

Damian Kaniewski, Tomasz Dziubak, Jacek Matulewski

MonoGame

Podstawowe konceptcje grafiki 3D

Mono
Game

Helion 

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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

Copyright © Helion S.A. 2023

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Wstęp	7
CZĘŚĆ I. Programowanie grafiki 3D z użyciem MonoGame	9
ROZDZIAŁ 1. Tworzenie gry MonoGame	11
Narzędzia i środowisko pracy	11
Instalacja i konfiguracja środowiska	12
Pierwszy projekt	16
Klasa gry	17
ROZDZIAŁ 2. Tworzymy pierwszy obiekt	21
Werteksy, efekt i rysowanie trójkąta	21
Nawijanie	23
Prymitywy	25
Kolory jako własności werteksów	26
ROZDZIAŁ 3. Macierze i kamera	29
Podstawowe przekształcenia	29
Rzutowanie sceny na ekran	31
Ustawienie kamery	32
Prosta animacja	33
Bufor werteksów	33
ROZDZIAŁ 4. Komponenty gry	35
Komponenty — wprowadzenie	35
Pierwszy komponent	35
Kontrola ułożenia prostopadłościanu	41
ROZDZIAŁ 5. Oświetlenie wirtualnej sceny. Model Phong	43
Model oświetlenia Phong	43
Wektory normalne	43
Definiowanie własnego formatu werteksów	44
Definiowanie wektorów normalnych modelu	45
Oświetlenie domyślne	47
Składowe światła w modelu oświetlenia Phong	48
Eksperymenty z własnościami materiału i źródłami światła	52

Różnice w cieniowaniu dla werteksów i dla pikseli	53
Uśrednienie i interpolacja normalnych	54
ROZDZIAŁ 6. Cienie rzucane	55
ROZDZIAŁ 7. Mieszanie kolorów	63
Alpha blending	63
Przezroczystość	64
Mgła	71
ROZDZIAŁ 8. Odzworowanie tekstur	75
Teksturowanie	75
Adaptacja aktora	76
Dodawanie tekstur do projektu	79
Zawijanie tekstur	82
Wiele obrazów w jednej teksturze	85
Przezroczystość	88
ROZDZIAŁ 9. Kontrolery gier	91
Sterowanie	91
Klawiatura	92
Wykrywanie zmian w stanie kontrolerów	94
Gamepad	95
Mysz	98
ROZDZIAŁ 10. Sfera, bufor indeksów i cieniowanie Phong'a	103
Komponent	103
Bufor indeksów	105
Konstrukcja sfery	106
Oświetlenie — uśrednianie normalnych	110
Teksturowanie	111
Ostatnie szlify	113
ROZDZIAŁ 11. Skybox i odzworowywanie otoczenia	115
Skybox	115
Alternatywne rozwiązanie	122
Odzworowanie otoczenia	125
 CZĘŚĆ II. Breakout 3D	 131
ROZDZIAŁ 12. Projekt gry	133
Tworzenie projektu gry	134
Ustawienie kamery	135
Wczytywanie i przygotowanie modelu	136

ROZDZIAŁ 13. Plansza, paletka oraz cegły	143
Plansza	143
Paletka	147
Cegły	151
ROZDZIAŁ 14. Piłka	157
ROZDZIAŁ 15. Ostatnie szlify	165
Zmiana pozycji kamery	165
Interfejs 2D	167
Optymalizacja obiektów gry	171
Różne kolory cegieł	173
CZĘŚĆ III. HLSL	177
ROZDZIAŁ 16. Język HLSL	179
Shadery	179
Środowisko programistyczne FX Composer	181
Typy zmiennych w HLSL	186
Semantyki	187
Funkcje, struktury, pętle i instrukcje warunkowe	189
Pliki efektu i techniki	191
ROZDZIAŁ 17. Tworzenie własnego efektu HLSL	195
Pierwszy własny efekt	195
Definiowanie zmiennych globalnych	198
Prezentacja pozycji za pomocą koloru	200
Wykorzystanie efektów w projekcie gry MonoGame	201
ROZDZIAŁ 18. Implementacja oświetlenia Phong'a w shaderach	207
Model Phong'a	207
Wprowadzenie do optyki	208
Implementacja modelu Phong'a w efekcie HLSL	213
Światło emisji i otoczenia	214
Światło rozproszone	215
Połysk	217
Oświetlenie per pixel	220
Użycie efektu w MonoGame	224
ROZDZIAŁ 19. Podstawy teksturowania z wykorzystaniem HLSL	227
Wprowadzenie	227
Implementacja	230
Użycie efektu w MonoGame	241
Skorowidz	247

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 `DrawableGameComponent`. 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 teksturuwanie.

