

Média- és Oktatásinformatikai Tanszék

Naprendszer Szimuláció Exobolygókkal

Dr. Horváth GyőzőTanszékvezető egyetemi docens (ELTE IK)Nyírő Levente Gyula

Programtervező informatikus MSc

Budapest, 2025

Tartalom

[1. Bevezetés 3](#_Toc196487570)

[2. Irodalmi áttekintés 5](#_Toc196487571)

[2.1. Teljesítményproblémák és optimalizálás JavaScriptben 6](#_Toc196487572)

[2.2. Webes 3D vizualizáció és kihívásai 7](#_Toc196487573)

[2.3. Háromdimenziós vizualizáció egy bútor webáruházban 8](#_Toc196487574)

[2.4. Fényhasználat optimalizálása háromdimenziós megjelenítésben 9](#_Toc196487575)

[2.5. BabylonJS és ThreeJS összehasonlítás 10](#_Toc196487576)

[3. Felhasználói dokumentáció 11](#_Toc196487577)

[3.1. Teljesítménymérés 12](#_Toc196487578)

[3.2. Menü 13](#_Toc196487579)

[3.2.1. Vonalak megjelenítése opció 13](#_Toc196487580)

[3.2.2. Metrikák exportálása 13](#_Toc196487581)

[3.2.3. Új bolygó hozzáadása 14](#_Toc196487582)

[3.2.4. Bolygók listájának a megjelenítése 15](#_Toc196487583)

[4. Fejlesztői dokumentáció 17](#_Toc196487584)

[4.1. Felhasznált technológiák 18](#_Toc196487585)

[4.1.1. Angular keretrendszer 18](#_Toc196487586)

[4.1.2. Planet adattípus 21](#_Toc196487587)

[4.1.3. ThreeJS – WebGL alapú háromdimenziós renderelés 23](#_Toc196487588)

[4.1.4. Webworker integrálása Angularban 30](#_Toc196487589)

[4.1.5. DataService és az API forrás 32](#_Toc196487590)

[4.1.6. MonitorService 32](#_Toc196487591)

[4.2. Optimalizáció 33](#_Toc196487592)

[4.2.1. Frames per second 34](#_Toc196487593)

[4.2.2. Memóriahasználat 35](#_Toc196487594)

[4.2.3. GPU használat 36](#_Toc196487595)

[4.2.4. CPU használat 37](#_Toc196487596)

[4.3. Telepítés 38](#_Toc196487597)

[4.4. Továbbfejlesztési lehetőségek 38](#_Toc196487598)

[5. Összegzés 39](#_Toc196487599)

[Forrásjegyzék 40](#_Toc196487600)

[Ábrajegyzék 41](#_Toc196487601)

# Bevezetés

Kétéves programtervező informatikus mesterképzésem keretében az utolsó félévben készítenem kell egy diplomamunkát, ahol bizonyíthatom, hogy képes vagyok az adott szakterület fejlesztésére, illetve kutatására. A témám keretein belül kutatási anyagokat kerestem, azokat dolgoztam fel, majd használtam az így megszerzett tudást, hogy a saját programomba beleépítsem, fejlesszem és teszteljem azt. A dokumentációban szóba fog kerülni a témaválasztás, a probléma, ami után kutatni kell és ennek magyarázata, eredményeket fogok ismertetni, beszélni fogok mások eredményeiről, akár más programnyelvekben szerzett tapasztalataikról, majd ismertetni fogom a technikai hátterét az általam írt programnak és az összeszedett megoldásomat a problémára.

Témakörként a Naprendszer modelleket választottam, ugyanis ennek ábrázolása nemcsak lenyűgöző látványvilágot kínál, hanem lehetőséget ad arra is, hogy a modern webes technológiák határait feszegetve, interaktív és valósághű modelleket hozzak létre. Már két féléve foglalkoztam ezzel a témával, hála az egyetemen elérhető kurzusnak, ahol a kiválasztott projektmunkával foglalkozhat a hallgató a félév során és egy mentor is segít a webes tudás elmélyítésében.

A szoftverfejlesztési pályafutásom során mindig is vonzott a frontend fejlesztés, hiszen itt kerül sor a felhasználói felület tervezésére és megvalósítására, amelynek kiemelt szerepe van a felhasználói élmény alakításában. Ehhez kapcsolódóan szerettem volna foglalkozni a háromdimenziós objektumok ábrázolásával weboldalaknál vagy mobilapplikációknál. Ezt azért tartottam fontosnak, mert a webdesign olyan irányba halad, ahol a vizuális megjelenítés és a felhasználóbarát megoldások rendkívül fontosak. A háromdimenziós modellek használata lehetővé teszi, hogy a felhasználók interaktív módon fedezzék fel a tartalmat, ami különösen fontos lehet olyan területeken, mint a tudományos vizualizáció, játékfejlesztés vagy akár az oktatás.

Technikai szempontból a webapplikáció, amiben a kutatásom készült Angular keretrendszerben készült és ebbe lett integrálva a ThreeJS egy canvas-on keresztül. A webes háromdimenziós képi megjelenítésre sok lehetőség elérhető webapplikációknál, ezek közül a legismertebb a ThreeJS, így ez volt az a technológia, amire a kutatásaimat és az általam megírt programot támasztom. Adatlekéréseket is alkalmazok a megírt programban, mégpedig egy francia API-tól kérem el a bolygóknak a pontos adatait.

A ThreeJS már több, mint egy évtizede velünk van az informatikában. 2010-ben publikálták és azóta rengeteg fejlődésen ment keresztül. Ez a JavaScript könyvtár lehetővé teszi, hogy a böngészőben közvetlenül rendereljünk 3D modelleket, anélkül, hogy külső szoftverekre vagy pluginokra lenne szükség. Közben megjelentek az ismert keretrendszerek is a webtechnológiában, mint a React, vagy az Angular, így elő kellett segíteni, hogy ezen technológiákba is importálható legyen a háromdimenziós csomag.

A diplomamunkám végső célja, hogy kutatást végezzek arról, hogyan lehet hatékonyabbá és optimalizáltabbá tenni a ThreeJS működését Angular keretrendszerben, akár mesterséges intelligencia beépítésével a fent említett Naprendszer modellben. Mindez, azért szükséges, mert a megfigyeléseim szerint az egész ThreeJS hatalmas potenciállal rendelkezhet a jövőben, azonban a komplex modellek és a dinamikus animációk kezelése gyakran hatalmas teljesítményigényeket követelhetnek. Ez különösen igaz lehet olyan esetekben, amikor a felhasználói élmény folyamatosan változik és interaktív háromdimenziós tartalmat igényel, mint például a Naprendszer modellek esetében. A kutatásom során olyan technikai megoldásokat és optimalizációs stratégiákat keresek, amelyekkel javítható a renderelési teljesítmény, csökkenthető a memóriahasználat, és biztosítható a felhasználók számára a zökkenőmentes és élvezetes élmény.

# Irodalmi áttekintés

A ThreeJS és a háromdimenziós ábrázolás optimalizálása nem egy új téma a szakmába. Számos kutató és fejlesztő is foglalkozott már ezzel a kérdéssel az elmúlt években. Háromdimenziós tartalmak webböngészőben történő megjelenítése, különösen komplex modellek és dinamikus jelenetek esetében mindig is kihívást jelentett, hogy a teljesítmény és a felhasználói élmény hasonló legyen, mint egy egyszerűbb modellnél. A ThreeJS, mint a WebGL egyik legnépszerűbb JavaScript könyvtára, lehetővé teszi a háromdimenziós modellek hatékony megjelenítését, de a nagy teljesítményigény és a böngésző korlátai miatt a fejlesztőknek gyakran kreatív megoldásokat kell keresniük a problémákra.

Számos kutatás és cikk foglalkozott már a ThreeJS teljesítményének a javításával. Volt, ahol a geometriák egyszerűsítésével és textúrák optimalizálásával foglalkoztak, mindezzel is csökkenthető volt a renderelési idő és a memóriahasználat. Egy másik gyakori megközelítés az instancing. Ez lehetővé teszi, hogy több objektumot egyetlen renderelési hívással jelenítsünk meg, ezzel jelentősen csökkentve a CPU és GPU terhelését.

A modern webes keretrendszerek (Angular, React) is komoly figyelmet kapott az elmúlt pár évben. A komponensalapú technológia és a state management hatékony használata, hogy a háromdimenziós tartalom zökkenőmentesen fusson együtt a webalkalmazás többi részével.

Ebben a fejezetben most egyes kutatások eredményeit fogjuk áttekinteni, amik hatással voltak az általam bemutatott kutatásokra. Azt is be fogom mutatni, hogy miképp volt mindez hatással.

## Teljesítményproblémák és optimalizálás JavaScriptben

Az első bemutatott cikk egy 2016-ban megírt publikáció, amely még nem kapcsolódik a ThreeJS-hez. A háromdimenziós megjelenítés weben a sok teljesítményigénytől fogva optimalizálni szükségeltetik. Ehhez meg kell alapozni azt, hogy magának a JavaScript is (amiben fejlesztünk) feltárjuk a problémáit teljesítmény tekintetében és azt vizsgálhassuk, hogy ezt hogy tudjuk optimalizálni. Ezt a tudást átültethetjük a ThreeJS síkjára is és így kaphatunk egy kielégítő teljesítménnyel futó háromdimenziós megjelenítést.

A tanulmány célja, hogy empirikus módszerekkel elemezze a gyakori teljesítményproblémákat és bemutassa, hogyan lehet ezeket optimalizálni. A kutatás során 98 rögzített teljesítményproblémát elemeztek 16 népszerű JavaScript projektből, mind kliensoldali, mind szerveroldali kódokból. A kutatás öt kérdésre keresett választ.

Az első a JavaScript teljesítményromlásának fő okaira próbál magyarázatot találni. Ezeket elsősorban a nem megfelelő API hívásokban, felesleges adatmásolásokban és ismétlődő műveletekben találta meg. Előbbi az esetek 52%-át tette ki a kutatásban. Ez egy elég nagy arány, hogy erre mindenképp odafigyeljen a fejlesztő.

Második a komplexitását kutatta az optimalizációs változtatásoknak. Itt a cikk leírta, hogy a legtöbb optimalizáció csak pár sor kódot érint, mégis ezek vonhatók felelősségre a JavaScript teljesítményromlásáért.

Harmadik esetben az optimalizálás hatására elért teljesítménynövekedést vizsgálja a cikk. Az eredmény jelentős növekedést mutatott ki (átlagosan 20-75%-os növekedés), azonban az eredmények nem voltak konzisztensek különböző JavaScript motorok között.

Negyedik esetben az előző válaszban található optimalizációknak biztosított konzisztens növekedés mértékét vizsgálták, itt azonban mindössze csak 42,68%-a volt minden motorban.

Az utolsó kérdésben a kutatás az ismétlődő mintázatokra tért ki és arra jutott, hogy az összegyűjtött 29 ismétlődő mintázat más projektben is alkalmazhatóak.

## Webes 3D vizualizáció és kihívásai

Ez a cikk az előző cikkhez képest kifejezetten a webes háromdimenziós implementáció során felmerülő teljesítményoptimalizációról szól. A cikk szintén leírja, hogy ez a technológia egyre elterjedtebb mind az e-kereskedelem, játékfejlesztés, építészeti tervezésben, kutatásokban, ahol a felhasználók interaktív és valósághű élményt várnak. Azonban a webes környezetnek sajátos korlátai és egyúttal kihívásai is vannak. Ezeket figyelembe kell venni a hatékony működés érdekében.

Figyelembe kell venni, hogy a böngészők korlátozott számítási teljesítménye és memóriahasználata miatt a komplex háromdimenziós modellek renderelése lassúvá válhat. A ThreeJS WebGL alapú, ami egy hatékony technológia, de nagy felbontású textúrák és komplex geometriák kezelése esetén nehézségekbe ütközhet. Arra is figyelni kell az implementáció közben, hogy különböző böngészők és eszközök eltérően támogatják a WebGL-t és a 3D grafika egyéb technológiáit, továbbá figyelembe kell venni a régebbi eszközökön való működést is. Fontos, hogy a felhasználó ne egy hosszú betöltési idővel rendelkező oldalt lásson, mert ez jelentősen ronthatja a felhasználói élményt. Ilyenkor megoldásként a fejlesztők a modellek tömörítésének és a progresszív betöltés használatának irányába mozdulhatnak el. Egy másik fontos tényező, hogy a felhasználói interakciók esetén is optimalizálni kell a renderelési folyamatot.

A cikk megoldásként javasolja a háromdimenziós modellek poligon számának csökkentését, illetve a Level of Detail technika alkalmazását, amely a modell részletességét dinamikusan állítja a kamera távolsága alapján. A másik ilyen hatásos módszer a fényoptimalizálás, fényforrások számának csökkentésével és a statikus árnyéktérképek használatával. További megoldásként javasolható, hogy a modellek és a textúrák tömörítése sokkal hatásosabb GLTF formátumban, mint mondjuk a hagyományosan igénybe vett OBJ formátumban. Mindemellett a progresszív betöltés és a lazy loading használata nagyobb fokú felhasználói élményt adhat főleg, ha a felhasználó már a teljes modell betöltése előtt interakcióba léphet a weboldalon megjelenő modellel. Fontos, hogy az eszköznek a kapacitását is vegyük igénybe úgy, hogy a felhasználó eszközének képességeitől függően dinamikusan változtatjuk a renderelési minőséget.

## Háromdimenziós vizualizáció egy bútor webáruházban

Ez a cikk egy olyan online kereskedési platform fejlesztését mutatja be, ahol a cél az volt, hogy a felhasználóknak minél nagyobb élményt adjanak azáltal, hogy a piacra szánt bútorokat pontosan és informatívan mutassa be. Azonban az ilyen fejlesztéseknek a hátránya az is, hogy nagy teljesítmény szükségeltetik és ehhez egy optimalizált frontendet és backendet kell párhuzamosan létrehozni.

A probléma az volt, hogy sok webáruház kétdimenziós képekre támaszkodik, ezzel viszont nem lehet bemutatni pontosan az eladni szánt termék méreteit, dizájnját és funkcionalitását. Ez viszont rosszul meghozott vásárlási döntésekhez, magasabb visszaküldési arányokhoz és elégedetlen ügyfelekhez vezet, különösen olyan iparágakban, mint a bútorkereskedelem, ahol a térbeli vizualizáció kulcsfontosságú.

A webáruház a ThreeJS integrálása mellett határozta el magát, ezzel elérhetővé téve a felhasználónak a termékek 360 fokos forgatását, közelítést a részletek megjelenítéséhez, illetve valós időben testre szabhatják az anyagokat és a színeket.

További megoldásként a MERN stack-re hivatkozik a cikk, ez annyit takar, hogy ezáltal a fejlesztés során létrejövő architektúra skálázhatóvá és modulárissá válik, mind frontend, backend és adatbázis szinten is. Ezzel elérve, hogy a teljesítmény a legjobbá váljon kevesebb erőforrás igénybevételével. Ez is egy fontos szempont, hogy az ügyfélelégedettség növekedésnek induljon.

