BEÁGYAZOTT RENDSZEREK ALAPJAI

Grafikus benchmark alkalmazás

A laborfoglalkozás célja

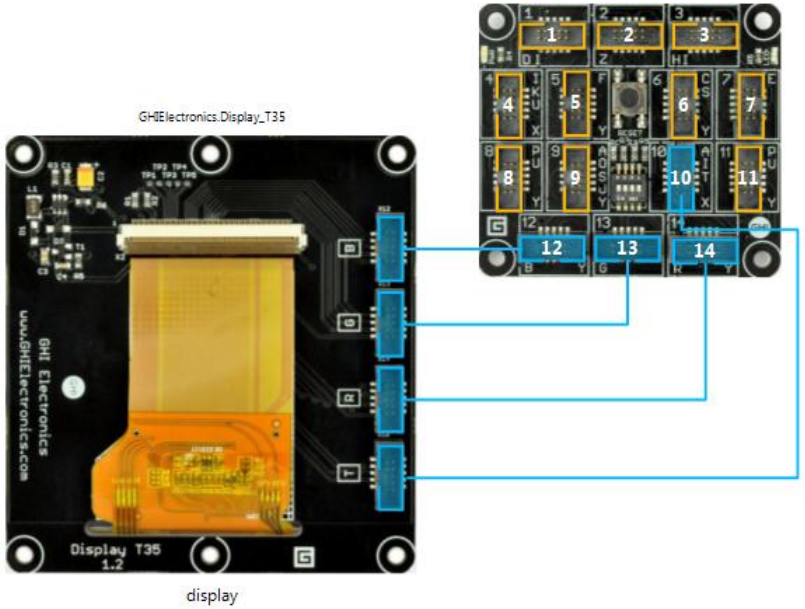
- A foglalkozás célja egy grafikus „benchmark” alkalmazás elkészítése, egyszerű alakzatok rajzolásával
  - A megjelenítendő objektumok a program elején kerülnek létrehozásra, típusuk véletlenszerű
  - Minden egyes jelenet kirajzolása előtt a grafikus objektumok véletlenszerűen kiválasztott, új pozíciót vesznek fel
  - A rajzolásra fordított időt mérjük, és kiszámítjuk az FPS (Frame Per Secundum) értéket, tehát azt, hogy az eszköz 1 másodperc alatt hányszor tudott új jelenetet generálni és azt kirajzolni

A laborfoglalkozás menete

1. A hardver konfiguráció összeállítása
2. Osztályhierarchia kialakítása
3. A program terve
4. Implementálás
5. Tesztelés

