# Rozdział 2,
# Kilka praktycznych uwag dotyczących projektów gier w MonoGame

*Wersja z 2014-03-07b*

Ten krótki rozdział jest czymś w rodzaju aneksu do rozdziału pierwszego. Omawiam w nim kilka zagadnień, nie związanych bezpośrednio z grafiką 3D, ale które często pojawiają się w kontekście tworzenia gier.

## Usuwanie obiektu SpriteBatch

Po utworzeniu projektu gry MonoGame, w klasie `Game1` znajdziemy pole `spriteBatch` typu `SpriteBatch`. Do tej referencji w metodzie `LoadContent` przypisywane jest odwołanie do obiektu, który ułatwia programowanie gier 2D. W szczególności umożliwia rysowanie tekstur i napisów, których położenie ustalane jest nie za pomocą współrzędnych trójwymiarowego układu odniesienia, jaki poznaliśmy w tym module, a za pomocą współrzędnych okna mierzonych w pikselach. W tym skrypcie, poza problemem 4 w laboratorium podstawowym modułu 5 i laboratorium rozszerzonym modułu 7 nie będziemy z tego obiektu korzystać. Wobec tego nic nie stoi na przeszkodzie, aby pole to usunąć z klasy. Jak to zrobić?

1. W pliku *Game1.cs* należy usunąć deklarację pola `spriteBatch` z klasy `Game1` (znajduje się na jej początku).
2. Z metody `Game1.LoadContent` (w tym samym pliku) należy usunąć polecenie `spriteBatch = new SpriteBatch(GraphicsDevice);` (jedyne w tej metodzie poza komentarzami).

## Wstrzymanie gry przy dezaktywacji okna

Większość gier reaguje na klawisz *P* lub *Pause/Break* wstrzymaniem gier. To samo robią, gdy gracz przełączy aktywne okno (np. klawiszami *Alt+Tab*). Można to łatwo uzyskać dodając na początku metod `Game1.Update` i `Game1.Draw` instrukcje warunkowe:

```
if (!this.IsActive) return;
```

## Tryb pełnoekranowy lub zmiana rozmiaru okna gry

Gry korzystające z grafiki trójwymiarowej uruchamiane są zwykle na pełnym ekranie. Przy czym jest w ich zwyczaju, że zmieniają rozdzielczość monitora tak, aby dostosować ilość pikseli dla jakich trzeba obliczyć kolor do możliwości karty graficznej i procesora. W MonoGame przełączenie w tryb pełnoekranowy jest dziecinnie proste dzięki temu, że klasa `Game1` implementująca okno gry jest już wyposażona w odpowiednie mechanizmy. Przełączanie w tryb pełnoekranowy ze zmianą rozdzielczości uzyskamy umieszczając w metodzie `Initialize`, przed ustaleniem macierzy rzutowania (ze względu na używaną do tego proporcję ekranu), następujący kod:
Bug: Zmiana wysokości, a przez to proporcji ekranu nie powoduje zmiany własności `graphics.GraphicsDevice.Viewport.AspectRatio`.

```
graphics.PreferredBackBufferWidth = 800;
```

```
graphics.PreferredBackBufferHeight = 600;
graphics.IsFullScreen = true; //tylko w Windows
graphics.ApplyChanges();
```

Jeżeli usuniemy instrukcję zmieniającą własność `IsFullScreen` to metoda zmieni jedynie rozmiar okna, w którym wyświetlana jest grafika dostosowując obszar roboczy do podanych szerokości i wysokości.

Oczywiste wydaje się otoczenie tych instrukcji konstrukcją `try..catch` przechwytującą wyjątki zgłaszane w razie podania nieodpowiednich rozmiarów okna (np. ujemnych), a przy przejścia w tryb pełnoekranowy, nieobsługiwanych rozdzielczości. Okazuje się jednak, że w MonoGame, inaczej niż w XNA, żadne wyjątki nie są w takich sytuacjach zgłaszane.