A kutatás kimutatta az elért előnyöket is, legyen szó a fokozott felhasználói elköteleződéstől, vagy épp a magabiztosabb vásárlói döntéshozataltól. Jelentősen csökkent továbbá a visszáruk száma is, ugyanis a pontos vizualizáció csökkentette az eltéréseket a várakozások és a valóság között.

## Fényhasználat optimalizálása háromdimenziós megjelenítésben

Ebben a cikkben a fényhasználat optimalizálását vizsgálják. Ez az én kutatásomban szintén egy fontos támaszpont, ugyanis a fénybeállítások általában több erőforrást tudnak igénybe venni és ezáltal nagyobb optimalizációra van szükségünk, hogy egy élvezhető teljesítményélményt lehessen adni a felhasználóknak. A cikkben kitérnek arra is, hogy a megfelelő beállítások a renderelési időt is jelentősen le tudják csökkenteni.

A fényforrások helyes beállítása elengedhetetlen a valósághű árnyékok, tükröződések és fényvisszaverődések létrehozásához. Ennek optimalizálása segít kiemelni a modellek részleteit, egyben mélységérzetet ad és növeli a vizuális hitelességet. Több fénytípust is megkülönböztetünk. Környezeti fény – alapvető megvilágítás, ami az egész teret bevilágítja. Irányított fény – ilyen a napfény is. Pontszerű fény – lámpa fényéhez hasonlítható. Az utolsó ilyen a fókuszfény – ami egy zseblámpa világítását imitálhatja.

Az optimalizálási technikák sokrétűek lehetnek, ezt a kutatás is leírja. Dinamikus árnyékok esetén, ahol a valós idejű árnyékszámítások kellenek a pontosabb megjelenítés céljából erőforrásigényesek lehetnek. Optimalizálás céljából ezt csakis a kritikus területeken érdemes használni. Másik ilyen technika az előre számított fény- és árnyékinformációk. Ezek csökkentik a valós idejű számítások terhelését, ami mindenképp előnyös renderelés szempontjából. Ezen kívül még a fénymintavételezés az, ami segítheti az optimalizálást. Itt a fényforrások hatásának mintavételezése csökkenti a renderelési időt.

Következtetésképpen a fényoptimalizálás, ahogy azt a fenti technikák is leírják, nemcsak a vizuális minőséget javítja, hanem egyben csökkenti is a renderelési időt és a hardverigényt. Arra is érdemes a fejlesztés során figyelni, hogy a felesleges fényforrásokat a fejlesztő eltávolítsa és a fényterheket jelentősen csökkentse, ugyanis ez egyben teljesítménynövekedést is eredményezhet. Ezekre a technikákra nagyon odafigyelnek az olyan gyakorlati területeken is, mint a játékfejlesztés, építészeti vizualizáció, vagy épp a termékdesign.

## BabylonJS és ThreeJS összehasonlítás

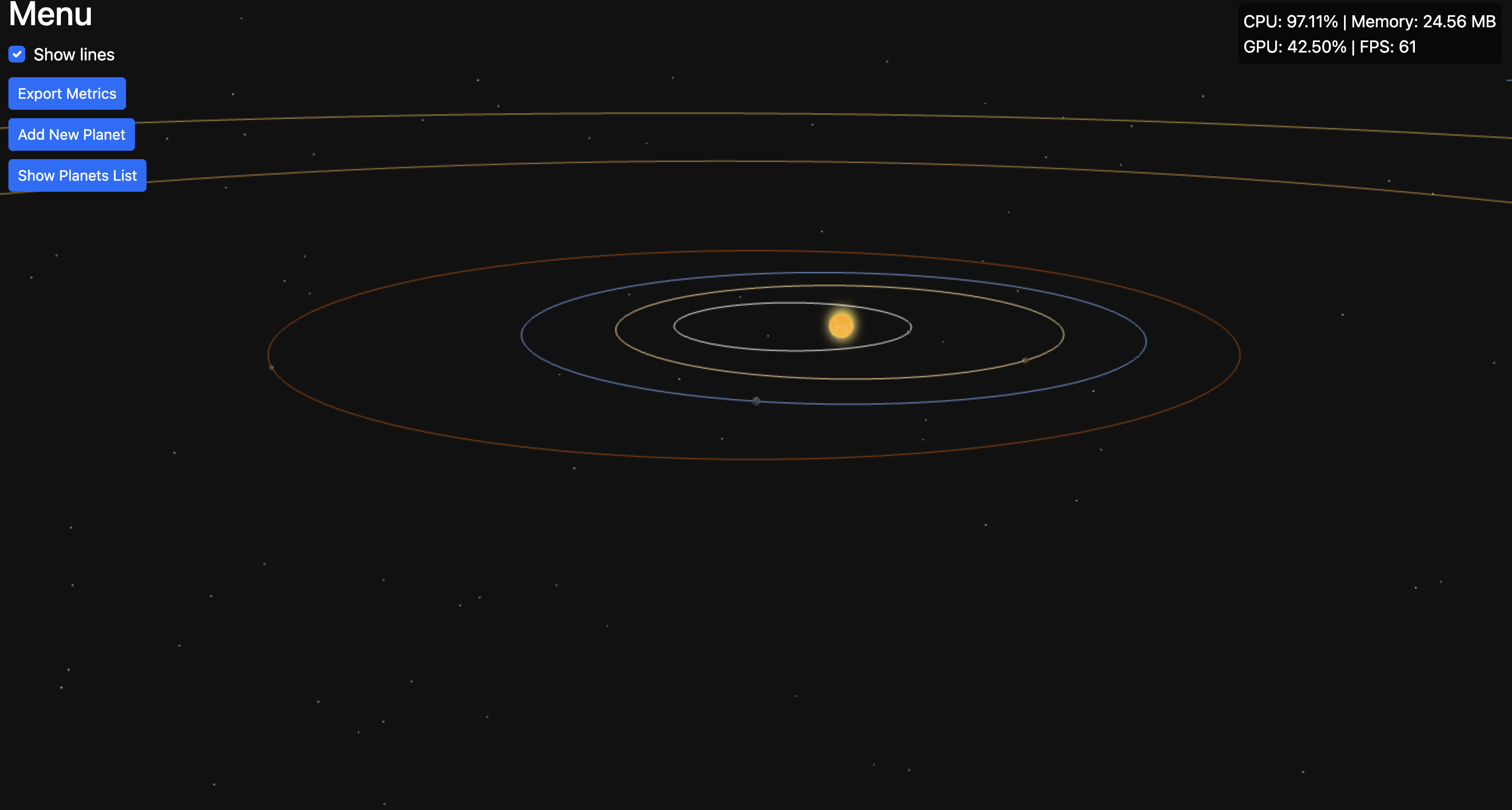
Fontosnak tartottam a kutatásom során, hogy egy másik keretrendszerrel való összehasonlítást is alapul vegyek a kutatásaim során. Az ebben a cikkben lévő tanulmány célja, hogy statisztikai adatokkal alátámasztva kiderüljön, hogy melyik WebGL könyvtár teljesít jobban nagy terhelés mellett. Mindez a kétdimenziós Voronoi-diagramokat (szigetvilágokat) háromdimenziós térképpé alakítja át. Három fő szempontból történtek a mérések: GPU, CPU és RAM használat.

Módszertant figyelembevéve a tesztek egy alap konfigurációban készültek, alap processzorral és 8 GB RAM igénybevételével. Ubuntu rendszeren futott és NodeJS-en voltak az automatizált tesztek. A teszt során 10 különböző méretű és komplexitású térképet ugyanazzal a jelenetettel rendereltek mindkét könyvtárat használva (kamera útvonal követése, gyors forgatás). Majd ezután az automatizált tesztrendszer 60 másodperces teszteket futtatott le, 3 másodpercenként rögzítve az erőforrás-használatot. A végén nem normális eloszlású adatok miatt Wilcoxon Rank Sum tesztet alkalmaztak a különbségek szignifikanciájának vizsgálatára.

Eredmények tekintetében érdekes adatok kerültek napvilágra. GPU-terhelés szempontjából a ThreeJS sokkal jobb eredményeket mutatott, átlagosan 20-30%-al kevesebbet a BabylonJS-hez képest. CPU-terhelésnél is a ThreeJS mutatott előnyt, viszont itt „csak” 15%-al. RAM használatot tekintve azonban nem jutott különbségre a két keretrendszer – a ThreeJS talán kicsivel több RAM-ot használt a renderelés közben. Ezekből az adatokból arra a következtetésre jutott a cikk, hogy a ThreeJS nagyobb komplexitású jelenetek mellett sokkal jobb futási időket tud produkálni szemben a BabylonJS-el. Utóbbi előnyére írható, hogy sokkal kezelhetőbb, egyszerűbb implementáció szempontjából, de a teljesítményhátránya nem elhanyagolható.

# Felhasználói dokumentáció

Ebben a fejezetben bemutatásra kerül részletesen a webapplikáció egyes funkciói, amelyek segítségével a felhasználó személyre szabhatja a háromdimenziós környezetét és konfigurálhatja a Naprendszer modellt.



1. ábra: felhasználói felület

A webapplikációba belépve a felhasználó a fenti ábrán megjelenített oldalt találhatja. Itt rögtön több funkciója is van, ezeket a következő fejezetekben pontról pontra bemutatom. Az alap dolgok, amiket a felületen találhatunk, a menü a bal felső sarokban beállításokkal és funkciókkal, illetve a jobb felső sarokban a mérési eredmények másodpercenkénti megjelenítése.

A teljes képernyőt viszont a Naprendszer modell tölti ki. Itt tudunk pásztázni az bal egérgomb használatával, ha elhúzzuk egyik pozícióból a másikba. Fel nyíl használatával lehet közelíteni, valamint le nyíl nyomva tartásával távolodni a modellben. Ha egy bolygóra rákattintunk bal egérgombbal, a program az adott bolygót kezdi el követni, valamint arra fókuszálni. Ez közelítésnél jó technika, amennyiben közelről szeretnénk megtekinteni az adott testet.

## Teljesítménymérés

A program jobb felső sarkában találhatóak a másodpercenkénti mérési eredmények. Felhasználók számára ez csak egy plusz érdekesség lehet, azonban fejlesztési szempontból igenis kulcsfontosságú adat lehet. Későbbi fejezetben bemutatom, hogy ezekből az adatokból hogyan lehet egy adatcsomagot exportálni további kutatási célokra.

Négy adatot találhatunk itt, ami folyamatosan változik a program erőforrásigényétől, vagy éppen teljesítményétől függően. Első a CPU, azaz a processzor használat igénye, ami itt egy becsült adat százalékos arányban mérve. A második adat a program memóriahasználatát mutatja megabájtokban. A második sorban az első adat GPU néven megmutatja, hogy mekkora a program erőforrásigénye a videókártyát tekintve. Az utolsó adat pedig megmutatja, hogy mennyi képkockát képes a program feldolgozni és a képernyőn a felhasználónak megmutatni másodpercenként (angol nevén frames per second). Ez utóbbi adatot sok helyen használják videójátékoknál is. 60 FPS-nél a program simának és folyékonynak tűnik, míg mindez 20 FPS szakadozna.



2. ábra: mérési adatok

## Menü

A program felső sarkában találhatjuk a főbb funkciókat, amelyekkel befolyásolhatjuk a Naprendszer modell működését, beállításait és mérési adatokat is exportálhatunk ennek a segítségével.

A képen szöveg, képernyőkép, Betűtípus, szám látható

Automatikusan generált leírás

3. ábra: menü lehetőségei

### Vonalak megjelenítése opció

Az első opció a „Show lines” checkbox. Ezáltal a felhasználó kikapcsolhatja, vagy újra megadhatja a programnak a bolygók pályáinak mozgási útját. Ezt azért tartottam a fejlesztés során fontosnak, hogy ne zavarja a felhasználót, ha épp a bolygó felszínét kívánja megtekinteni és a vonalak csak elvonnák a figyelmét.

### Metrikák exportálása

A következőkben három gomb lesz bemutatva, ebből az első az „Export Metrics”. Ennek a gombnak a segítségével – ahogy az már fentebb említésre is került – az eddig eltelt másodpercek adatait hívatott mérni, ezzel grafikonok, kimutatások elkészítésének lehetőségeit segítve elő. A fájlt a felhasználó CSV formátumban kapja meg, amelyet jegyzettömb vagy Excel használatával könnyedén meg is nyithat. A fájl megtekintésével az alábbi sorokkal találkozhatunk.

A screen shot of a computer

AI-generated content may be incorrect.

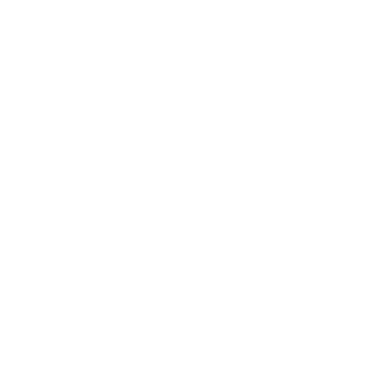
4. ábra: generált CSV fájl tartalma

A fájl első sorában a fejlécet találhatjuk. A „timestamp” a mérés kezdetétől eltelt másodperceket mutatja, CPU a százalékos arányt, a memória MB-ot használ mutatóként. A negyedik oszlopban a képkockák száma találhatóak másodpercenként eltelve, míg az ötödik oszlopban a GPU használatot láthatjuk szintén százalékos arányban.

A második sorban látható az első mérési adat. 43.27%-os CPU használattal, 23.97 MB memóriafelhasználással, 0 FPS-el (ez egy kezdeti betöltési értéke lesz a programnak), illetve a 90%-os GPU felhasználással – ez utóbbi később csökkenő tendenciát mutat).

### Új bolygó hozzáadása

A második gomb „Add New Planet” névvel található meg. Itt adhatunk hozzá új exobolygót a modellbe. Ennek a gombnak a megnyomásával átkerülünk egy olyan fázisba, ahol már kötelező kiválasztanunk a bolygó pályáját az egér mozgatásával. Amennyiben a bal egérgombra rányomunk, megbizonyosodva az általunk kiválasztott pálya helyességéről, visszakerülünk a kiindulási fázisra a felhasználói kezelőfelület tekintetében és láthatóvá válik az új bolygónk a modellben.

A solar system in space

AI-generated content may be incorrect.

5. ábra: új bolygó hozzáadása

Az új bolygó már bele is került a rendszerbe és megkapta a saját textúráját a háttérben egy hőmérsékleti tulajdonsággal. Előfordulhat, hogy egy bolygó ellipszispályája túl közeli a Naphoz, vagy épp túl távoli. Ez mind meglátszódik az objektumon. Amennyiben megfelelő pályán forog a Nap körül – megfelelő távolságban, akkor kialakulhatnak rajta az exobolygók által uralt tulajdonságok, amelyek az élethez elengedhetetlenek.

A planet with lines of light

AI-generated content may be incorrect.

6. ábra: exobolygó kialakulása megfelelő környezetben

### Bolygók listájának a megjelenítése

A „Show Planets List” gomb megnyomása egy lentebb megnyíló adatoszlopot eredményez. Ezzel láthatóvá válik az összes eddig létrehozott bolygó az alapértelmezett bolygókkal együtt. Minden sorban láthatjuk a bolygó nevét, egy „Follow” és egy „Delete” gombot. Ha a bolygók listáját megjelenítettük a fenti gomb neve megváltozik és az elrejtésre utal „Hide Planets List” néven. Természetesen ezzel bezárhatjuk az imént megjelent ablakot.

