

## Eseményvezérelt alkalmazások: 9. gyakorlat

A munkafüzet az animációk és triggerek használatát mutatja be, MVVM architektúrában.

A mai gyakorlaton egy Ping-Pong játékot fogunk elkészíteni animációkkal.

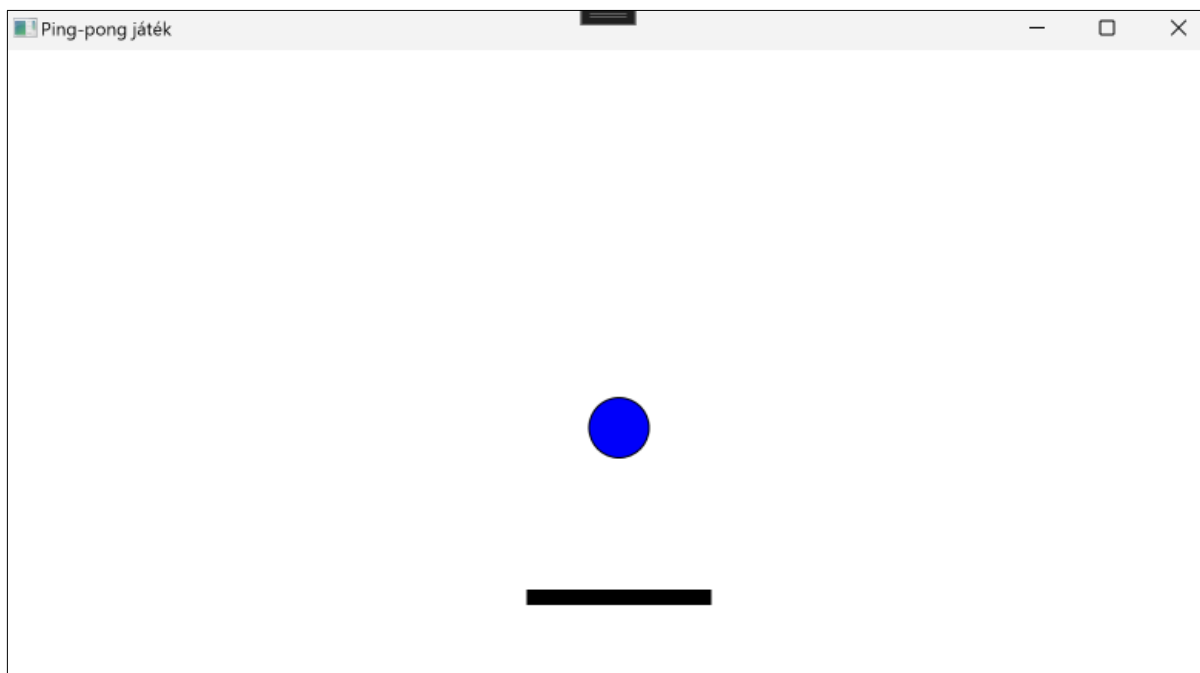


Figure 1: Ping-pong játék

A WPF támogatja az animációk végrehajtását, amely lényegében függőségi tulajdonságok adott időn keresztül történő folyamatos módosítását jelenti. A grafikus vezérlők lényegében bármelyik tulajdonságát animálhatjuk, definiálva a kezdőállapotot (**From**), a végállapotot (**To**), valamint az időt (**Duration**). A különböző típusokhoz megfelelő animációs típus létezik: **DoubleAnimation**, **ThicknessAnimation**, **ColorAnimation**, stb.

