Damian Kaniewski, Tomasz Dziubak, Jacek Matulewski

MonoGame
Podstawowe koncepcje grafiki 3D

Helion

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki: Studio Gravite/Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/twoapl
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-9350-9

Printed in Poland.

Spis treści

Rozdział 4.

Komponenty gry

Budowanie gry, nawet stosunkowo prostej, to spore wyzwanie dla programisty, szczególnie jeżeli działa w pojedynkę, a liczba linii kodu stale rośnie. Bardzo łatwo zgubić się w zawiłościach skomplikowanej logiki gry, obsługi poszczególnych jej trybów czy interakcji graczy w grze wieloosobowej. Aby uniknąć utraty orientacji we własnym projekcie, należy go podzielić na moduły, a moduły na klasy, które będą realizować w miarę autonomiczne zadania i które łatwiej jest testować. Z klas można budować większe całości bez konieczności kontrolowania niezliczonej liczby zmiennych. W MonoGame poza klasami mamy także do dyspozycji tzw. komponenty gry. Są to specjalne klasy, które są odświeżane i rysowane w podobnym schemacie, jak odświeżana jest klasa gry (dziedzicząca po klasie `Game`).

Komponenty — wprowadzenie

Wiele elementów gry można zamknąć w komponentach. Mówiąc precyzyjniej, każdy większy fragment kodu, który jest wielokrotnie wykonywany, może być umieszczony w komponencie. Komponenty nie muszą być widoczne na ekranie, mogą być „niewizualne"; dziedziczą wówczas z klasy `GameComponent`. Przykładem takiego komponentu może być obiekt kontrolujący zmieniany dynamicznie podkład muzyczny lub komponent sterujący dialogiem postaci w grze RPG. Takie komponenty posiadają metodę `Update`, która po zarejestrowaniu komponentu jest automatycznie wywoływana z taką samą częstością, jak metoda `Update` klasy gry. Komponenty mogą także być wyposażone w metodę `Draw`. Powinny wówczas dziedziczyć po `DrawableGame` ↪`Component`. W tej części samouczka przedstawię przykład komponentu „wizualnego", którym będzie zwykły prostopadłościan. Wybór tej bryły nie jest przypadkowy. Na prostopadłościanie wygodnie będzie w następnych rozdziałach omawiać i testować oświetlenie oraz teksturowanie.

Pierwszy komponent

Rozpocznijmy od stworzenia nowego projektu gry.

1. Uruchamiamy Visual Studio Code z MonoGame.
2. W oknie terminala przejdźmy do folderu, w którym chcemy umieścić nowy projekt. Stwórzmy dla niego folder o nazwie *MojaDrugaGra*.
3. Następnie przejdźmy do tego folderu poleceniem `cd MojaDrugaGra`.

4. Wykorzystajmy komendę `dotnet new mgdesktopgl` do stworzenia projektu.
5. Otwórzmy folder w Visual Studio Code, tak jak opisano w poprzednim rozdziale.
6. Do edytora wczytajmy plik *Game1.cs*.

W MonoGame możemy wyróżnić dwa profile graficzne: *HiDef* oraz *Reach*. Profil *HiDef* został zaprojektowany dla systemów, które zapewniają bardziej zaawansowane możliwości graficzne. Ogranicza to liczbę platform i systemów obsługujących profil, ale pozwala na bogatszy zestaw możliwości. Natomiast profil *Reach* zapewnia, że kod i zawartość gry będą działać na wielu platformach i w wielu systemach, ale możliwości graficzne są mniejsze. Jeżeli chcemy przełączyć grę do trybu *Reach*, możemy w konstruktorze klasy `Game1` umieścić polecenie:

```
graphics.GraphicsProfile = GraphicsProfile.Reach;
```

Następnie zajmijmy się przygotowaniem efektu. W tym celu w klasie `Game1` zdefiniujmy pole o nazwie `efekt` typu `BasicEffect` i w metodzie `Game1.Initialize` zainicjujmy je, wpisując kod z listingu 4.1.

