Valósidejű C# grafika a böngészőben

Szabó Dávid[1], Dr. habil. Illés Zoltán[2]

{ [1]sasasoft, [2]illes }@inf.elte.hu
ELTE IK

Absztrakt. A legfrissebb .NET Core 3.0 futtatókörnyezet felhasználásával már nem csak webalkalmazásaink szerver oldalát, de akár a böngészőben végrehajtandó kliens oldalát is fejleszthetjük C# nyelven. Felhasználhatjuk a .NET világában elérhető könyvtárakat és névtereket, továbbá a Javascript funkcionalitáshoz is hozzáférhetünk. Kutatásaimban böngészőben futó valósidejű grafikai alkalmazásokkal vizsgálom az új technológia lehetőségeit.

Kulcsszavak: .NET Core, C#, Blazor, valós idő, grafika

1. Bevezetés

Webszerverek és webszolgáltatások fejlesztésében a mai napig jelentős szereplő az ASP .NET keretrendszer. A funkcionalitásban és könyvtárakban gazdag .NET világ és C# nyelv kombinációjával egyszerűen készíthetünk statikus, vagy komplexebb dinamikus weboldalakat is. Az ASP keretrendszer rengeteg konfigurációs terhet levesz a vállunkról. Átlátható navigációt és átirányítási lehetőségeket biztosít, gyorsan implementálhatunk authentikációt és adatbázis támogatást, illetve az egyszerű IT biztonsági eszközök támogatásával és beállításával segíti a szolgáltatásaink biztonságossá tételét. A Razor szintaxis [1] segítségével C# logikát ágyazhatunk a HTML lapokba, melyet a szerver a lap előállításakor hajt végre. Dinamikusan módosíthatjuk a HTML lapunk tartalmát, feltételeket és ciklusokat használhatunk HTML elemek létrehozásához és paraméterezéséhez. A több lapos weboldalakat és webszervereket az MVC architektúra segítségével szervezhetjük egy rendezett projektté, mely megkönnyíti kapcsolatok kiépítését a Razor HTML lapok és az alkalmazásunk üzleti modellje között.

Az elmúlt években a .NET és C# technológiák fő fejlesztési iránya a multi-platform eszközökké alakítás volt. Ennek az irányvonalnak a Microsoft által fejlesztett nyílt forráskódú .NET Core futtatókörnyezet [2] az egyik legjelentősebb eredménye, mellyel C# alkalmazásainkat eljuttathatjuk nem csak Windows, de Linux és MacOS környezetekre is (az Android és iOS támogatás is fejlesztés alatt áll). Tehát webalkalmazásaink már nem csak Windows rendszereken futhatnak. A további platformokon a hosszú múltra visszatekintő Mono futtatókörnyezetet használhatjuk, mely a mai modern operációs rendszerek mindegyikén képes futtatni C# alkalmazásainkat. A Mono nyílt forráskódú környezetet a Mono-Project vállalat fejleszti és céljuk, hogy nem csak asztali, de mobil és egyéb kompakt okos eszközökre is eljuttassa a C# nyelvet.

Láthatjuk, hogy az előbbi technológiák segítségével, célplatformtól függetlenül választhatjuk a C# nyelvet. Bármilyen mai platformra fejleszthetünk C#-ban… egyetlen kivétellel: Webalkalmazásainkat kiszolgáló szerverink készülhetnek C# nyelven, de a kliens oldalon, a felhasználó eszközén futó weboldalhoz tartozó kód kizárólag Javascript lehet. A webböngésző alkalmazások világában ugyanis a Javascript a sztenderd kliens oldali nyelv, melyet minden böngésző képes végrehajtani.

A legújabb .NET Core 3.0 keretrendszerrel és Blazor könyvtárral viszont ez megváltozott, mostantól a felhasználók eszközén a webböngészőben végrehajthatunk C#-ban készült alkalmazásokat!

2. C# a böngészőben

Korábban, ha a weboldalunkon dinamikus működést szeretnénk elérni, azaz miután a felhasználó eszközén betöltődött a lap további kódot és logikát szeretnénk végrehajtani a Javascript programozási nyelvet kellett használnunk. Ez az a nyelv melyet beágyazhatunk a HTML lapjainkba és minden webböngésző alkalmazás képes futtatni. Mindez igaz volt a 2017-ben bemutatott WebAssembly [3] megjelenéséig.