Az animációkat deklaratíván XAML kóddal vagy C# háttérkódban is megadhatjuk. Több animációt forgatókönyvbe (**Storyboard**) foglalhatunk - XAML kód esetén kötelező. Például:

```
<Button Name="myButton" ...>

<Storyboard Storyboard.TargetName="myButton" Duration="0:00:04">
  <DoubleAnimation From="1" To="0" Storyboard.TargetProperty="Opacity" />
</Storyboard>
```

Ugyanez C# háttérkóddal:

```
DoubleAnimation myAnimation = new DoubleAnimation
{
    From = 1,
    To = 0,
    Duration = new Duration(TimeSpan.FromSeconds(4))
};
myButton.BeginAnimation(Button.OpacityProperty, myAnimation);
```

## 1 Modell

Készítsük el a játék modell rétegét; vagy induljunk ki a *skeleton* projektből, amely a modell megvalósítását már tartalmazza. Utóbbi esetben is tekintsük át a modell felépítését, hiszen a nézetmodell elkészítéséhez az interfészének ismerete szükséges lesz.

### 1.1 Element

A pályán szereplő elemek (labda és ütő) reprezentálására hozunk létre egy **Element** osztályt. Az osztály az alábbiakat tartalmazza:

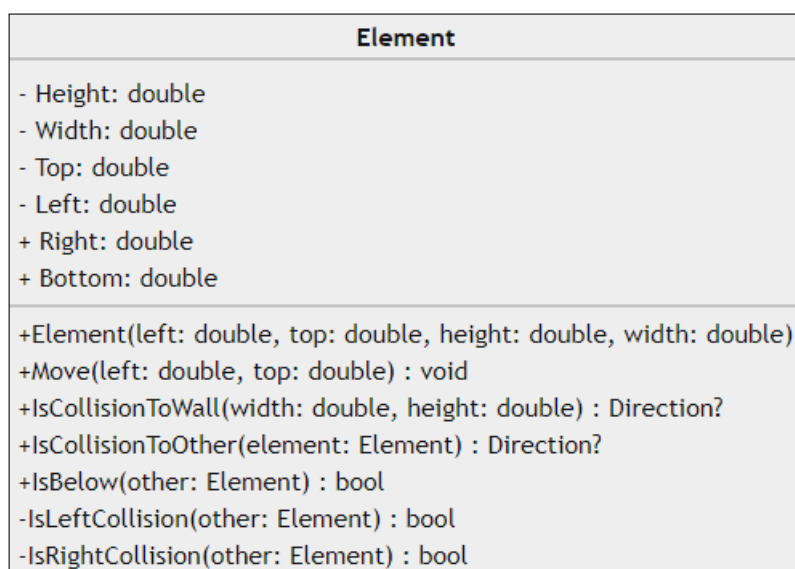


Figure 2: Az **Element** osztálydiagramja

- **Height**: Az elem magassága
- **Width**: Az elem szélessége
- **Top**: Az elem tetejének helyzete
- **Left**: Az elem bal oldalának helyzete
- **Right**: Az elem jobb oldalának helyzete. Vegyük észre, hogy ez igazából a **Left** + **Width** értékeként áll elő
- **Bottom**: Az elem aljának helyzete. Vegyük észre, hogy ez igazából a **Top** + **Height** értékeként áll elő

Az osztály tartalmazza az alábbi segédfüggvényeket:

- **Move(double left, double top)**: Az elem beállítása a kapott pozícióra
- **IsCollisionToWall(double width, double height)**: Ellenőrzi, hogy az elem falnak ütközött-e. A visszatérési érték legyen az ütközés iránya, ami
  - balról, ha az elem bal széle kisebb vagy egyenlő mint 0.
  - jobbról, ha az elem jobb széle nagyobb vagy egyenlő mint a kapott szélesség.
  - felülről, ha az elem teteje kisebb vagy egyenlő 0.

- alulról, ha az elem alja nagyobb vagy egyenlő mint a kapott magasság.
- **IsCollisionToOther(Element element)**: Ellenőrzi, hogy az elem egy másik elemnek ütközött-e. A visszatérési érték legyen az ütközés iránya, ami
  - balról, ha az elem alja nagyobb vagy egyenlő mint a másik elem teteje de az elem teteje kisebb mint a másik elem teteje (azaz nekiért, de még nem haladt át rajta), illetve a jobb széle nagyobb egyenlő mint a másik elem bal széle, de a bal széle kisebb egyenlő mint a másik elem fele.
  - jobbról, ha az elem alja nagyobb vagy egyenlő mint a másik elem teteje de az elem teteje kisebb mint a másik elem teteje (azaz nekiért, de még nem haladt át rajta), illetve a jobb széle nagyobb egyenlő mint a másik elem fele, de a bal széle kisebb egyenlő mint a másik elem jobb széle.
- **IsBelow(Element element)**: Visszaadja, hogy az adott elem a másik elem alá került-e, azaz a teteje nagyobb vagy egyenlő mint a másik elem alja.