LISTING 4.1. Inicjacja efektu

```
private BasicEffect efekt;

...

protected override void Initialize()
{
    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;

    base.Initialize();
}
```

Następnie dodajmy do projektu komponent gry o nazwie `Prostopadloscian`[1]. Dodajemy do projektu zwykłą klasę. Następnie przekształcimy ją w komponent. W tym celu w rozwijanym drzewku z lewej strony naszego edytora klikamy prawym przyciskiem myszy nasz folder główny (wolne pole pod listą plików) i wybieramy polecenie *Nowy plik/Add File*, w polu nazwy wpisujemy *Prostopadloscian.cs*. Możemy również użyć menu *Plik* lub ikony widocznej w górnej części okna *Eksploratora*. Po utworzeniu nowego pliku kod klasy tworzymy od zera (nie ma szablonu klasy jak w „pełnym" Visual Studio). Na początku dodajemy przestrzenie nazw związane

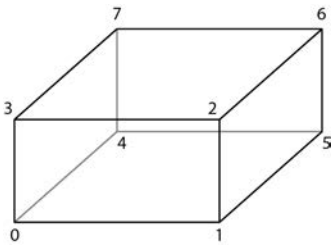
[1] Istnieje również możliwość tworzenia nazw z polskimi znakami, ale aby uniknąć kłopotów z kompilacją na inne platformy, staramy się tego unikać.

z MonoGame (wyróżnione na listing 4.2). Wskazujemy też jej klasę bazową, tj. `Drawable ↪GameComponent`. W klasie `Prostopadloscian` definiujemy trzy pola. Jedno z nich to efekt typu `BasicEffect`. Nie używamy ogólniejszej klasy `Effect`, bo w dalszej części używać będziemy zdefiniowanej w `BasicEffect` macierzy świata do określenia pozycji prostopadłościanu na scenie. Pozostałe pola to referencja typu `GraphicsDevice`, która jest argumentem niemal każdej ważnej metody MonoGame, oraz bufor werteksów.

LISTING 4.2. Klasa nowego komponentu

```
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 MojaDrugaGra
{
    class Prostopadloscian : DrawableGameComponent
    {
        private GraphicsDevice gd;
        private BasicEffect efekt;
        private VertexBuffer buforWerteksow;
    }
}
```

W nowej klasie definiujemy także konstruktor przyjmujący sześć argumentów (listing 4.2). Jego argumenty `dx`, `dy` i `dz` określają szerokość, wysokość i głębokość prostopadłościanu. Aby wygodniej określać współrzędne wierzchołków, dzielimy w konstruktorze trzy liczby przez dwa. W konstruktorze inicjujemy także pole `gd` i klonujemy efekt. Dzięki sklonowaniu komponent będzie dysponował własnym, niezależnym od gry efektem. Następnie tworzymy lokalną tablicę ośmiu punktów, które wykorzystywać będziemy do budowania werteksów (rysunek 4.1). W konstruktorze zdefiniujmy także zmienne typu `Color` określające trzy kolory dla trzech par powierzchni prostopadłościanu.



RYSUNEK 4.1. Punkty, z których zbudowany będzie prostopadłościan

LISTING 4.3. Konstruktor komponentu

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

```
        gd = game.GraphicsDevice;
        this.efekt = (BasicEffect)efekt.Clone();
        Vector3[] punkty = new Vector3[8]
        {
            new Vector3(-dx, -dy, dz),
            new Vector3(dx, -dy, dz),
            new Vector3(dx, dy, dz),
            new Vector3(-dx, dy, dz),
            new Vector3(-dx, -dy, -dz),
            new Vector3(dx, -dy, -dz),
            new Vector3(dx, dy, -dz),
            new Vector3(-dx, dy, -dz)
        };
        Color kolor1 = kolor ?? Color.Cyan;
        Color kolor2 = kolor ?? Color.Magenta;
        Color kolor3 = kolor ?? Color.Yellow;
    }
```

Teraz czeka nas przygotowanie najbardziej żmudnej części kodu konstruktora — definiowanie tablicy werteksów (listing 4.4). Musimy je zdefiniować w taki sposób, aby każda ściana była wyświetlana jako ciąg złożony z dwóch trójkątów, co nieco ograniczy liczbę werteksów Następnie całą tę tablicę kopiujemy do karty graficznej, korzystając z bufora werteksów. Kolejność werteksów jest ważna — powinny być tak ułożone, żeby wszystkie trójkąty ustawione były przodem na zewnątrz bryły.