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* zdefiniujemy pole o nazwie `efekt` typu `BasicEffect` i w metodzie `Game1.Initialize` zainicjujemy 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 *Prostopadloscian1*. 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. `DrawableGameComponent`. 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

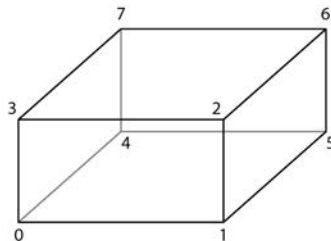
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;

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 zdefiniujemy 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 `DrawablesGameComponent`, 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 `Prostopadloscian`, 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()
{

```

```

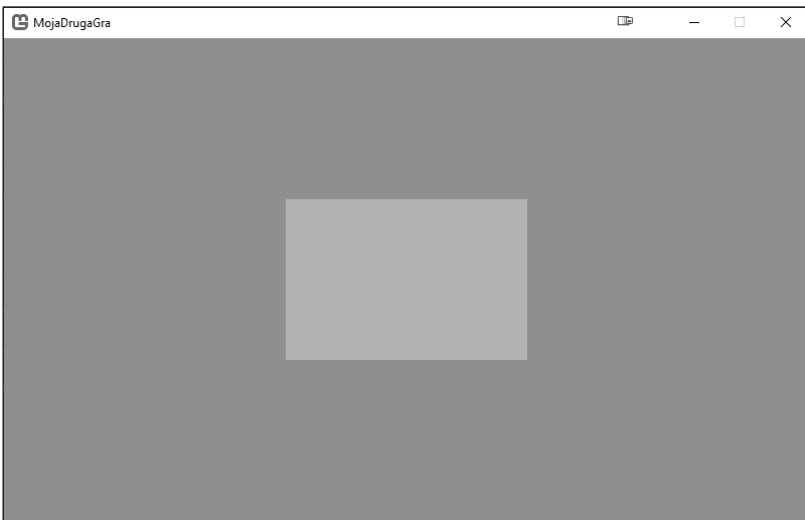
efekt = new BasicEffect(graphics.GraphicsDevice);
...
efekt.World = Matrix.Identity;

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).