## 1.2 PingPongModel

A **PingPongModel** osztály fogja tartalmazni a játéklogikát. Az inicializáláshoz a pálya méretének, illetve a labda és ütő pozíciójának és méretének megadására van szükségünk.

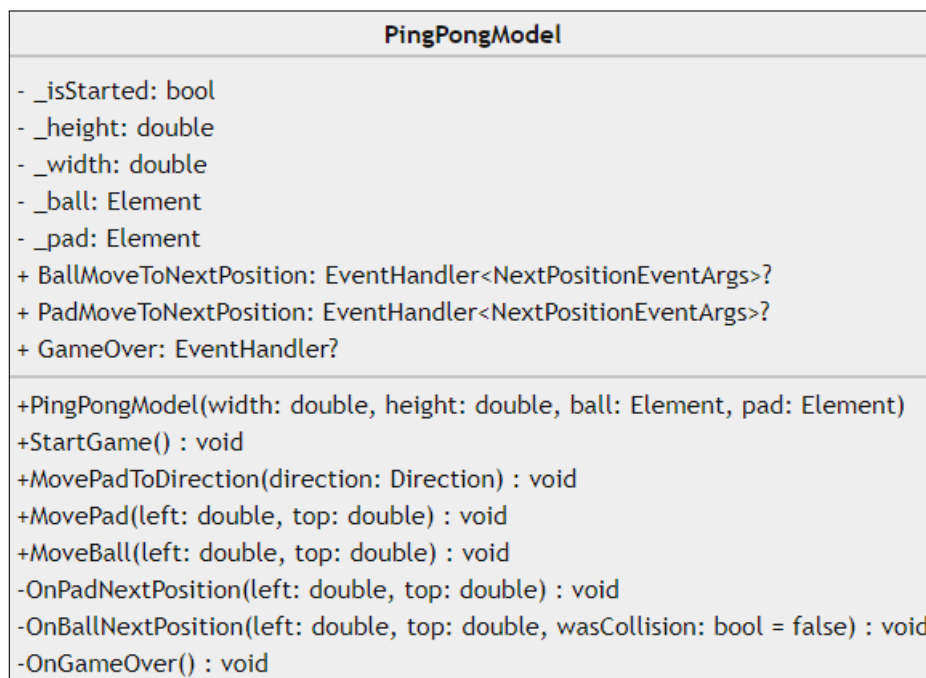


Figure 3: A **PingPongModel** osztálydiagramja

A játék elindítására a **StartGame()** függvény meghívásával van lehetőség. A labdát “indítsuk el” egy új célpozíció megadásával. Az új célpozíció legyen a játékpálya teteje vagy alja, jobb vagy bal irányba. A **BallMoveToNextPosition** event kiváltásával jelezzük, hogy egy új célpozíció került beállításra.

Hozzuk létre az alábbi függvényeket a játék vezérlésére:

- **MovePadToDirection(Direction direction)**: Az ütő új célpozícióját állítsuk be a jelenlegi helyzet + 100 pixelre. Amennyiben az új pozíció kívül esne a játéktáblán, úgy a pálya szélére állítsuk azt be. A **PadMoveToNextPosition** event kiváltásával jelezzük, hogy egy új célpozíció került beállításra. Eseményparaméterként adjuk át az új pozíciót.
- **MovePad(double left, double top)**: Állítsuk be az ütő pozícióját a kapott pozícióra.
- **MoveBall(double left, double top)**: Állítsuk be a labda pozícióját a kapott pozícióra. Vizsgáljuk meg, hogy a labda nem érte-e el valamelyik falat, vagy pedig az ütőt. Ehhez használjuk az **Element** osztály **IsCollisionToWall** illetve **IsCollisionToOther** függvényeit. Amennyiben a labda falat vagy ütőt ért, úgy a következő célpozíciót állítsuk be az ellenkező irányba és jelezzük

