**ENERGETIKAI TECHNIKUM ÉS KOLLÉGIUM**

**TET-RISZ**

**2023.**

Konzulens: Készítette:

Tóth ZoltánTarczal Dániel János

Nyilatkozat

Alulírott Tarczal Dániel János büntetőjogi felelősségem teljes tudatában nyilatkozom arról, hogy az itt szereplő záródolgozat saját, önálló munkám eredménye és sem részeiben sem egészében nem került még kereskedelmi forgalomba, ill. publikálásra, a GPL licenszelésű programrészek kivételével.

Paks, 2023. április 14.

Tarczal Dániel János

**Tartalomjegyzék**

[I. Bevezetés 4](#_Toc1)

[II. Témaválasztás indoklása 4](#_Toc2)

[III. Fejlesztői dokumentáció 5](#_Toc3)

[1. Specifikáció 5](#_Toc4)

[2. Adatmodell leírása 6](#_Toc5)

[3. Algoritmusok leírása 10](#_Toc6)

[4. Egységtesztelés 15](#_Toc7)

#### **Bevezetés**

Szakdolgozatom témájának alapját a híres Tetris nevű játék adja, amit eredetileg az orosz Alexej Pazsitnov fejlesztett ki a szovjet Elektronika 60 nevezetű számítógépre 1984-ben. Azóta rengeteg iteráción ment keresztül a szórakoztató szoftver, és az egyik legeladottabb játékká nőtte ki magát. Ez a játék hozzám először egy régi mobiltelefon egyik előre telepített játékaként jutott el, és nagyon megtetszett, ahogy az egyszerűségének ellenében milyen kreatív, és akár nehéz is tud lenni, mely tulajdonságai miatt szinte bárkinek élményt tud okozni a program használata.

Az ötlet a játékkal kapcsolatban egy iskolai projektmunka során fogalmazódott meg bennem, ahol osztálytársammal, Bene Dániellel a játékmotor, Godot Engine felhasználásával, C# nyelven írtunk meg a Google Chrome böngésző offline módjának dinós játékához hasonló videojátékot. Megtetszett a motor és ebben kívánkoztam megírni a szakdolgozatomat, de mivel kezdetben nem sokat tudtam a játékfejlesztésről és a Godot-ról sem, nagyon sok információt magamtól kellett összegyűjtenem, a fejlesztés menetéről, illetve a játékmotor képességeiről, funkcióiról. A játékmotor az úgynevezett belső mag egy videojátékban, ami valós idejű grafikával rendelkezik, ezért megkönnyíti a fejlesztést és akár több platformra is fejleszthetjük a játékot.

A játékban a Tetris játék adottságain túl próbáltam hozzáadni témázhatóságot, illetve ranglistákat a többjátékos-szerű élmény érdekében.

#### **Témaválasztás indoklása**

Választásom azért erre a projektre esett, mert az egyszerű videojátékok széles körben érdekeltek az emberek között, ráadásul ez a játék ezek közt is kiemelten közkedvelt. Célom a bevált ötlet újra gondolása, és átültetése egy modern, platformfüggetlen környezetbe. Linux felhasználóként fontosnak tartom, hogy a meglevő technológiával tegyük elérhetővé a lehető legtöbb platformra szoftverünket, hogy minél több ember számára elérhetőek legyenek azok. A választásom fő okozója viszont az volt, hogy C#-ban szerettem volna programot írni webalkalmazás helyett.

#### **Fejlesztői dokumentáció**

###### **Specifikáció**

Felhasználói követelmények

* Könnyen áttekinthető játéktér
* A tetromínó (alakzat) építőkockáinak könnyű megkülönböztetése, akár színekkel
* A soron következő tetromínó megjelenítése
* Pontok kiírása
* Mozgás, forgatás, helyes működés, megfelelő teljesítmény
* Nehézségi fokozat választása
* Irányítás módosítása
* Játék megállítása

Fejlesztői környezet, fejlesztői függőségek

A Godot motor fejlesztői környezet főként a GUI részében volt mérvadó, az üzleti logikák nagy részében csak vanilla **.NET**-et használtam. A Godot 3.5.1-es verziójával kezdtem a munkámat, ami nem a .NET Core-ra épül, hanem a régebbi .NET Framework 4.7.2-őn alapuló, nyílt forráskódú és platformfüggetlen Mono keretrendszerre, viszont idő közben megjelent a rég várt **Godot 4.0-**ás verziója, ami többek között már a **.NET Core 6.0** keretrendszeren alapul, ami újabb, jobban támogatott, illetve gyorsabb mint elődje, így idő közben átváltottam erre. A motor letöltése az alábbi [linken érhető el](https://godotengine.org/download) a .NET-es gombra kattintva az oldalon. A C# kód megfelelő futásához továbbiakban szükséges a [.NET SDK](https://dotnet.microsoft.com/download) 6-os verziója is.