Alapvetően a program a Napra fókuszál és arra is tudunk közelíteni és távolítani a kamerát a nyilak segítségével, azonban egy kiválasztott bolygó követését itt is megoldhatjuk a „Follow” gomb segítségével. Ilyenkor a kamera az adott bolygóra helyezi a fókuszt és annak a pályáját kezdi el követni. Ennek a segítségével erre a bolygóra már rá is tudunk közelíteni minden gond nélkül.

Előfordulhat, hogy valamelyik bolygót el szeretnénk távolítani a modellünkből. Erre lehetőséget is ad a program a felhasználónak a bolygók nevei mellett megjelenő „Delete” gombbal. A Napon kívül mindegyik objektumot törölni lehet. Ha egy olyan bolygó kerül törlésre, amelyiken a fókuszunk van éppen, akkor a program azzal kezeli ezt, hogy újra a Nap kerül fókuszpontba. Azokat a bolygókat is törölhetjük, amelyeket a program automatikusan hozott létre a program megnyitása elején. Egy bolygó törlése esetén a hozzá tartozó elliptikus pálya íve is törlésre kerül és nem lehet már megjeleníteni a képernyőn.

A screenshot of a black screen

AI-generated content may be incorrect.

ábra 7: bolygók listája

# Fejlesztői dokumentáció

A Naprendszer szimuláció megvalósítása során számos technológiai és architekturális döntést kellett meghozni, hogy a rendszer hatékonyan működjön, mindeközben valósághű vizualizációt nyújtson, és optimalizált legyen a teljesítmény szempontjából. Ebben a fejezetben részletesen bemutatom a felhasznált technológiákat, az alkalmazott architektúrát, valamint a megvalósítás során alkalmazott főbb módszereket és optimalizációs technikákat.

A modern webes technológiák gyors fejlődése lehetővé teszi, hogy egyre komplexebb háromdimenziós tartalmakat jelenítsünk meg a böngészőkben, azonban ez a lehetőség számos kihívással is jár. A teljesítmény, skálázhatóság és felhasználói élmény szempontjából kritikus volt, hogy a megfelelő eszközöket és módszereket válasszam. A következő fejezetekben bemutatom, hogyan választottam ki a megfelelő keretrendszereket, miként épült fel a rendszer architektúrája, és milyen technikákat alkalmaztam a hatékony működés érdekében.

A fentiek mellett kitérek a külső adatforrások integrálásának folyamatára is, amely lehetővé tette, hogy a szimuláció ne csak vizuálisan legyen valósághű, hanem a tudományos megalapozottságát is biztosítsam. Végül pedig áttekintésre kerül a fejlesztés során felmerülő kihívások és az azokra talált megoldások, amelyek nélkülözhetetlenek voltak a stabil és optimálisabban futó alkalmazás létrehozásához.

Ez a fejezet nem csupán a jelenlegi implementációt mutatja be, hanem betekintést nyújt a jövőbeli fejlesztési lehetőségekbe is, mint például a renderelési folyamat további optimalizálása vagy a virtuális valóság támogatásának bevezetése.

## Felhasznált technológiák

A választott technológiák lehetővé tették egy olyan alkalmazás létrehozását, amely nemcsak jól teljesít, hanem könnyen bővíthető és karbantartható is. Mindezt úgy, hogy a tudományos pontosság és a felhasználói élmény sem került háttérbe.

### Angular keretrendszer

Az alkalmazás alapját az Angular keretrendszer képezi, amelyet számos előnye miatt választottam és mert a régebbi fejlesztések során, amelyek a ThreeJS-re irányultak, ebben a technológiában volt releváns tapasztalatom. A komponensalapú architektúra lehetővé teszi a kód újrafelhasználását és könnyű karbantarthatóságát. Az Angular erős típusossága (TypeScript) pedig segít elkerülni a gyakori programozási hibákat. A keretrendszer beépített eszközei, mint a dependency injection és a reaktív programozás (RxJS) lehetővé tették az alkalmazás hatékony és moduláris kialakítását.

Ebben a webapplikációban tulajdonképpen az „app.component.ts”-nek és a illetve a „threejs.worker.ts”-nek volt nagyobb főszerepe. Azonban idő kellett, míg kikutattam, hogy hogyan kell a kettőt összekötni, hogy működőképes legyen a háttérben futó szál és a köztük lévő kommunikáció is zavarmentes lehessen. Ennek megvalósításáról egy későbbi fejezetben térek ki.

Elsősorban az „app.component.html”-ről írnék, ugyanis itt vannak azok a funkciók, amik elengedhetetlenek felhasználói szempontból, így nyújtva azt az élményt, hogy a felhasználó is szerkesztheti vagy kutathatja a Naprendszert. A html kód gyakorlatilag csak a menüt, a canvast és a teljesítményadatokat mutató „div” elementet tartalmazza. A menü eseményhívásokat is tartalmaz, amely arra készteti a TypeScript oldalt, hogy valamely adatot manipulálja. Ahhoz, hogy minél strukturáltabban nézzen ki a kód és kevés styling adatot kelljen hozzáadnom, így a Bootstrap 5 hozzáadását tartottam ésszerűnek az Angularhoz. Ezt a csomagot npm install segítségével kellett hozzáadni a programhoz, majd az „angular.json” fájlban hozzá kellett adni a megfelelő helyre az elérési útvonalát. Az Angular egyik hatalmas előnye a komponens alapú programozás mellett a html-be való interpolálás. Bármelyik változó értékét megjeleníthetjük és futási időben megváltoztathatjuk a TypeScript fájlban és a html-ben ez azonnal láthatóvá is válik. A kódban gyakorlatilag csak ez utóbbit kellett használni, ugyanis a fő szerep a „threejs.worker.ts” fájlon van.

Az „app.component.ts” fájl szolgál arra, hogy az inicializálást elkezdje, elküldve az adatokat a worker felé és egyben betöltve az adatokat a html-re, hogy a felhasználó kész adatokkal szembesüljön, illetve az események kezelését intézze. A fájl első soraiban az alap deklarációk és a hozzájuk tartozó inicializációk találhatóak. A „worker” a mellékszál kezelésével, a „loggingInterval” a mérési mutatók ütemezésével, a „canvas” pedig a háttérben megjelenő képpel foglalkozik. Lejjebb találhatóak a méréssel foglalkozó változók: a „cpuUsage”, „memoryUsage”, „gpuUsage” és az „fps”. Ezek a „loggingInterval” segítségével másodpercentént változnak. Az „isAddingPlanet” egy flag, ami megmutatja, hogy éppen folyamatban van-e új bolygó hozzáadása, és a „newPlanetData” tárolja el az új bolygóval kapcsolatos adatokat miközben a felhasználó éppen az új bolygó hozzáadásával van elfoglalva.

Az „ngOnInit” metódusban történnek meg az adatok lekérései a „data.service.ts”-en keresztül minden egyes bolygó tekintetében, amiket a rendszer a program elején létrehoz. Ezeket az adatokat a metódus elmenti a listába, majd továbbküldi a worker threadnek, hogy dolgozzon vele tovább. A Napot is ebben a metódusban adjuk hozzá, azonban ennek adtam egy „deletable” adattagot, ami világossá teszi, hogy ehhez az objektumhoz nem tartozhat törlés gomb.

Az inicializálás után fontos, hogy olyan dolgok is lefussanak, amelyeknek be kell várniuk ezt a lépést. Erre szolgál az „ngAfterViewInit”. Itt inicializáljuk a workert és adjuk át az összes alap eseménykezelést a worker threadnek. Ilyen esemény amikor lenyomjuk az egér bal gombját, vagy felengedjük azt, mozgatjuk az egeret, lenyomjuk a fel-le nyílakat, görgetünk az egér görgőjével. Itt található még egy worker threadből bejövő információ az “FPS”-el kapcsolatosan. Ez a méréshez egy elengedhetetlen adat, ezzel a program dolgozik is tovább.

A privát hozzáférési szintű „monitorSystemStats” feladata, hogy kiszámolja és elmenti a fent említett változóba, hogy az adott másodpercben mekkora volt az egyes perifériák terheltsége. A metódus elején törli az intervallumot a “loggingInterval” változóról, megnézi, hogy mi az aktuális gép típusa és ez alapján hozzárendel egy értéket. Ezután egy belső metódus az FPS segítségével tesz egy becslést a GPU használattal kapcsolatban. Ezután a memória- és CPU-használatra teszi meg ugyanezt a metódus.

Egy másik privát metódus is van az osztályban. Ez a „sortPlanetsByDistance” nevet kapta. A lényege, hogy miután a felhasználó megnyitotta a bolygók listáját, akkor mindez a Naptól számított távolságot figyelembe véve teszi sorrendbe a bolygókat. A felhasználó által létrehozott bolygók a lista végére kerülnek, hogy a felhasználó számára mindez szembetűnőbb lehessen.

A következő pár metódus mind publikus a html számára, így ezek emiatt eseménykezelőként is funkcionálnak. Az első ebből az „onShowLines”, aminek a célja, hogy a továbbküldi a worker threadnek a checkbox éppen aktuális állapotát.

Az „exportMetricsToCSV” egy egyszerűbb metódus, itt igazából csak a monitorService-nek az erre vonatkozó metódusát hívja meg, ezzel egy CSV fájlt generálva, de ennek a belső logikája egy későbbi fejezetben kerül bemutatásra.

Következő metódus egy kicsit bonyolultabb az előzőkhöz képest. Arra funkcionál, hogy elkezdődjön az új bolygó hozzáadása. „startAddingPlanet” néven fut és az első feladata, hogy az „isAddingPlanet” változót igazra állítsa, ezzel is blokkolva a gombot, ami az új bolygó hozzáadására irányul, hogy a fókusz a bolygó tulajdonságainak kialakítására helyeződjön. Mindez egy objektumot fog eredményezni, amit a bolygó behelyez az osztályszíntű listába és az új objektumot elküldi a worker threadnek. A végén még egy rendezést végrehajt a listán az erre hivatott rendező metódus meghívásával, hogy a létrehozott bolygó a megfelelő pozícióba kerüljön.

Az utolsó három metódus már a meglévő bolygókkal kapcsolatos. Az első ilyen, ami segít megjeleníteni a listát „toggleShowPlanetsList” néven fut és csak a „showPlanetsList” változót állítja át annak ellentétére, hogy a felhasználó láthassa, vagy elrejtse a bal oldalon megjelenő panelt a bolygók gyűjteményével. Ez a metódus nem küld tovább adatot a worker-nek, ugyanis ez csak a külső megjelenítéssel (GUI) kapcsolatos. Minden bolygó mellett megjelenik két gomb, az egyiknek az eseménykezelője a „followPlanet”. Ez továbbküldi a workernek a kiválasztott bolygó „englishName” adattagját, majd a worker teszi a dolgát, ahogy azt a következő fejezetben látni fogjuk. A „deletePlanet” is ugyanezeket a lépéseket teszi meg, csak más típusnév alatt küldi tovább az adatot, hogy a worker tisztában legyen vele, milyen lépést is kell eszközölnie.

Az „app.component.ts” fájl végén egy „ngOnDestroy” metódussal megbizonyosodunk róla, hogy a logolás befejeződött és nem terheli tovább a memóriát.

### Planet adattípus

Mielőtt a worker szállal kapcsolatos tudnivalókat ismertettem volna, fontosnak tartottam, hogy bemutassam a hozzászabott adattípust, amelyet „Planet”-nek neveztem el. Ennek segítségével sikerült implementálni és optimalizálni a kódot a worker-ben, ezért nagy szereppel bír. Ezt a fájlt a models mappán belül találjuk.

Négy fontos alapadattagja van. A „data”, amelyik a bolygó alapinformációit hívatott eltárolni, a „mesh” a ThreeJS-ben való megjelenítést kezeli, „orbitLine” néven a bolygó elliptikus pályáját mentjük el, illetve a „spotLight”, ami a Naptól az éppen aktuális bolygó felé érkező fényt tárolja el. Az utóbbi három típusa „any”, azaz nincs megkötve, hogy milyen típusú lehet, ez futási időben egyébként is meg lesz határozva.

A „data” adattagnak bonyolultabb a típusa, ugyanis ezen belül még több alapadattal kapcsolatos információ található meg.

Az első a legegyszerűbb, „name”, ami az azonosítását mutatja meg a bolygónak, ez szöveges adattípusként van eltárolva.

A „color” segít abban, hogy a bolygó pályájának színét tároljuk el, ami a legjobban hozzá illik és a felhasználói szempontból növelhetjük így a meglévő élményt. Ez az adat lehet string vagy number, attól függően, hogy milyen színkódot kap meg az app.component.ts-től.

A következő adattagok már csak number típusúak lesznek. A „size” adattag a bolygó méretét adja meg, hogy megkülönböztethessük őket ez alapján is.

A „speed” adattag a sebességét tárolja el a bolygónak, azonban ahogy ezt látni fogjuk, ki kell számolni további meglévő adatok alapján.

A „distance” segít meghatározni, hogy mekkora a egyes bolygók másik objektumtól vett távolsága egyes esetekben. Ezt az adatot csak a Hold esetében használtam fel, hogy egy bizonyos távolságot meghatározzak tőle a Földtől. Minden más esetben a „perihelion” és „aphelion” adat segít ezt eltárolni, majd a worker-ben meghatározni az éppen aktuális távolságot a Naptól a pozíció függvényében. A Föld esetében a napközelség 147 millió kilométer, míg a naptávolság 152 millió kilométer.

A „semimajorAxis” az általánosabb bolygók egyfajta pályatávolságként szolgál. Ez az ellipszis pálya leghosszabb átmérőjének a fele kilométerben. A Föld esetén ez 149.6 millió kilométer.

A következő fontos adat, ami minden bolygó pályájának megadja a sajátosságát, az az „eccentricity”, magyar nevén excentricitás. Ez a pálya elnyúltságát mutatja meg. Amennyiben az érték nulla, úgy egy tökéletes kört ad meg a pálya, de ha közelíti az 1-et, úgy egy nagyon elnyújtott ellipszisű pálya keletkezik. Információképpen a Föld excentricitása kb. 0.017, tehát majdnem kör alakú pályán kering.

Tengelyferdeséget is meg kell határozni és szemléltetni kell a bolygóknál a modellben. Erre szolgál az „axialTilt” adattag. Ezt fokban tároljuk el, a Föld esetén ez 23,5°, ez is okozza az évszakokat.

Az egyik lényeges adattag az „angle”. Az aktuális pályaszöget mutatja meg. A pozíciók kiszámolásához és animálásához elengedhetetlen.

Azt is fontos tudni, hogy mennyi idő alatt keringi körül a bolygó a központi csillagunkat egy teljes kör alatt. Erre szolgál a „sideralOrbit” adattag. A Föld esetén ez 365.25 nap.

