Rozdział 4
Komponenty gry na przykładzie prostopadłościanu

Wersja z 2014-03-21

Budowanie gry, nawet stosunkowo prostej, to spore wyzwanie dla programisty. Bardzo łatwo zgubić się w dużej ilości linii kodu z dużą ilością instrukcji warunkowych. Do tego dochodzi obsługa wielu ekranów, trybów gry, wielu graczy itd. Każdy programista wie, że aby uniknąć utraty orientacji we własnym projekcie powinien podzielić kod na klasy, które będą realizować autonomiczne zadania i które łatwiej jest testować. Z klas można budować większe całości bez konieczności kontrolowania niezliczonej ilości zmiennych. W MonoGame klasom owym wygodnie jest nadać status komponentów – wówczas ich aktualizacja i ewentualne rysowanie kontrolowane będzie automatycznie.
Wiele elementów gry można zamknąć w komponentach. W zasadzie każdy większy fragment kodu, który może się powtarzać zasługuje na utworzenie dla niego komponentu. Komponenty mogą być „niewizualne"; dziedziczą wówczas z klasy `GameComponent`. Przykładem takiego komponentu może być obiekt kontrolujący zmieniany dynamicznie podkład muzyczny lub komponent sterujący dialogiem postaci napotykanych w grze RPG. Komponenty mogą także być wyposażone w metodę `Draw`. Powinny wówczas dziedziczyć z `DrawableGameComponent`. W tym drugim przypadku mogą to być postaci kontrolowane przez gracza lub komputer, poruszane wiatrem drzewo lub cały ekran menu.
Najwygodniej projektować komponenty autonomiczne tj. takie, których komunikacja z pozostałymi komponentami jest ograniczona do minimum. W praktyce to założenie jest jednak niemożliwe do zrealizowania. Dlatego opiszę również możliwość implementacji usługi, która będzie odpowiedzialna za uzgadnianie stanów różnych komponentów. Usługa ta będzie czymś w rodzaju centrum komunikacji komponentów, pośrednikiem między nimi.

Celem tego rozdziału jest wprowadzenie idei modularnego programowania gier, w którym niemal wszystkie elementy gry są samodzielnymi komponentami komunikującymi się za pomocą usług. Jako przykładu użyję komponentu kontrolującego pozycję postaci w dynamicznie wyznaczanej formacji ścigającej wyznaczony cel.

USUNIĘTA CZĘŚĆ O PUNKTORACH
(włącznie z komunikacją między komponentami – w MonoGame chyba i tak jej nie ma)

Komponent prostopadłościanu

Niezaprzeczalną wadą trójkąta, który dotychczas służył nam do prezentacji możliwości MonoGame, jest to że jest… płaski. Potrzebujemy jakiejś figury przestrzennej. Proponuję, żeby to był prostopadłościan i to z kilku powodów. Pierwszym jest fakt, że jego ściany są czworokątami, których nie ma wśród prymitywów dostępnych w MonoGame i trzeba je budować samodzielnie. Drugim - że prostopadłościan szczególnie dobrze nadaje się do teksturowania. A trzecim, najważniejszym, jest to że prostopadłościany będą wygodnymi klockami, z których będziemy budowali inne obiekty. Wartością dodaną będzie użycie komponentu gry, który będzie zapewniał automatyczną aktualizację i rysowanie.

1. Stwórzmy nowy projekt gry:
    a. Uruchom Visual Studio z MonoGame
    b. Przyciśnij klawisze *Ctrl+Shift+N*.
    c. W oknie *New Project* przejdź do kategorii *MonoGame*, zaznacz projekt *MonoGame Windows OpenGL Project*, wpisz nazwę *MojaDrugaGraMonoGame* i kliknij *OK*.
    d. Jeżeli chcesz przełączyć grę do trybu `Reach`, możesz w konstruktorze `Game1` umieścić polecenie

```
GraphicsDevice.GraphicsProfile = GraphicsProfile.Reach;
```