A forráskód zöme **Visual Studio 2022**-vel, illetve **Visual Studio Code**-dal íródott, a felhasználói felület pedig a **Godot IDE**-vel.

A Godot IDE-t, illetve a Godot motort több szempontból is összehasonlíthatjuk egy általunk korábbról is ismert grafikus könyvtárhoz, a WinForms-hoz és az ezt szerkesztő Visual Studio-hoz. A WinForms-ban létrehozott különböző objektumoknak, például cimkéknek, vagy szövegdobozoknak különböző tulajdonságaik, illetve eseményeik vannak. Ezek általában statikusak, de dinamikusan is létrehozhatjuk őket kód segítségével. A Godot is lényegében ezeket az objektumakat kínálja, amiken felül teljesítmény optimalizációkat követ el.

A forráskódot a **Git** verziókövető rendszerrel követtem és rögzítettem, amit az [alábbi linken](https://git-scm.com/download/win) keresztül érhetünk el, ha a forráskód történetét nem Visual Studio-ban kívánjuk böngészni

A dokumentációt nagyrészt **Onlyoffice**-ban írtam, ami egy nyílt forráskódú office csomag, ami nagy kompatibilitással bír a Microsoft által használt Open XML formátumokhoz.

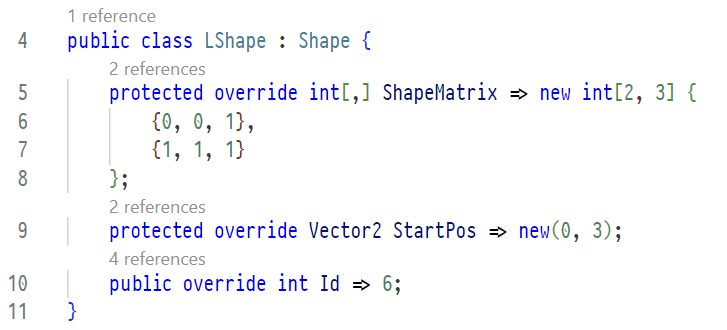
Néhány képet a GNU Image Manipulation Programban (azaz a **GIMP**-ben) készítettem el, ami szintén egy nyílt forráskódú szoftver

A fejlesztői környezet rendszerkövetelményei közé tartozik 4 GB RAM, egy 15 éves Intel Core 2 Duo és egy **Vulkan 1.0** kompatibilis grafikus meghajtó, amit átválthatunk **OpenGL 3.3**-ra is, hogyha nem rendelkezünk ilyen eszközzel, szóval elég alacsonyok a futási igényei.

A tesztelést **Arch Linux**-on, **Windows 10 Enterprise 22H2**-őn, illetve **Windows 11 Pro 22H2**-őn folytattam.

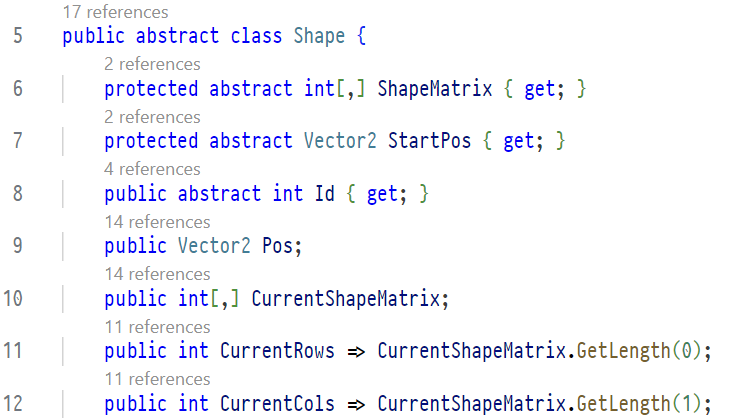
###### Adatmodell leírása

A programot objektum orientált módon próbáltam lemodellezni, ennek következtében a projekt 17 osztályt tartalmaz, amiből nyolcat le lehet egyszerűsíteni egyre a felhasznált absztrakció miatt, így négy üzleti logika, illetve öt GUI osztály.