LISTING 4.4. Fragment konstruktora odpowiedzialny za definiowanie werteksów

```
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),
    //tylna 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)
};
buforWerteksow = new VertexBuffer(
    gd,
    VertexPositionColor.VertexDeclaration,
    werteksy.Length,
    BufferUsage.WriteOnly);

buforWerteksow.SetData<VertexPositionColor>(werteksy);
```

Klasa `DrawableGameComponent`, której użyliśmy jako klasy bazowej komponentu `Prostopadloscian`, 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 `Drawa` ↪`bleGameComponent`, która rozszerza klasę `GameComponent`, implementując jednocześnie interfejs `IDrawable`, co zmusza ją do posiadania metody `Draw`. Metoda `Draw` będzie jednak potrzebna do wyświetlenia zdefiniowanych przed chwilą werteksów. Dlatego nadpisujemy ją w klasie `Prostopadloscian` zgodnie ze wzorem widocznym na listingu 4.5.

LISTING 4.5. Nadpisana metoda Draw komponentu

```
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);
}
```

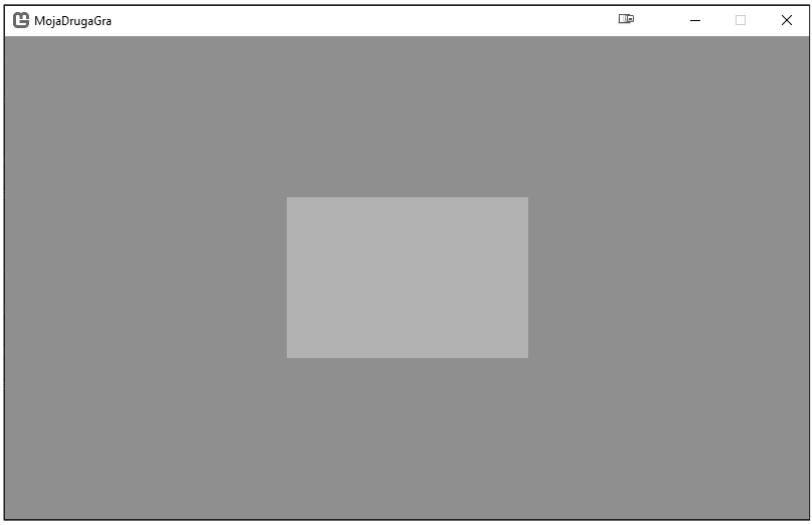
To dobry moment, aby sprawdzić, jak działa nasz komponent. W tym celu musimy utworzyć instancję klasy `Prostopadloscian` i zarejestrować ją w klasie gry, czyli dodać ją do listy komponentów w kolekcji `Game1.Components`. Wracamy zatem do edycji klasy `Game1` (plik *Game1.cs*), definiujemy w niej pole `prostopadloscian` typu `Prostopadlo` ↪`scian`, inicjujemy je w metodzie `Initialize`, tworząc obiekt tej klasy, i dodajemy go do listy komponentów gry (listing 4.6). Ostatni argument w konstruktorze komponentu ustalamy jako równy `null`. To oznacza, że użyjemy predefiniowanych kolorów, które ułatwią nam dostrzeżenie głębi pomimo braku oświetlenia. Oczywiście pustą wartość `null` możemy zastąpić np. przez `Color.White`. Wówczas utworzymy biały prostopadłościan.