A hardver konfiguráció összeállítása

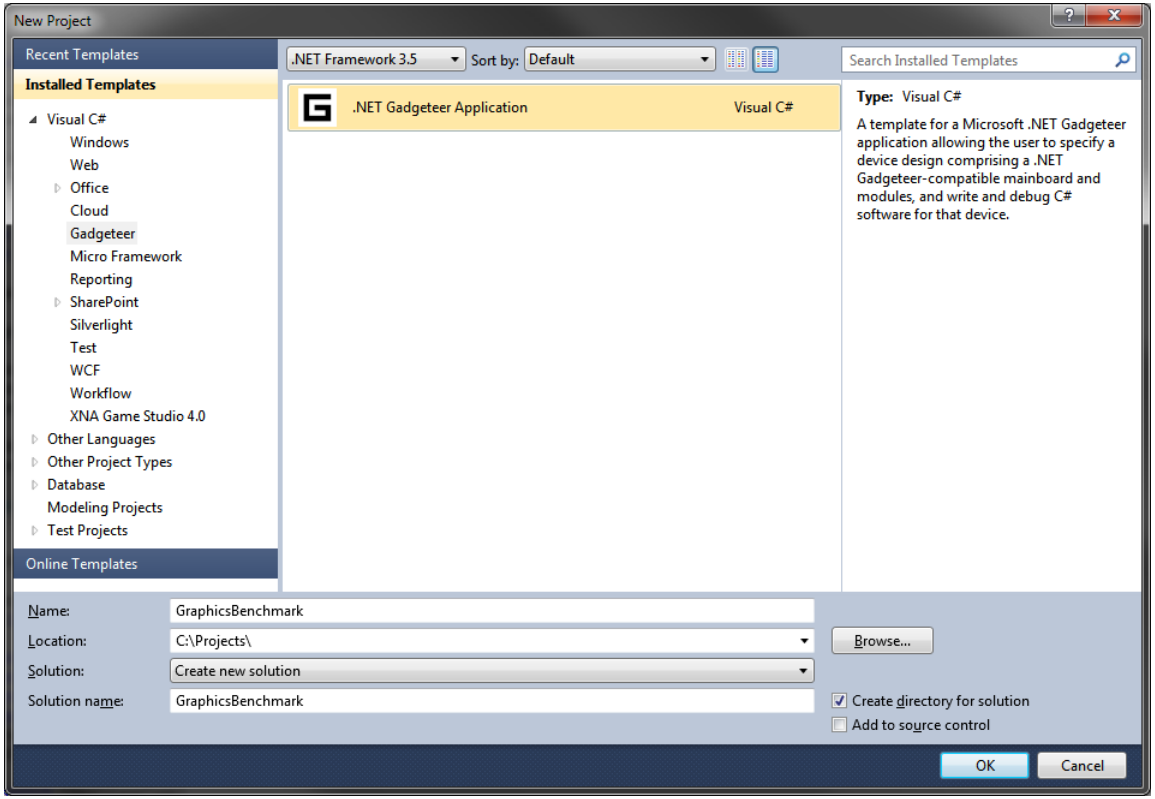
- Állítsuk össze a következő konfigurációt:



display

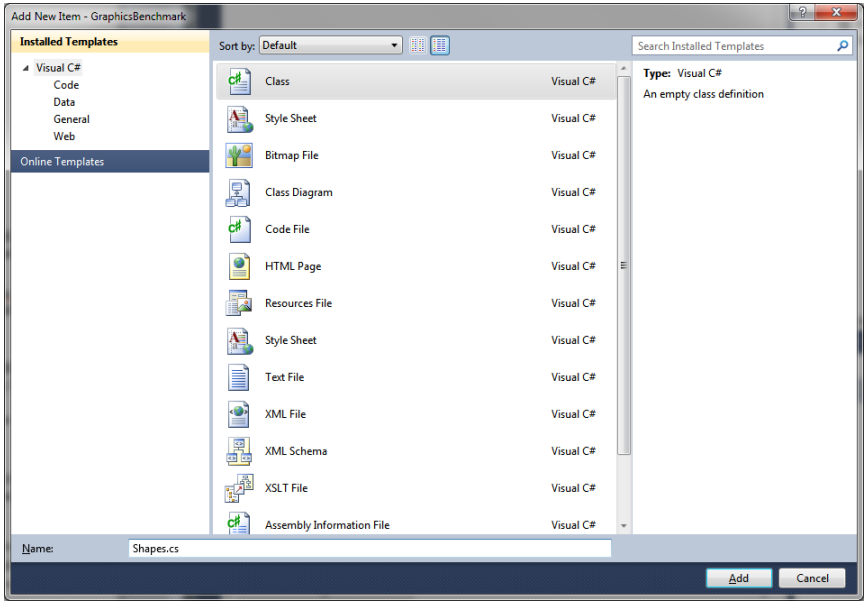
Új alkalmazás létrehozása

- Hozzunk létre egy új Gadgeteer alkalmazást GraphicsBenchmark néven:



Osztályhierarchia kialakítása

- Az összes grafikus objektum egy új kódfájlba fog kerülni: „Shapes.cs”
- Ennek létrehozásához kattintsunk a „GraphicsBenchmark” projektre a jobb-egérgombbal, majd az „Add/New Item” menüpontra
- A megjelenő ablakban válasszuk ki a „Class” elemet, alul pedig adjuk meg névnek a következőt: „Shapes.cs”