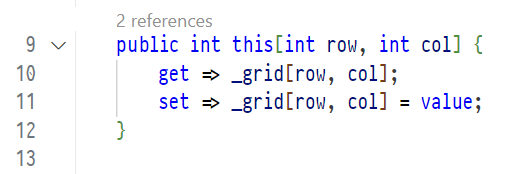
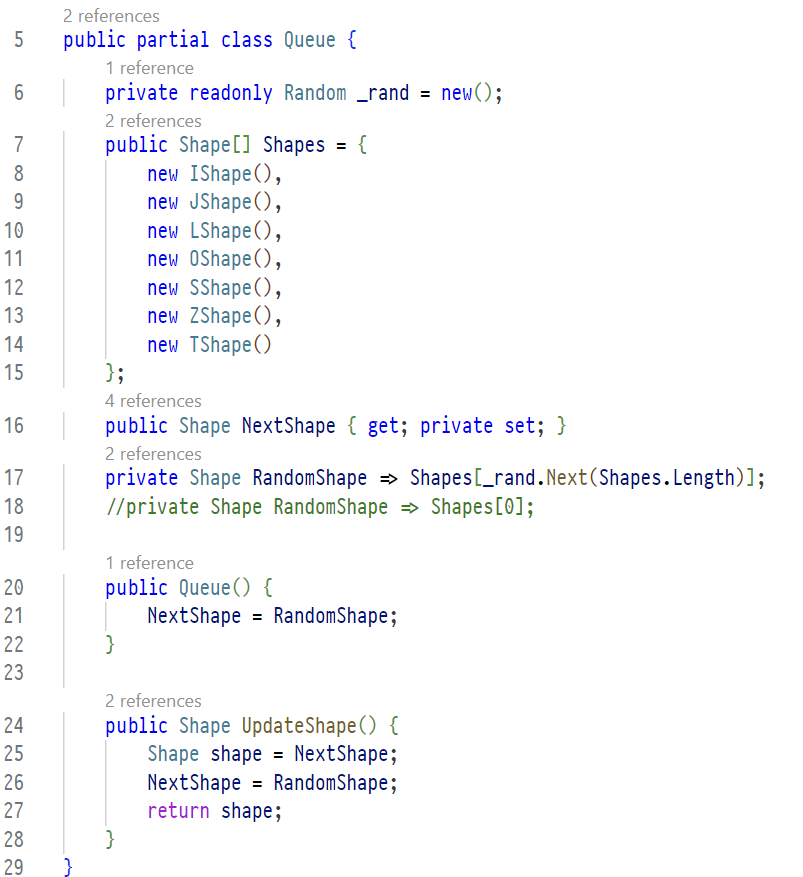


A projekt alapját a különböző *Shape* osztályok adják amikben a tetrominók építőkockáinak elhelyezkedését, kezdőpozízióját, illetve egy megkülönböztetésre szolgáló id-t írnak le. Ezen alosztályok a *Shape* absztrakt osztály elemeinek adnak konkrét értékeket.

A *Grid* nevű osztály az elhelyezett tetrominók blokkjainak tárolásáért, és ellenőrzéséért felel.



Az osztályon belüli grid mátrixot egy úgynevezett indexerrel tettem könnyen elérhetővé más osztályok számára.



A *Queue* osztályban történik a soron következő tetrominók kiválasztása. A *RandomShape* tulajdonság egy random változó segítségével kiválasztja a *Shapes* tömbből a következő alakzatot, és azt példányosítja.



A fő osztálynak a *Manager* osztályt lehetne nevezni, amely felelős a magas szintű irányításokért, illetve az üzleti logikai műveletek összehangolásáért, amik alacsonyabb szintű megvalósításáért más osztályok felelnek. Főbb elemei a következő tetromino kiválasztása, annak pozíciójának ellenőrzése, forgatása, mozgatása, illetve a lehelyezése.

A *GameField* osztály tartalmazza a játékmenet megejelenítéséhez szükséges objektumokat, a tetromínók kirajzolását, illetve a menük felső kezelését.



A *ControlMenu* az irányítások módosításáér felel a *PopMenu* osztály segítségével.

A *PauseMenu* osztály a játék megállítására hivatott.

* IShape
* JShape
* LShape
* OShape
* SShape
* ZShape
* TShape

Queue

Shape

Grid

Manager

GameField

PauseMenu

Menu

ControlMenu

PopMenu

###### **Algoritmusok leírása**

A játék az objektum orientált mivolta miatt számos osztályt és azokon belül különböző algoritmusokat tartalmaz.