Van egy adat, aminek nem number a típusa, ellentétben a fenti pár adattaggal. Ez a „mass”, vagyis a bolygó tömegét adja meg. Nem lehet egyszerűen number típusként ezt eltárolni, mivel olyan nagy szám, hogy a hozzá tartozó „value”-t és „exponent”-et is el kell menteni. Ezek mind számbeli típusok. Eredményül kapjuk azt a tudományos alakot, hogy

A Föld esetében ez a két érték 5.972 és 24

### ThreeJS – WebGL alapú háromdimenziós renderelés

A projektemben, ahogy azt már a bevezetőben bemutattam, ThreeJS könyvtár segítségével valósítottam meg a Naprendszer interaktív háromdimenziós szimulációját, amely lehetővé teszi a bolygók valósághű megjelenítését és viselkedésének szimulálását. A megoldásom a WebGL technológiára épül, így hardveres gyorsítás kihasználásával képes komplex háromdimenziós jelenetek megjelenítésére közvetlenül a böngészőben anélkül, hogy külső bővítményekre lenne szükség.

A fájl elején találhatjuk az importokat és az inicializációkat. Itt vesszük át a főszáltól a canvas-t, illetve a renderer is itt kerül kialakításra a háttérszínével és az árnyékbeállításokkal együtt. A scene-t is létrehozzuk, majd ezután jönnek a kamerával kapcsolatos beállítások. PerspectiveCamera-t állítottam be a programnak, ami azt jelenti, hogy a valóságos perspektívát szimulálja – vagyis amit a szemünk is érzékel: a távolabbi tárgyak kisebbnek látszanak, a közelebbi tárgyak nagyobbnak. Ezzel együtt a kamera kezdőpozícióját is meg kell adni y és z tengelyt tekintve. Itt kerül inicializálásra a yaw, pitch és a previousMousePosition is, amik a kamera pozíciójának mozgatásáért felelnek.

A programnak kell egy alapfény, hogy lássuk az objektumokat. AmbientLight-al oldottam meg és ennek az értékét minimálisra állítottam, hogy minél valóság közelibb élményt nyújthassak a felhasználóknak. A fény tiszta fehér színt eredményez.

Meg kell adni az alap statikus objektumokat, a Napot és a Holdat, valamint inicializálni kell a bolygók listáját, amelybe az előbb felsorolt két objektum nem tartozik bele, mivel azok más paraméterekkel lesznek felvértezve. A bolygók listájának minden egyes eleme és a Hold a Planet adattípust fogják kapni, amelynek adattagjai fentebb részletezésre kerültek.

A kezdeti fázisban kerülnek felsorolásra az új bolygó hozzáadásával kapcsolatos segédváltozók, mint az isAddingPlanet boolean változó, previewPlanet és a previewOrbit. Utóbbi kettőnek a célja, hogy eltárolja és megjelenítse a scene-ben az objektumot, amit a felhasználó még csak hozzá tervez adni a modellhez. Ezek a változók egyelőre null értékben maradnak.

A képkockák számlálása is a workerben történik, amit a főszál a futás ideje alatt megkap. A frameCount segítségével a program megszámlálja, hogy egy másodperc alatt hány képkocka került feldolgozásra és továbbküldésre.

Az általánosabb változók pedig az inicializálások alján találhatóak. Például, hogy a program mutassa-e az elliptikus pályákat, hogy éppen a felhasználó húzza-e az egeret, mert pásztázni akar a kamerával. Olyan változó is van, ami azt jelöli ki, hogy épp mi az aktuális objektum, amit a kamerának követnie kell, de vannak olyan konstans adatok is, amelyek a bolygóközi távolságok arányát, vagy épp az animáció sebességét hívatott meghatározni.

Az első metódus a sorban az „addStars” nevet kapta. Ennek célja, hogy még realisztikusabbá tegye a modellt azáltal, hogy egy képzeletbeli gömb síkjára csillagszerű objektumokat helye a következő matematikai egyenlet használatával.

x = r sin(ϕ) cos(θ)

y = r sin(ϕ) sin(θ)

z = r cos(ϕ)

ahol

r a gömb sugara.

ϕ a magassági szög [0, π].

θ a horizontális szög [0, 2π]:

Ahhoz, hogy létrehozhassunk új elliptikus vonalakat vissza kell adnunk egy ThreeJS vonalat, amit a „createOrbitLine” metódusban generálunk a következő egyenlettel:

ahol

r(θ) az adott szögű távolságot jelenti.

„a” a fél-nagy tengely (semi-major axis).

„e” az excentricitás.

θ az aktuális szög, amely 0 ≤ θ ≤2π között mozog.

Amikor új bolygó adatát kapjuk meg, a sebességet nem tudjuk egyből lekérdezni, hanem azt a bolygó volatilitásából, súlyából, excentricitás és az animáció sebességéből kell kikövetkeztetnünk. Erre implementáltam a „calculateSpeedFromVolatility” függvényt, ami a bolygó adatait várja. A függvénymagban a bolygó tömegét logaritmikusan normalizálja, az excentricitás pedig közvetlenül befolyásolja a volatilitást. A volatilitás összeadódik, és az ANIMATION\_SPEED szorzó segítségével adja vissza alapsebességet.

Mivel az API-ból a bolygók tengelyferdülését fokban kapja meg a program és a ThreeJS-ben radiánban kell megadni a forgatást, így szükség volt egy „getAxialTiltInRadian” függvény implementálására is. A függvény a szokásos formulát használva a bejövő fok értéket megszorozza a π és a 180 hányadosával és ezzel visszaadja az új értéket.

Mivel cél volt egy realisztikus modell kialakítása, ezért fontos volt, hogy a Nap kapjon egy ragyogó effektust. Ezt a ThreeJS-ben „lensflare” kifejezéssel illetik, azonban az implementációja egyszerű. Erre a „getGlow” függvényt hoztam létre, ami egy fénylő sprite-ot hoz létre a megadott textúrával és mérettel, és beállítja annak áttetszőséget, fényességét. Ezt a visszatérő adatot majd a Naphoz hozzá kell adni annak felparaméterezésekor.

Ahhoz, hogy árnyékok legyenek, fény is kell. Alapfényről már esett szó a modellben, azonban minden Nap – bolygó közötti kapcsolathoz meg kell adni egy „spotlight”-ot, azaz egyfajta reflektorfényt. Erre szolgál a „createPlanetSpotlight” függvény, aminek meg kell adni paraméterben a bolygó nevét és mindez egy fényt fog visszaadni, amit később hozzá kell adni a modellhez.

A „getInitialAngle” függvény a bolygó kezdeti szögét számítja ki a bolygó „sideralOrbit” pályája és egy adott dátum alapján. Az eredmény a bolygó helyzetét mutatja a pályáján, figyelembe véve az eltelt időt a J2000 referencia dátumtól. A szög a 0 és 2π közötti értékre van korlátozva.

A következő függvény a „getPlanetByName” nevet kapta. Mivel az összes bolygó egy listában található, így szükséges volt, hogy valahogy megtaláljuk a bolygót és azt visszakapva műveleteket tudjunk végezni, vagy épp egy lekérdezést lehessen végrehajtani. Egy nevet vár paraméterként, ugyanis ez az elsődleges azonosítója a bolygóknak a modellben. A biztonság kedvéért a lambdában a listában lévő neveket kisbetűsre állítja, hogy a keresés sikeressége biztos legyen.

A „createNewPlanet” függvény egy új bolygót hoz létre a Three.js-ban, amely gömb formájú, textúrázott és fényforrással rendelkező objektum. Beállítja a bolygó méretét, pozícióját, tengelyferdülését, valamint biztosítja, hogy árnyékot vessen és árnyékot is kapjon. Végül visszaadja a bolygó 3D-s reprezentációját.

Új bolygó létrehozásánál csináltam egy segédfüggvényt, ami meghatározza annak távolságát a Naptól. Így kialakítja a program a megfelelő körpályát a bolygó számára. Ez a függvény a „calculateOrbitDistance” nevet kapta.

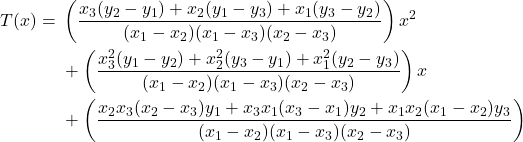
A következő függvény a bolygó átlaghőmérsékletére ad becslést. Az „estimateAvgTemperature” metódus csak egy távolság adatot vár. Ebben a metódusban három ismert távolság-hőmérséklet adat alapján másodfokú interpolációval létrejövő függvényre volt szükség. A három távolság-hőmérséklet adatot a Merkúr, Föld és a Mars adataiból tudtuk megszerezni. A Vénusz paramétereit annak hőmérsékleti anomáliája miatt nem tartottam célszerűnek belevenni az interpolációba.

(d1, t1) = (57475497.74, 167) – Merkúr

(d2, t2) = (149339835.23, 15) – Föld

(d3, t3) = (226914680.3, -63) – Mars

A másodfokú egyenletet a következőképpen lehet kiszámolni:



A távolságot az x helyére beillesztve pedig könnyedén megkaphatjuk az újonnan generált bolygó hőmérsékletét. Eredményként a következő egyenletet kapjuk.

A graph with a line

AI-generated content may be incorrect.

ábra 8: Nap távolságától számított becsült felszíni hőmérséklet

A kamerakezelésre is kellett függvényt írni. Ez a „changeTargetPlanet” nevet kapta és paraméterként a felhasználó által kiválasztott bolygót veszi át. Mindezt úgy végzi el a metódus, hogy a kamera megtartja a korábbi távolságot és irányt, amelyről nézett az előző célobjektumra.

Az animáció során kell egy függvény arra, hogy a bolygónak folyamatosan tudjuk változtatni a pálya menti szögét a sebessége alapján, így a hozzá tartozó ellipszis pályán mozog tovább. Kiszámítja az aktuális pozícióját a bolygónak, figyelembe véve a pálya excentricitását és félnagytengelyét. Emellett folyamatosan forgatja a bolygót a saját tengelye körül, hogy ezzel minél élethűbb animációt érjen el a modell.

Mivel a worker fájlnál nem tudjuk statikusan elérni az adatokat kívülről, így JavaScriptben használatos aszinkron fetch-ekkel kell megoldani a textúrák lekérését. Ehhez két aszinkron függvényt is írtam. Az egyik a „loadTextures” nevet viseli magán. Ez egy paraméter nélküli metódus, ami bitmapekké alakítja a lekérdezett .jpg fájlokat és azokat visszaadja egy objektumként. Később erre iratkozik fel a program és inicializálja a bolygókat textúra tekintetében is.

Egy másik metódus a „loadTextureByPath”. Itt már paraméterként egy elérési utat is vár a függvény, majd ennek segítségével szintén aszinkron hívás által visszaadjuk az elvárt textúrát.

Amikor lekérdezzük az összes kezdeti objektum textúráját, elkezdődik a Nap és a bolygók inicializálása. Itt hívjuk meg a Nap ragyogását és a csillagok létrehozását is. Egy bolygó inicializálásánál megadjuk, hogy mi legyen a neve, párosítjuk a textúrákat és beállítjuk azokat, méretet és árnyékot állítunk. Fényeket is itt kérünk le, ahogy a kezdeti mozgást is, majd az eljárás végén hozzáadjuk a létrehozott bolygókat a fényekkel és az elliptikus pályákkal együtt a jelenethez. A Hold is itt kerül beállításra. Az „animate” meghívásával elindítjuk a mozgásokat.

Az „animate” metódus az előző implementációkban jóval hosszabb volt, azonban a sikeres optimalizációnak köszönhetően sikerült lerövidíteni a programkódot és gyorsabbá tenni a programot. Az első sorokban a Holdnak állítjuk be a haladási irányát és a pozícióját annak „mesh” adattagján keresztül. Utána az „FPS” kiszámolása következik, amit a metódus azonnal is visszaküld a főszálnak, hogy az adatokat elmentse és abból kalkulációkat végezzen. A bolygók listáján is végig kell iterálni, hogy mindegyiknek újra állítsunk a pozícióját a „setPlanetPosition” függvény segítségével. Majd a kijelölt bolygónkat is követnünk kell, erre camera.lookAt függvényét hívjuk meg. Finomítani kellett a kamerakezelésen a „lerp” technológia segítségével is, így amikor valamilyen változás áll be a kamera mozgásában, az egészet finoman indítja el és állítja meg, így sokkal emberibb környezetet teremtünk a felhasználónak interakciók szempontjából. Az utolsó két sor pedig kirajzolja a jelenetet a megadott kameranézetből a képernyőre és meghívja az „animate” függvényt a következő képkockánál, ezáltal létrehozva egy folyamatos és hatékony animációs ciklust.

Az utolsó blokk fogadja a user interakciókat és a főszálról érkező adatokat is. A főszáltól „event”-et kapunk paraméterként, ez a „data.type”-on belül tartalmazza a bejövő adat típusát és egy iránymutatást ad, hogy hogyan kell azt kezelni. Mindemellett más adatok is lehetnek az „event.data”-n belül.

Az első ilyen event a „mousedown”. Ez azokat az eseményeket tartalmazza, amik akkor történnek, amikor a felhasználó lenyomja az egér bal gombját. Itt kezeljük, hogy vajon objektumra nyomtunk-e rá, csak a pásztázást szeretnénk elindítani, vagy épp aktív-e az „isAddingPlanet” adattag és épp új bolygót kell inicializálnunk annak alapadataival. Amennyiben utóbbi történik, akkor itt történik meg a textúra hívása azok alapján, amilyen becsült hőmérséklet lehet az adott bolygón. Az eljárás végén hozzáadjuk a bolygók listájához az új bolygót és a jelenethez a bolygót a fényekkel és az elliptikus pályával együtt.

Egérfelengedésre is kell egy esemény, ez a „mouseup”, aminek a hatására az „isDragging” változót hamis értékre állítjuk, ezzel megakadályozva a kameramozgást az egér mozgatásának hatására.

A „mousemove” esemény esetén a kamerának a pozícióját állítjuk át abban az esetben, ha az „isDragging” flag igaz értékben van. De van másik eset, amikor új bolygót adunk hozzá a modellhez. Itt folyamatosan, minden mozgás esetén létre kell hozni és törölni kell az előző bolygót, így a felhasználó annyit lát a modellben, hogy a bolygó folyamatosan áthelyeződik, követi az egeret, amíg le nem helyezi azt a megfelelő helyre a bal egérgombbal.

A „toggleLines” esemény a „showLines” flag értékét állítja be a bejövő adat szerint. Végigmegy az összes bolygón, amit a listába eltároltunk és visszavonjuk, vagy épp hozzáadjuk a jelenethez az elliptikus pályájukat.

A „keydown” egy olyan event, ahol a kamera közelítést és a távolodást kezeljük le az „event.data.key” alapján. Ez két értéket vesz fel, „ArrowUp” – ilyenkor közelítés történik, valamint az „ArrowDown” – ez a távolítást fogja véghezvinni egy step érték segítségével.

A következő pár bejövő esemény a bolygók adatait tartalmazza. Egy ilyen például az „earthData”. Itt egy bolygót inicializálunk, átvesszük az alapadatait, majd megadjuk az elliptikus pálya színét, elliptikus pályáját és a sebességét, majd az eljárás végén hozzáadjuk ezt az osztályszintű bolygók listájához.

