Rozdział 1
Podstawowe koncepcje grafiki 3D

Wersja z 2014-03-20

Projektowanie gry musimy zacząć od przygotowania sobie środowiska pracy. Zakładam, że Czytelnik posiada już doświadczenie w programowaniu w języku C#, a w szczególności zna środowisko Visual Studio – zaczynać będziemy od aplikacji dla Windows, które tworzy się właśnie w Visual Studio. Konieczne jest jeszcze zainstalowanie MonoGame. Należy pobrać pakiet instalacyjny (plik *.exe*) ze strony: *http://monogame.codeplex.com/*.

Błędy w trakcie i po instalacji:
1. Pomimo dołączenia do pakietu instalacyjnego MonoGame biblioteki OpenAL (biblioteka obsługująca dźwięk firmowana przez Creatibe Labs), przy próbie uruchomienia pierwszego projektu może pojawić się błąd z komunikatem o jej braku. Należy wówczas samodzielnie zainstalować OpenAL (plik instalacyjny oalinst.exe jest w pakiecie instalacyjnym MonoGame (można go otworzyć za pomocą 7-Zip) w katalogu *$SHELL[17]\MonoGame\v3.0* (zob. *https://monogame.codeplex.com/discussions/357014*).
2. Po instalacji powinny być dostępne szablony w Visual Studio 2010 i 2012. Jeżeli ich nie ma należy odpowiednie pliki skopiować do katalogu *c:\Users\[użytkownik]\Documents\Visual Studio 2012\Templates\ProjectTemplates\Visual C#* (można je wydobyć z pliku instalacyjnego lub skopiować z innego komputera). W ten sam sposób można dodać szablony do Visual Studio 2013.
3. Pojawia się również problem braku pliku *SDL.dll* (to pochodna problemu 2), także wtedy, gdy przenosimy projekt do innego katalogu lub komputera. Obejście: skopiować plik *SDL.dll* samodzielnie, a w projekcie (w *Solution Explorer*) zmienić jego własność *Copy to Output Directory* na *Do not copy*.

Polskie litery w kodzie źródłowym z zajęć (np. `werteksyTrójkąta`)

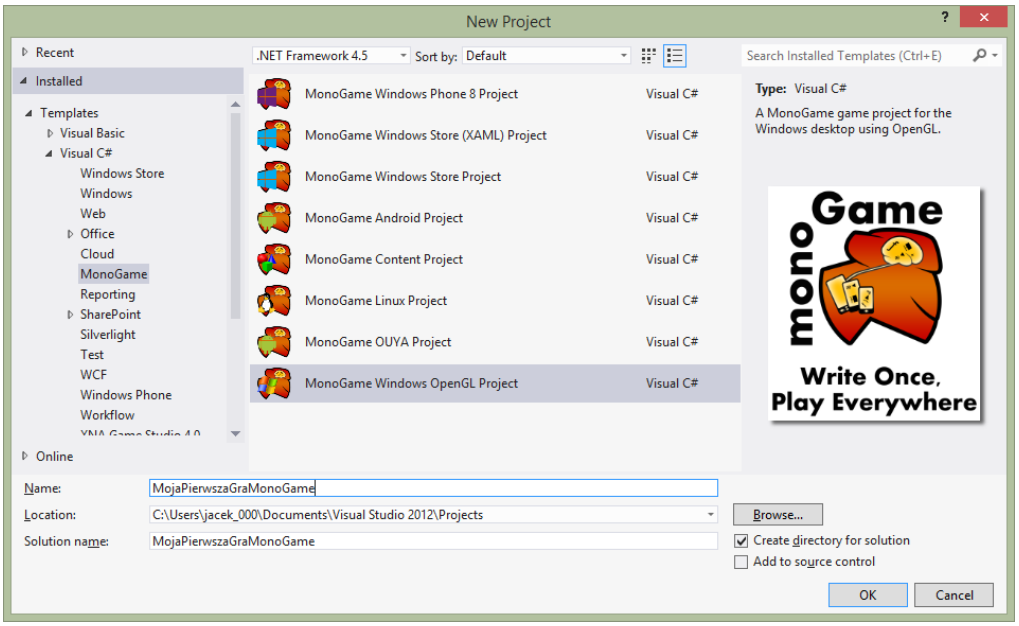
Tworzenie projektu gry

WSKAZÓWKA: W książce, do opisu MonoGame korzystam z szablonu *Visual C#*, *MonoGame*, *MonoGame Windows OpenGL Project*. Windows 8 (na końcu rozdziału suplement) + Android.

WSKAZÓWKA: Ustalmy, że grą nazywać będziemy każdą aplikację utworzoną za pomocą MonoGame bez względu na to, czy jest to rzeczywiście gra, czy tylko program rysujący trójkąt.

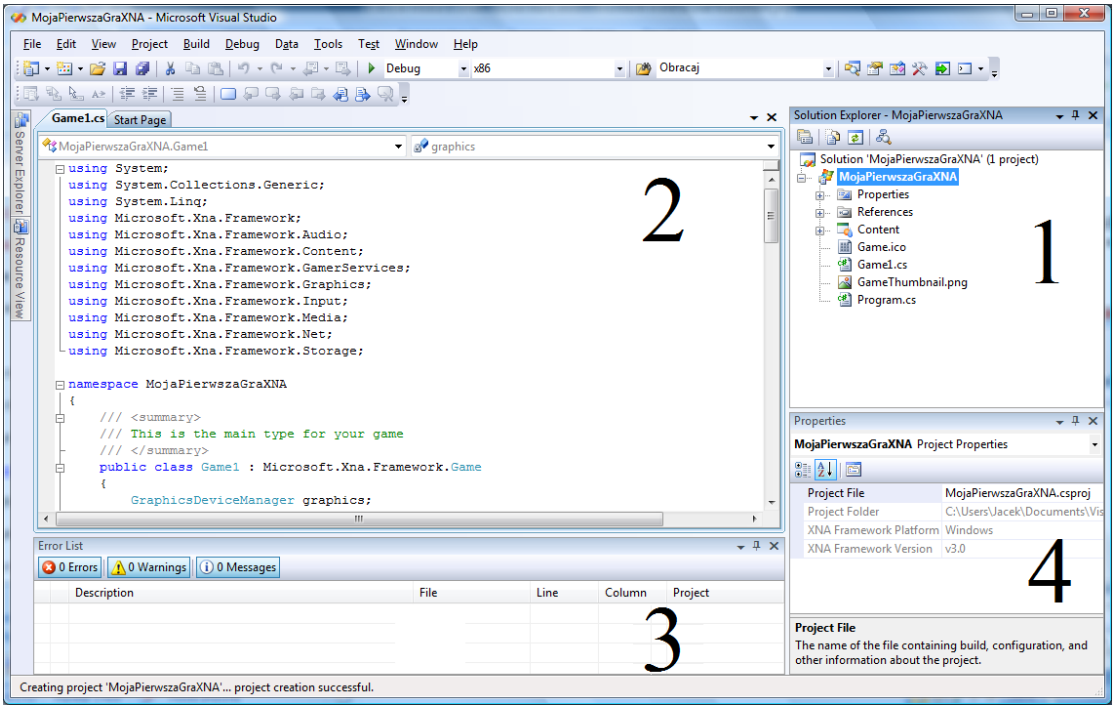
Aby stworzyć pierwszą grę:

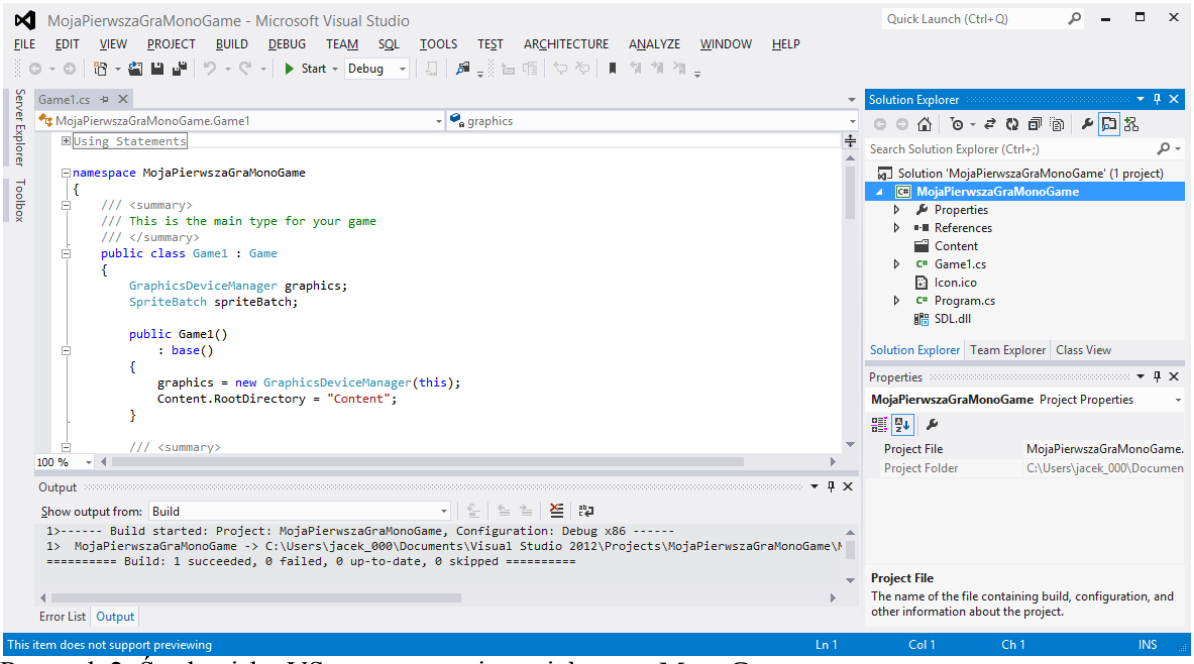
1. Uruchamiamy Visual Studio z zainstalowanym szablonem MonoGame.
2. Z menu *File* wybieramy podmenu *New*, a następnie *Project…* (alternatywnie naciskamy *Ctrl+Shift+N*).
3. Pojawi się typowe dla Visual Studio okno pozwalające na wybór typu projektu (rysunek 1), w którym
    a. na drzewie zawierającym kategorie szablonów zaznaczamy *MonoGame*;
    b. z listy szablonów projektów w prawej części owego okna zaznaczamy *MonoGame Windows OpenGL Project*,
    c. w polu *Name* wpisujemy nazwę projektu np. *MojaPierwszaGraMonoGame*,
    d. wreszcie klikamy przycisk *OK*.



Rysunek 1. Wybór projektu

Po chwili Visual Studio stworzy projekt składający się z kilku plików. Wszystkie one przedstawione zostaną w podoknie o nazwie *Solution Explorer* (numer 1 na rysunku poniżej). W środkowej części okna widoczny jest kod C# najważniejszego pliku projektu *Game1.cs* (numer 2). Na dole widoczny jest pasek, na którym pojawiać się będą komunikaty o błędach i ostrzeżenia (numer 3). W prawym dolnym roku znajduje się domyślnie okno własności (numer 4). W przypadku projektów MONOGAME ma ono mniejszą rolę i nie będziemy z niego korzystać zbyt często.





Rysunek 2. Środowisko VS po utworzeniu projektu gry MonoGame

Klasa gry

Plik *Game1.cs* zawiera kod klasy `Game1`. Jest to odpowiednik klasy `Form1` ze zwykłych okienkowych projektów Visual C#. Klasa ta implementuje okno gry. Już w tej chwili możemy nacisnąć klawisz *F5* (lub zielony trójkąt na pasku narzędzi), aby skompilować i uruchomić projekt. W efekcie zobaczymy jednak tylko okno z radośnie błękitnym obszarem użytkownika. Nie należy jednak niedoceniać szablonu – dla naszej wygody klasa `Game1` zawiera wszystkie istotne z punktu widzenia programisty metody. Prześledźmy zatem wspólnie obecną zawartość klasy `Game1`.

- Zadaniem konstruktora klasy `Game1` jest utworzenie instancji obiektu `graphics` typu `GraphicsDeviceManager`. Jak przekonamy się jeszcze w tym module jest to obiekt kluczowy z punktu widzenia programowania grafiki w MonoGame. W jednej grze może być zainicjowany tylko jeden obiekt tego typu. W konstruktorze można również zmienić profil grafiki (o tym więcej poniżej).
- Metoda `Initialize` jest miejscem przeznaczonym na wszelki kod inicjujący nasz świat 3D. Jeżeli na scenie chcemy umieścić samolot, to ustalenie jego pozycji początkowej i prędkości powinno nastąpić właśnie w tym miejscu.
- Metoda `LoadContent` jest najlepszym miejscem do wczytania tekstur, dźwięków, modeli i innych danych z plików. Zwalnianie tych zasobów, jeżeli to jest konieczne, powinno być wykonane w metodzie `UnloadContent`.
- Wszystkie powyższe metody były uruchamiane tylko raz tuż po uruchomieniu programu (poza `UnloadContent`, która jest uruchamiana przed jego zakończeniem). Dwie kolejne metody tj. `Update` i `Draw` są natomiast uruchamiane cyklicznie z częstością odpowiadającą mniej więcej częstości odświeżania karty graficznej i monitora (~60Hz w przypadku monitora LCD). Metoda `Update` służy do aktualizacji stanu obiektów odpowiedzialnych za wygląd sceny np. pozycji i kierunku samolotu, podczas gdy metoda `Draw` służy do renderowania (rysowania) kolejnej klatki gry.

Profile

Tu wstawić informacje o profilowaniu grafiki:
Reach – tu WDDX 1.0, WP7, PS 2.0
HiDef – WDDX >=1.1, PS 3.0

W MonoGame, w ustawieniach projektu nie ma zakładki XNA Game Studio, na której możnaby wybrać profil. Ale profile są nadal dostępne w kodzie. Wybór w konstruktorze (graphics.GraphicsProfile=GraphicsProfile.Reach;)? Nie da się wyklikać, można tylko zmienić domyślny HiDef na Reach z poziomu kodu.

Werteksy i rysowanie prymitywu (trójkąta)

Jak narysować prostą figurę płaską np. trójkąt? Przede wszystkim potrzebujemy tablicę werteksów opisujących jego położenie. Zadeklarujmy zatem prywatne pole klasy `Game1`, które będzie tablicą obiektów typu `VertexPositionColor`.

```
private VertexPositionColor[] werteksyTrojkata =
   new VertexPositionColor[3]{
      new VertexPositionColor(new Vector3(0.5f, -0.5f, 0), Color.White),
      new VertexPositionColor(new Vector3(-0.5f, -0.5f, 0), Color.White),
      new VertexPositionColor(new Vector3(0, 0.5f, 0), Color.White)};
```

Werteksy to punkty w przestrzeni trójwymiarowej, które **są wierzchołkami figur**. Dlaczego zatem nie mówi się o nich po prostu punkty? Dlatego, że z werteksami związana jest większa ilość informacji niż tylko trzy współrzędne wektora położenia. Z werteksem związany może być też kolor wierzchołka, co widzimy na powyższym listingu, ale także wektor normalny, czy współrzędna tekstury. My na razie ograniczamy się tylko do położenia opisywanego wektorem typu `Vector3` i koloru.

Układ współrzędnych, którego używamy definiując położenie werteksu jest takim samym kartezjański układem współrzędnych, jaki poznajemy w szkole podstawowej. Oznacza to, że jego osie są prostoliniowe i prostopadłe, a sam układ prawoskrętny (oś OZ skierowana jest do nad, a nie w głąb ekranu).