2. Przygotuj efekt. W tym celu
    a. W klasie `Game1` zdefiniuj pole o nazwie `efekt` typu `BasicEffect`.
    b. W metodzie `Game1.Initialize` zainicjuj je:

```
efekt = new BasicEffect(graphics.GraphicsDevice);
efekt.VertexColorEnabled = true;
efekt.Projection = Matrix.CreatePerspective(
   2.0f*graphics.GraphicsDevice.Viewport.AspectRatio,
   2.0f,
   1.0f,
   10.0f);
efekt.View = Matrix.CreateLookAt(
   new Vector3(0, 0, 2.5f),
   new Vector3(0, 0, 0),
   new Vector3(0, 1, 0));
efekt.World = Matrix.Identity;
```

3. Dodaj do projektu komponent gry o nazwie `Prostopadloscian`
    a. Rozwijamy menu *Project* i wybieramy z niego polecenie *Add Class…*.
    b. W oknie *Add New Item* zaznaczamy pozycję *Game Component*.
    c. W polu *Name* wpisujemy *Prostopadloscian.cs*.
    d. Klikamy przycisk *Add*.
    e. Po utworzeniu nowego komponentu w edytorze kodu wskazujemy klasę bazową nowej klasy tj. `DrawableGameComponent`. Jednocześnie na początku pliku podajemy przestrzenie nazw, których będziemy używać, a w klasie `Prostopadloscian` definiujemy trzy pola. Jedno z nich to efekt typu `BasicEffect`. Nie używamy ogólniejszej klasy `Effect`, bo w dalszej części używać będziemy macierzy świata do określenia pozycji prostopadłościanu na scenie.

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

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

namespace MojaDrugaGraMonoGame
{
    class Prostopadloscian : DrawableGameComponent
    {
        GraphicsDevice gd;
        BasicEffect efekt;
        VertexBuffer buforWerteksow;
```

4. Definiujemy konstruktor przyjmujący 6 argumentów:

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

Typ Color? jest równoznaczny z Nullable<Color> i oznacza, że do zmiennej kolor może być przypisany nie tylko obiekt typu Color (typ wartościowy), ale również null.

a. Argumenty dx, dy i dz określają szerokość, wysokość i głębokość prostopadłościanu. Aby wygodniej określać współrzędne wierzchołków dzielimy te argumenty w konstruktorze przez dwa:

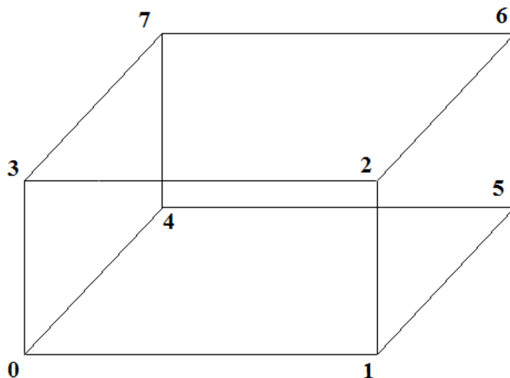
```
public Prostopadloscian(Game game, BasicEffect efekt
   float dx, float dy, float dz, Color? kolor)
   : base(game)
{
   dx /= 2;
   dy /= 2;
   dz /= 2;
```

b. Inicjujemy pole gd i klonujemy efekt. Dzięki sklonowaniu komponent będzie dysponował własnym, niezależnym od gry efektem:

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

c. Tworzymy lokalną tablicę ośmiu punktów, które wykorzystywać będziemy do budowania werteksów (zob. rysunek poniżej):

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



d. Zdefiniujmy lokalne zmienne typu Color określające trzy kolory dla trzech par powierzchni:

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

e. Teraz najbardziej żmudna część kodu konstruktora – definiowanie tablicy werteksów. Musimy je zdefiniować w taki sposób, aby każda ściana była wyświetlana jako ciąg złożony z dwóch trójkątów. To nieco ograniczy liczbę werteksów.

Dalsze zmniejszenie liczby nadmiarowych werteksów moglibyśmy uzyskać korzystając z bufora indeksów. Jednak w odróżnieniu od gładkiej sfery, bufor ten nie nadaje się do prostopadłościanu. Wprawdzie w każdym jego wierzchołki powtarzają się trzy razy, ale każdy może mieć inny kolor, a w przyszłości będzie miał także inną normalną.