LISTING 4.6. Tworzenie i rejestrowanie komponentu w klasie gry

```
private Prostopadloscian prostopadloscian;

...

protected override void Initialize()
{
```

```
    efekt = new BasicEffect(graphics.GraphicsDevice);
    ...
    efekt.World = Matrix.Identity;

    prostopadloscian = new Prostopadloscian(
        this, efekt, 1.5f, 1.0f, 2.0f, null);
    this.Components.Add(prostopadloscian);

    base.Initialize();
}
```

Po uruchomieniu gry efekt, jaki zobaczymy na ekranie, nie będzie jeszcze zbyt widowiskowy. Macierz świata jest jednostkowa, więc prostopadłościan jest ustawiony do nas jedną ze ścian w taki sposób, że nie widać pozostałych (rysunek 4.2). Warto byłoby zatem umożliwić dowolne ustawienie prostopadłościanu na scenie. To wymaga dostępu do macierzy świata efektu sklonowanego w komponencie. W tym celu najprostsze będzie zdefiniowanie w klasie `Prostopadloscian` własności udostępniającej tę macierz (listing 4.7).



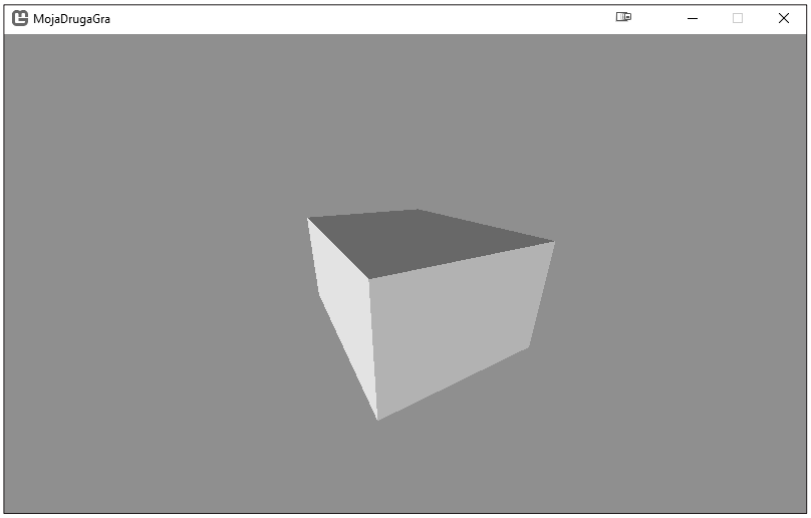
RYSUNEK 4.2. Prostopadłościan w domyślnym ustawieniu na scenie

LISTING 4.7. Zdefiniowana w komponencie własność udostępniająca macierz świata

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

Kontrola ułożenia prostopadłościanu

Wykorzystując własność komponentu udostępniającą macierz świata, możemy sterować orientacją prostopadłościanu np. za pomocą klawiszy lub gamepada (por. rysunek 4.3). (Więcej na temat kontrolerów gier w rozdziale 9.). Wystarczy do metody `Game1.Update` dodać polecenia widoczne na listingu 4.8. Oczywiście, jeżeli z jakiegoś powodu udostępnianie macierzy świata nam nie odpowiada, możemy sprawdzić stan klawiatury w klasie komponentu, w jego metodzie `Prostopadloscian.Update`, i z jej poziomu modyfikować obiekt macierzy `efekt.World`.