2.1 WebAssembly

A WebAssembly-t a World Wide Web Consortium együttműködésével a Mozilla, Microsoft, Google és Apple fejleszti. A WebAssembly (WASM) egy új típusú kód és programozási nyelv, melyet futtatni képesek a modern webböngészők. Ezt a nyelvet nem arra tervezték, hogy közvetlenül fejlesszenek benne alkalmazásokat. A WebAssembly nyelve inkább egy fordítási célnyelv más alacsony vagy akár magas szintű programozási nyelvek számára. A modern webböngészőkben dolgozó virtuális gép már nem csak Javascript kódot, hanem WASM kódot is képes kiszolgálni és végrehajtani. Tehát a WASM és Javascript párhuzamosan futhat a weblapunkon és a két kódbázis API-kon keresztül interaktálhat egymással. A WebAssembly nem a Javascript-et leváltó nyelvnek készül, hanem a lehetőségeinek kiegészítését segíti.

A WebAssembly célja, hogy a böngészőben futó alkalmazásaink egyes részeit Javascript-től különböző nyelven készíthessük el. Célszerű a WASM-hez fordulni, ha egy meglévő más nyelven íródott kódbázist szeretnénk felhasználni, vagy kifejezetten teljesítmény igényes feladatokat egy alacsonyabb szintű nyelven, szigorúbb memóriamodell mellett szeretnénk fejleszteni. Ezeket a modulokat írhatjuk C#, C/C++, vagy bármilyen nyelven, melyhez elérhető eszköz, amely WASM kódra fordítja a forrást. A lefordított WASM kódot pedig, mint egy könyvtárat felhasználhatjuk a weboldalunkon.

2.2 Blazor

A C# alkalmazásaink böngészőben futtatását is a WebAssembly nyelv teszi lehetővé további .NET technológiák kombinálásával, melyet összefoglaló néven Blazor-nek neveznek.

Az új .NET Core 3.0 keretrendszerben elérhető egy új projekttípus, a BlazorWASM projekt. A sablon egy .NET Core 3.0 projektet készít, melyben konfigurált egy ASP webszervert statikus fájlok kiszolgálásához. Egy webszervert kaptunk, melyet Windows, Linux és MacOS rendszereken is képesek vagyunk futtatni. A szerver jelenleg egyetlen index.html weboldalt szolgál ki, mely a BlazorWASM webalkalmazásunk. Az index.html által hivatkozott mono.wasm állományt is letöltik a kliens böngészők, mely a weboldalunkhoz tartozó a felhasználó eszközén futó C# kódunkat és a kiszolgálásához szükséges eszközöket tartalmazza. Kliens oldalon a Blazor a Mono futtatókörnyezetet [4] használja a C# kód kiszolgálásához és végrehajtásához. A Mono-Project implementálta a Mono futtatókörnyezetet a WebAssembly nyelvre. Tehát a böngészők virtuális gépében egy Mono futtatókörnyezetet dolgozik, melyben C# kódot hajthatunk végre.

Egy Blazor alkalmazás Komponensekből áll, melyeket a .razor kiterjesztésű állományok írnak le. Ezekkel a komponensekkel a HTML elemekhez hasonlóan fel tudjuk építeni a weboldalunk megjelenését. A sablon projekt tartalmaz komponenseket, például az App komponenst, mely a különböző kliens oldali Blazor lapok között tud navigálni. Míg a .NET Core szerver egy útvonalon egy Blazor webalkalmazást szolgál ki, addig ezen webalkalmazáson belül további al-lapok is lehetnek, melyeket az App komponens keres fel és jelenít meg az URL alapján. Ezek a Blazor lapok is komponensek, melyekben a @page kulcsszóval ellátva megadhatjuk, hogy milyen relatív URL címen szeretnénk elérni azt az adott lapot. Egy komponens nem csak egy lap lehet, akármilyen egyéb HTML elemeket vagy akár másik komponenseket is tartalmazhat. Ezzel szegmentálhatjuk a HTML lapjaink leírását különböző komponensek között.

A komponensekben a @ karakterrel C# kódot ágyazhatunk be, mely a kliens oldalon a felhasználó böngészőjében fog futni a Mono futtatókörnyezetben. A kód beágyazás a már korábbról

ismert Razor szintaxist követi. Az elágazásokba helyezett HTML tag-ek csak akkor kerülnek a megjelent HTML kódba, ha az elágazás ágába fut a végrehajtás. A ciklusok minden iterációja példányosítja a ciklusmagba helyezett HTML tag-eket. Akár egy teljes, csak C# kódot tartalmazó kódblokkot is létrehozhatunk @code blokk elhelyezésével. Változókat és azok értékeit is felhasználhatjuk a HTML előállításakor. A különbség a hagyományos ASP.NET Razor és a Blazor között, hogy míg az előbbi csak egyszer, a szerveren értékelődik ki, mikor a HTML lapot előállítjuk, addig a Blazor képes a megjelenített HTML állományt futás közben alterálni a felhasználó eszközén.