Powyższa metoda zmienia rozdzielczość na najniższą rozdzielczość większości obecnie dostępnych kart graficznych tj. 800x600. Jeżeli chcemy zachować bieżącą rozdzielczość zamiast używać stałych wartości odczytajmy bieżące wartości szerokości i wysokości obrazu w pikselach:

```
graphics.PreferredBackBufferWidth = graphics.GraphicsDevice.DisplayMode.Width;
graphics.PreferredBackBufferHeight = graphics.GraphicsDevice.DisplayMode.Height;
```

Zamiast zmieniać własność `IsFullScreen` obiektu `graphics`, a potem wywoływać jego metodę `ApplyChanges`, aby wprowadzić zmiany w życie możemy użyć metody `graphics.ToggleFullScreen`. Już wiemy, że to nie działa (psuje i poza tym też wymaga ApplyChanges)

## Częstość wywoływania metody Update

W ramach końcowych ustawień projektu dotyczących obsługi okna przez platformę MonoGame. Warto na przykład wiedzieć, że można zmienić częstość wywoływania metody `Update` równą domyślnie 60Hz. Nie warto natomiast majstrować przy częstości wywoływania metody `Draw`, która jest wyświetlana w miarę możliwości z częstością odświeżania karty graficznej i monitora.

Częstość 60Hz (typowa dla odświeżania monitorów LCD) oznacza, że metoda `Update` wywoływana jest co 16.667 milisekund. Zmniejszanie tego okresu nie ma sensu – i tak nie zobaczymy efektu. No chyba, że dysponujemy monitorem z większą częstością odświeżania np. monitorem CRT. Załóżmy, że chcemy zwiększyć częstość odświeżania do 75Hz. Wówczas okres między wywołaniem metody `Update` powinien być równy 13.333. To oznacza, że w metodzie `Initialize` należy umieścić instrukcję:

```
this.TargetElapsedTime = TimeSpan.FromMilliseconds(13.333f);
```

lub

```
this.TargetElapsedTime = TimeSpan.FromSeconds(1.0f/75.0f);
```

Ja wolę jednak inne rozwiązanie. Naturalne wydaje mi się, żeby funkcja `Update` zmieniająca ustawienia aktorów na ekranie wywoływana była zawsze przed wywołaniem metody `Draw`, która aktorów narysuje. Aby to uzyskać należy ustawić własność `IsFixedTimeStep` bieżącego obiektu klasy `Game1` na `false`:

```
this.IsFixedTimeStep = false;
```

Zwróćmy uwagę na zasadniczą różnicę względem tego, jak zwykle rozwiązuje się ten problem w aplikacjach Windows. Jeżeli nie mamy do czynienia z animacją odświeżanie zawartości sceny może być wykonywane jedynie w momencie, w którym do okna aplikacji dociera komunikat `WM_PAINT` lub inny komunikat związany z naciśnięciem klawisza lub ruchem myszki. W MonoGame przy domyślnych ustawieniach wymuszony jest natomiast stały cykl odświeżania.

## Własności okna gry

Teraz zupełna drobnostka – tytuł okna. Jeżeli nie zmienimy trybu na pełnoekranowy, możemy zadbać o jego zmianę. Odpowiada za to własność `this.Window.Title`, np.:

```
Window.Title = "Moja pierwsza gra MonoGame";
```