RYSUNEK 4.3. Kolorowanie ścian kompensuje brak oświetlenia

LISTING 4.8. Sterowanie orientacją komponentu. Metoda Update klasy Game1

```
protected override void Update(GameTime gameTime)
{
    KeyboardState stanKlawiatury = Keyboard.GetState();
    GamePadState stanGamepada = GamePad.GetState(PlayerIndex.One);

    if (stanGamepada.Buttons.Back == ButtonState.Pressed ||
        stanKlawiatury.IsKeyDown(Keys.Escape))
        Exit();

    float katObrotu = 0.01f;
    if (stanKlawiatury.IsKeyDown(Keys.LeftShift) ||
        stanKlawiatury.IsKeyDown(Keys.RightShift))
        katObrotu *= 10;

    if (stanKlawiatury.IsKeyDown(Keys.Left) ||
        stanGamepada.IsButtonDown(Buttons.DPadLeft))
        prostopadloscian.MacierzSwiata *= Matrix.CreateRotationY(katObrotu);
    if (stanKlawiatury.IsKeyDown(Keys.Right) ||
        stanGamepada.IsButtonDown(Buttons.DPadRight))
        prostopadloscian.MacierzSwiata *= Matrix.CreateRotationY(-katObrotu);
```

```
        if (stanKlawiatury.IsKeyDown(Keys.Up) ||
            stanGamepada.IsButtonDown(Buttons.DPadUp))
            prostopadloscian.MacierzSwiata *= Matrix.CreateRotationX(katObrotu);
        if (stanKlawiatury.IsKeyDown(Keys.Down) ||
            stanGamepada.IsButtonDown(Buttons.DPadDown))
            prostopadloscian.MacierzSwiata *= Matrix.CreateRotationX(-katObrotu);
        if (stanKlawiatury.IsKeyDown(Keys.OemPeriod))
            prostopadloscian.MacierzSwiata *= Matrix.CreateRotationZ(katObrotu);
        if (stanKlawiatury.IsKeyDown(Keys.OemComma))
            prostopadloscian.MacierzSwiata *= Matrix.CreateRotationZ(-katObrotu);

        Vector2 wychylenieLewegoGrzybka = stanGamepada.ThumbSticks.Left;
        prostopadloscian.MacierzSwiata *=
            Matrix.CreateRotationY(katObrotu * wychylenieLewegoGrzybka.X);
        prostopadloscian.MacierzSwiata *=
            Matrix.CreateRotationX(-katObrotu * wychylenieLewegoGrzybka.Y);

        Vector2 wychyleniePrawegoGrzybka = stanGamepada.ThumbSticks.Right;
        prostopadloscian.MacierzSwiata *=
            Matrix.CreateRotationY(10 * katObrotu * wychyleniePrawegoGrzybka.X);
        prostopadloscian.MacierzSwiata *=
            Matrix.CreateRotationX(-10 * katObrotu * wychyleniePrawegoGrzybka.Y);

        base.Update(gameTime);
    }
```

Stworzony w tym rozdziale komponent prostopadłościanu będzie naszym modelem w kolejnych częściach samouczka. W następnych rozdziałach przetestujemy na nim oświetlenie i teksturowanie.

Skorowidz

Poznaj tajniki programowania grafiki 3D we frameworku MonoGame, dzięki któremu powstały takie hity jak *Stardew Valley* czy *Carrion*. MonoGame jako kontynuator XNA oferuje twórcom gier ogromne perspektywy, swobodę i możliwość dostarczania gier na wszystkie najpopularniejsze systemy operacyjne, łącznie z mobilnymi, a przy tym pozostaje całkowicie darmowym narzędziem. Książka jest adresowana do wszystkich zainteresowanych tworzeniem gier i mających przynajmniej podstawową wiedzę o używanym w MonoGame języku programowania C#.

MonoGame. Podstawowe koncepcje grafiki 3D

dotyczy programowania grafiki 3D, co oznacza, że do jej tworzenia używa się kodu, a nie edytora sceny obsługiwanego myszką. Autorzy wyczerpująco omówili kluczowe zagadnienia, jak rozdzielenie definiowania figur i brył, poruszanie nimi na scenie, oświetlenie i cienie, mieszanie kolorów czy odwzorowanie tekstur. Wyjaśnili także kwestie wykraczające poza podstawy grafiki 3D, a dotyczące tworzenia gier, jak komponentyzacja ich produkcji czy projektowanie systemów sterowania. W rezultacie Czytelnik otrzymuje książkę na temat grafiki 3D w MonoGame łączącą teorię z wykorzystaniem praktycznych przykładów.

Dzięki książce:

- **zgłębisz programowanie grafiki 3D**
- **zaczniesz tworzyć trójwymiarowe gry wideo**
- **poznasz tajniki frameworka MonoGame**
- **odkryjesz zalety użycia C# w gamedevie**

Helion

helion.pl

HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-9350-9

Cena: 69,00 zł