```
VertexPositionColor[] werteksy = new VertexPositionColor[24]{
    //przednia sciana
    new VertexPositionColor(punkty[3], kolor1),
    new VertexPositionColor(punkty[2], kolor1),
    new VertexPositionColor(punkty[0], kolor1),
    new VertexPositionColor(punkty[1], kolor1),

    //tylnia sciana
    new VertexPositionColor(punkty[7], kolor1),
    new VertexPositionColor(punkty[4], kolor1),
    new VertexPositionColor(punkty[6], kolor1),
    new VertexPositionColor(punkty[5], kolor1),

    //gorna sciana
    new VertexPositionColor(punkty[3], kolor2),
    new VertexPositionColor(punkty[7], kolor2),
    new VertexPositionColor(punkty[2], kolor2),
    new VertexPositionColor(punkty[6], kolor2),

    //dolna sciana
    new VertexPositionColor(punkty[0], kolor2),
    new VertexPositionColor(punkty[1], kolor2),
    new VertexPositionColor(punkty[4], kolor2),
    new VertexPositionColor(punkty[5], kolor2),

    //lewa sciana
    new VertexPositionColor(punkty[3], kolor3),
    new VertexPositionColor(punkty[0], kolor3),
    new VertexPositionColor(punkty[7], kolor3),
    new VertexPositionColor(punkty[4], kolor3),

    //prawa sciana
    new VertexPositionColor(punkty[1], kolor3),
    new VertexPositionColor(punkty[2], kolor3),
    new VertexPositionColor(punkty[5], kolor3),
    new VertexPositionColor(punkty[6], kolor3)
};
```

f. Całą tę tablicę kopiujemy do karty graficznej korzystając z bufora werteksów:

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

3. Kolejnym krokiem jest nadpisanie metody `Draw` komponentu

Klasa `DrawableGameComponent`, której użyliśmy jako klasy bazowej komponentu `Prostopadłościan` dziedziczy po klasie `GameComponent`. Ta klasa implementuje interfejs `IUpdateable`, który wymusza obecność metody `Update`. A ponieważ jest ona zdefiniowana już w klasie `GameComponent` nie ma w zasadzie konieczności nadpisywania jej w klasie potomnej tj. w klasie naszego komponentu. Podobnie jest z klasą `DrawableGameComponent`, która rozszerza klasę `GameComponent` implementując jednocześnie interfejs `IDrawable`, co zmuszą ją do posiadania metody `Draw`.

a. Przygotowujemy publiczną metodę `Draw` nadpisującą metodę z klasy bazowej (modyfikator `override`) i wyświetlającą listę trójkątów na podstawie werteksów z bufora:

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

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

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

    base.Draw(gameTime);
}
```

4. Prostopadłościan na razie będzie sam odpowiedzialny za swój ruch. To da nam pretekst, żeby nadpisać jego metodę `Update`, w której umieścimy polecenia modyfikujące macierz świata. Do klasy dodamy jednak włącznik, który będzie włączał i wyłączał animację – wystarczy nacisnąć *F5*.