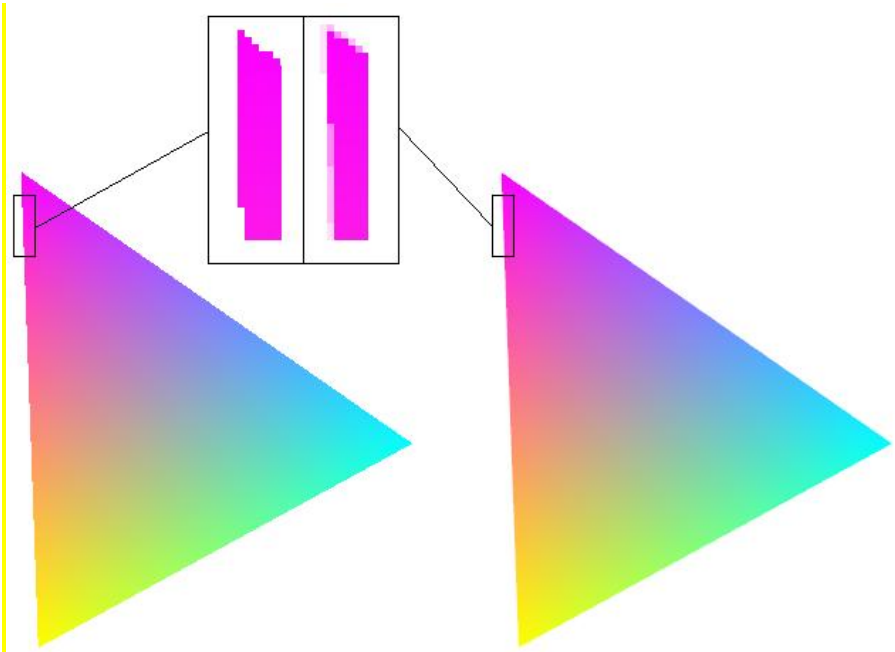
Domyślnie rozmiar okna jest ustalony i użytkownik nie może go zmieniać (pomijam zaprezentowaną powyżej możliwość zmiany jego rozmiaru z poziomu kodu). Można jednak również umożliwić zmianę rozmiaru okna użytkownikowi np. za pomocą myszy. Należy w tym celu przełączyć własność `Window.AllowUserResizing` na `true`. Aby to rzeczywiście działało, w MonoGame, inaczej niż było w XNA, konieczne jest także przechwycenie zdarzenia Window.SizeChanged, z którym należy związać metodę widoczną na poniższym listingu:

```
public Game1()
    : base()
{
   graphics = new GraphicsDeviceManager(this);
   Content.RootDirectory = "Content";

   this.Window.ClientSizeChanged += new EventHandler<EventArgs>(Window_ClientSizeChanged);
}

void Window_ClientSizeChanged(object sender, EventArgs e)
{
   System.Reflection.FieldInfo field = typeof(OpenTKGameWindow).GetField(
      "updateClientBounds",
       System.Reflection.BindingFlags.NonPublic | System.Reflection.BindingFlags.Instance);
   if (field != null) field.SetValue(this.Window, false);

   efekt.Projection = Matrix.CreatePerspectiveFieldOfView(
      MathHelper.PiOver2, graphics.GraphicsDevice.Viewport.AspectRatio, 1, 100);
}
```

Pamiętajmy, że użytkownik może zmieniając rozmiar okna zmienić również jego stosunek szerokości do wysokości. To powinno być odzwierciedlone w aktualizacji macierzy rzutowania. Należy więc w takiej sytuacji ustalenie wartości tej macierzy aktualizować w przy każdej zmianie rozmiaru okna (ostatnia instrukcja metody `Window_ClientSizeChanged`).

## Wygładzanie krawędzi (antyaliasing)

Obracanie krawędzi trójkąta lub kwadratu uwypukla zjawisko, które jest zmorą grafiki komputerowej w ogóle, nie tylko grafiki trójwymiarowej. Punkt na ekranu może zajmować jedną z dyskretnych pozycji wyznaczonych przez piksele monitora. To powoduje, że nie można narysować gładkiej linii innej niż pozioma lub pionowa (może jeszcze linie nachylone pod kątem 45° wyglądają znośnie). Wszystkie pozostałe linie ukośne, a te w grafice trójwymiarowej w rzucie perspektywicznym dominują, są „schodkowe”. Efekt ten nazywa się *aliasingiem*. Jest tym wyraźniejszy, im mniejsza jest rozdzielczość ekranu. Trudno go jednak wyeliminować zwiększeniem rozdzielczości, gdyż to bardzo obniża wydajność przetwarzania grafiki. Można jednak wykorzystać pewien trik, aby nieco oszukać nasze oko. Sztuczka ta polega na takim wymieszaniu kolorów w pobliżu krawędzi, aby uzyskać gładkie przejście między kolorami z obu jej stron (zob. rysunek poniżej). Mówiąc wprost chodzi o rozmycie obrazu w pobliżu krawędzi. Oczywiście procedura taka musi być realizowana sprzętowo – inaczej kosztowałaby zbyt wiele czasu głównego procesora CPU.