ezt a `BallMoveToNextPosition` event kiváltásával. Amennyiben a labda túlhaladt az ütőn (ezt az `IsBelow` függvény segítségével ellenőrizhetjük) úgy a játék véget ért. Ezt a `GameOver` event kiváltásával jelezhetjük.

Mindhárom eljárás elején ellenőrziük, hogy a játék tart-e még, és csak ebben az esetben végezzük el a leírt tevékenységeket.

## 2 Nézetmodell

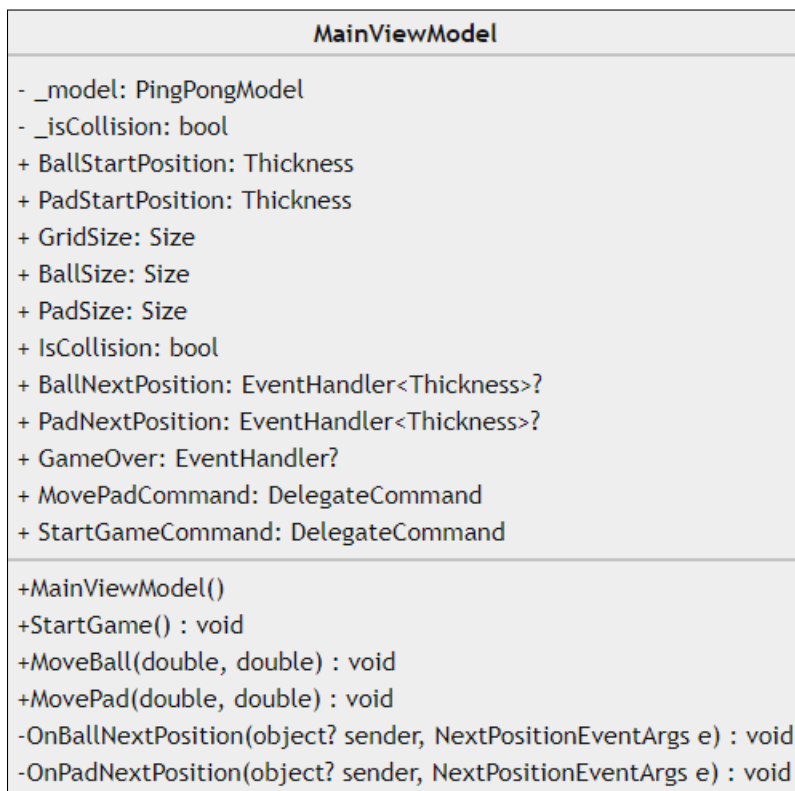


Figure 4: A `MainViewModel` osztálydiagramja

A nézetmodell konstruktorában állítsuk be:

- A játéktábla méretét (`GridSize`) 800\*400-ra
- A labda méretét (`BallSize`) 40\*40-re
- Az ütő méretét (`PadSize`) 120\*10-re
- A labda kezdőpozícióját (`BallStartPosition`) horizontálisan és vertikálisan is a képernyő közepére
- Az ütő kezdőpozícióját (`PadStartPosition`) horizontálisan a képernyő közepére, vertikálisan pedig a képernyő aljától 100 pixelre

Ezen kívül példányosítsunk egy új model objektumot, a játékpálya méretének, illetve a labda és ütő méretének és kezdőpozíciójának átadásával.

Készítsünk egy `StartGameCommand` command-ot, amely meghívja a model `StartGame()` metódusát, ezzel elindítva a játékot.