Azt is kezelni kell, ha a főszálról új bolygó iránti kérelem érkezik. Ezt a „startAddingPlanet” kezeli. Itt a kezdeti „previewPlanet”-et fogjuk létrehozni és adjuk hozzá a „scene”-hez, amit később a felhasználó a fent említett módon egérmozgatással beállíthatja a bolygótól vett távolságát.

A törlésnél a „deletePlanet” hívódik meg. Megkapjuk a célbolygó nevét az „event.data.planetName” segítségével és töröljük a jelenetből és a bolygók listájából. A törlés előtt még, amennyiben a törlendő bolygón volt a kamera fókusza, akkor az új célbolygónk a Nap lesz. A bolygóval együtt a Naptól érkező reflektorfényt és az elliptikus pályát is töröljük. Ha a Földet töröljük, azzal együtt a Holdat is törli a kód.

Az utolsó eseménykezelés a „followPlanet”. Itt az előzőhöz képest szintén egy objektum nevét kapjuk meg. Lekérjük a bolygót a jelenetből és meghívjuk a „changeTargetPlanet” függvényt, hogy állítsa be az aktuális objektumunkat főfókuszra.

### Webworker integrálása Angularban

Az egyik leglényegesebb dolog, amivel egy jövőbeli fejezetben is fogom bizonyítani, hogy a program teljesítménye és erőforrásigénye valóban javulást mutatott az a Webworker beépítése Angularban. Ezzel rengeteg számítás, illetve programfutást lehet kiszervezni egy mellékszálra, ahelyett hogy a főszálat terhelnénk plusz funkciókkal.

Ehhez elsősorban a „\_workers” nevezetű mappába létre kell hozni a „threejs.worker.ts” fájlt. Majd a létrehozott projektben egy másik fájlt is be kell konfigurálni „tsconfig.worker.ts” néven. Ebbe a fájlba az 1. ábrán látható kódot kell belefoglalni. Majd az „angular.json” fájlon belül hozzá kellett adnom a következő sort a „projects/{projectname}/architect/build/option”-ön belül: "webWorkerTsConfig": "tsconfig.worker.json".

A képen szöveg, képernyőkép, Betűtípus, tervezés látható

Automatikusan generált leírás

9. ábra: worker importálása

Az összekötés html oldalon egy canvas-sel valósul meg, és az angular typescript oldalán inicializálni kell a workert, majd ott át lehet adni postMessage segítségével a canvast, illetve bizonyos eseménykezeléseket a hozzájuk tartozó paraméterekkel. Erről a 2. ábrán láthatjuk a példakódot.

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

10. ábra: worker meghívása a főszál indításakor

A workerben insideWorker-t és a THREE csomagot inicializálni kell elsősorban, majd az insideWorkeren belül kell a canvas-re hivatkozva megadni a rendereléssel kapcsolatos adatokat. A self.onmessage kóddal megkaphatjuk az eseményt megnevezve a megfelelő paramétereket, amint felhasználói interakció érkezik a főszálról, ahogy az a 3. ábrán látható.

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

11. ábra: esemény fogadása a mellékszálon

### DataService és az API forrás

A modell implementálása közben fontos volt, hogy a webapplikáció olyan adatokat szerezzen meg a futás elején, amely valódi adatokkal ábrázolja a bolygókhoz tartozó információkat, így még inkább valóssá téve a modellt.

Az Angular-on belüli data.service.ts fájl szolgál arra, hogy service layer-ként adatokat húzzon le API-ról és ezt továbbítsa azokhoz a metódusokhoz, ahol a service egyes függvényei meg lettek hívva. Ilyen metódusok az egyes bolygókhoz tartozó lekérések.

Az API viszont egy francia oldalról lett lekérve, ez egy publikusan és reális adatokkal elérhető végpont, amelyet bárki használhat céltól függetlenül. [https://api.le-systeme-solaire.net/rest.php/bodies/{planetName](https://api.le-systeme-solaire.net/rest.php/bodies/%7bplanetName)} címen érhető el „GET” metódus használatával és a „planetName” helyére kell írni a megfelelő bolygó nevét francia nyelven.

### MonitorService

A monitoring rész foglalkozik azzal, hogy a bejövő mérési adatok a megfelelő helyre el legyenek mentve és azt is beállítja, hogy mennyi ilyen adatot mentsen el egy időben. A régi adatokat célszerű törölni, hogy a memória ne legyen tele felesleges adatokkal, így a „maxDataPoints” adattag 60-ra lett állítva.

A konstruktor kezdetben törli az eddigi összes metrikát a „clearMetrics” metódus meghívásával, ami a lokális adattárból visszavonja a megfelelő névvel ellátott tömböt.

A „getMetrics” megszerzi az adattárból az összes logot és visszaadja a hívásnál. Ezt a függvényt két helyen is hívjuk ebben a fájlban. Az egyik a „logMetrics” ahol az eddig meglévő tömb végére illesztjük az új adatnégyesünket, beleértve az FPS, GPU, CPU és RAM adatokat, ha pedig túllépjük a fent megadott keretet, akkor az első adatokat shifteli a program, ami azt jelenti, hogy azokat eldobja és az egész tömböt elcsúsztatja egyel előrébb.

A másik hely, ahol a „getMetrics” metódust használja a program az az „exportToCSV”. Itt foglalom össze a több soros string fájlt, amit később egy fájllétrehozáskor belefűzzük az adatok közé. A metódus végén meghívódik a „clearMetrics”, hogy ezek után már friss adatokkal tudjon dolgozni a program és a felhasználó egyaránt.

## Optimalizáció

Mint azt a szakirodalmi összefoglalóban olvasni lehetett fontos, hogy a program minél jobban optimalizált legyen. Ez az egyik fő pillére a felhasználói élmény kialakításának a funkcionalitás mellett. Ha maga a webapplikáció optimalizálatlan lenne, vagy épp strukturálatlan és gyakran szakadozna, az a felhasználói bizalomba is kerülne. Ehhez meg kellett találni a megoldásokat és az olyan lépéseket, amelyek segítenek az optimalizációba. Mindezek után vizsgálatokkal és kutatási módszerekkel be kell bizonyítani, hogy a webapplikáció teljesítménye nőtt, míg erőforrásigénye fordítottan arányosan csökkent.

Ahogy azt a fejlesztői dokumentációban bemutattam, a „monitor.service.ts” pont ezt a feladatot hívatott ellátni és az általa előállított CSV fájl segítségével lehetőség nyílt mindezt grafikon formájában is szemléltetni.

A tesztelés 54 másodpercig futott változó terhelés mellett worker thread-el és anélkül három számítógép igénybevételével. Változó terhelés alatt azt érthetjük, hogy volt, amikor állt egyhelyben a kamera és figyelte az eseményeket, illetve olyan is, amikor erős felhasználói funkciók igénybevétele történt, vagy épp fókuszt váltottam egy másik bolygóra és pásztáztam a kamerával. Négy teljesítménybeli tényezőt mértem a kutatás során: a képkockák számát másodpercenként (FPS), memória-, videókártya-, illetve processzorhasználatot. Ezeket fogom most a következő fejezetekben részletesebben is elemezni.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Megnevezés | CPU | GPU | RAM | OS |
| Apple Macbook Air M1 | M1 chip | Integrált | 8 GB | macOS 15 |
| Számítógép 1 | Ryzen 5 5600G | Integrált | 32 GB | Windows 11 |
| Számítógép 2 | Ryzen 5 2600X | AMD RX570 | 16 GB | Windows 11 |

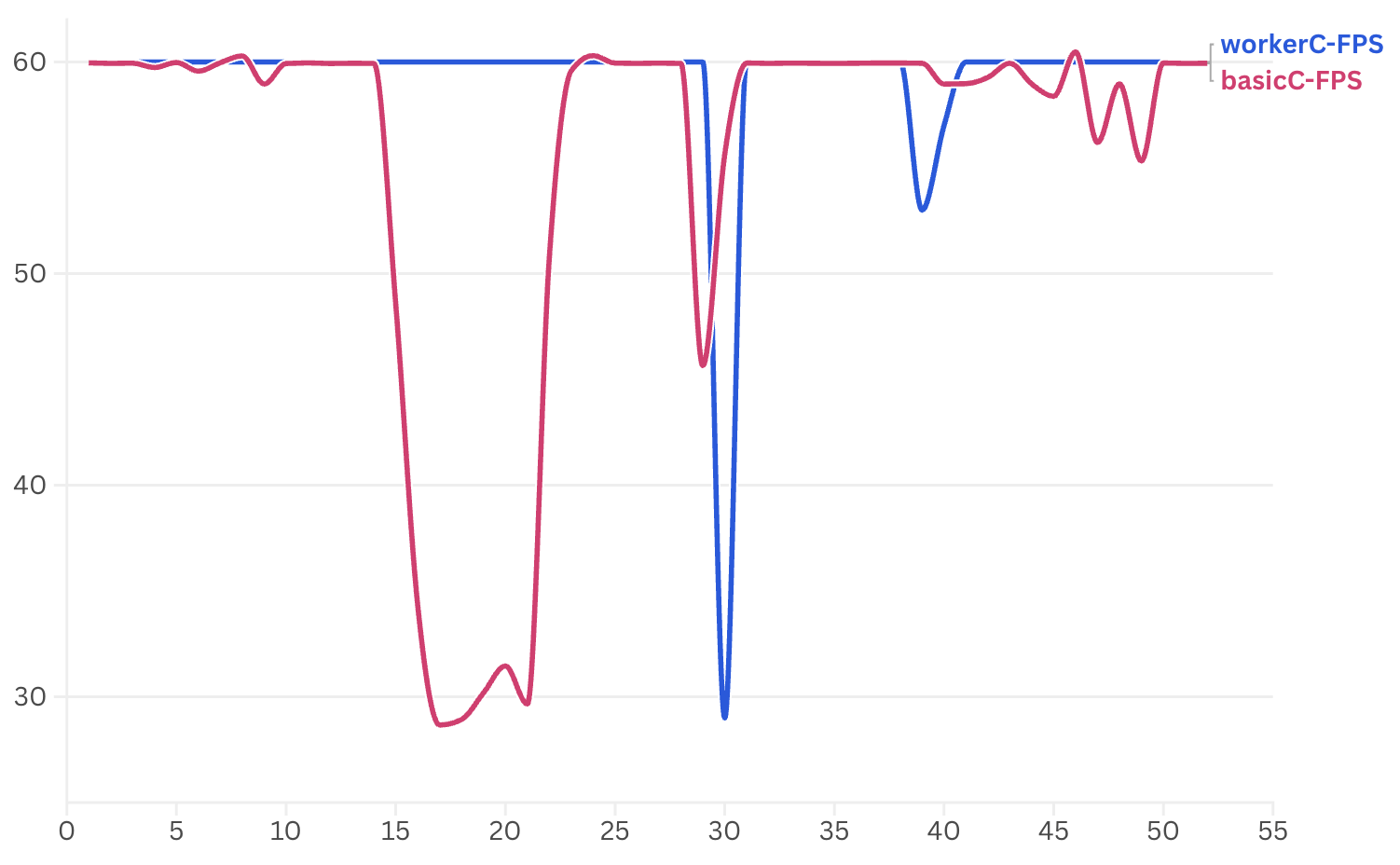
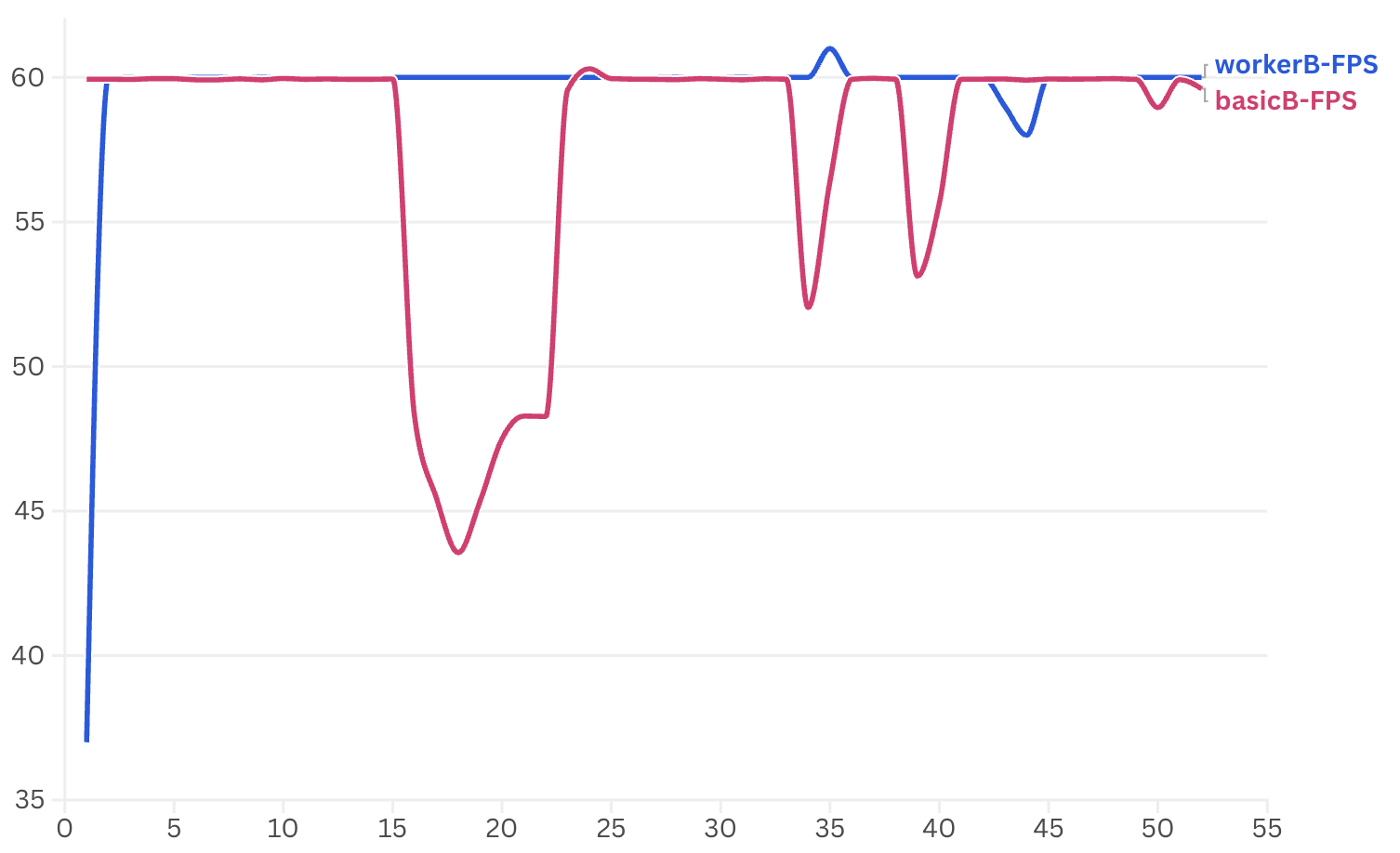
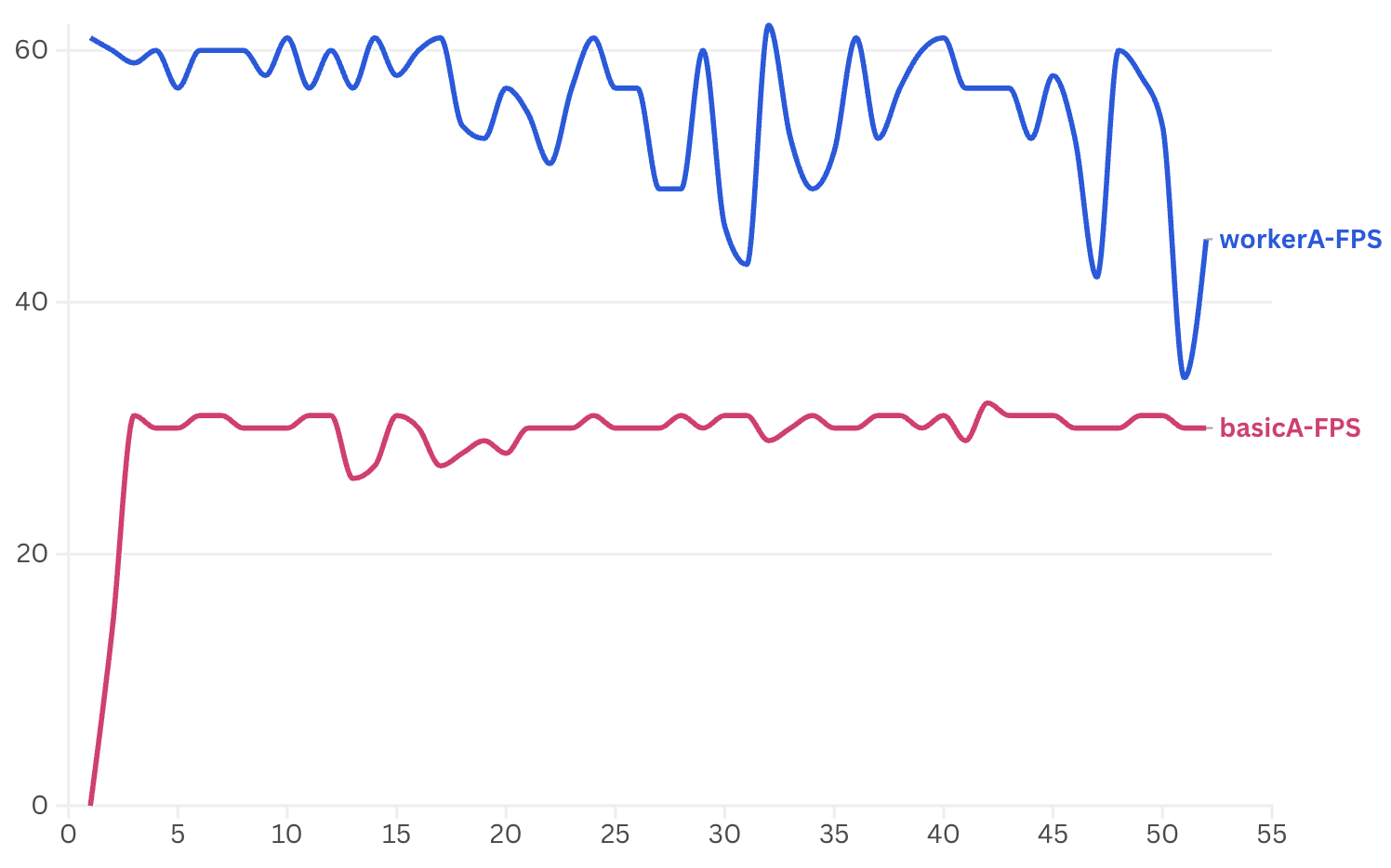
ábra 12: futtatott számítógépek adatai