RYСУNEK 4.2. Prostopadłościan w domyślnym ustawieniu na scenie

LISTING 4.7. Zdefiniowana w komponencie własność udostępniająca macierz świata

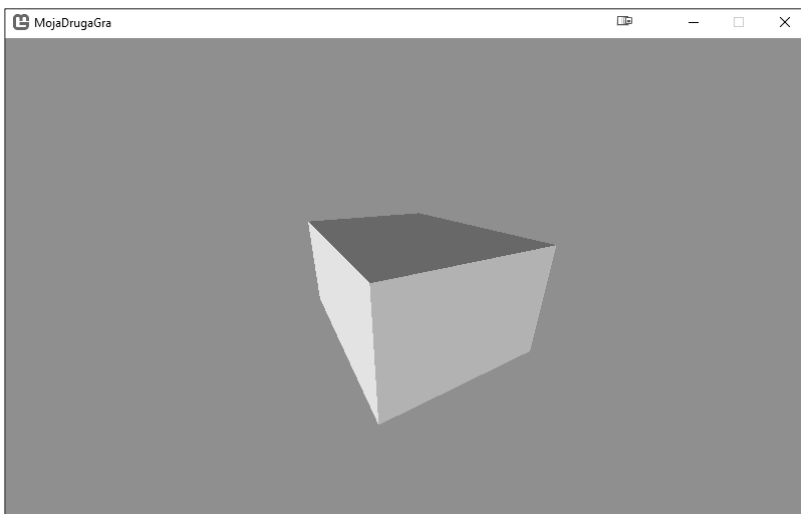
```

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 tekstuowanie.

Skorowidz

A

aktor, 76
alpha blending, 63
animacja, 33

B

bitmapa, 75
Breakout3D
 projekt gry, 134
bufor
 indeksów, 105, 108
 werteksów, 33

C

C# for Visual Studio Code, 15
cienie rzucane, 55
cieniowanie, 28
 dla pikseli, 53
 dla werteksów, 53
 Phonga, 103
culling, 192

E

efekty
 HLSL, 213
 inicjowanie, 36
 mgły, 72
 odwzorowania otoczenia,
 126
 w projekcie gry MonoGame,
 201

F

format FBX, 137
FX Composer, 181
 generowanie domyślnego
 obiektu, 183
interfejs programu, 182,
 195, 213
okno
 Editor, 184
 Effect Wizard, 184
 Materials, 184
 startowe, 182

G

gamepad, 95
gra
 Breakout, 133
 komponenty, 35
 tworzenie projektu, 16

H

HLSL, 179
funkcje, 189
instrukcje warunkowe, 189
mapowanie nierówności,
 235
multitexturing, 238
pętle, 189
prezentacja pozycji, 200
program do teksturowania,
 230
semantyki, 187
struktury, 189
teksturowanie proste, 232
tworzenie własnego efektu,
 195
typy tekstur, 227
typy zmiennych, 186
zmiennie globalne, 198

I

implementacja modelu
 Phonga, 213
instalacja środowiska, 12
interfejs 2D, 167

J

język HLSL, 179

K

kamera
 ustawianie, 32, 135
 zmiana pozycji, 165
kanał alfa, 193
klasa
 aktora, 76
 Ball, 157, 158
 BasicEffect, 21

Board, 143, 144
Breakout3D, 145, 149, 154,
 162, 174
Brick, 152, 153, 172, 174
Camera, 135
Game1, 17, 41, 47, 52, 56,
 59, 244
GameObject, 134
nowego komponentu, 37
Paddle, 147, 148, 173
Prostopadloscian, 86, 203,
 245
Sfera, 107
Skybox, 117, 122
TextureCube, 122
klawiatura, 92
 wykrywanie zmiany stanu,
 94
kolor światła
 emisyjnego, 209
 otoczenia, 209
 rozproszonego, 210
kolory, 26
 mieszanie, 63
komponent
 konstruktor, 37, 46
 rejestrwanie, 39
 sterowanie orientacją, 41
 tworzenie, 35, 39, 103
konfiguracja środowiska, 12
konstruktor komponentu, 37,
 46
kontrolery gier, 91


M

macierze, 29
 mnożenie, 29
 projekcji, 31
 rzutowania, 31, 58
mapowanie nierówności,
 bump mapping, 229, 234
mgła, 71
mieszanie
 kolorów, alpha blending,
 63, 193
 tekstur, multitexturing,
 229, 238

- model Phonga, 207
 MonoGame, 11
 Content Builder, 11, 15
 implementacja efektów, 224, 241
 multitexturing, 238
 mysz, 98
- N**
- nawijanie, 23
 normalna, 54, 210
 werteksu, 110
- O**
- obraz rastrowy, 75
 obrót cienia, 60
 obsługa
 gamepada, 96
 klawiatURY, 92
 myszy, 101, 102
 odwzorowanie
 otoczenia, 115, 125
 tekstur, 75
 optyka, 208
 oświetlenie, 43
 domyślne, 47
 per pixel, 220
 Phonga, 43
 Phonga w shaderach, 207
 uśrednianie normalnych, 54, 110
- P**
- pliki efektu, 191
 połysk, 211, 217
 prawo
 Lamberta, 210
 odbicia, 208
 program
 Blender, 136
 FBX Converter, 137
 FX Composer, 181
 MGCB Editor, 80, 193
 projekt gry Breakout3D, 133
 cegły, 151
 dodawanie modelu, 140
 eksport modelu, 138
 interfejs 2D, 167
 kolory cegieł, 173
 optymalizacja obiektów gry, 171
 paletka, 147
- piłka, 157
 plansza, 143
 przetwarzanie modelu, 139
 tworzenie, 134
 ustawienie kamery, 135
 zmiana pozycji kamery, 165
- prostopadłością, 37, 41
 z teksturą, 82
 prymitywy, 25, 27
 przebiegi, passes, 191
 przekształcenia, 29
 przetwarzanie modelu, 136
 przezroczystość, 64, 88, 193
- R**
- rejestrowanie komponentu, 39
 rotacja, 30
 rysowanie
 bryły, 108
 trójkąta, 21
 rzut równoległy, 62
 rzutowanie, 31
 cieni, 55
- S**
- sfera
 konstruowanie, 106
 shadery, 179
 implementacja oświetlenia Phongą, 207
 pikseli, pixel shader, 179
 werteksów, vertex shader, 179
 skybox, 115
 słowo kluczowe Technique, 192
 stany renderowania, render states, 192
- Ś**
- środowisko programistyczne
 FX Composer, 181
 światło
 emisyjne, emission, 50, 209, 214
 otoczenia, ambient, 49, 209, 214
 rozbłyску, specular, 49
 rozproszzone, diffuse, 49, 210, 215
- T**
- tablica werteksów, 21, 33
 techniki, techniques, 191
 teksturowanie, 75, 111
 z wykorzystaniem HLSL, 227
 tekstury
 dla skyboksów, 116
 dodawanie do projektu, 79
 dodawanie wielu obrazów, 85
 filtrowanie, 86
 interpolacja liniowa, 85
 sfery, 112
 tablica sześciu tekstur, 122
 typy w HLSL, 227
 zawijanie, 82
 translacja, 30
 tworzenie
 efektu HLSL, 195
 komponentu, 39
 obiektu, 21
 projektu gry, 16, 134
- V**
- Visual Studio Code, 11
- W**
- wektory normalne, 43, 210
 definiowanie, 45
 interpolacja, 54
 uśrednianie, 54, 110
 werteksy, 21
 bufor, 33
 cieniowanie, 53
 kolorowanie, 200
 tablice, 21, 33
 własności, 26
 własny format, 44
 własności
 materiałów, 52
 werteksów, 26
 współczynnik połysku, 212
- Z**
- zawijanie tekstur, 82
- Ż**
- źródła światła, 52

PROGRAM PARTNERSKI

— GRUPY HELION —

- 
1. ZAREJESTRUJ SIĘ
 2. PREZENTUJ KSIĄŻKI
 3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA
Helion

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 rozdzielanie 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**
- **zacznesz tworzyć trójwymiarowe gry wideo**
- **poznasz tajniki frameworka MonoGame**
- **odkryjesz zalety użycia C# w gamedevie**

	<p>KOD KORZYŚCI <i>Sięgnij po więcej!</i> ▶</p> 
 helion.pl	<p>ISBN 978-83-283-9350-9</p>  9 788328 393509
 <p>HELION SA ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl</p>	<p>Cena: 69,00 zł</p>