Készítsünk egy `MovePadCommand` commandot is, amely segítségével az ütőt fogjuk majd mozgatni. Feltételezzük, hogy paraméterként egy jobb vagy bal irányt kapunk. Ennek segítségével hívjuk meg a model `MovePadToDirection(direction)` függvényét.

A nézetmodellben iratkozzunk fel a modell `PadMoveToNextPosition` és `BallMoveToNextPosition` eseményeire. A kapott pozícióból hozzunk létre egy új `Thickness` példányt, majd ezt felhasználva invokáljuk a megfelelő `PadNextPosition` vagy `BallNextPosition` eseményeket.

Hozzunk létre továbbá az alábbi publikus függvényeket:

- `MoveBall(double left, double top)`: A modell megegyező függvényének meghívása a labda mozgatására.
- `MovePad(double left, double top)`: A modell megegyező függvényének meghívása az ütő mozgatására.

### 3 Nézet (deklaratív leírás XAML-ben)

A nézethez adjunk hozzá egy:

- **Grid** panelt, amelynek háttere legyen fehér, szélessége és magassága pedig legyen kötve a nézetmodellben szereplő **GridSize** tulajdonsághoz. A kötés iránya legyen **OneTime**.
- **Ellipse** controlt amelynek színe legyen kék, körvonala fekete, horizontálisan igazítsuk balra, vertikálisan pedig fel. A **Margin** adattagot kössük a nézetmodell **BallStartPosition** tulajdonságához. A kötés iránya legyen **OneTime**. A szélessége és magassága pedig legyen kötve a nézetmodellben szereplő **BallSize** tulajdonsághoz. A kötés iránya legyen szintén **OneTime**. A controlnak állítsunk be egy tetszőleges nevet is.
- **Rectangle** controlt amelynek színe és körvonala legyen fekete, horizontálisan igazítsuk balra, vertikálisan pedig fel. A **Margin** adattagot kössük a nézetmodell **PadStartPosition** tulajdonságához. A kötés iránya legyen **OneTime**. A szélessége és magassága pedig legyen kötve a nézetmodellben szereplő **PadSize** tulajdonsághoz. A kötés iránya legyen szintén **OneTime**. A controlnak állítsunk be egy tetszőleges nevet is.

A nézetben iratkozzunk fel az 'S', illetve és jobb és bal nyíl billentyűk lenyomására. Előbbi fogja a **StartCommand** parancsot kiváltani és ezáltal új játékot indítani, míg utóbbi kettő a **MovePadCommand** kiváltásával az ütő mozgatásáért lesz felelős.

```
<Window.InputBindings>
  <KeyBinding Command="{Binding MovePadCommand}"
    CommandParameter="Right" Key="Right" />
  <KeyBinding Command="{Binding MovePadCommand}"
    CommandParameter="Left" Key="Left"/>
  <KeyBinding Command="{Binding StartGameCommand}" Key="Space" />
</Window.InputBindings>
```

### 4 Nézet (*code behind*)

MVVM architektúrában a *code behind* osztályba (a nézetekhez tartozó *.xaml.cs* állomány) csak olyan logikák kerülhetnek, amelyek kizárólag a nézet manipulálását szolgálják. Ilyenek az animációk is, ezért ezeket a könnyebb programozhatóság érdekében helyezzük a *code behind* osztályba.