A háttérben minden komponenst egy ComponentBase osztályból származtatott osztállyá fog alakítani a fordító. Az ősosztályon keresztül hozzáférhetünk a komponens életciklusának metódusaihoz, melyekkel kódot futtathatunk a komponens inicializálásakor (OnInitialized) vagy a HTML leírásba behelyezéskor (OnAfterRender). A @code blokkban felülbírálhatjuk ezeket a függvényeket, illetve paramétereket és tulajdonságokat is vihetünk fel a komponensünkhöz, melyeket kívülről elérhetnek más komponensek és módosíthatják értéküket.

A Blazor projekt létrehozásakor ügyeljünk a felhasznált projektsablonra. Több Hosztolási módot is választhatunk a Blazor projektekhez. Ez a cikk első sorban a BlazorWASM hosztolási módot mutatja be, mely a kliens oldalon C# kódot futtat a Mono keretrendszer segítségével. A szerver oldali .NET Core alkalmazás kihagyható, de ebben az esetben egy saját webszerverre lesz szükségünk, mely képes a statikus Blazor fájlokat kiszolgálni a kliensek számára. Elérhető a BlazorServer mód, melyben a komponensekbe írt C# kód továbbra is a szerveren hajtódik végre és a hívások eredményét a SignalR valósidejű kommunikációs könyvtár segítségével juttatja el a szerver a klienshez. A komponensek programozása és Blazor felhasználása ugyanúgy történik mindkét projekt típusban, viszont a háttérben a webalkalmazás működése jelentősen különbözik.

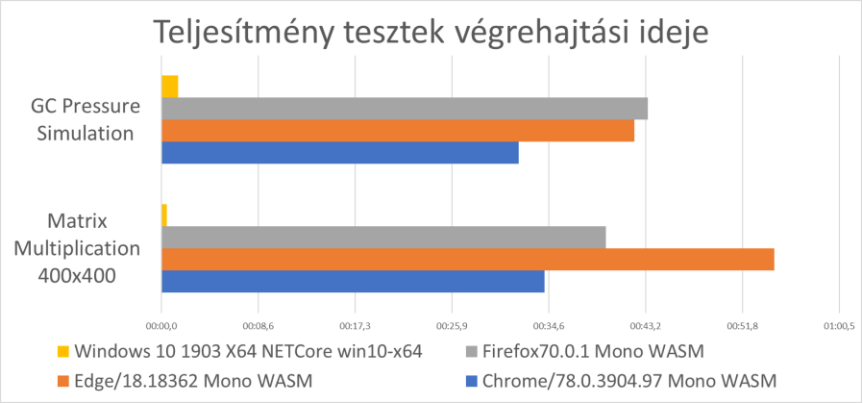
2.3 Javascript hívások

A webalkalmazások kliens oldali logikájában szükség lehet hozzáférni különböző Javascript API-khoz, ugyanis a webes világban a legtöbb webböngésző szolgáltatás és böngészőkben elérhető könyvtár Javascript API-kal kommunikál. Ahhoz, hogy ezeket az eszközöket használni tudjuk a Blazor rendszerben Javascript invokálást kell végrehajtanunk.

A C# kódból képesek vagyunk Javascript függvényeket meghívni, adatokat és paramétereket átadni ezeknek a függvényeknek, illetve a visszatérési értékület kiolvasni. Ehhez szükség van a komponensünkben a @inject IJSRuntime JSRuntime utasítással igényelni egy Javascript futtatókörnyezet implementációt. Ezen az implementáción keresztül tudunk a Javascript futtatókörnyezetben függvényeket hívni. A hívás során a Mono és Javascript virtuális gépek között adatkonverzió és szinkronizáció történik emiatt ezek a hívások jelentősen lassabbak, mint egy egyszerű függvényhívás. A hívások aszinkron függvények, ugyanis a Mono környezetben futó szál és a Javascript végrehajtási szála között szinkronizációra van szükség.