### Frames per second

Az egyik legszembetűnőbb tényező egy háromdimenziós webes modellnél a képkockák száma másodpercenként. Szabad szemmel látható bármilyen kijelzőn, hogy mekkora különbség is van egy harminc, valamint egy hatvan képkocka per másodperc megjelenítés között, utóbbi sokkal kifinomultabbnak, simának tűnik.

Ahogy azt az eredmények is mutatják egy mellékszál létrehozása és a sok optimalizációs művelet rengeteget tud segíteni az elosztáson és egy stabilabb, jobb teljesítménnyel rendelkező webapplikáció tud megjelenni a felhasználó előtt. Worker szál nélkül az FPS számossága stabilan mozgott huszonöt és harmincöt között, az elején való képernyő betöltést leszámítva. Worker-rel átlagosan negyven és hatvan FPS között mozog a modell megjelenítése, azonban sokkal nagyobb kilengések vannak ellenben a mellékszállal nem rendelkező modellnél.

Az eredmények alapján kijelenthető, hogy a worker beépítése egy szükséges lépés volt a teljesítmény javítása érdekében.



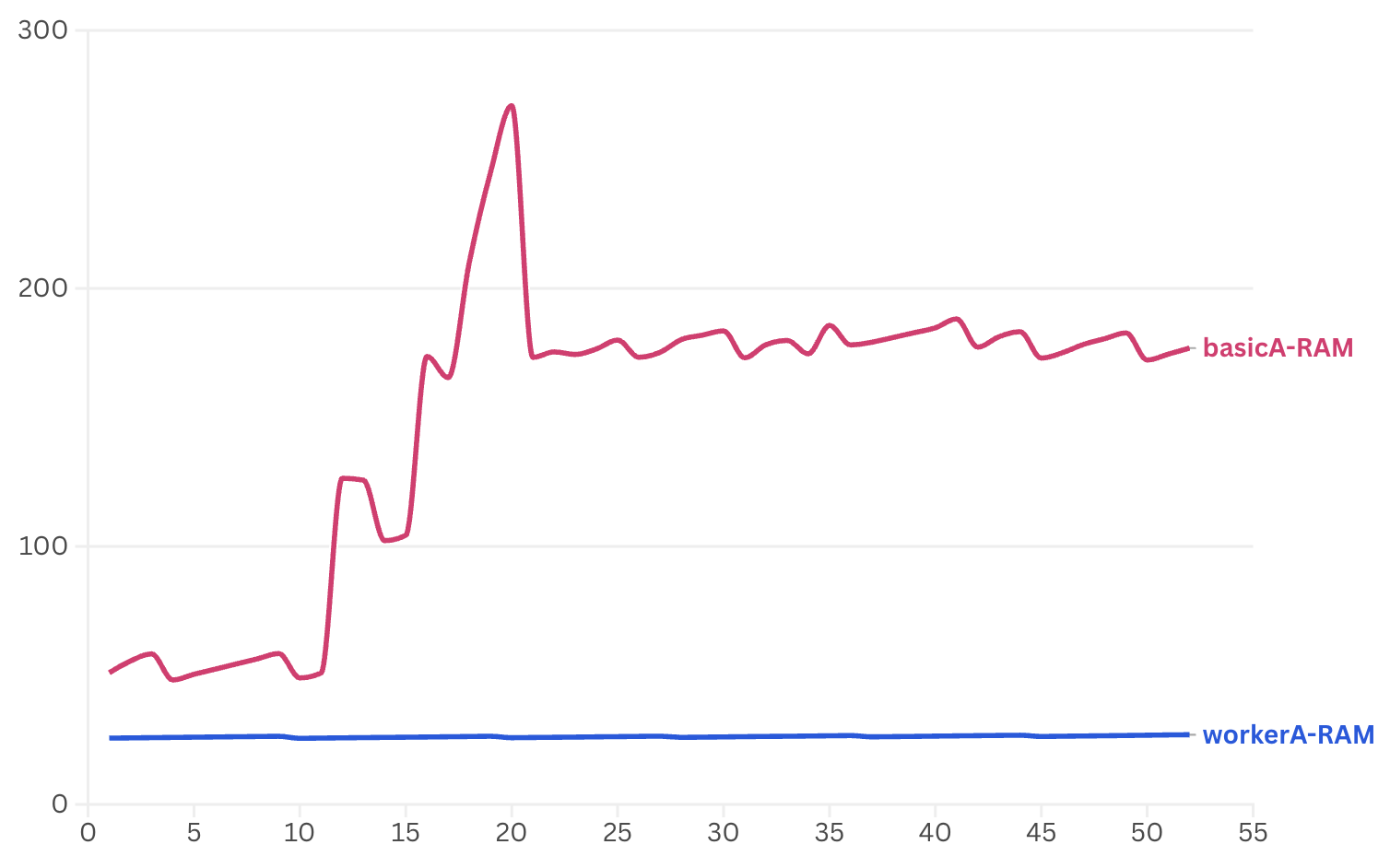
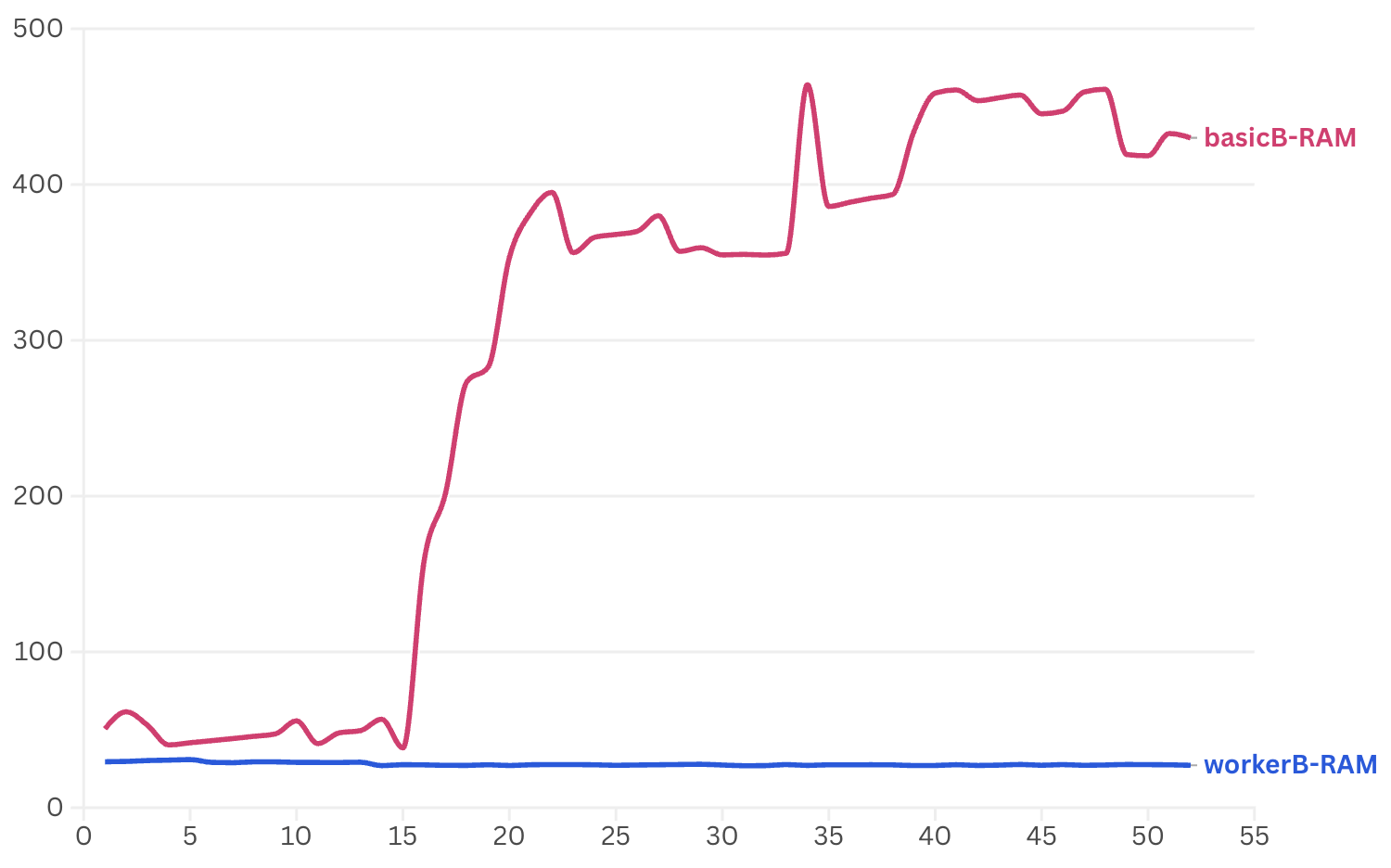
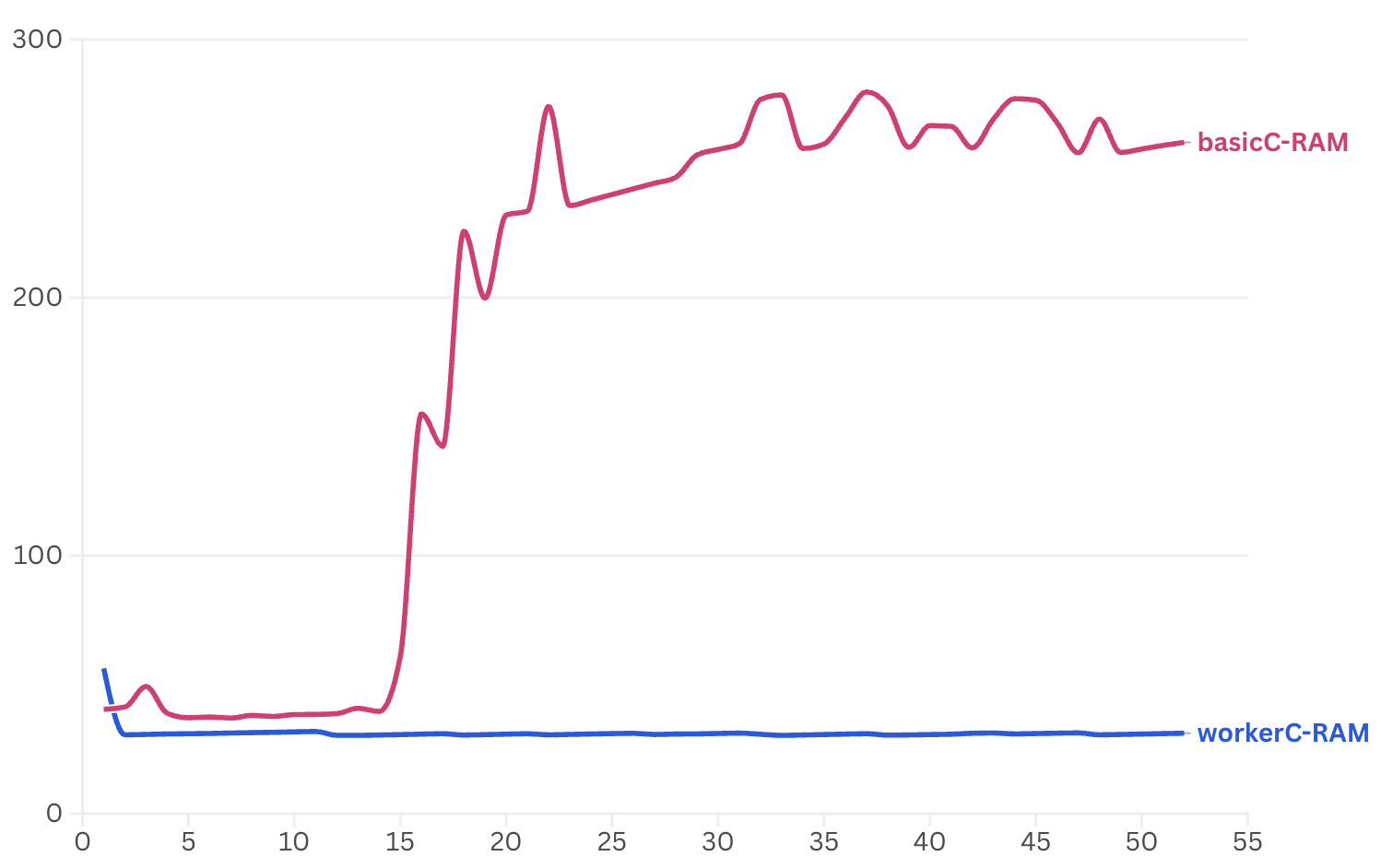
13. ábra: FPS worker thread-el és anélkül

### Memóriahasználat

Memóriahasználatot fontos figyelembe venni, ugyanis ez is befolyásolhatja a teljesítményét a programnak. Nem olyan szembetűnően, mint az FPS, de túlzott memóriaterhelés esetén főleg, ha több program is fut az applikáció mellett, leállhat a webapplikáció.

