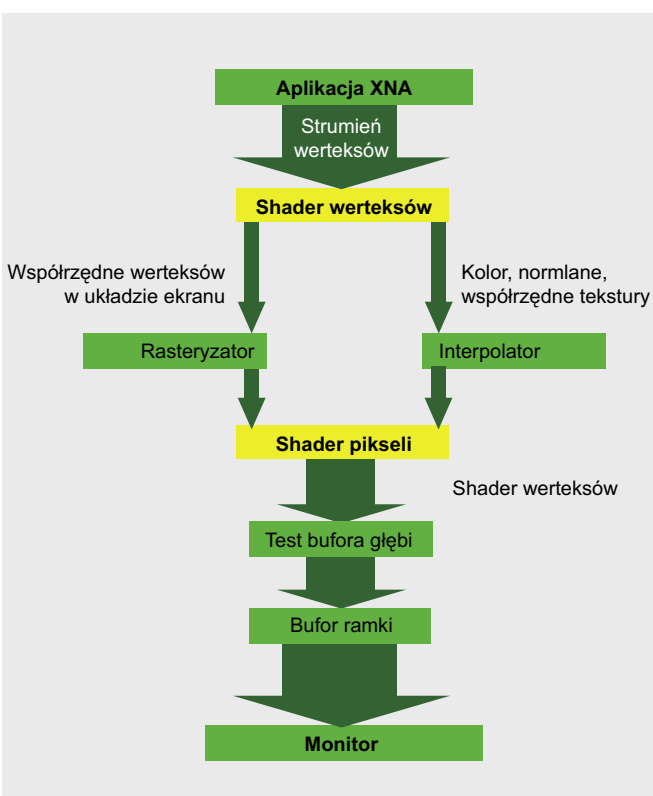
```
Effect superimposingEffect;
```

Następnie w metodzie `Game1.LoadContent` wczytujemy nowy efekt poleceniem:

```
superimposingEffect = Content.Load<Effect>("Stereoscopy
                    Effect");
```

Pozostaje jedynie przygotować kwadrat przesłaniający ekran. Jego werteksy są następujące:

```
VertexPositionTexture[] werteksyEkranu = new
                    VertexPositionTexture[4]
{
 new VertexPositionTexture(new Vector3(-1, 1, 0f), new
                    Vector2(0, 0)),
```



Rysunek 5. *Uproszczony schemat potoku renderowania. Nie ma w nim shadera geometrii, do którego i tak jednak nie ma jeszcze dostępu w XNA 4.0*

Listing 4. *Zapisany w pliku StereoscopyEffect.fx kod efektu odpowiedzialny za złożenie i wyświetlenie tekstur*

```
texture TextureLeft;
sampler TextureSamplerLeft = sampler_state
{
    texture = <TextureLeft>;
    magfilter = POINT;
    minfilter = POINT;
    mipfilter = POINT;
    AddressU = Clamp;
    AddressV = Clamp;
};

texture TextureRight;
sampler TextureSamplerRight = sampler_state
{
    texture = <TextureRight>;
    magfilter = POINT;
    minfilter = POINT;
    mipfilter = POINT;
    AddressU = Clamp;
    AddressV = Clamp;
};

float3 FilterLeft = float3(1,0,0);
float3 FilterRight = float3(0,1,1);

struct Vertex
{
    float4 Position : POSITION; //it's not used
    float2 TexCoord : TEXCOORD0;
};

Vertex VS(Vertex input)
{
    Vertex output;
    output.Position = input.Position;
    output.TexCoord = input.TexCoord;
    return output;
}

float4 PS(Vertex input): COLOR0
{
    float4 texelLeft = tex2D(TextureSamplerLeft,
                    input.TexCoord);
    float4 texelRight = tex2D(TextureSamplerRight,
                    input.TexCoord);

    float4 color;
    color.rgb = FilterLeft*texelLeft +
                    FilterRight*texelRight;
    color.a = 1;
    return color;
}

technique Technique1
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 VS();
        PixelShader = compile ps_2_0 PS();
    }
}
```