2.4 Teljesítmény

A kliens oldalon futó C# teljesítmény összehasonlítva a .NET Core 3.0 keretrendszer teljesítményével elég rossz eredményeket produkál. A tesztek során nagy méretű mátrixok szorzását, illetve a szemétgyűjtő terhelését végeztem string-ek és byte tömbök létrehozásával [5]. Jellemzően legalább 50x-es teljesítmény veszteséggel fut a böngészőben C# kód a .NET Core 3.0-hoz képest. Egyes tesztekben a különbség akár 100x-osra is növekedhet.



1. ábra: Teljesítmény tesztek futási idejének összehasonlítása

A mérésekhez használt számítógép konfiguráció: i7-2630QM, 16 Gb memória

3. WebGL

Kutatásaim során a .NET és C# nyelv lehetőségeit vizsgálom multi-platform valósidejű [6] grafikai megjelenítés témájában. A Blazor felhasználásával egy új platform és egyben új világ tárult ezen témakör opciói közé: a webböngészők és webalkalmazások világa.

Ha a web-en komplexebb valósidejű grafikai alkalmazásokat szeretnénk fejleszteni jellemzően a WebGL grafikus API vagy egy WebGL-re épülő könyvtár felhasználása közül választhatunk. A WebGL az OpenGL grafikus API mintájára készült grafikus processzorok alacsony szintű felhasználását lehetővé tévő interfész. Egy Javascript API-n keresztül hozzáférhetünk az OpenGL-ből ismert erőforrások webes változataihoz, adatokat küldhetünk a grafikus processzornak, shader programokat készíthetünk és futtathatunk, illetve ezek eredményeit megjeleníthetjük a böngészőben egy Canvas HTML elemben.

Az OpenGL-hez hasonlóan az állapotot a grafikus kontextus tárolja, illetve az állapotot módosító és rajzoló utasításokat a kontextuson értelmezzük. Ha WebGL-t szeretnénk megjelenítéshez használni, akkor először egy WebGL grafikus kontextusra van szükségünk, mely képes kezelni a grafikus erőforrásainkat és végrehajtani velük a rajzolási és megjelenítési parancsokat. OpenGL-ben egy grafikus kontextus egy végrehajtási szálhoz van kötve és ez határozza meg, hogy az adott szálon végrehajtott utasítások, melyik kontextusban hajtódnak végre. WebGL esetén az API biztosít egy WebGLContext objektumot és ezen osztályon értelmezett metódusokkal tudjuk a kontextus állapotát módosítani.

A kontextust a Canvas elemen keresztül tudjuk létrehozni. Az API automatikusan konfigurálja a megjelenítés platformfüggő részeit (a SwapChain erőforrást, melyet a WebGL el is rejt a fejlesztők elől).

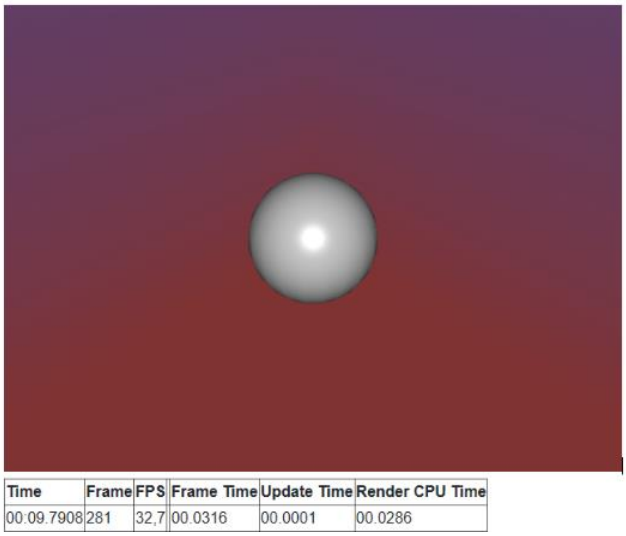
A Blazor segítségével a Canvas grafikák és WebGL elérhetővé vált a C# fejlesztők számára is. A BlazorExtensions/Canvas NuGet csomag felhasználásával egy átlátható C# API-t kapunk a HTML Canvas eleme köré, mellyel elérhetjük a Canvas rajzoló függvényeit és a WebGL grafikus API-t [7].

3.1 Blazor és WebGL

A BlazorCanvas könyvtáron keresztül a WebGL API első verzióját érhetjük el. Ez a legtöbb böngészőben támogatott, bár funkcionalitása felett eljárt az idő. A modernebb és gyorsabb grafikus eszközök eléréséhez használható WebGL 2 egyelőre nem támogatott a könyvtárban (ahogy minden webböngészőben sem).