Ahogy a grafikon is mutatja, worker szál igénybevétele nélkül a memória folyamatos növekedést produkált egy kiugrással az elején egészen 270 MB-ig, majd ezek után stabilan mozgott a 170-180 MB-os tartományban. Átlagban 150 MB memóriára volt szüksége így a webapplikációnak. Worker thread beépítésével megoldódott, hogy a modellünk stabil memóriaigénybevétellel működjön, tartotta magát 26 MB körül.

Következésképp kijelenthető, hogy a mellékszál hatékonyan kezeli a memóriát, míg a worker nélküli verzió memóriaszivárgást mutat.

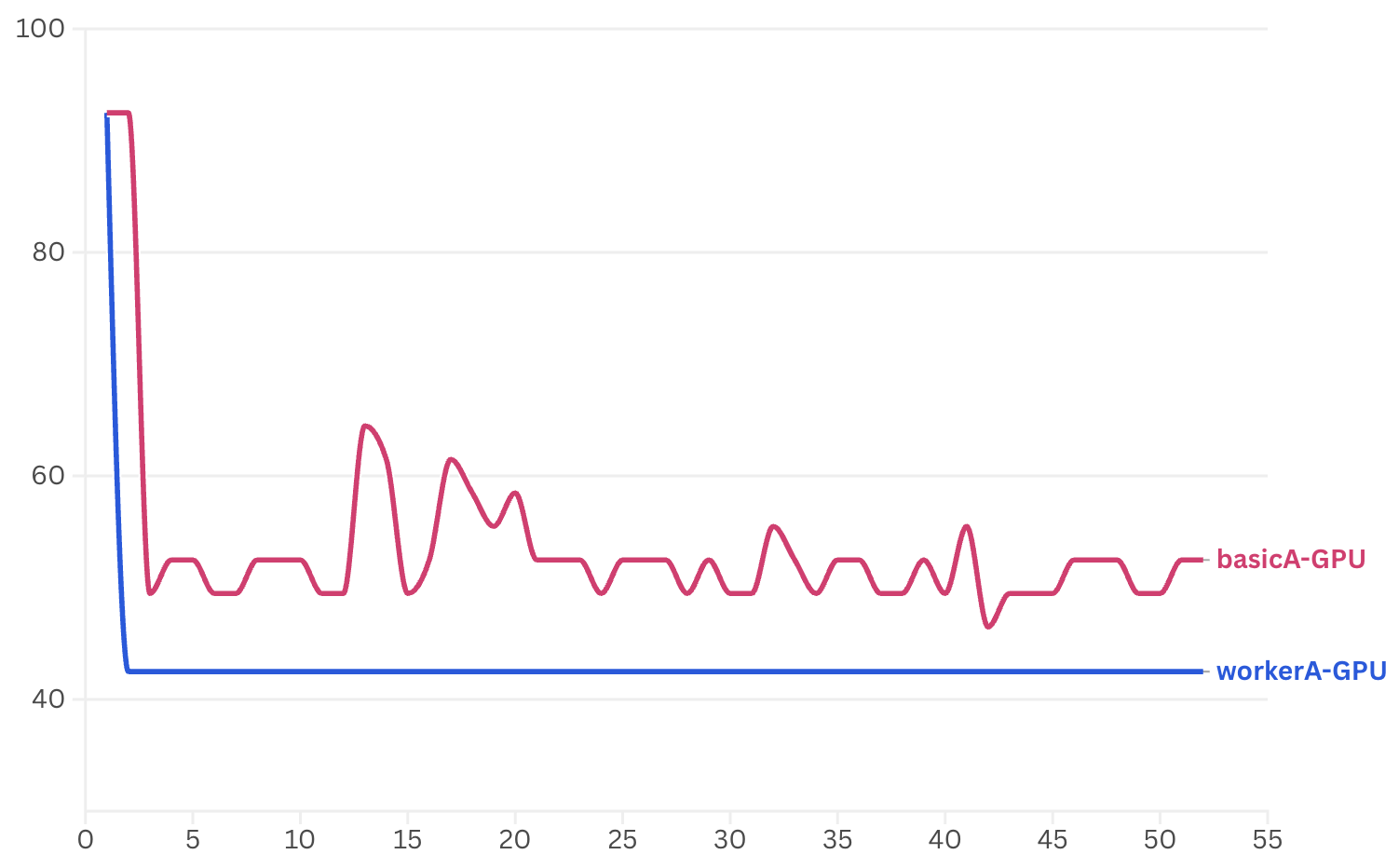
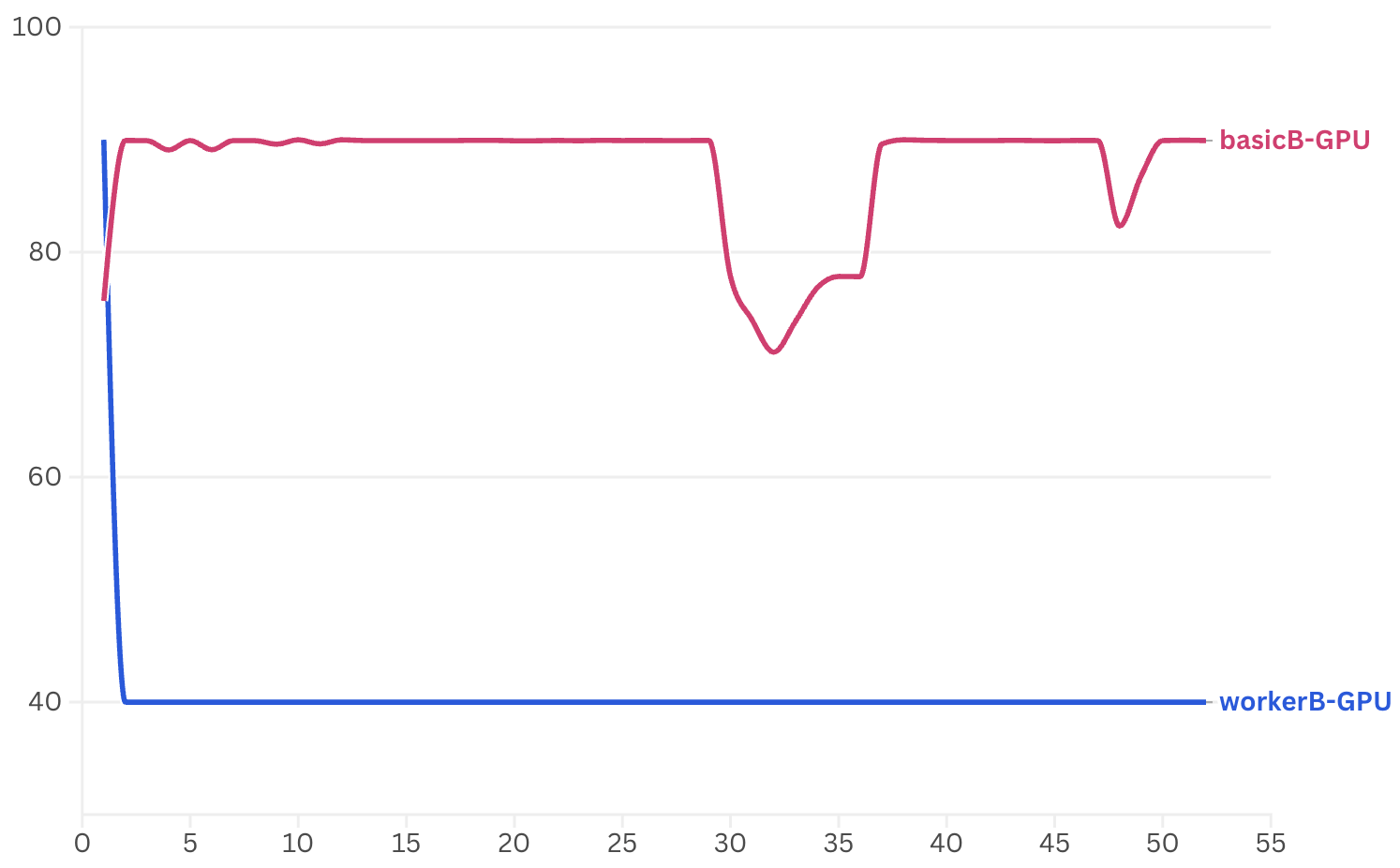
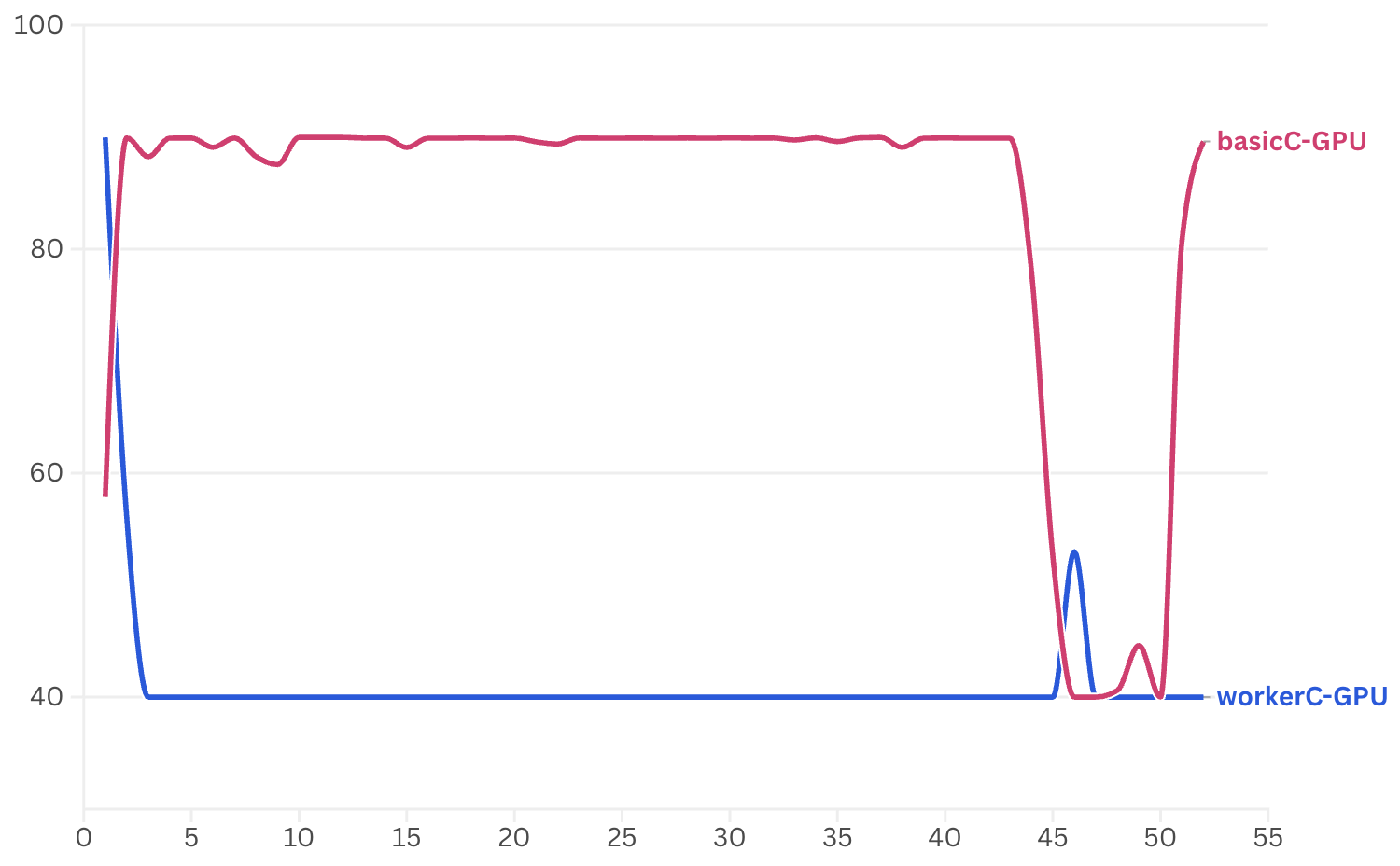
14. ábra: Memóriahasználat használat worker thread-el és anélkül

### GPU használat

A háromdimenziós modellezés egyik tulajdonsága, hogy a GPU az egyik legfontosabb eszköz arra, hogy a legjobb megjelenítést biztosítsa. Emiatt érdemes erről az eszközről is méréseket végezni, hogy láthatóvá váljon a mellékszál hatékonysága.

Worker thread nélkül 52% körüli volt a videókártya terhelése és elég ingadozó volt, az elejét leszámítva 44 és 64 százalék között mozgott. A mellékszál bevezetésével pedig egy elég stabil terhelést tudott mutatni az applikáció, a maga 42.5%-ával.

Szemmel látható, hogy worker thread segítségével a GPU terhelése alacsonyabb is és egyben stabilabb is.