Osztályhierarchia kialakítása

- Ha mindent jól csináltunk, akkor egy fájl jött létre, melynek tartalma új ablakban jelenik meg:

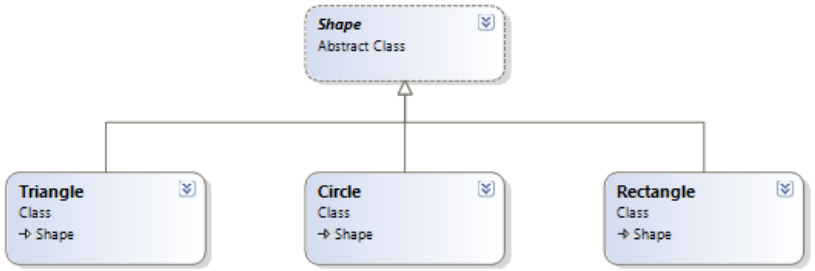
```
using System;
using Microsoft.SPOT;

namespace GraphicsBenchmark
{
    class Shapes
    {
    }
}
```

- Az alakzatok „ismerik” saját típusukat, melyet egy „ShapeType” nevű enum segítségével fogunk tárolni:

```
enum ShapeType
{
   Circle,
   Rectangle,
   Triangle
}
```

Osztályhierarchia kialakítása

- Egyetlen fájlban definiáljuk az összes alakzatot, ezért az osztály neve „Shape” lesz az eredetileg felkínált „Shapes” helyett (az s betűt töröljük ki az osztály nevéből)
- A „Shape” osztály lesz az összes grafikai alakzat ősosztálya



Osztályhierarchia kialakítása

```
abstract class Shape
{
     public int X { get; set; }
     public int Y { get; set; }
     public int Width { get; set; }
     public int Height { get; set; }
     public ShapeType Type { get; private set; }
     public Gadgeteer.Color Color { get; set; }

     public Shape(ShapeType shapeType)
     {
         this.Type = shapeType;
     }

      public abstract void DrawToBitmap(Bitmap bitmap);
}
```

- `abstract class Shape`: az alakzatot reprezentáló absztrakt osztály
- `public int X`: az alakzat X koordinátája a képernyőn
- `public int Y`: az alakzat Y koordinátája a képernyőn
- `public int Width`: az alakzat szélessége pixelben
- `public int Height`: az alakzat magassága pixelben
- `public ShapeType Type`: az alakzat típusa (a `private set` minősítő azért kell, hogy a példányosítást követően ne lehessen módosítani az alakzat típusát
- `public Gadgeteer.Color Color`: az alakzat színe
- `public Shape(ShapeType shapeType)`: konstruktor, amely paraméterként vár egy `ShapeType` típusú változót(ezzel a módszerrel kikényszerítjük, hogy az objektum létrehozásakor mindenképpen definiálva legyen a létrehozandó alakzat típusa)
- `this.Type = shapeType`: beállítjuk az objektum típusát
- `public abstract void DrawToBitmap(Bitmap bitmap)`: az alakzat kirajzolja önmagát a bitmap nevű bitképre. Ez a metódus absztrakt, tehát a leszármazott implementálja, hogy hogyan kell kirajzolnia önmagát.