Listing 5. *Ostateczna postać metody Draw*

```
protected void Draw(GameTime gameTime)
{
    //Left camera
    gd.SetRenderTarget(renderTargetLeft);
    gd.Clear(Color.Black);
    if (cameraEnabledLeft)
    {
        solid.View = viewLeft;
        solid.Draw();
    }
    gd.SetRenderTarget(null);


    //Right camera
    gd.SetRenderTarget(renderTargetRight);
    gd.Clear(Color.Black);
    if (cameraEnabledRight)
    {
        solid.View = viewRight;
        solid.Draw();
    }
    gd.SetRenderTarget(null);


    //Superimposing
    superimposingEffect.CurrentTechnique = superimposingEffect.Techniques[0];
    superimposingEffect.Parameters["TextureLeft"].SetValue(renderTargetLeft);
    superimposingEffect.Parameters["TextureRight"].SetValue(renderTargetRight);
    superimposingEffect.Parameters["FilterLeft"].SetValue(filterLeft);
    superimposingEffect.Parameters["FilterRight"].SetValue(filterRight);


    foreach (EffectPass pass in superimposingEffect.CurrentTechnique.Passes)
    {
        pass.Apply();
        gd.DrawUserPrimitives<VertexPositionTexture>(PrimitiveType.TriangleStrip, werteksyEkranu, 0, 2);
    }


    if (showSeparateViews)
    {
        spriteBatch.Begin(SpriteSortMode.Immediate, BlendState.Opaque, null, DepthStencilState.Default, null);
        spriteBatch.Draw(renderTargetLeft, new Rectangle(0, 0, Window.ClientBounds.Width / 4, Window.ClientBound
                    s.Height / 4), Color.White);
        spriteBatch.Draw(renderTargetRight, new Rectangle(3 * Window.ClientBounds.Width / 4, 0, Window.ClientBou
                    nds.Width / 4, Window.ClientBounds.Height / 4), Color.White);
        spriteBatch.End();
    }


    base.Draw(gameTime);
}
```

```
 new VertexPositionTexture(new Vector3(1, 1, 0f), new
                   Vector2(1, 0)),
 new VertexPositionTexture(new Vector3(-1, -1, 0f), new
                   Vector2(0, 1)),
 new VertexPositionTexture(new Vector3(1, -1, 0f), new
                   Vector2(1, 1))
};
```

Po tych przygotowaniach możemy użyć nowego efektu do połączenia tekstur i ich wyświetlenia. W tym celu należy zmodyfikować końcówkę metody Game1.Draw (listing 5). Po tej zmianie nie będzie już z pewnością problemu z nienadążaniem za częstością odświeżania. Program zaczyna działać płynnie, a jednocześnie użycie CPU spada niemal do zera.

Nasza implementacja animowanego anaglifu jest trójprzebiegowa. Dwa razy wywoływany jest standardowy BasicEffect i na koniec napisany samodzielnie StereoscopyEffect. Nie wątpię, że korzystając z różnych trików można by ilość przebiegów ograniczyć do dwóch. Szczególnie jeżeli skorzystamy z mieszania kolorów (*alpha blendingu*), który jednak narzuca pewne ograniczenia w sposobie rysowania sceny. Nasza implementacja jest wolna od tych założeń, a jednocześnie na tyle ogólna, że może być bez trudu użyta w dowolnym programie XNA. Wszystko dzięki temu, że dwa pierwsze przebiegi mogą być zrealizowane w zupełnie dowolny sposób.

Więcej w sieci

- *http://theinstructionlimit.com/?p=123* – Strona gry HyperCUBE, która była naszą inspiracją, a blog jest twórców – przewodnikiem.

WOJCIECH STAŃCZYK, JACEK MATULEWSKI

Jacek Matulewski - Fizyk zajmujący się na co dzień optyką kwantową i układami nieuporządkowanymi na Wydziale Fizyki, Astronomii i Informatyki Stosowanej Uniwersytetu Mikołaja Kopernika w Toruniu. Jego specjalnością są symulacje ewolucji układów kwantowych oddziaływujących z silnym światłem lasera.

Od 1998 interesuje się programowaniem dla systemu Windows. Ostatnio zainteresowany platformą .NET i językiem C#. Wierny użytkownik kupionego w połowie lat osiemdziesiątych "komputera osobistego" ZX Spectrum 48k.

Wojciech Stańczyk - Programista pracujący na co dzień w firmie DOT.NET (http://www.dot.net.pl), gdzie zajmuje się tworzeniem rozwiązań programistycznych dla platformy Microsoft .NET. Poza pracą stale rozwija swoje umiejętności w zakresie programowania grafiki komputerowej, w tym efektów stereoskopowych. Od ponad 12 lat wielki pasjonata chińskich sztuki walki Kung Fu Tang Lang Men.

Artykuł jest fragmentem pracy magisterskiej WS napisanej pod kierunkiem JM.