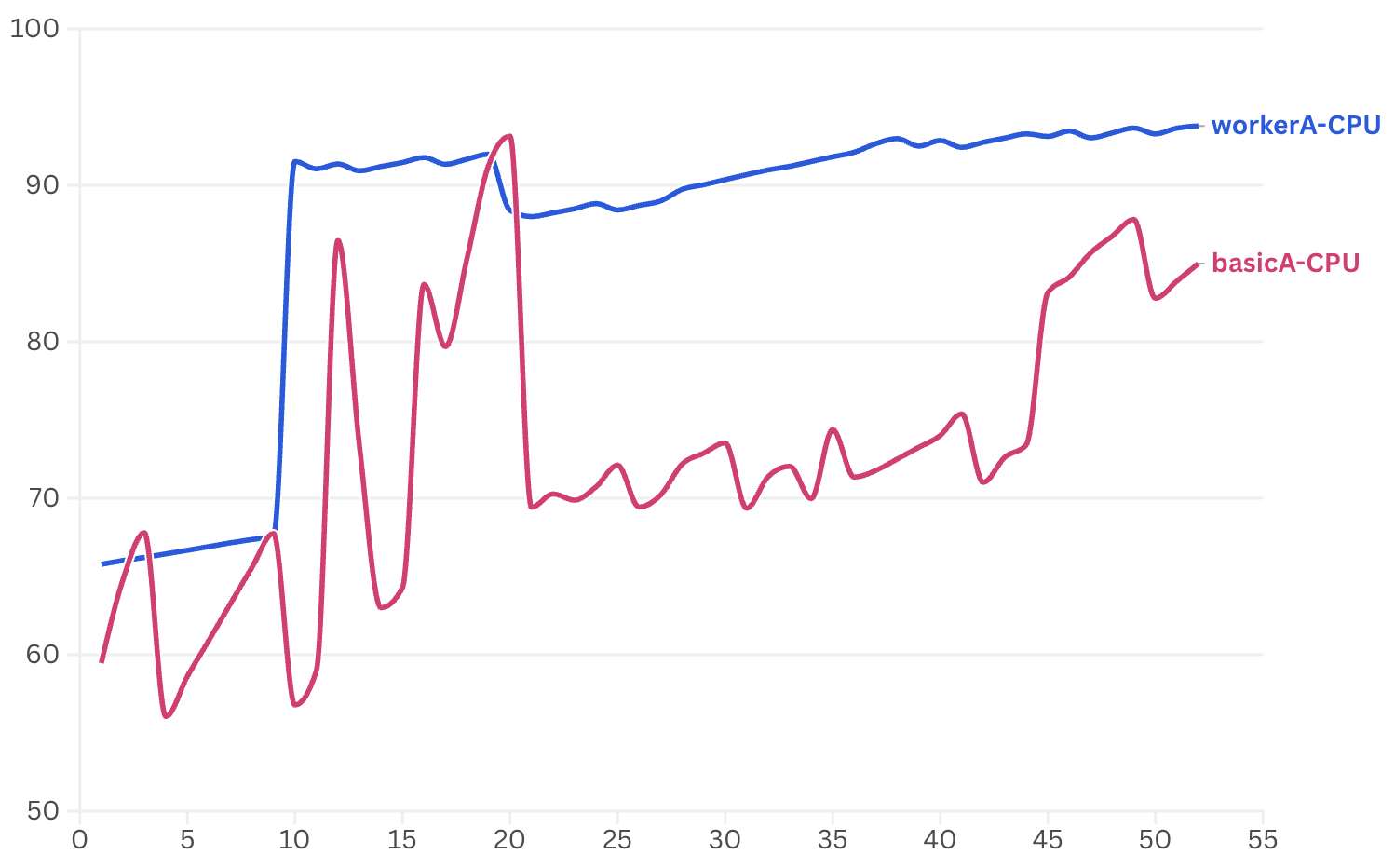
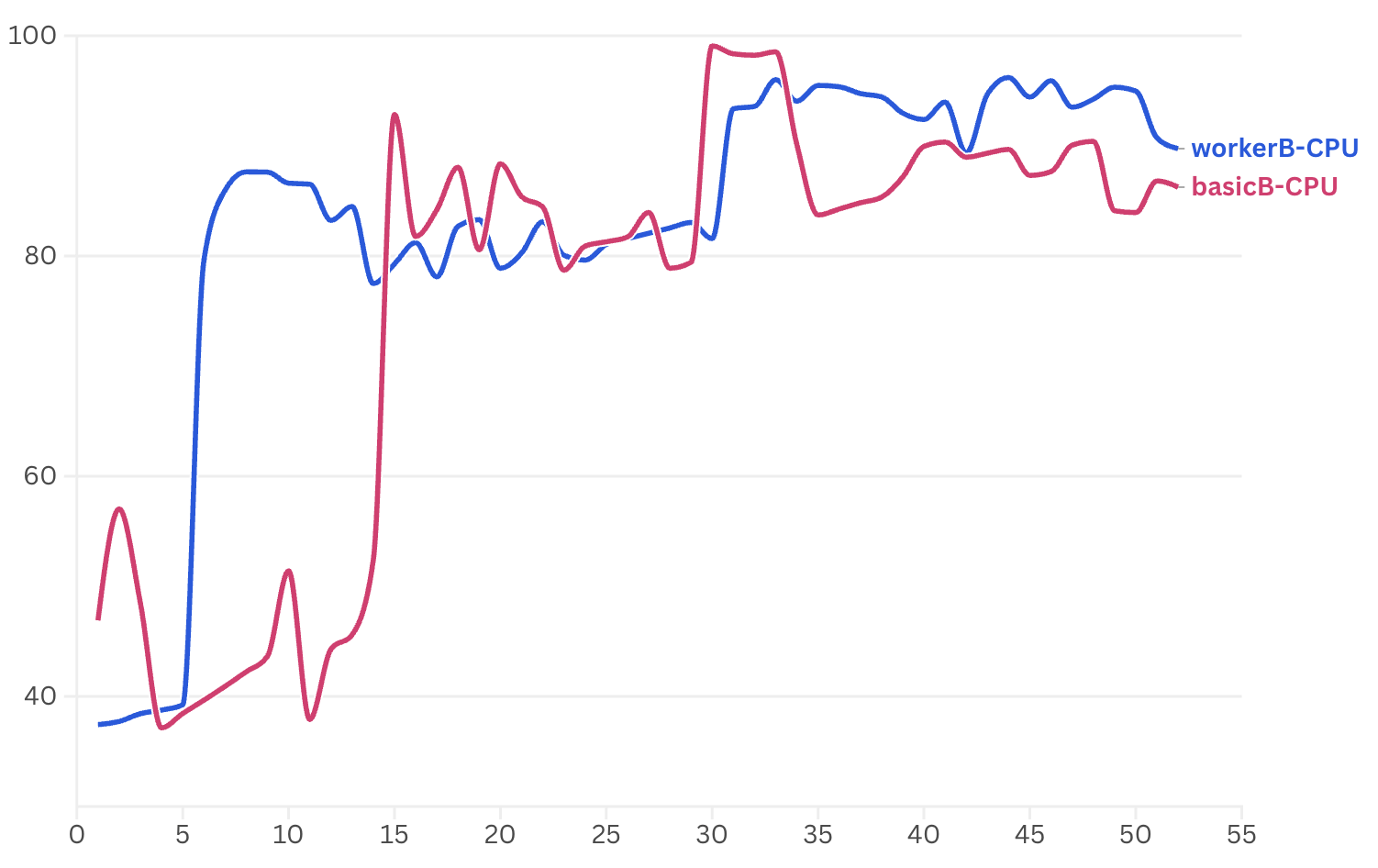
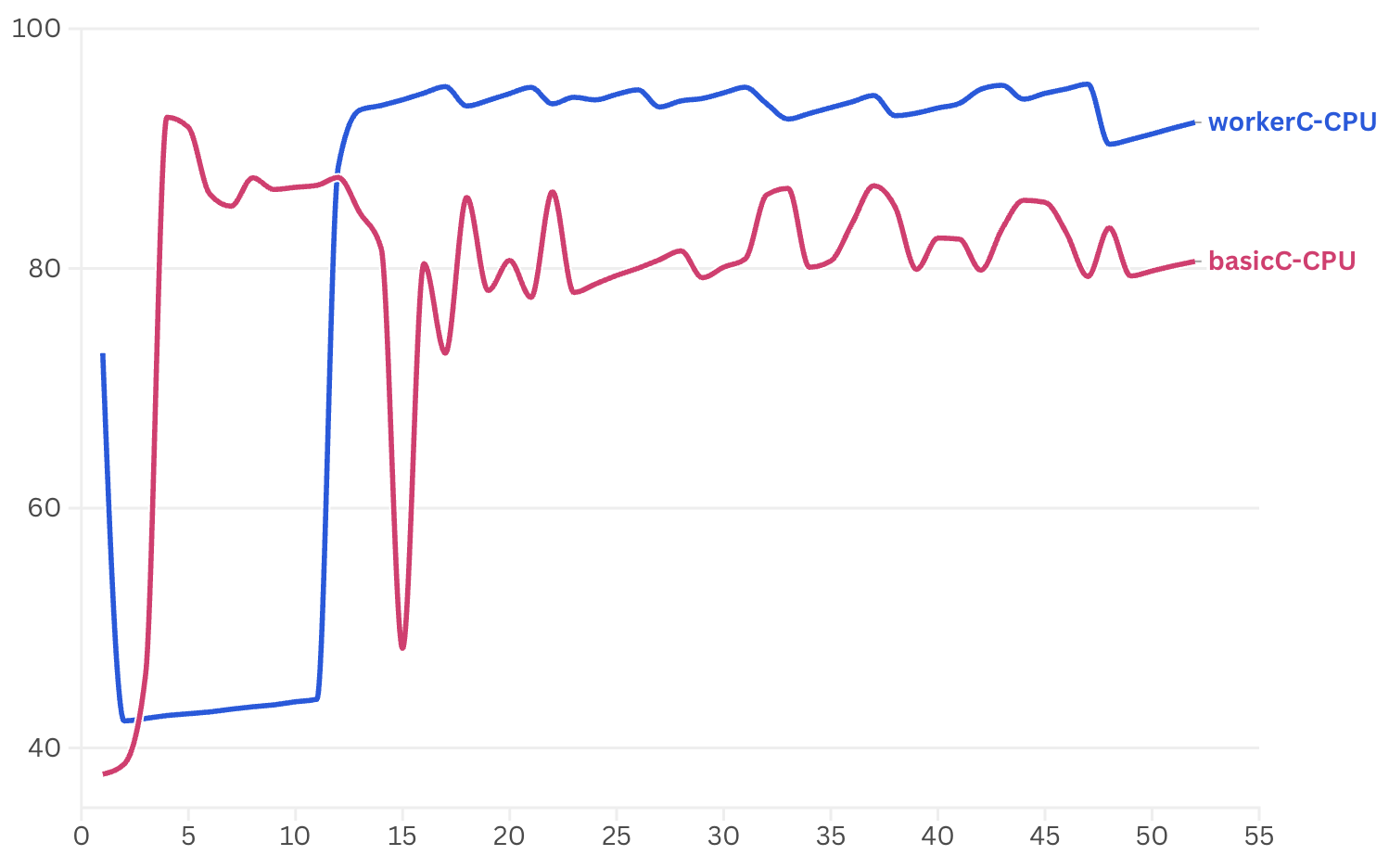
15. ábra: GPU használat worker thread-el és anélkül

### CPU használat

Ahogy azt a mindennapjainkban megtanulhattuk, mindennek megvan a maga ára. Azért, hogy minél jobb teljesítményt tudjunk elérni a modellben, valamit fel kell áldoznunk, vagy legalább más erőforrásigényt kell igénybe venni. Ez az elmélet szembe is tűnik a processzor használat elemzésénél is.

Az előző alfejezetekhez hasonlóan kezdjük a worker nélküli esettel. Egy helyen történtek a számítások, így nem is volt akkora igénybevétele a CPU-nak, ellenben a mellékszálas esettel, azonban néhol még így is meghaladta annak értékeit a tizenkilencedik másodperc környékén. Átlagosan 70% körül mozgott kisebb-nagyobb ingadozásokkal, attól függően, hogy épp milyen felhasználói interakciók történtek.

Mellékszál igénybevétele esetén átlagosan 80%-os CPU terhelés figyelhető meg. Itt is voltak ingadozások az elején, de később stabilizálódott a helyzet. A nagyobb terhelés a háttérben futó számítási folyamatoknak tudható be. De ahogy az előző három esetből (FPS, memóriahasználat, GPU terhelés) láttuk, még így is jobban megéri mellékszálon futtatni a számításokat a jobb teljesítmény reményében.

16. ábra: CPU használat worker thread-el és anélkül

## Telepítés

Ebben a fejezetben bemutatom, hogy hogyan kell ezt Node alapú Angular webapplikációt telepíteni és elindítani. Fontos ehhez, hogy a gépen telepítve legyen az előbb említett két technológia. Ezeket a <https://nodejs.org/en/download> és a <https://angular.dev/installation> linkeken találjuk meg. Ha ezek a programok fent vannak a célszámítógépen, akkor a telepítéshez be kell lépnünk a program mappájába és a konzolban és ki kell adni a következő két parancsot: npm install majd az alapcsomagok telepítésével futtatható a program az npm run start parancssal. Amennyiben a 4200-as port nem foglalt, így ott találhatjuk a webapplikációt.

## Továbbfejlesztési lehetőségek

// TODO: resize, elliptikus pályák módosítása, hőmérséklet megjelenítése,

# Összegzés

A téma teljesen más megvilágításba helyezte nekem az informatika világát. Sose foglalkoztam még ilyen komoly szinten háromdimenziós megjelenítéssel. Sose kutattam még ennyire mélyen, de a kutatásmódszertanon szerzett tapasztalatok is rengeteget tudtak segíteni.

# Forrásjegyzék

[1] Performance Issues and Optimizations in JavaScript: An Empirical Study. <https://software-lab.org/publications/icse2016-perf.pdf>, 14-22 May 2016

[2] Improvement of Model Simplification Algorithm Based on LOD and Implementation of WebGL. <https://ieeexplore.ieee.org/abstract/document/9182590>, 27-29 June 2020

[3] 3D Visualization in Furniture Ecommerce. [https://isjem.com/download/3d-visualization-in-furniture-ecommerce/#](https://isjem.com/download/3d-visualization-in-furniture-ecommerce/), 5 June 2024

[4] Using the Three.js library to develop remote physical laboratory to investigate diffraction. <https://ceur-ws.org/Vol-3662/paper23.pdf>, 16 Apr 2024

[5] BabylonJS and Three.js : Comparing performance when it comes to rendering Voronoi height maps in 3D. <https://www.diva-portal.org/smash/get/diva2:1228221/FULLTEXT01.pdf>, 2018

[6] The Solar System OpenData. <https://api.le-systeme-solaire.net/en/>, 2025

[7] Combine CSV. <https://combine-csv.de/>, 2025.04.15.

[8] Flourish. <https://app.flourish.studio>, 2025.04.15.

[9] Angular Interpolation. <https://docs.angular.lat/guide/interpolation>, 2025.04.23.

# Ábrajegyzék

[1. ábra: felhasználói felület 11](#_Toc196484892)

[2. ábra: mérési adatok 12](#_Toc196484893)

[3. ábra: menü lehetőségei 13](#_Toc196484894)

[4. ábra: generált CSV fájl tartalma 14](#_Toc196484895)

[5. ábra: új bolygó hozzáadása 14](#_Toc196484896)

[6. ábra: exobolygó kialakulása megfelelő környezetben 15](#_Toc196484897)

[ábra 7: bolygók listája 16](#_Toc196484898)

[ábra 8: Nap távolságától számított becsült felszíni hőmérséklet 26](#_Toc196484899)

[9. ábra: worker importálása 28](#_Toc196484900)

[10. ábra: worker meghívása a főszál indításakor 29](#_Toc196484901)

[11. ábra: esemény fogadása a mellékszálon 30](#_Toc196484902)

[ábra 12: futtatott számítógépek adatai 32](#_Toc196484903)

[13. ábra: FPS worker thread-el és anélkül 33](#_Toc196484904)

[14. ábra: Memóriahasználat használat worker thread-el és anélkül 34](#_Toc196484905)

[15. ábra: GPU használat worker thread-el és anélkül 35](#_Toc196484906)

[16. ábra: CPU használat worker thread-el és anélkül 36](#_Toc196484907)