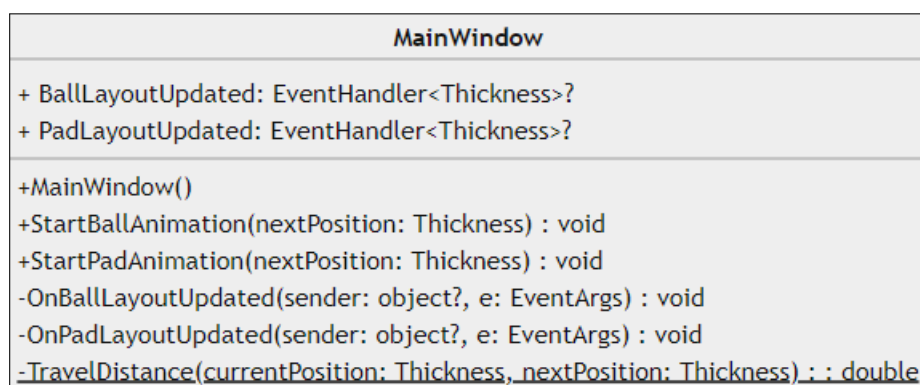


Figure 5: A MainWindow osztálydiagramja

A `StartBallAnimation(Thickness nextPosition)` függvény lesz felelős azért, hogy a labdát a jelenlegi pozíciójából animáltan elmozgassa a kapott új pozícióba. Ehhez hozzuk létre az alábbi animációt:

```
ThicknessAnimation animation = new ThicknessAnimation
{
    From = Ball.Margin,
    To = nextPosition,
    Duration = new Duration(TimeSpan.FromMilliseconds(5)),
    SpeedRatio = 1 / TravelDistance(Ball.Margin, nextPosition)
    // Speed depends on the distance to travel
};

Ball.BeginAnimation(Ellipse.MarginProperty, animation,
    HandoffBehavior.SnapshotAndReplace);
```

A `SpeedRatio` segítségével módosíthatjuk az idő relatív múlását az adott animáció tekintetében. Erre azért van szükség, hogy a labda a különböző távolságokat ugyanolyan sebességgel tegye meg. Ehhez határozzuk meg a két pozíció közötti távolságot a `TravelDistance(Thickness currentPosition, Thickness nextPosition)` függvény segítségével.

A `BeginAnimation` metódus a következő paramétereket várja:

- melyik propertyt kell beállítani (`Ellipse.MarginProperty`),
- mire (`animation`),
- mi történjen, ha még a befejezés előtt újra meghívásra kerül ez a metódus (azt szeretnénk, ha megszakítaná az animáció kirajzolását, és újat kezdene: `HandoffBehavior.SnapshotAndReplace`).

A `StartPadAnimation(Thickness nextPosition)` függvény lesz felelős azért, hogy az ütőt a jelenlegi pozíciójából animáltan elmozgassa a kapott új pozícióba. Ehhez hozzuk létre az alábbi animációt:

```
ThicknessAnimation animation = new ThicknessAnimation
{
    From = Pad.Margin,
    To = nextPosition,
    Duration = new Duration(TimeSpan.FromMilliseconds(100)),
};

Pad.BeginAnimation(Rectangle.MarginProperty, animation,
    HandoffBehavior.SnapshotAndReplace);
```

Az osztályban iratkozzunk fel az `Ellipse`, illetve `Rectangle` controlunk `LayoutUpdated` eseményére egy-egy eseménykezelővel. Ez az esemény akkor váltódik ki, amikor az adott control layout-ja (pl. margin) megváltozik. Esetünkben ahogy halad előre az animáció, ez az esemény többször kiváltódik. Ennek segítségével tudjuk frissíteni az adott elem haladását és ellenőrizni, hogy időközben a labda az ütőnek vagy pedig a falnak ütközött-e.

Ehhez a megfelelő eseménykezelőben váltsuk ki a `BallLayoutUpdated` illetve `PadLayoutUpdated` eseményeket, eseményparaméterben pedig adjuk át a megfelelő control `Margin` adattagjának értékét. Ez fogja tartalmazni az aktuális pozíciót is.

## 5 Alkalmazás környezeti réteg

Az `App` osztályunkat fogjuk arra használni, hogy a nézet és nézetmodell eseményeit összekössük. Ehhez példányosítsunk egy `MainWindow` és `MainViewModel` példányt. A nézetmodellt állítsuk be a nézet objektum `DataContext`-jének.