A *Shape* osztály a tetrominók mozgásának leírásáért felel. Ebben az osztályban találhatóak az aktuálisan aktív alakzatmátrixra vonatkozó adatok.



A forgatást végrehajtó *Rotate* metódust kissé rendhagyóbb módon, dinamikusan hozom létre egy általánosabbnak mondható forgásmátrix helyett, amelyben megtalálhatóak lennének a soron következő forgatás blokkpozíciói. Ebben az algoritmusban létre először létre kell hoznom egy új alakzatmátrixot az előző mátrix sorainak és oszlopainak megcserélésével, tehát egy 2 sorból és 3 oszlopból álló mátrixot 3 sorossá és 2 oszlopossá alakítom át. Következik a feltöltés, ami egy általánosnak nevezhető dupla for-ciklus hajt meg. A feltöltés a forgatni kívánt alakzat bal-alsó sarkával kezd, és oszloponként helyezi be a 0, vagy 1 értékeket a létrehozandó alakzat bal-felső sarkától soronként. Ennek a függvénynek létrehoztam a fordítottját is a könnyebb ütközéskezelés érdekében. Itt az algoritmus a jobb-alsó sarokból helyezi át a bal-felső sarkokba az megfelelő értékeket.

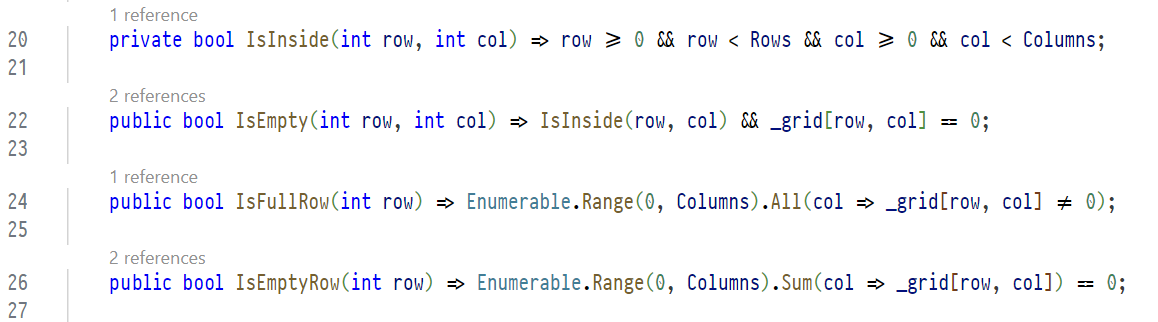


Az alakzatot mozgató *Move* függvény a paraméterként megadott sorral illetve oszloppal változtatja meg a *pos* nevű Vector2 típus értékét. A Vector2 egy Godot specifikus adattípus, ami lényegében egy számpárt hivatott eltárolni, általában pozíciót.

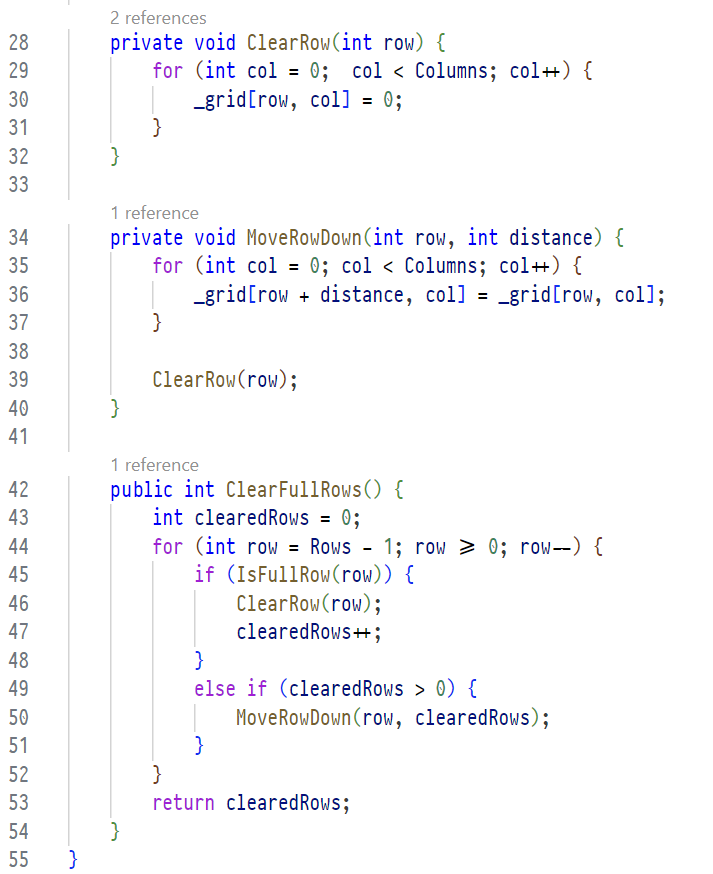
A tetrominók letétele után szükség van ezt az osztályt “visszállítani”, hogy a soron következő alakzat a megfelelő helyről indulhasson el.