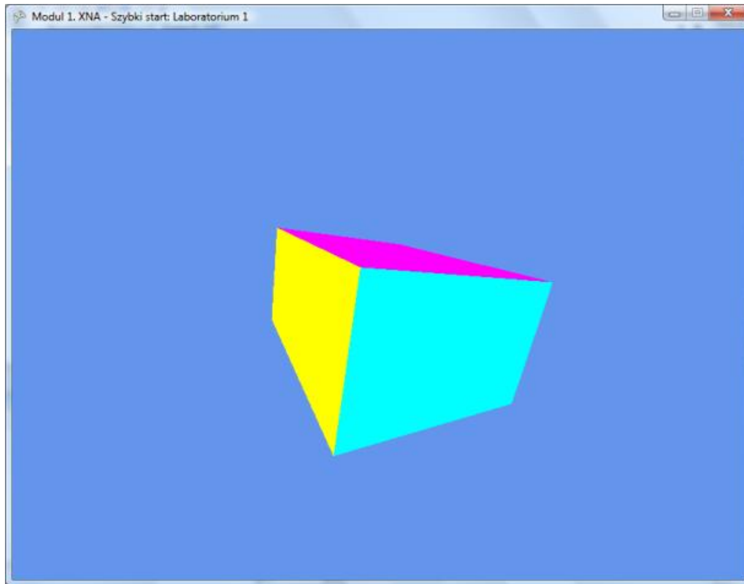
```
public bool AnimacjaAktywna = false;
private KeyboardState poprzedniStanKlawiatury = Keyboard.GetState();

public override void Update(GameTime gameTime)
{
    KeyboardState stanKlawiatury = Keyboard.GetState();
    if (stanKlawiatury.IsKeyDown(Keys.F5) &&
        !poprzedniStanKlawiatury.IsKeyDown(Keys.F5))
        AnimacjaAktywna = !AnimacjaAktywna;
    poprzedniStanKlawiatury = stanKlawiatury;

    if (AnimacjaAktywna)
    {
        efekt.World *=
            Matrix.CreateRotationX(gameTime.ElapsedGameTime.Milliseconds / 1000.0f) *
            Matrix.CreateRotationY(gameTime.ElapsedGameTime.Milliseconds / 1000.0f);
    }

    base.Update(gameTime);
}
```

5. Dodaj komponent do listy komponentów gry
   a. Wracamy do edycji klasy `Game1` (plik *Game1.cs*).
   b. W jej metodzie `Initialize` tworzymy obiekt reprezentujący prostopadłościan i dodajemy go do listy komponentów gry:

```
this.Components.Add(
      new Prostopadloscian(this, 1.5f, 1.0f, 2.0f, Color.White));
```

   c. Jeżeli chcemy użyć predefiniowanych kolorów ostatni argument zastąpmy przez `null`:

```
this.Components.Add(
      new Prostopadloscian(this, 1.5f, 1.0f, 2.0f, null));
```



Udostępnienie macierz świata komponentu

Jeżeli chcemy mieć możliwość kontrolowania pozycji i orientacji prostopadłościanu spoza komponentu, należy macierz świata z efektu komponentu udostępnić za pomocą własności:

1. W klasie Prostopadloscian zdefiniuj własność:

```
public Matrix World
{
   get
   {
      return efekt.World;
   }
   set
   {
      efekt.World = value;
   }
}
```

2. Stwórz dwa odsunięte od siebie prostopadłościany
   a. Wracamy do klasy `Game1` (plik *Game1.cs*).
   b. W metodzie `Initialize` zastąpmy poprzednie polecenie tworzenia komponentu przez nowe, wykorzystując nową własność komponentu do ustalenia jego pozycji i skalowania.:

```
Prostopadloscian p1 = new Prostopadloscian(this, efekt, 1.5f, 1.0f, 2.0f, Color.White);
p1.World=Matrix.CreateScale(0.75f) * Matrix.CreateTranslation(-1, 0, 0);
this.Components.Add(p1);

Prostopadloscian p2 = new Prostopadloscian(this, efekt, 1.5f, 1.0f, 2.0f, Color.Crimson);
p2.World=Matrix.CreateScale(0.75f) * Matrix.CreateTranslation(1, 0, 0);
this.Components.Add(p2);
```

   c. Teraz w metodzie `Game1.Update` możemy kontrolować położenie i orientację prostopadłościanów tak, jak wcześniej robiliśmy to z metody `Prostopadloscian.Update`:

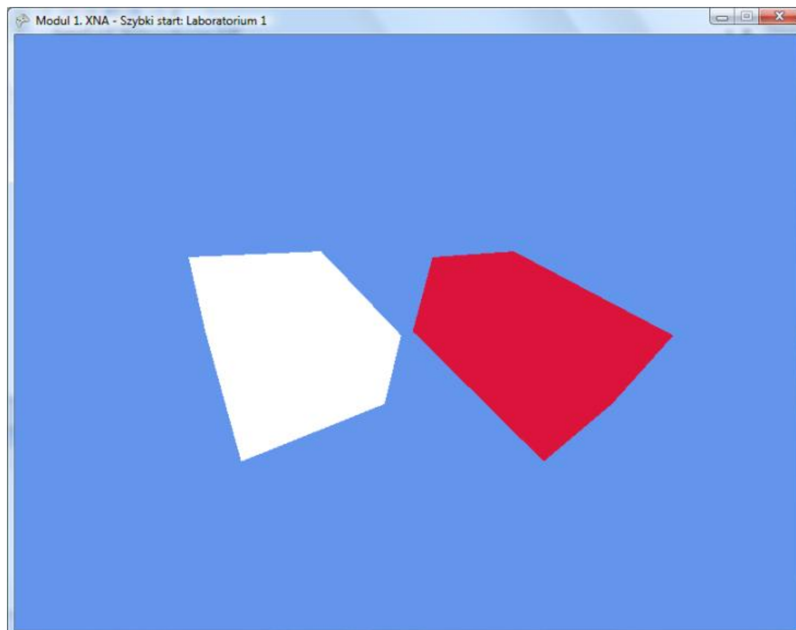
```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed ||
        Keyboard.GetState().IsKeyDown(Keys.Escape))
        Exit();
```

```
    foreach (GameComponent komponent in this.Components)
    {
        if (komponent is Prostopadloscian)
            (komponent as Prostopadloscian).World *=
                Matrix.CreateRotationX(gameTime.ElapsedGameTime.Milliseconds / 1000.0f) *
                Matrix.CreateRotationY(gameTime.ElapsedGameTime.Milliseconds / 1000.0f);
    }

    base.Update(gameTime);
}
```

Po uruchomieniu zobaczymy dwa prostopadłościany obracające się wokół wspólnego środka. Sprawdź co się stanie, gdy prawostronne mnożenie macierzy zmienimy na lewostronne:

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

    foreach (GameComponent komponent in this.Components)
    {
        if (komponent is Prostopadloscian)
            (komponent as Prostopadloscian).World =
               Matrix.CreateRotationX(gameTime.ElapsedGameTime.Milliseconds / 1000.0f) *
               Matrix.CreateRotationY(gameTime.ElapsedGameTime.Milliseconds / 1000.0f) *
               (komponent as Prostopadloscian).World;
    }

    base.Update(gameTime);
}
```

Efekt widoczny jest na rysunku poniżej:



[KOMENTARZ/INFORMACJA]
Dzięki interfejsowi IVertexType możnaby napisać sparametryzowaną wersję klasy Prostopadłościan<VertexType> : GameComponent where VertexType : IVertexType ale ja inicjuję werteksy wewnątrz komponentu, żeby nadać im położenie i normalne. Zmiennych określonych jedynie typem interfejsu nie mógłbym konfigurować.

Zadania

Zadanie 1

Przygotuj komponent o nazwie `GameScene` dziedziczący z `DrawableGameComponent`, który na wzór klasy gry `Game` wyposażony będzie w listę komponentów `Components` i który ułatwi zarządzanie „ekranami" w grze. Powinien wywoływać metodę `Update` każdego z komponentów w swojej metodzie `Update`, a metodę `Draw` – w swojej metodzie `Draw`. W efekcie komponenty zarejestrowane w instancji `GameScene` będą odświeżane jeżeli własność `Enabled` instancji `GameScene` będzie ustawiona na `true`, a rysowane jeżeli równa prawdziwości będzie własność `Visible`. Ponadto zdefiniuj menedżera ekranów, który nie będzie komponentem i odpowiadać będzie tylko za zmianę aktywnego ekranu tzn. będzie rejestrował lub usuwał instancje klasy `GameScene` w liście komponentów gry.

Zadanie 2

Do komponentu `GameScene` dodaj zdarzenie, które pozwoli wykonać kod zdefiniowany w metodzie zdarzeniowej w pętli efektu. Do metody zdarzeniowej prześlij efekt, aktualną instancję `GraphicDevice` i instancję `EffectPass` z bieżącej iteracji pętli.
Dodaj również zdarzenia `Updated` i `Drawed` pozwalające na wykonanie dowolnego kodu po odświeżeniu i narysowaniu komponentów sceny.

Zadanie 3

Skompilować prostopadłościan do biblioteki DLL lub PCL.