Czasem stosuje się opis położenia we współrzędnych jednorodnych. Co więcej to właśnie w tym układzie współrzędnych pracuje karta graficzna. Ten układ współrzędnych poza trzema „zwykłymi" współrzędnymi położenia (x,y,z) używa także czwartej współrzędnej w nazywanej współczynnikiem skalowania. Ich najważniejszą zaletą jest to, że we współrzędnych jednorodnych wszystkie przekształcenia, włączając w to translacje i rzutowania z perspektywą, mogą być opisane przez macierze 4x4. Dzięki temu karta graficzna musi jedynie szybko mnożyć macierze.

Zalecane jest skopiowanie tablicy werteksów do przechowywanego w pamięci karty graficznej bufora werteksów (tym zajmiemy się niżej). Związane jest to z wyraźnym nastawieniem MonoGame, tak jak i jego pierwowzoru – XNA, na pisanie własnych shaderów werteksów i pikseli. Definiowaniu własnych shaderów poświęcona jest druga część książki (rozdziały 11-14). A na razie użyjemy domyślnych shaderów zaszytych w klasie `BasicEffect`. Zadeklarujmy zatem pole, które będzie przechowywać referencję do obiektu tego typu.

```
private BasicEffect efekt;
```

Do jego inicjacji doskonale nadaje się metoda `Initialize`. Umieśćmy w niej instrukcję tworzącą obiekt klasy `BasicEffect`[1].

```
protected override void Initialize()
{
   efekt = new BasicEffect(graphics.GraphicsDevice);

   base.Initialize();
}
```

[1] W wersji XNA 4.0 konstruktor klasy `BasicEffect` nie pozwala już na użycie puli efektów – konstruktor jest jednoargumentowy.

Następnie zlokalizujmy metodę `Draw` klasy `Game1`. Jak już wiemy, to ona jest odpowiedzialna za renderowanie poszczególnych kadrów wyświetlanych przez MonoGame. W kontekście rysowania sceny kluczowe znaczenie ma obiekt reprezentujący urządzenie, na którym wyświetlana będzie nasza grafika (karta graficzna + monitor). W klasie `Game1` jest to `graphics.GraphicsDevice`, przy czym `graphics` jest polem klasy przechowującym referencję do wspomnianego już obiektu typu `GraphicsDeviceManager`. Obiekt `GraphicsDevice` pozwala na pobranie i ustawienie wielu parametrów wyświetlania (od rozdzielczości ekranu po sposób ukrywanie jednej ze stron figur). Dla ułatwienia proponuję zatem w metodzie `Draw` zdefiniować referencję `gd`, która będzie ułatwiać odwoływanie się do tego obiektu.

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

   GraphicsDevice gd = graphics.GraphicsDevice;
   gd.Clear(Color.Black);

   ...
```

Przy okazji zmieniłem błękitne tło, charakterystyczne dla „pustego” projektu MonoGame na bardziej gustowne - czarne. Kolor tła jest argumentem metody `gd.Clear` tj. wartością, która zapisywana jest do bufora reprezentującego zawartość prezentowaną w oknie gry w momencie usuwania (nadpisywania) jego zawartości.

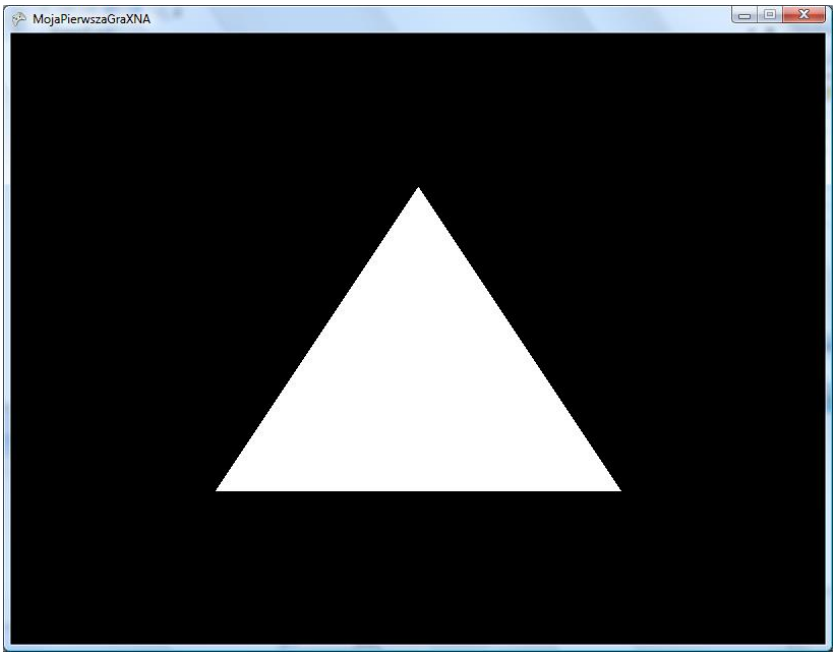
I wreszcie skorzystamy z domyślnego shadera dostępnego przez referencję `efekt` do narysowania trójkąta. Kod, który do tego celu przygotujemy zawiera pętlę `foreach`, która ma w ogromnej większości przypadków taką postać, jak ta przedstawiona poniżej. Pętla iteruje przebiegi (ang. *passes*) zdefiniowane w bieżącej technice efektu. Oba terminy: technika i przebieg staną się jasne, gdy poznamy język HLSL i będziemy tworzyć własne efekty (rozdziały 11-14). Na razie skazani jesteśmy na użycie tego fragmentu kodu bez jego pełnego zrozumienia. Ostatecznie cała metoda będzie wyglądała następująco:

```
protected override void Draw(GameTime gameTime)
{
   GraphicsDevice gd = graphics.GraphicsDevice;
   gd.Clear(Color.Black);
   gd.VertexDeclaration = new VertexDeclaration(gd, VertexPositionColor.VertexElements);

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

      //Instrukcje rysujace figury (prymitywy)
      gd.DrawUserPrimitives<VertexPositionColor>(
            PrimitiveType.TriangleList,
            werteksyTrojkata,
            0,
            1);
   }

   base.Draw(gameTime);
}
```



Rysunek 3. Pierwsza figura

Do rysowania użyliśmy funkcji `gd.DrawUserPrimitives` parametryzowanej klasą `VertexPositionColor`. Rysuje ona figurę typu wskazanego w pierwszym argumencie. Drugi argument powinien być tablicą wierzchołków. W trzecim i czwartym informujemy od którego wierzchołka w tablicy powinniśmy zacząć i ile trójek wierzchołków (w przypadku trójkątów) wykorzystać tj. ile figur narysować.

Przyjąłem konwencję używania polskich nazw zmiennych i klas poza sytuacjami, w których nie mają one dobrego polskiego tłumaczenia (np. *passes*).

WSKAZÓWKA: W klasie werteksu jest pole `VertexDeclaration`. Pole to, w naszym przypadku `VertexPositionColor.VertexDeclaration`, jest opcjonalnym piątym argumentem metody `gd.DrawUserPrimitives`.

Nawijanie

Proszę teraz zamienić kolejność werteksów w tablicy `werteksyTrojkata` np. pierwszego i drugiego. Zmieni to kolejność tzw. **nawijania werteksów**, które wyznacza przednią i tylnią stronę figury. W oryginalnej wersji werteksy nawijane były w taki sposób (zgodnie ze wskazówkami zegara), że przód trójkąta skierowany jest w stronę kamery. Zasada jest bowiem taka, że jeżeli staniemy „za" figurą, to werteksy powinny być nawijanie odwrotnie do wskazówek zegara. Jest to o tyle ważne, że przy domyślnych ustawieniach jeżeli figura zwrócona jest do kamery tylną stroną, to nie jest renderowana. W efekcie zmiana kolejności werteksów powoduje, że trójkąt odwraca się do nas tylną stroną i tym samym staje się niewidoczny. Dzięki temu karta graficzna nie traci czasu na renderowanie wnętrz brył, o ile dopilnujemy, aby te zbudowane były w z figur (trójkątów), których przody skierowane są na zewnątrz.
To domyślne ustawienie można jednak zmienić. Następujące instrukcje