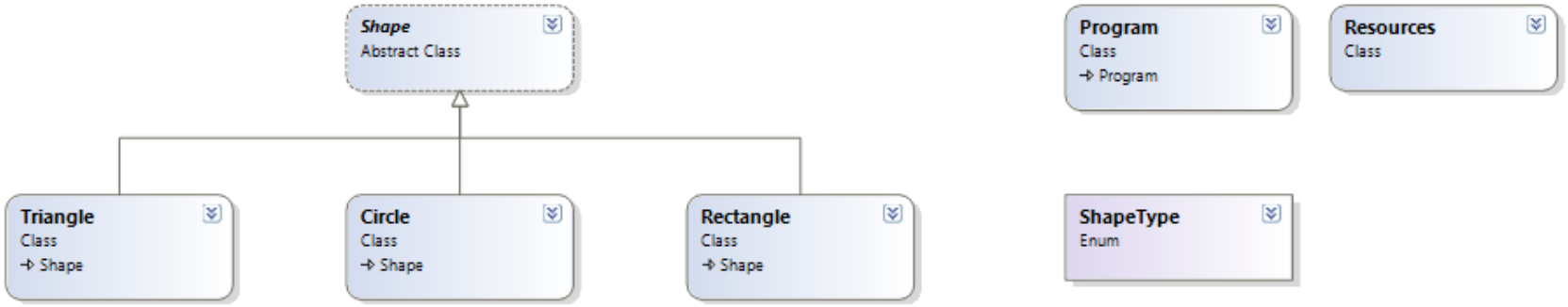
Osztályhierarchia kialakítása

- Definiáljunk háromféle alakzatot:
  - Kör (Circle)
  - Négyzet (Rectangle)
  - Háromszög (Triangle)
- Az osztályhierarchia kialakításával minden alakzat (leszármazott) számára előírjuk a kirajzolás módját (DrawToBitmap() metódus)
- Minden osztály konstruktorából meghívjuk az őskonstruktort, melyben paraméterül át kell adni az alakzat típusát (típus definiálása)

Osztályhierarchia kialakítása

```
class Circle: Shape
{
    public Circle()
        : base(ShapeType.Circle)
    {
    }

    public override void DrawToBitmap(Bitmap bitmap)
    {
        int halfWidth = Width / 2;
        int halfHeight = Height / 2;
        bitmap.DrawEllipse(Color, X + halfWidth, Y + halfHeight, halfWidth, halfHeight);
    }
}

class Rectangle: Shape
{
    public Rectangle()
        : base(ShapeType.Rectangle)
    {
    }

    public override void DrawToBitmap(Bitmap bitmap)
    {
        bitmap.DrawRectangle(Color, 1, X, Y, Width, Height, 1, 1, Color, 1, 1, Color, 1, 1, 255);
    }
}

class Triangle: Shape
{
    public Triangle()
        : base(ShapeType.Triangle)
    {
    }

    public override void DrawToBitmap(Bitmap bitmap)
    {
        int downY = Y + Height;
        int rightX = X + Width;
        int centerX = X + Width / 2;
        bitmap.DrawLine(Color, 1, X, downY, rightX, downY);
        bitmap.DrawLine(Color, 1, X, downY, centerX, Y);
        bitmap.DrawLine(Color, 1, centerX, Y, rightX, downY);
    }
}
```

Osztályhierarchia kialakítása



Shape: alakzatokat definiáló ősosztály
Triangle, Circle, Rectangle: alakzat leszármazottak
Program: a programot reprezentáló, futtatást végző osztály
Resources: erőforrásokat tartalmazó osztály (tervező generálja)
ShapeType: az alakzatok típusát meghatározó enum

A program terve

- A program futtatása során a következő műveleteket kell elvégezni:
    1. Inicializálás
        a) kijelző méret lekérdezése
        b) buffer létrehozása
        c) alakzatok létrehozása, inicializálása
    2. Alakzatok rajzolása a buffer-be (Bitmap)
    3. Buffer kirajzolása a képernyőre

Implementálás

- A futtatást végző „Program” osztályt (Program.cs) a következő mezőkkel kell kiegészíteni:

```
public partial class Program
{
      int screenWidth;
      int screenHeight;

      Shape[] shapes;
      Bitmap renderBitmap;
      …
```

- `int screenWidth:` a kijelző szélességét tároló változó
- `int screenHeight:` a kijelző magasságát tároló változó
- `Shape[] shapes:` az alakzat objektumokat tároló tömb
- `Bitmap renderBitmap:` buffer a rajzoláshoz