Az *IsEmpty* függvény az *IsInside* hibamentesítő függvény segítségével ellenőrzi az adott sor és oszlop blokkjának ürességét.



Az *IsFullRow*, illetve az *IsEmptyRow* függvény az egész sort veszi figyelembe, amit egy Enumerable-lel és egy Linq kifejezéssel egy sorba le tudtam rövidíteni.



A *ClearFullRows* a *ClearRow* és a *MoveRowDown* függvény igénybevételével tisztítja ki a teli sorokat a játék szabályainak megfelelően. A függvényben megtalálható *clearedRows* változónak két funkciója is van. Minden egyes sort ellenőrzi a sor telítettségét, igaz eredménynél kitisztítja a for-ciklus *row* által meghatározott sorát, majd hozzáad egyet a *clearedRows* változóhoz. Ezen változó alapján tudja az algoritmus később a feljebbi sorokat lejjebb hozni megfelelően. Másik funkciója a változónak a pontszámítás, ugyanis a pontokat a játékos a kitisztított sorok után kapja meg.

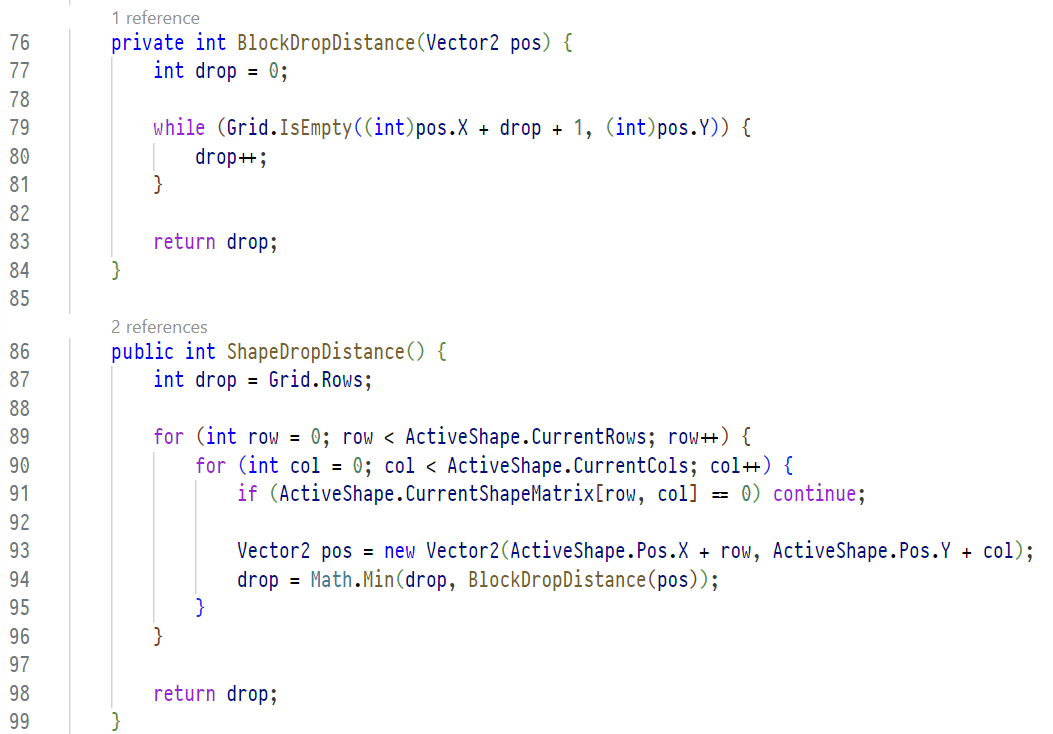


A szabályos mozgásért az *IsLegal* bool típusú függvény felel, ami az alakzatmátrix és annak pozíciójának az összegével a grid osztály *IsEmpty* metódusával határozza meg hogy az adott pozícióban található-e már lehelyezett kocka.