Rysunek 9. Efekt włączenia wygładzania krawędzi

Aby włączyć wygładzanie krawędzi, należy zmodyfikować konstruktor klasy gry `Game1` oraz zdefiniować metodę zdarzeniowej związaną ze zdarzeniem `GraphicsDeviceManager.PreparingDeviceSettings`.

```
public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";

    graphics.PreferMultiSampling = true;
    graphics.PreparingDeviceSettings += new
        EventHandler<PreparingDeviceSettingsEventArgs>(graphics_PreparingDeviceSettings);
}

void graphics_PreparingDeviceSettings(object sender, PreparingDeviceSettingsEventArgs e)
{
    //antyaliasing
    e.GraphicsDeviceInformation.PresentationParameters.MultiSampleQuality = 0;
    e.GraphicsDeviceInformation.PresentationParameters.MultiSampleType =
        MultiSampleType.FourSamples;
}
```

Poniższe ustawienia, w których do ustalenia koloru pikselu korzysta się z czterech sąsiednich pikseli jest dość bezpieczne – takie próbkowanie wspierane jest przez większość kart graficznych, także przez procesor graficzny *ATI R500 Xenos* montowany w Xbox 360.
Nie ma odpowiednich własności w GraphicsDeviceInformation; nie ma wsparcia dla MSAA w MonoGame (zarówno w OpenGL, jak i DirectX):
http://monogame.codeplex.com/discussions/531445
KONKURS: +1/2 dla osoby, która znajdzie obejście, dzięki któremu można włączyć MSAA np. dzięki sięgnięciu bezpośrednio do OpenGL via PInvoke.

## Tylko jedna instancja gry

Gry często nie pozwalają na uruchamianie wielu instancji. Chodzi przede wszystkim o wydajność i wyłączność korzystania z zasobów. Najprościej zablokować uruchamianie kolejnych instancji korzystając z mutexu (ang. *mutual exclusion*, wzajemne wykluczanie). Jest to technika wykorzystywana w programowaniu współbieżnym (wielowątkowym) i polega na stworzeniu w jednym z wątków obiektu mutexu, którego obecność jest sygnałem dla innych wątków, że mają wstrzymać swoje działanie do momentu gdy pierwszy wątek zakończy krytyczny fragment i usunie

mutex z pamięci. Ponieważ obiekty mutexów tworzone są na poziomie jądra systemu (klasa `Mutex` z platformy .NET jest tylko „opakowaniem” dla funkcji WinAPI), dotyczą nie tylko wątków jednego procesu, ale również wielu procesów.
Gra będzie sprawdzała, czy obecny jest mutex o specyficznej dla tej gry nazwie. Jeżeli takiego nie ma, utworzy go. Utworzenie mutexu uda się tylko pierwszej instancji gry. Kolejne będą wykrywały jego obecność, co będzie dla nich sygnałem, że powinny niezwłocznie zakończyć swoje działanie.

```
#region Using Statements
using System;
using System.Collections.Generic;
using System.Linq;
#endregion

using System.Threading;

namespace MojaPierwszaGraMonoGame
{
#if WINDOWS || LINUX
    /// <summary>
    /// The main class.
    /// </summary>
    public static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            bool czyPierwszaInstancja;
            Mutex m = new Mutex(true, "MojaPierwszaGraMonoGame", out czyPierwszaInstancja);
            if (!czyPierwszaInstancja)
            {
                Console.Beep();
                return;
            }

            using (var game = new Game1())
                game.Run();

            m.ReleaseMutex();
        }
    }
#endif
}
```