Implementálás

1. Inicializálás:

```
void ProgramStarted()
{
      screenWidth = (int) display.Width;
      screenHeight = (int) display.Height;
      renderBitmap = new Bitmap(screenWidth, screenHeight);
      InitShapes();
      Render();
      Debug.Print("Program Started");
}
```

Implementálás

1. Inicializálás:

```
void InitShapes()
{
      shapes = new Shape[3];
      shapes[0] = new Circle()
      {
          X = 10,
          Y = 20,
          Width = 10,
          Height = 10,
          Color = GT.Color.Red
      };
      shapes[1] = new Rectangle()
      {
          X = 10,
          Y = 100,
          Width = 10,
          Height = 10,
          Color = GT.Color.Yellow
      };
      shapes[2] = new Triangle()
      {
          X = 10,
          Y = 180,
          Width = 10,
          Height = 10,
          Color = GT.Color.White
      };
}
```

Implementálás

2. Alakzatok rajzolása a buffer-be:

```
void DrawToBitmap()
{
      for (int i = 0; i < shapes.Length; ++i)
      {
          Shape actualShape = shapes[i];
          actualShape.DrawToBitmap(renderBitmap);
      }
}
```

Implementálás

3. Buffer rajzolása a képernyőre:

```
void Render()
{
	DrawToBitmap();
	display.SimpleGraphics.DisplayImage(renderBitmap, 0, 0);
}
```

Implementálás

- Az „F5” billentyű segítségével elindítható a program „Debug” módban
- Amennyiben mindent jól csináltunk, megjelenik a háromféle alakzat a képernyőn, tehát a definiált osztályok jól működnek
- Fejlesszük tovább a programot úgy, hogy a kirajzolásra kerülő alakzatok legyenek véletlenszerűek és azok számát egy konstans segítségével lehessen megadni!

Implementálás

- A Program osztályon belül további konstansokra és mezőkre lesz szükség:

```
const int SHAPE_COUNT = 50;
const int MAX_SIZE = 20;
const int MIN_SIZE = 10;

Random randomizer = new Random();
```

- `const int SHAPE_COUNT`: az alakzatok száma
- `const int MAX_SIZE`: az alakzatok maximális mérete
- `const int MIN_SIZE`: az alakzatok minimális mérete
- `Random randomizer`: véletlen szám generátor

Implementálás

- Az InitShapes() metódust a következőképpen kell módosítani:

```
void InitShapes()
{
        shapes = new Shape[SHAPE_COUNT];
        for (int i = 0; i < SHAPE_COUNT; ++i)
        {
            int nextShapeType = randomizer.Next(3);
            int width = MIN_SIZE + randomizer.Next(MAX_SIZE - MIN_SIZE);
            int height = MIN_SIZE + randomizer.Next(MAX_SIZE - MIN_SIZE);
            byte[] rgb = new byte[3];
            randomizer.NextBytes(rgb);
            GT.Color color = GT.Color.FromRGB(rgb[0], rgb[1], rgb[2]);
            switch (nextShapeType)
                 {
                case 0:
                    shapes[i] = new Circle();
                    break;
                case 1:
                    shapes[i] = new Rectangle();
                    break;
                case 2:
                    shapes[i] = new Triangle();
                    break;
            }
            shapes[i].Width = width;
            shapes[i].Height = height;
            shapes[i].Color = color;
        }
}
```

Implementálás

- Az alakzatok pozíciója szintén véletlenszerűen kerül meghatározásra az inicializálás során, illetve a későbbiek folyamán minden egyes „frame” rajzolásánál a SetShapePositions() metódus hívásával:

```
void SetShapePositions()
{
        for (int i = 0; i < SHAPE_COUNT; ++i)
        {
            Shape actualShape = shapes[i];
            actualShape.X = randomizer.Next(screenWidth - actualShape.Width);
            actualShape.Y = randomizer.Next(screenHeight - actualShape.Height);
        }
}
```