Az API első látásra kényelmes WebGL osztályok és rajtuk értelmezett metódusok formájában tárja elénk a grafikus lehetőségeket. A WebGL Javascript erőforrásaihoz (Context, Buffer, Texture, stb.) kapunk C# wrapper osztályokat. A WebGLContext példányhoz hozzájuthatunk a Canvas komponenstől, mely inicializálja nekünk a grafikus kontextust minden a weboldal felületén megjelenítéshez szükséges funkciójával. A kontextus függvényei C# primitív típusokat, definiált enumerátorokat, illetve WebGL erőforrás osztálypéldányokat várnak paraméterben, így a legtöbb függvény kényelmesen és intuitívan használható.

Sajnos ez nem minden függvényre igaz. A buffer-ek feltöltése és uniform változók értékeinek megadása float, integer vagy egyéb primitív számtípusok tömbjeinek átadásával történik. Tehát, ha ezeket a változókat a System.Numerics névtér típusaiban tároljuk (Vector3, Matrix4x4 stb.) vagy akár komplexebb saját struktúrában [8], akkor minden WebGL hívásnál konvertálnunk kell az adatokat tömbökké, mely sűrűn fog szemétgyűjtést előidézni. Másik lehetőségünk, hogy kényelmetlenül tömbök formájában kell kezeljük az adatainkat. A WebGL Javascript API is tömbök formájában várja ezeket a paramétereket, ezért ez a konverzió elkerülhetetlennek tűnik.



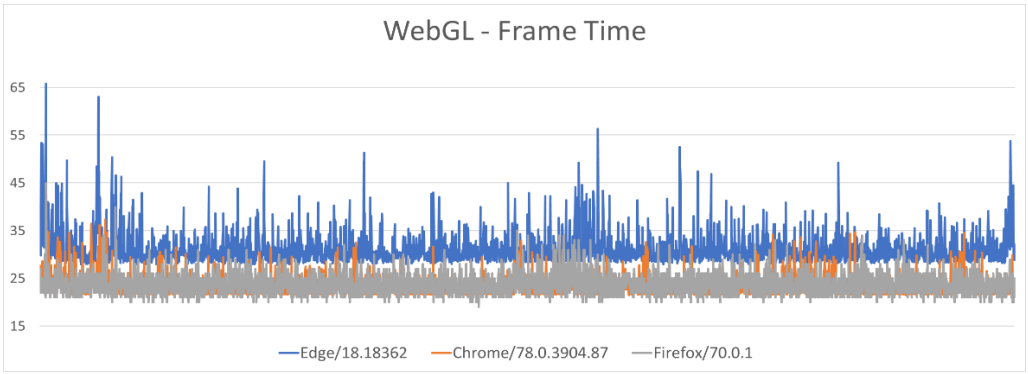
2. ábra: Árnyalt gömb és procedurális Skybox Blazor WebGL-ben

A .NET Standard közös kódbázist támogató lehetőségeinek köszönhetően a korábbi OpenGL grafikai alkalmazásaimhoz készített kódot kényelmesen fel tudtam használni a Blazor WebGL projektekben is. Több C# algoritmus és adatszerkezet melyeket korábban már elkészítettem néhány kattintást követően felhasználhatóvá váltak a böngészőben futó webalkalmazásban. Blazor nélkül ezt a kódbázist teljesen újra kellett volna írnom Javascript-ben.

3.2 Valósidejű Grafika?

Az egyébként is az elvártnál lassabb kliens oldali sebességet tovább súlyosbítja a sűrű Javascript invokáció. Az összes WebGL hívást továbbítani kell a Javascript futtatókörnyezetnek ugyanis ezeket a hívásokat mind a Canvas DOM elemen és a Javascript WebGLContext példányon kell végrehajtani. Így egy egyszerű WebGL parancs végrehajtása modern processzorokon is, akár 2-3 milliszekundumot is igénybe vehet, amely a valósidejű grafikában a másodpercenként 60 képkocka előállításához elfogadhatatlan. A WebGL-hez hasonló alacsony szintű grafikus API-nkak célja, hogy nagy részletességgel tudjuk konfigurálni a grafikus szerelőszalag adatait és működését. Ha minden erőforrást megfelelően előkészítünk, akár 15-20 WebGL utasítás (30-60 milliszekundum) is szükséges lehet egy olyan egyszerű grafika megjelenítése, mint egy 3 dimenziós gömb. Ez azt jelenti, hogy a jelenlegi lehetőségeinket tekintve ez a technológia még nem alkalmas valósidejű grafika megjelenítésére. A Javascript invokációk csökkentése érdekében a könyvtárban lehetőségünk van