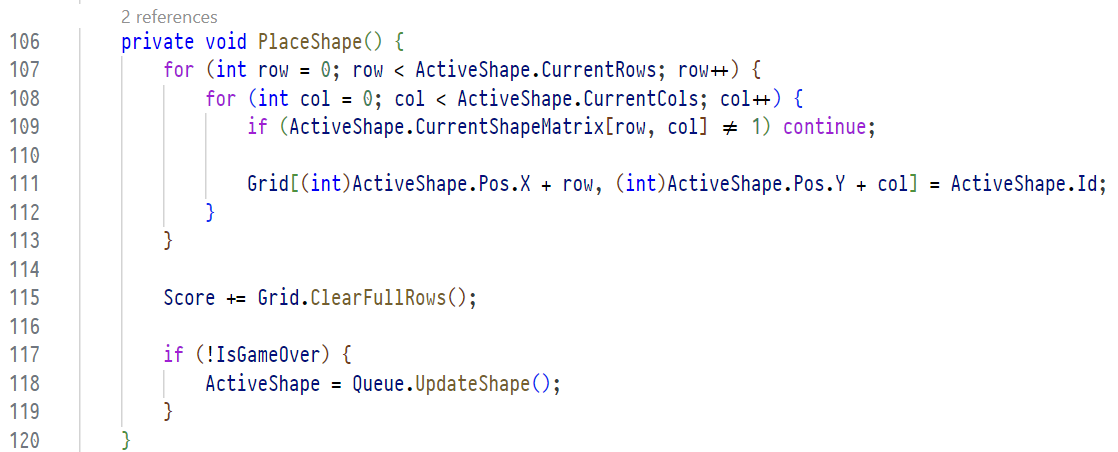
A mozgatásér felelős függvények először elmozgatják az alakzatot, és ha annak az új pozíciója nem szabályos helyen van, akkor visszateszi az alakzatot az eredeti pozíciójába.



A Tet-Risz-ben lehetőség van a tetrominó ledobására is. Ehhez meg kell határozni a lehető legnagyobb dobási távolságot a *BlockDropDistance* és a  *ShapeDropDistance* föggvények segítségével.

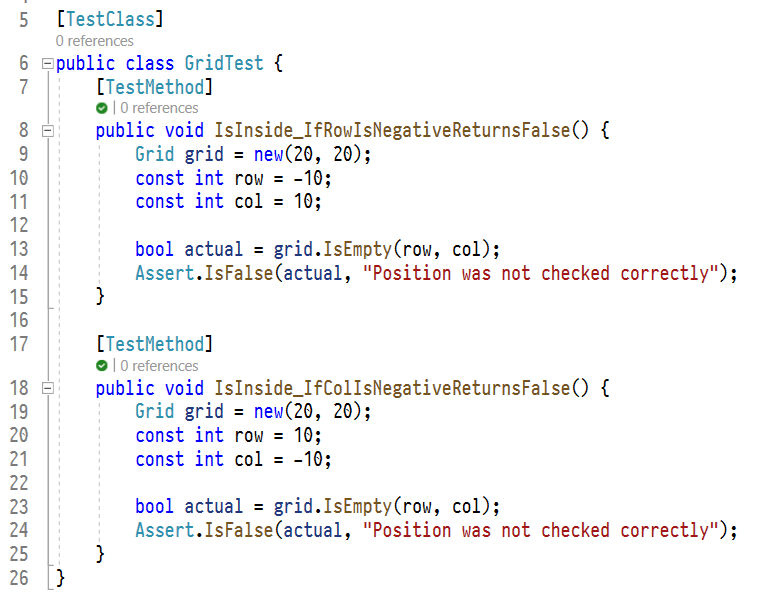


Amikor az alakzatunk a játéktér aljára érkezik, el kell azt helyeznünk a grid mátrixán belül, ami a *PlaceShape* függvényben történik meg. Itt növekszik meg esetlegesen a pontszámunk, illetve itt kerül kiválasztásra a soron következő tetrominó.



###### Egységtesztelés

A tesztelések végrehajtásához az MSTest framework-öt használtam fel, ami az alapértelmezett a Visual Studio 2022-ben.



A teszteseteket a *Grid* osztályban, leginkább az azon belüli függvények könnyebb tesztelhetőségeiért. Mindkét esetben az *IsEmpty* függvényt hívom le, viszont közvetve az *IsInside*  függvényt tesztelem, ami annak a privát függvénye. Ezeken azt tesztelem, hogyan viselkedik a függvény negatív értékek esetén. A tesztek eredményei helyes működésre utalnak.