```
gd.RasterizerState = RasterizerState.CullCounterClockwise;
```
Jeżeli poza metodą Draw, trzeba zdefiniować pole gd

odtwarza domyślne ustawienia, a więc podczas renderowania pomijane są te figury, których werteksy nawijane są (patrząc od strony kamery) w kierunku przeciwnym do wskazówek zegara, a więc tych, które są skierowane do nas tyłem. Możemy użyć instrukcji, która zmienia usuwaną stronę figur:

```
gd.RasterizerState = RasterizerState.CullClockwise;
```

A jeżeli nie chcemy w ogóle zajmować się kolejnością wierzchołków, co nie jest dobrym pomysłem, ale przy jednej figurze jest do zaakceptowania, możemy wyłączyć cały mechanizm ukrywania instrukcją:

```
gd.RasterizerState = RasterizerState.CullNone;
```

My jednak pozostaniemy przy ustawieniach domyślnych (pierwsza z powyższych wartości), a więc musimy również przywrócić pierwotną kolejność werteksów.

Predefiniowane wartości obiektu `RasterizerState` dotyczą jedynie ukrywania tylnich powierzchni. Jednak ten stan zawiera więcej ustawień dotyczących wyświetlania pikseli. Możemy na przykład zmienić sposób pracy rasteryzatora (o nim poniżej) tak, żeby zamiast wskazania wszystkich pikseli wewnątrz trójkąta, wskazywał tylko te leżące na jego krawędziach. Wymaga to jednak samodzielnego stworzenia i skonfigurowania obiektu:

```
RasterizerState rs = new RasterizerState();
rs.CullMode = CullMode.None;
rs.FillMode = FillMode.WireFrame;
this.graphics.GraphicsDevice.RasterizerState = rs;
```

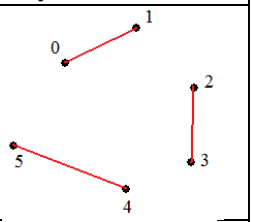
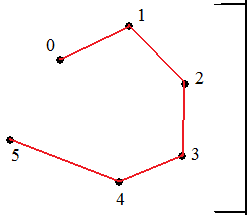
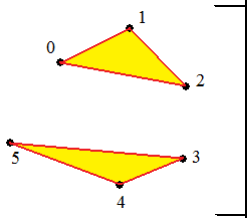
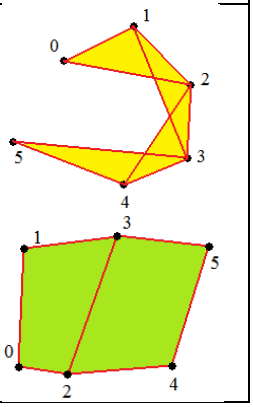
Prymitywy

Prymityw to określenie **predefiniowanej figury geometrycznej rozpiętej na werteksach**. W MonoGame możemy korzystać z kilku podstawowych prymitywów. Wszystkie wymienione są w typie wyliczeniowy `PrimitiveType`. Są wśród nich linie oraz trójkąty. Prymityw określany stałą `PrimitiveType.LineList` wymaga parzystej ilości werteksów. Oznacza to, że możemy zobaczyć tylko jedną linię tego typu. Możemy natomiast wybrać, czy ma to być linia między pierwszym i drugim wierzchołkiem, czy może między drugim a trzecim. Decyduje o tym trzeci argument metody `gd.DrawUserPrimitives`, czyli offset np.

```
gd.DrawUserPrimitives<VertexPositionColor>(
      PrimitiveType.LineStrip,
      werteksyTrojkata,
      1,
      1);
```

Dwie linie możemy narysować korzystając z ciągu linii (`PrimitiveType.LineStrip`). Wówczas trzeci argument musi być w naszym przypadku równy 0, a czwarty 2. Niestety MonoGame nie pozwala na narysowanie konturu – aby domknąć trójkąt rysowany liniami, musielibyśmy tablice werteksów uzupełnić o kopię pierwszego werteksu. [Zawsze można przełączyć tryb wypełniania w RasterizerState.Fill na WireFrame]
W przypadku trójkątów, w MonoGame 3 mamy już tylko dwie możliwości: 1) oddzielne trójkąty „konsumujące” trzy werteksy z tablicy na każdy z nich lub 2) ciąg trójkątów, w którym każdy nowy rysowany jest z dwóch ostatnich werteksów poprzedniego trójkąta i nowego werteksu.
Wszystkie prymitywy zostały zebrane w tabeli widocznej poniżej. Jak widać wśród nich nie mamy czworokąta. Taka decyzja twórców jest podyktowana najprawdopodobniej faktem, że czworokąt jest figurą, której cztery wierzchołki mogą być niewspółpłaszczyznowe, a więc i tak musi być rysowana z dwóch trójkątów. A jeżeli skorzystamy do tego z wachlarza lub ciągu trójkątów, wymagane są wówczas i tak tylko cztery werteksy. Wśród prymitywów nie ma też wielokąta o dowolnej ilości wierzchołków.

Tabela 1. Prymitywy dostępne w MonoGame 3

Prymityw	Stała z `PrimitiveType`	Przykład
Linie	`PrimitiveType.LineList`	
Ciąg linii	`PrimitiveType.LineStrip`	
Trójkąty	`PrimitiveType.TriangleList`	
Ciąg trójkątów	`PrimitiveType.TriangleStrip`	

Kolory jako własności werteksów

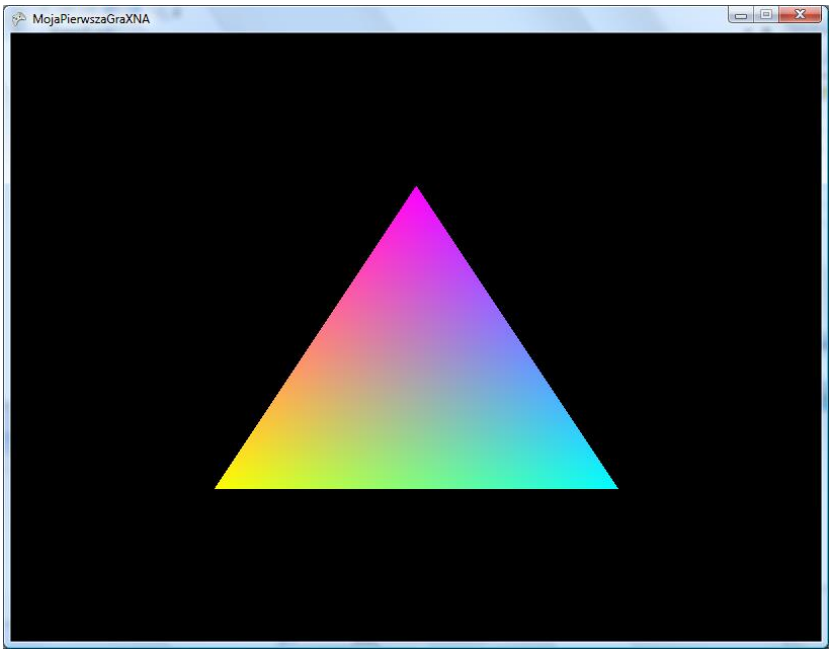
Używany przez nas typ werteksu `VertexPositionColor` zakłada, że poza pozycją ustalamy również kolor każdego z trzech werteksów. Definiując tablicę `werteksyTrojkata` z wszystkimi werteksami związałem kolor biały, co oczywiście można uznać jedynie za minimum. Nietrudno wyobrazić sobie, że ich kolor zmieniamy na żółty lub błękitny. Trudniej natomiast – że z każdym werteksem wiążemy inny kolor, np.:

```
private VertexPositionColor[] werteksyTrojkata =
   new VertexPositionColor[3]{
      new VertexPositionColor(new Vector3(0.5f, -0.5f, 0), Color.Cyan),
      new VertexPositionColor(new Vector3(-0.5f, -0.5f, 0), Color.Yellow),
      new VertexPositionColor(new Vector3(0, 0.5f, 0), Color.Magenta)};
```

Zanim wyświetlimy kolorowy trójkąt musimy jeszcze uaktywnić obsługę kolorów przez efekt. W tym celu w metodzie `Game1.Initialize` dodajmy instrukcję:

```
efekt.VertexColorEnabled = true;
```

Po kompilacji uzyskamy cieniowany trójkąt jak na poniższym rysunku.



Rysunek 4. Cieniowanie (interpolacja kolorów przypisanych do wierzchołków)

Podstawowe przekształcenia

Jak w każdej bibliotece grafiki trójwymiarowej, także i w MonoGame możemy rysowanych aktorów obracać, przemieszczać i skalować. Odpowiada za to macierz świata (dostępna przez referencję `efekt.World`), która przekształca werteksy przed ich narysowaniem. Jest to macierz o rozmiarach 4x4 (por. wspomniane wyżej współrzędne jednorodne). Przekształcenia wykonane na tej macierzy kumulują się jeżeli nowe macierze mnożymy przez macierz zastaną. Należy jednak pamiętać, że przekształcenia wykonane zostaną od ostatniego w listingu do pierwszego i co jeszcze ważniejsze kolejność wykonanych operacji ma zasadnicze znaczenie dla końcowego położenia i orientacji obiektów.

Weź do ręki książkę. Wzdłuż jej krawędzi wyznacz trzy osie układu współrzędnych: OX, OY i OZ. Następnie obrócić ją wzdłuż osi OX, a następnie wokół osi OY. Zapamiętaj orientację książki. Teraz odwróć książkę wpierw wokół osi OY, a później wokół osi OX. Jak teraz książka jest zorientowana?
Czy obroty są przemienne? Czy w ogóle mnożenie macierzy, w tym macierzy opisujących obroty, jest przemienne? Czy translacje są przemienne? A czy przemienne są translacje z obrotami?

Poniższy fragment kodu, który należy umieścić w metodzie `Update`, i w którym można usuwać i przestawiać poszczególne instrukcje, może pomóc w zrozumieniu problemu kolejności przekształceń:

```
Matrix macierzPrzeksztalcenia = Matrix.Identity;
macierzPrzeksztalcenia *= Matrix.CreateScale(0.5f);
macierzPrzeksztalcenia *= Matrix.CreateRotationZ(-MathHelper.PiOver2);
macierzPrzeksztalcenia *= Matrix.CreateTranslation(0.5f, 0, 0);
efekt.World = macierzPrzeksztalcenia;
```

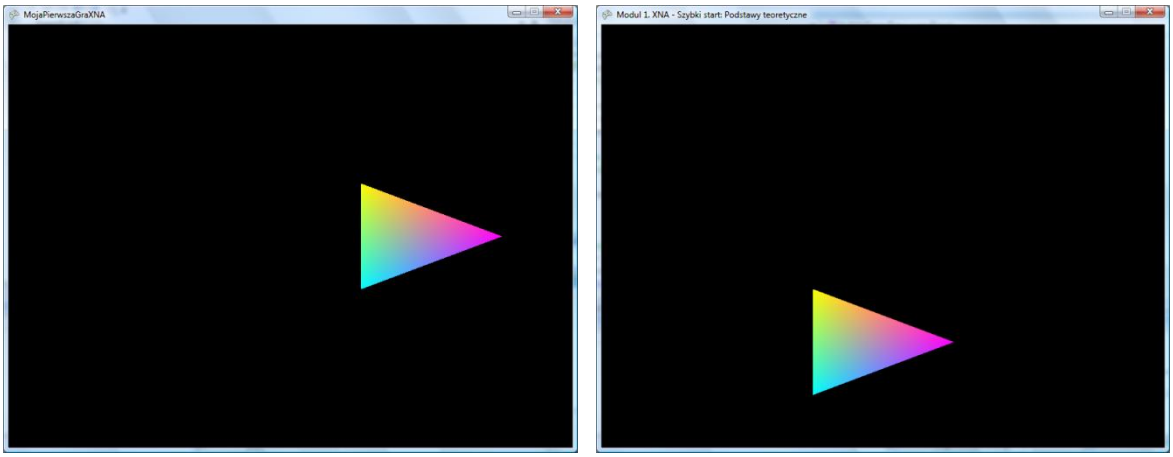
W szczególności proponuję zamienić miejscami obrót o kąt prosty i translację:

```
Matrix macierzPrzeksztalcenia = Matrix.Identity;
macierzPrzeksztalcenia *= Matrix.CreateScale(0.5f);
macierzPrzeksztalcenia *= Matrix.CreateTranslation(0.5f, 0, 0);
macierzPrzeksztalcenia *= Matrix.CreateRotationZ(-MathHelper.PiOver2);
efekt.World = macierzPrzeksztalcenia;
```

Proszę zwrócić uwagę, że jako argumentu metody statycznej `Matrix.CreateRotationZ` użyliśmy stałej z klasy `MathHelper` tj. `-MathHelper.PiOver2`. Oszczędza to czas procesora CPU, który musiałby wykonywać operacje dzielenia i rzutowania, gdybyśmy użyli tradycyjnego rozwiązania tj. `(float)-Math.PI / 2.0f`.

Jak wspomniałem wcześniej w MonoGame, podobnie, jak w OpenGL, a przeciwnie niż w Direct3D, stosowany jest prawoskrętny układ współrzędnych. Takiego układu uczyłeś się w szkole. Oznacza to, że oś Z skierowana jest do kamery, a nie w głąb sceny, jak w Direct3D. Zatem jeżeli chcemy odsunąć obiekt od kamery (przesunąć go w głąb sceny), należy jego współrzędne z ustawić na ujemne wartości.

Umieściłem powyższy zestaw instrukcji w metodzie `Update`. To najlepsze dla nich miejsce. Można również wstawić je do metody `Draw`, ale przed narysowaniem trójkąta – choć ta powinna zasadniczo służyć wyłącznie do operacji związanych *stricte* z renderowaniem. Oczywiście powyższe przekształcenia nie zmieniają się w czasie, można by więc je nawet przenieść do `Initialize`. Nie robię tego jednak ze względu na późniejsze modyfikacje idące w kierunku animacji.



Rysunek 5. Efekt translacji i rotacji w różnej kolejności

Należy pamiętać, że jeżeli w wyniku obrotu trójkąt obróci się do nas tyłem, stanie się niewidoczny. Możemy temu zapobiec zmieniając tryb pomijania stron figur np. ustawiając `gd.RasterizerState = RasterizerState.CullNone;`.

Ten sam efekt uzyskamy przypisując macierz przekształcenia macierzy widoku `efekt.View`. Jaka jest zatem różnica między macierzą widoku, a macierzą świata? Intencją autorów platformy jest, aby macierz świata opisywała przekształcenia sceny, podczas gdy macierz widoku opisuje zmianę pozycji i kierunku kamery. Jednak, jako że ruch jest względny, z punktu widzenia widza nie ma między tymi ruchami zasadniczej różnicy (poza tym, że przekształcenia kamery i aktora, które mają skutkować takim samem ruchem na ekranie, powinny być skierowane w przeciwnych kierunkach). Tak jest również w rzeczywistości. Jeżeli siedzimy w stojącym na peronie pociągu na idealnie gładkich szynach i nagle widzimy ruch względem pociągu na sąsiednim torze, nie wiemy, czy rusza nasz pociąg, czy sąsiedni. Dlatego np. w OpenGL obie macierze tworzyły jedną – macierz model-widok. Jest jednak różnica praktyczna – ponieważ przekształcenia na macierzach należy wykonywać w odwrotnej kolejności, wygodnie jest mieć osobną macierz widoku, w której ustalamy położenie kamery i więcej już się nią nie zajmujemy.

Podsumujmy. **Macierz świata** opisuje transformacje układu odniesienia sceny (aktorów względem sceny). **Macierz widoku** przekształca współrzędne werteksów w nieruchomym układzie odniesienia świata do układu odniesienia kamery, z punktu widzenia której tworzony jest obraz. Następnie tak

przetransformowane współrzędne, za pomocą **macierzy rzutowania**, rzutowane są na dwuwymiarowy układ współrzędnych ekranu.

W bieżącym stanie projektu gry próba przesunięcia obiektu w głąb sceny nie powoduje jego zmniejszenia. Dlaczego?

Rzutowanie sceny na ekran

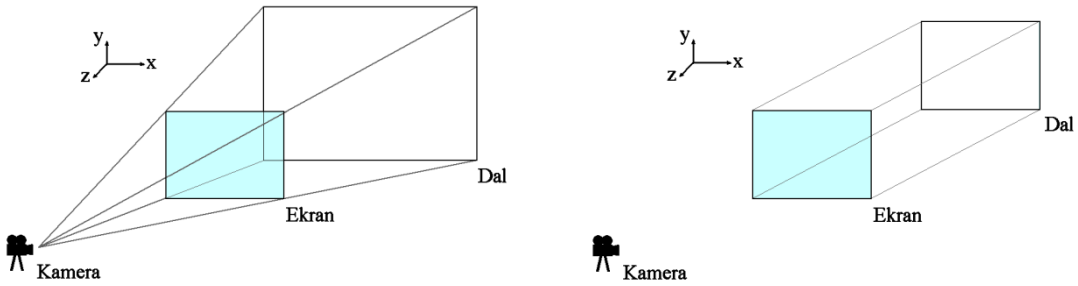
Po obrocie trójkąt wydłużył się w poziomie. Co gorsze już w położeniu pierwotnym nasz trójkąt ma wypaczony kształt – jest zbyt szeroki. To oczywiście niezamierzone zachowanie wynikające z faktu, że układ współrzędnych sceny rozciąg się od -1 do +1 w obu kierunkach. Skrajne wartości -1 i +1 układ współrzędnych osiąga na dolnym, górnym, lewym i prawym brzegu okna. Ale okno wcale kwadratowe nie jest. Należy ten fakt uwzględnić określając macierz rzutowania.
Domyślna macierz rzutowania (projekcji) jest **izometryczna**, co oznacza, że bez względu na odległość od kamery wszystkie przedmioty mają jednakową wielkość (translacja w kierunku *OZ* nie przynosi widocznego efektu). Jej postać jest wyjątkowo prosta – jest to macierz jednostkowa, w której składowa M_{ZZ} równa jest zero. Dzięki temu wszystkie współrzędne *z* werteksów, po przemnożeniu przez macierz stają się zerowe (zrzutowane na płaszczyznę ekranu). Możemy ograniczyć się do poprawienia stosunku wysokości do szerokości frustum tj. do ustalenia następującej macierzy projekcji:

```
efekt.Projection = Matrix.CreateOrthographic(
   2.0f * graphics.GraphicsDevice.Viewport.AspectRatio, //szerokość
   2.0f, //wysokość
   0.0f, //bliż
   100.0f); //dal
```

Możemy także pójść dalej i zmienić ją na macierz perspektywy, w której dalsze przedmioty stają się mniejsze:

```
efekt.Projection = Matrix.CreatePerspective(
   2.0f * graphics.GraphicsDevice.Viewport.AspectRatio, //szerokość
   2.0f, //wysokość
   1.0f, //bliż
   100.0f); //dal
```

Argumenty obu metod tworzących macierze, poza trzecim, są identyczne. Podajemy w nich szerokość i wysokość sceny w bliży oraz bliż i dal (objaśnienie terminów na rysunku poniżej). W przypadku perspektywy bliż musi mieć wartość nieujemną (większą od zera). Zauważmy jednak, że współrzędne *z* wszystkich trzech werteksów naszego trójkąta równe są zero. W konsekwencji trójkąt przestanie być widoczny (znajdzie się poza bryłą widzenia - frustum). Możemy zmienić położenie trójkąta, albo zmieniając współrzędne *z* werteksów, albo stosując translację, ale ja lubię mieć model w początku układu współrzędnych (ułatwia to np. obroty). Proponuję zatem odsunąć kamerę o jedną jednostkę w kierunku dodatnich wartości osi OZ (w kierunku widza).



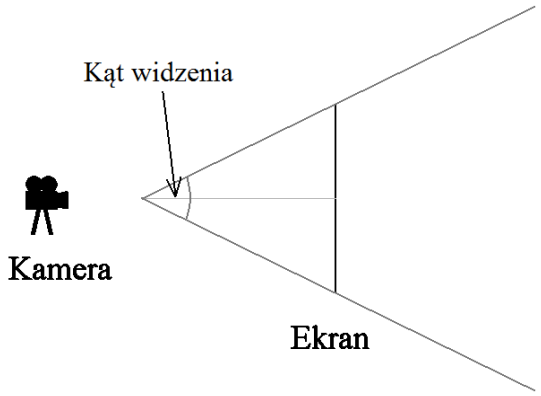
Rysunek 6. Frustum w rzutowaniu perspektywicznym i ortonormalnym

Do ustalenia bryły widzenia z perspektywą można również użyć alternatywnej metody

```
efekt.Projection = Matrix.CreatePerspectiveFieldOfView(
```

```
MathHelper.PiOver2, //kąt widzenia
graphics.GraphicsDevice.Viewport.AspectRatio, //proporcje obrazu
1.0f, //bliż
100.0f); //dal
```

Zgodnie z poniższym rysunkiem kąt widzenia zależy od odległości kamery od ekranu i szerokości ekranu (tangens połowy kąta widzenia równy jest stosunkowi połowy szerokości ekranu do odległość kamery od ekranu).



Rysunek 7. Kąt widzenia

Ustawianie kamery

Pamiętajmy, że w MonoGame mamy do czynienia z prawoskrętnym układem współrzędnych (oś Z skierowana jest do kamery). Należy zatem macierz widoku przemnożyć w metodzie `Initialize` przez macierz translacji o wektor (0,0,1). Oznacza to odsunięcie świata o wektor (0,0,-1):

```
efekt.View = Matrix.CreateTranslation(0, 0, -1);
```

Podejdźmy jednak do zagadnienia bardziej kompleksowo. W końcu kamera ma sześć stopni swobody, a nie tylko trzy. Poza trzema współrzędnymi wektora położenia możemy ustalić również kierunek kamery tj. punkt, na który jest skierowana. Możemy również zmienić tzw. polaryzację, a więc kierunek, w który skierowana jest góra kamery. To daje razem dziewięć liczb, ale nie wszystkie są niezależne. Dla przykładu górę kamery po ustaleniu położenia i kierunku można by ustawić za pomocą jednej wartości kąta (przechylenia). Podobnie kierunek można ustalić za pomocą dwóch kątów (odchylenia i nachylenia). Te trzy kąty tworzą kąty Eulera, które razem z trzema współrzędnymi położenia w jednoznaczny sposób opisują ustawienie kamery. Wrócimy do tego tematu, gdy będziemy zajmować się kontrolą kamery za pomocą myszy (rozdział 6). Tak, czy inaczej do ustawienia wszystkich parametrów ustawienia kamery najwygodniej użyć funkcji wzorowanej na funkcji z biblioteki GLU:

```
efekt.View = Matrix.CreateLookAt(
   new Vector3(0, 0, 1), //położenie kamery
   new Vector3(0, 0, 0), //punkt, na który kamera jest skierowana
   new Vector3(0, 1, 0)); //kierunek góry kamery (polaryzacja)
```

Animacja

W metodzie `Update` zastąpmy ustalanie wartości macierzy świata przez następującą instrukcję:

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

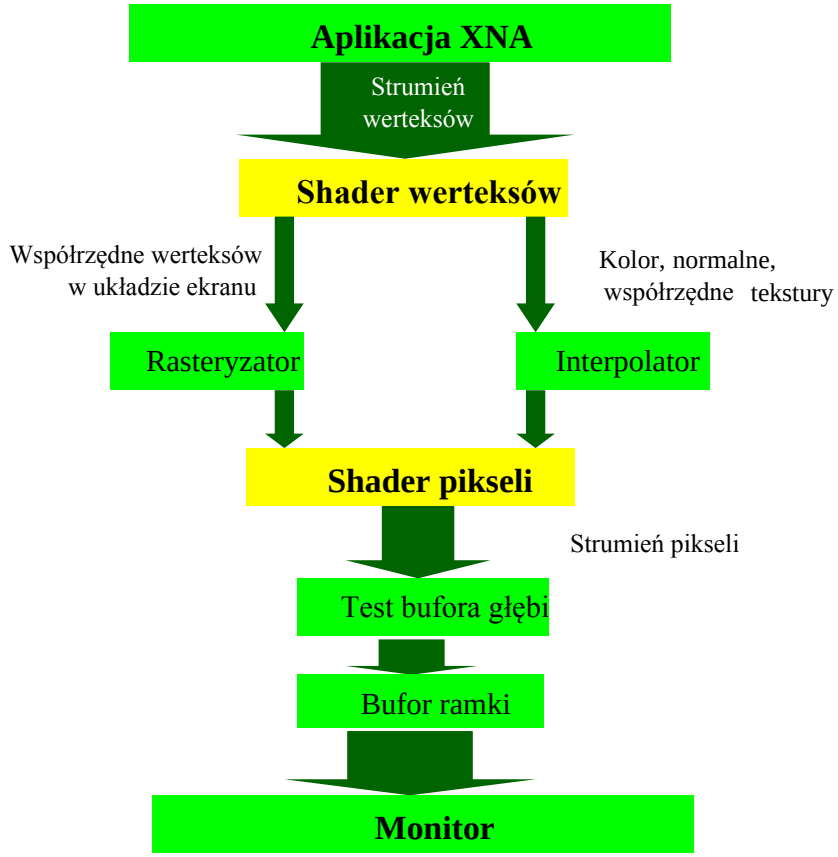
Zwróćmy uwagę, że tym razem do macierzy świata nie jest przypisywana gotowa wartość, a zamiast tego jej bieżąca wartość jest tylko mnożona przez macierz obrotu. W ten sposób sumaryczny kąt obrotu wokół osi Z będzie rósł z każdym kadrem i na ekranie zobaczymy animację obracającego się w płaszczyźnie XY trójkąta.

Zwróćmy także uwagę na fakt, że w argumencie metody `Matrix.CreateRotationZ` używamy argumentu metody `Update` – `gameTime`. Dzięki niemu można łatwo sprawdzić ile rzeczywiście upłynęło czasu między kolejnym wywołaniami metody `Update` i uniezależnić animację od częstości wywoływania tej metody lub chwilowych korków na drodze do procesorów CPU lub GPU.

Bufor werteksów

Nasz projekt ma jedną zasadniczą wadę - nie korzysta z bufora werteksów. Bufor werteksów umożliwia przesyłanie wierzchołków do pamięci karty graficznej. Przyspiesza to dostęp do nich i tym samym przyspiesza rendering. Może nie będzie to zysk widoczny przy rysowaniu jednego trójkąta, ale planując pisanie gry warto już teraz nabierać dobrych nawyków.

Bufor werteksów to już nie jest prosta tablica bajtów. Związana jest z nim teraz deklaracja werteksów. Jest to wobec tego pojemnik z kontrolą typów (*strongly typed container*).



Rysunek 8. Schematyczne i uproszczone przedstawienie potoku renderowania

1. Zacznijmy od utworzenia bufora. W tym celu zdefiniujmy następujące pole klasy:

```
VertexBuffer buforWerteksowTrojkata;
```

2. Bufor zainicjujmy w metodzie `Initialize`, a następnie wypełnijmy werteksami z tablicy `werteksyTrojkata`:

```
buforWerteksowTrojkata = new VertexBuffer(
   graphics.GraphicsDevice,
   VertexPositionColor.VertexDeclaration,
   werteksyTrojkata.Length,
   BufferUsage.WriteOnly);
buforWerteksowTrojkata.SetData<VertexPositionColor>(werteksyTrojkata);
```

3. Po tych poleceniach bufor zostaje zapełniony werteksami i w zasadzie sama tablica werteksów nie będzie już dłużej potrzebna (można ją zatem tworzyć lokalnie w metodzie `Initialize`). Aby wykorzystać werteksy z bufora werteksów do narysowania figury należy zmodyfikować metodę `Draw` zastępując wywołanie metody `gd.DrawUserPrimitives` następującymi dwiema instrukcjami:

```
gd.SetVertexBuffer(buforWerteksowTrojkata);

foreach (EffectPass pass in efekt.CurrentTechnique.Passes)
{
   pass.Apply();
   gd.DrawUserPrimitives<VertexPositionColor>(
      PrimitiveType.TriangleList,werteksyTrojkata, 0, 1);
   gd.DrawPrimitives(PrimitiveType.TriangleList, 0, 1);
}
```

PrimitiveType.TriangleList – gdy mamy więcej trójkątów, lepiej użyć ciągu. Tu jest jeden.

Pierwsza wskazuje na bufor werteksów, z którego mają być pobrane werteksy i w postaci strumienia werteksów przesłane do shaderów. **Ponieważ korzystamy tylko z jednego bufora, instrukcja ta może być umieszczona już w metodzie `Initialize`, po zainicjowaniu bufora.** Druga zajmuje się interpretacją i rysowaniem werteksów. Podobnie, jak metoda `gd.DrawUserPrimitives`, powyższe instrukcje najlepiej wywołać wewnątrz pętli po przebiegach efektu (po zawartości kolekcji `efekt.CurrentTechnique.Passes`).
Od teraz zawsze będziemy stosować **bufor werteksów**.

Dynamiczny bufor werteksów

WSKAZÓWKA: W tym miejscu warto zrobić kopię projektu. W dalszej części książki, poza zadaniem 3 w tym rozdziale, z dynamicznego bufora werteksów nie będziemy korzystać.

Bufor werteksów, jaki poznaliśmy przed chwilą jest z założenia statyczny. Po zapisaniu do niego werteksów i przesłaniu ich do karty graficznej mogą być one tylko odczytywane – to pozwala na optymalizację i wydajne renderowanie bryły. Nie ogranicza to możliwości obrotów i przesunięć opisywanej werteksami bryły – do tego korzystamy przecież z macierzy świata, ograniczona jest jednak możliwość jej deformowania. Jeżeli bardzo nam zależy na częstych zmianach buforowanych werteksów powinniśmy zamiast `VertexBuffer` użyć klasy `DynamicVertexBuffer`. Jest to klasa implementująca dynamiczny bufor werteksów. Używa się jej bardzo podobnie jak zwykłego bufora werteksów, ale trzeba zwrócić uwagę na kilka nowych szczegółów.
W ramach ćwiczenia proponuję narysować trójkąt, którego werteksy będziemy obracać bez korzystania z macierzy świata. Po zaktualizowaniu położenia i koloru będziemy werteksy wysyłać do bufora.

1. Deklarujemy referencję do bufora jako pole klasy `Game1`:

```
DynamicVertexBuffer dynamicznyBuforWerteksowTrojkata = null;
```

2. W metodzie `Game1.Initialize` inicjujemy bufor poleceniami:

```
dynamicznyBuforWerteksowTrojkata =
   new DynamicVertexBuffer(graphics.GraphicsDevice,
                           VertexPositionColor.VertexDeclaration,
                           werteksyTrojkata.Length,
                           BufferUsage.WriteOnly);
dynamicznyBuforWerteksowTrojkata.SetData(werteksyTrojkata, 0, werteksyTrojkata.Length,
                                         SetDataOptions.NoOverwrite);
```

Zwróćmy uwagę, że metoda `SetData` pozbawiona jest parametru. Obecny musi być również czwarty argument tej metody równy `SetDataOptions.NoOverwrite`. W MonoGame to nie jest konieczne! (zob.

http://blogs.msdn.com/b/shawnhar/archive/2010/07/07/setdataoptions-nooverwrite-versus-discard.aspx)

3. Instrukcja rysowania, jaką w metodzie `Game1.Draw` rysujemy bryłę korzystając z bufora, którego zawartość może być ustalana dynamicznie nie różni się niczym od instrukcji rysowania dla zwykłego bufora werteksów (w naszym przypadku zmienia się tylko referencja do bufora):

```
//wnętrze pętli po przebiegach w metodzie Draw (po instrukcji pass.Apply();)
gd.SetVertexBuffer(dynamicznyBuforWerteksowTrojkata);
gd.DrawPrimitives(PrimitiveType.TriangleList, 0, 1);
```

Podobnie, jak w przypadku zwykłego bufora werteksów, pierwsza instrukcja może być przeniesiona do metody `Initialize`.

4. W dynamicznym buforze werteksów ponowne wywołanie metody `SetData` modyfikuje zawartość bufora. Zmodyfikujmy zatem w metodzie `Game1.Update` położenie werteksów i ich kolory, a następnie podeślijmy zmienione werteksy do bufora.

```
for(int i=0;i<werteksyTrojkata.Length;i++)
{
   werteksyTrojkata[i].Position = Vector3.Transform(
      werteksyTrojkata[i].Position,
      Matrix.CreateRotationZ(gameTime.ElapsedGameTime.Milliseconds / 1000f));
   Color kolor = werteksyTrójkąta[i].Color;
   kolor.R += 1;
   werteksyTrójkąta[i].Color = kolor;
}
dynamicznyBuforWerteksowTrojkata.SetData(werteksyTrojkata,0,3,SetDataOptions.NoOverwrite);
```

W odróżnieniu od XNA, w MonoGame, w klasie `DynamicVertexBuffer` nie ma zdarzenia `ContentLost` informującego o „przypadkowym” opróżnienia bufora. Jest jedynie pole `IsContentLost`.

WSKAZÓWKA: Przywróć wersję projektu ze statycznym buforem werteksów.

Bardzo krótko o klawiaturze i gamepadzie

Odczytywaniem urządzeń wejścia zajmiemy się w rozdziałach 6 i 7, ale już teraz warto wskazać w jaki sposób sprawdzić czy naciśnięty został jakiś klawisz na klawiaturze lub przycisk na gamepadzie. Zresztą metoda `Update` utworzona w szablonie MonoGame zawiera już linię zamykającą aplikację przy naciśnięciu jednego z klawiszy na gamepadzie:

```
if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed) this.Exit();
```

Dodajmy do tego linię sprawdzającą, czy naciśnięty jest klawisz *Escape*. Jego przyciśnięcie spowoduje zamknięcie gry:

```
if (Keyboard.GetState().IsKeyDown(Keys.Escape)) this.Exit();
```

Możemy także spróbować związać ze spacją przełączanie między pełnym ekranem a oknem. W XNA wystarczało do tego wywołanie funkcji `graphics.ToggleFullScreen`. W MonoGame to nie działa zbyt dobrze. Po pierwsze wymaga wywołania funkcji `graphics.ApplyChanges`, a po drugie w przypadku powrotu z pełnego ekranu do okna, scena pozostaje przesunięta. Znalezione na forum MonoGame obejście tego problemu może wyglądać następująco:

```
if (Keyboard.GetState().IsKeyDown(Keys.Space))
{
    if (!graphics.IsFullScreen)
    {
        graphics.PreferredBackBufferWidth =
```

```
            graphics.GraphicsDevice.PresentationParameters.BackBufferWidth;
        graphics.PreferredBackBufferHeight =
            graphics.GraphicsDevice.PresentationParameters.BackBufferHeight;
    }
    graphics.GraphicsDevice.Viewport = new Viewport(
        0, graphics.GraphicsDevice.DisplayMode.Height - graphics.PreferredBackBufferHeight,
        graphics.PreferredBackBufferWidth, graphics.PreferredBackBufferHeight);
    graphics.ToggleFullScreen();
    graphics.ApplyChanges();
}
```

Zadania

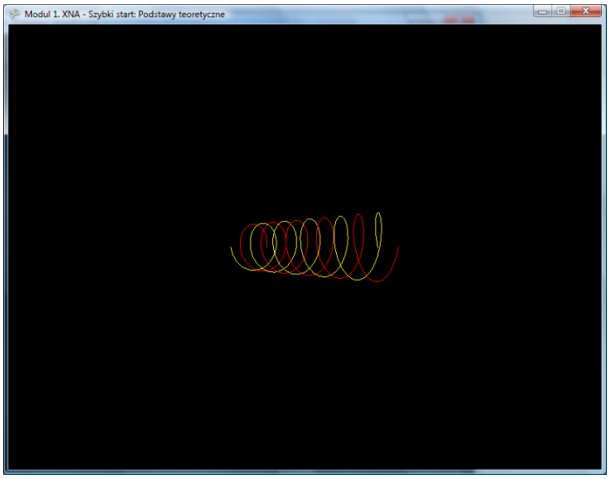
Zadanie 1

Zmodyfikować i rozszerzyć werteksy przechowywane w buforze tak, aby na scenie rysować kwadrat.

Zadanie 2

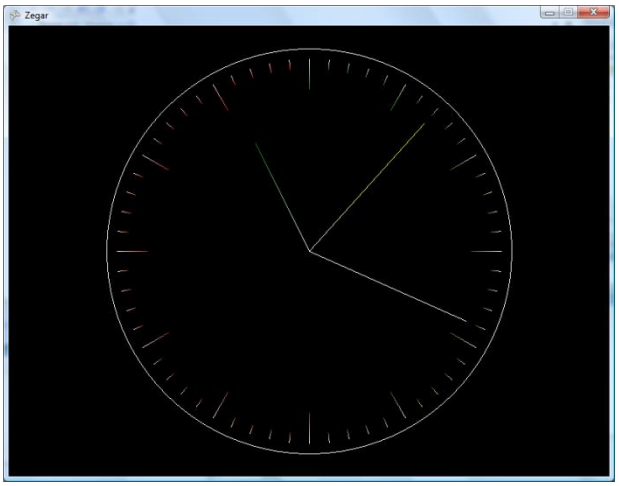
Zdefiniuj metody odpowiedzialne za przygotowanie tablicy werteksów opisujących elipsę, helisę oraz „płaską” spiralę i spiralę wyciągniętą w kształt stożka. Proponuję umieścić je wszystkie w jednej klasie statycznej. Narysować podwójną helisę.

Podpowiedź: zacznij od projektu z poprzedniego zadania i zmodyfikuj tablicę i bufor werteksów.



Zadanie 3

Zaprojektuj zegar analogowy z możliwością ustalenia wielkości i położenia. Do animacja wskazówek, zamiast zmiany macierzy świata, użyć dynamicznego bufora werteksów.


Zegar