csoportosítani a WebGL utasításainkat és egyszerre több utasítást leküldeni a Javascript futtatókörnyezetnek, de a két környezet közötti szinkronizáció, illetve a hívások szerializálása és deszerializálása ugyanúgy értékes feldolgozási időbe kerül. Az alábbi diagrammon láthatjuk a gömb megjelenítésének 2 perces futása során az egyes képkockák előállításához szükséges időt milliszekundumban. Láthatjuk, hogy ez az idő végig 16 milliszekundum felett áll, tehát a másodpercenként kevesebb, mint 60 képkockát állítunk elő.



3. ábra: Blazor WebGL képkocka előállítási idő milliszekundumban

Amint a WebGL hívások eljutottak a Javascript futtatókörnyezetbe a grafikus processzor által végzett munkák már nem szenvednek teljesítmény veszteséget. Tehát, ha nagy adatszerkezeteket dolgozunk fel vagy hosszú shader programokat kell futtatnunk és csak ritkán például felhasználói adatmódosítás esetén szükséges frissíteni a grafikát, akkor mindezt megtehetjük anélkül, hogy a C# nyelvet elhagynánk. Amennyiben valós időben folyamatosan frissülő grafikát és megadott másodpercenkénti képkockaszámot szeretnénk elérni a Blazor keretrendszer és Blazor Canvas könyvtár még nem áll készen a feladatra.

4. Összegzés

A Blazor segítségével egyszerűen juttathatunk C# kódot a webböngészőkben futó alkalmazásokba. A Razor szintaxissal kiegészített HTML lapok képesek dinamikusan változtatni a tartalmukat és frissíteni a megjelenített információkat. Korábbi alkalmazásaink C# kódbázisát felhasználhatjuk a böngészőkben, ezzel drasztikusan rövidítve a webalkalmazások fejlesztési idejét.

A technológia fiatalsága révén a futási teljesítmény nagyságrendekkel lassabb, mint egy hagyományos alkalmazásé. Ebből kifolyólag jelenleg csak rugalmasabb puha valósidejű határidőkkel rendelkező alkalmazások fejlesztésére alkalmas a Blazor. Szigorú határidők [9] vagy rövid időkorlátok tartása egyelőre nem konzisztens a böngészőben futtatható C# kód segítségével.

Köszönetnyilvánítás

EFOP-3.6.3-VEKOP-16-2017-00001: Tehetséggondozás és kutatói utánpótlás fejlesztése autonóm járműirányítási technológiák területén – A projekt a Magyar Állam és az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg.

Hivatkozások

1. Adam Freeman: *Pro ASP.NET Core MVC*, Apress, 2016, [101-122], ISBN: 978-1-4842-0397-2
2. Andrew Troelsen, Philip Japikse: *Pro C# 7: With .NET and .NET Core*, Apress, 2017, [1245-1251], ISBN: 978-1-4842-3017-6

3. *WebAssembly*, 2019, https://webassembly.org/ (utoljára megtekintve: 2019.11.06.)
4. Laurent Sansonetti: *Mono and WebAssembly - Updates on Static Compilation*, 2018, https://www.mono-project.com/news/2018/01/16/mono-static-webassembly-compilation/ (utoljára megtekintve: 2019.11.06.)
5. Szabó Dávid: *C# Multi-Platform Környezetben*, MSc Diplomamunka, 2018, [58-59] https://edit.elte.hu/xmlui/bitstream/handle/10831/38929/Diplomamunka.pdf (utoljára megtekintve: 2019.11.06.)
6. Dávid Szabó, Dr. Zoltán Illés, Viktória B. Heizlerné: *Valós idejű funkcionalitás Windows-ban*, INFODIDACT 2018, [263-264], ISBN: 978-615-80608-2-0
7. *Blazor Canvas repository on GitHub*, 2019, Letöltve 2019. november 06., https://github.com/BlazorExtensions/Canvas
8. Illés Zoltán: *Programozás C# nyelven*, Jedlik Oktatási Stúdió 2005, [51-54], ISBN: 963-86514-1-5
9. Dr. Illés Zoltán, Heizlerné Bakonyi Viktória, Illés Zoltán: *Valós időben, valós világban*. INFODIDACT 2015, Zamárdi (2015), ISBN: 978-963-12-3892-1