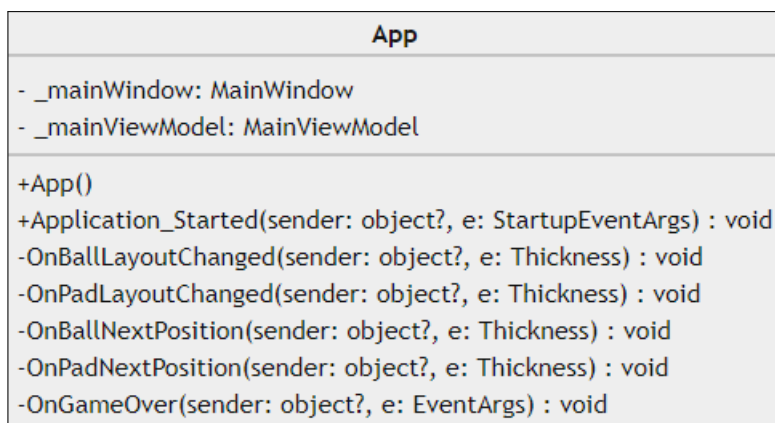


Figure 6: Az App osztálydiagramja

Iratkozzunk fel a nézetmodell megfelelő eseményeire az alábbi logikákkal:

- **BallNextPosition:** Az eseményparaméterben kapott pozícióval hívjuk meg a nézet **StartBallAnimation** függvényét, elindítva ezzel a labda mozgását az új pozíció felé.
- **PadNextPosition:** Az eseményparaméterben kapott pozícióval hívjuk meg a nézet **StartPadAnimation** függvényét, elindítva ezzel az ütő mozgását az új pozíció felé.
- **GameOver:** A játék végét jelezzük a felhasználó felé egy popup feldobásával. Az OK gomb megnyomására az ablak záródjon be.

Iratkozzunk fel a nézet megfelelő eseményeire az alábbi logikákkal:

- **BallLayoutUpdate:** Meghívja a nézetmodell **MoveBall** függvényét, frissítve ezzel a labda helyzetét a kapott pozícióra
- **PadLayoutUpdate:** Meghívja a nézetmodell **MovePad** függvényét, frissítve ezzel a labda helyzetét a kapott pozícióra

## 6 A játékpálya háttérének frissítése <sup>OP</sup>

Nemcsak egy-egy elem mozgását, hanem a háttér változását is lehetőségünk van animálni.

Hozzunk létre egy animációt, amely a játékpálya háttérét pirosra változtatja egy rövid időre abban az esetben, ha a labda az ütőnek ütközik. Ehhez egészítsük ki a **Grid** elemünket az alábbival:

```
<Grid.Style>
  <Style>
    <Style.Triggers>
      <DataTrigger Binding="{Binding IsCollision}" Value="True">
        <DataTrigger.EnterActions>
          <BeginStoryboard>
            <Storyboard>
              <ColorAnimation
                Storyboard.TargetProperty="(Grid.Background).(SolidColorBrush.Color)"
                From="White" To="Red"
                FillBehavior="Stop" AutoReverse="True" Duration="0:0:0.5" />
            </Storyboard>
          </BeginStoryboard>
        </DataTrigger.EnterActions>
      </DataTrigger>
    </Style.Triggers>
  </Style>
</Grid.Style>
```

A háttér változását egy **DataTrigger** segítségével fogjuk elérni, amely akkor fogja az animációt elindítani, ha ütközés történt.

Az ütközés jelzésére adjunk hozzá egy `IsCollision` bool tulajdonságot a nézetmodell osztályunkhoz. A modell osztályunk `BallMovedToNextPosition` esemény eseményargumentumait egészítsük ki úgy, hogy az új pozíció mellett egy `bool` adattagban átadja azt az információt is, hogy történt-e az ütővel ütközés vagy nem.

Amennyiben igen, úgy a nézetmodell megfelelő eseménykezelőjében állítsuk az `IsCollision` adattag értékét igazra, majd amikor a labda elmozdult a célpozíció felé, állítsuk vissza hamisra.

A `DataTrigger` az `IsCollision` igazra állítása esetén fog aktiválódni, meghívva ezzel a beállított `ColorAnimation` animációt. Az `AutoReverse="True"` beállításnak köszönhetően miután a háttér pirosra váltott, automatikusan vissza is vált fehérre.