- A Render() metódus szintén módosítást igényel:

```
void Render()
{
        SetShapePositions();
        DrawToBitmap();
        display.SimpleGraphics.DisplayImage(renderBitmap, 0, 0);
}
```

- Futtatással meggyőződhetünk róla, hogy a véletlenszerűen kiválasztott alakzatok megfelelően kerülnek kirajzolásra, véletlen pozícióra

Implementálás

- A rajzolást folyamatosan kell végezni egy cikluson belül, illetve minden iteráció során mérni kell a jelenet rajzolásra fordított időt. Ebből kiszámítható, hogy egy másodperc alatt hány „frame” rajzolható ki (FPS)
- Az idő méréséhez két új „DateTime” típusú mezőre lesz szükség a Program osztályon belül, az érték szöveges megjelenítéséhez pedig egy „Font” objektumra:

```
DateTime before;
DateTime after;
Font font = Resources.GetFont(Resources.FontResources.NinaB);
```

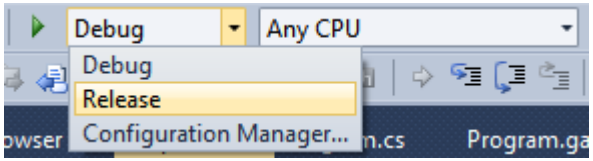
Implementálás

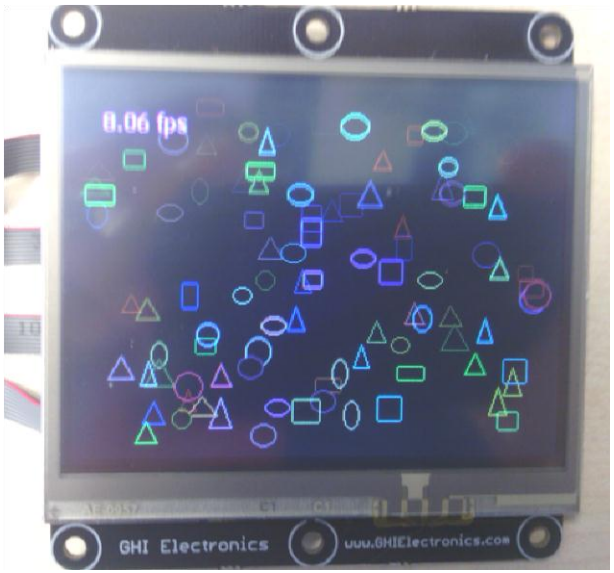
- A rajzolás előtt és azt követően is lekérjük a rendszeridőt
- A két idő különbségéből megkapjuk a „jelenet” elkészítésére fordított időt milliszekundumban
- Ebből kiszámítható az FPS érték :

```
void Render()
{
    while (true)
    {
        before = DateTime.Now;
        SetShapePositions();
        renderBitmap.Clear();
        DrawToBitmap();
        after = DateTime.Now;
        TimeSpan renderTime = after.Subtract(before);
        float fps = 1000.0f / renderTime.Milliseconds;
        renderBitmap.DrawText(fps.ToString("F2") + " fps", font, GT.Color.Magenta, 20, 20);
        display.SimpleGraphics.DisplayImage(renderBitmap, 0, 0);
    }
}
```

Tesztelés

- A tesztelés során a bal-felső sarokban látható az „FPS” érték, mely megmutatja, hogy egy másodperc alatt hányszor képes az eszköz az új jelenet előállítására és kirajzolására
- A programot teszteljük „Debug” (F5) és „Release” (Ctrl+F5) módban is, majd figyeljük meg a sebesség különbséget
- Vizsgáljuk meg, hogy az alakzatok száma hogyan befolyásolja a teljesítményt





+Feladat

- Az alakzatok ne önmagukat rajzolják ki, hanem a Render() metódus végezze a kirajzolásukat a típusuk szerint (Type mező).