

ELTE | IK
INFORMATIKAI KAR

Média- és Oktatásinformatikai Tanszék

Naprendszer Szimuláció Exobolygókkal

Nyíró Levente Gyula

Programtervező informatikus MSc

Belső témavezető:

Dr. Horváth Győző

Tanszékvezető egyetemi docens (ELTE IK)

Külső témavezető:

Zvara Zoltán

Programtervező informatikus MSc (ELTE IK)

Budapest, 2025

Tartalom

1. Bevezetés	4
2. Irodalmi áttekintés	6
2.1. Teljesítményproblémák és optimalizálás JavaScriptben	6
2.2. Webes 3D vizualizáció és kihívásai	7
2.3. Háromdimenziós vizualizáció egy bútor webáruházban	8
2.4. Fényhasználat optimalizálása háromdimenziós megjelenítésben	9
2.5. BabylonJS és ThreeJS összehasonlítás	10
3. Felhasználói dokumentáció	12
3.1. Teljesítménymérés	13
3.2. Menü	13
3.2.1. Vonalak megjelenítése opció	14
3.2.2. Metrikák exportálása	15
3.2.3. Új bolygó hozzáadása	15
3.2.4. Animáció megállítása vagy elindítása	17
3.2.5. Modell újragenerálása	17
3.2.6. Bolygók listájának a megjelenítése	17
4. Fejlesztői dokumentáció	19
4.1. Felhasznált technológiák	19
4.1.1. Angular keretrendszer	20
4.1.2. Planet adattípus	23
4.1.3. ThreeJS – WebGL alapú háromdimenziós renderelés	25
4.1.4. Webworker integrálása Angularban	33
4.1.5. DataService és az API forrás	35
4.1.6. MonitorService	36
4.2. Telepítés	36

5. Eredmények.....	38
5.1. Frames per second	39
5.1.1. Apple MacBook Air M1	39
5.1.2. Windows számítógép 1	40
5.1.3. Windows számítógép 2	40
5.2. Memóriahasználat	41
5.2.1. Apple MacBook Air M1	41
5.2.2. Windows számítógép 1	42
5.2.3. Windows számítógép 2	43
5.3. GPU-használat	44
5.3.1. Apple MacBook Air M1	44
5.3.2. Windows számítógép 1	44
5.3.3. Windows számítógép 2	45
5.4. CPU-használat	46
5.4.1. Apple MacBook Air M1	46
5.4.2. Windows számítógép 1	47
5.4.3. Windows számítógép 2	47
6. Jövőbeli fejlesztések	49
6.1. Továbbfejlesztési lehetőségek	49
6.2. További kutatási irányok	49
7. Konklúzió	51
Forrásjegyzék.....	53
Ábrajegyzék	55

1. 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 kutatást végeztem és ennek magyarázata. Ezek után az eredményeket fogom előadni, összefoglalom mások eredményeit, 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 szoftvertechnológia kurzusnak, ahol a kiválasztott projektmunkával foglalkozhat a hallgató több félév során átívelően és egy mentor is segít a projektben lévő 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 szempontjait nem szabad elhanyagolni. 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 [\[11\]](#) keretrendszerben készült és ebbe lett integrálva a ThreeJS [\[12\]](#) 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 [\[6\]](#) 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, a fent említett Naprendszer modellben megvalósításán keresztül. 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.

2. Irodalmi áttekintés

A ThreeJS és a háromdimenziós ábrázolás optimalizálása nem egy új téma a szakmában. 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 rendszernél. 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 [\[19\]](#). Volt, ahol a geometriák egyszerűsítésével és textúrák optimalizálásával foglalkoztak, mindezzel is csökkenthető volt a renderelési idő és a memóriahasználat [\[2\]](#). Egy másik gyakori megközelítés az instancedMesh. Ez lehetővé teszi, hogy több objektumot egyetlen renderelési hívással jelenítsünk meg, ezzel jelentősen csökkentve a CPU és GPU terhelését [\[13\]](#).

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

Ebben a fejezetben most egyes kutatások eredményeit és okait fogjuk áttekinteni, amelyek hatással voltak az általam bemutatott kutatásra.

2.1. Teljesítményproblémák és optimalizálás JavaScriptben

Az első bemutatott cikk egy 2016-ban megírt publikáció [\[1\]](#), amely még nem kapcsolódik a ThreeJS-hez. A háromdimenziós webes megjelenítés jelentős erőforrásigénye miatt elengedhetetlen az optimalizálás. Ezért különösen fontos, hogy JavaScriptben tiszta, átlátható kódot írjunk, és folyamatosan vizsgáljuk, hogy hogyan tudjuk a működést tovább finomítani. Ezt a tudást átültethetjük a ThreeJS síkjára is és így kaphatunk egy kielégítő teljesítménnyel futó programot.

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 ahhoz, 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 módosításával érték el a legjelentősebb teljesítményjavulást.

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.

A negyedik esetben az előző válaszban bemutatott optimalizációk konzisztens növekedési ütemét vizsgálták, azonban itt az eredmény mindössze a motorok teljesítményének 42,68%-át érte el.

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

2.2. Webes 3D vizualizáció és kihívásai

Ez a cikk [\[2\]](#) 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 valóság-hű élményt várnak. Azonban a webes környezetnek sajátos korlátjai és egyúttal kihívásai is vannak. Ezeket figyelembe kell venni a hatékony működés érdekében.

Fontos továbbá, 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 poligonszámának csökkentését, illetve a Level of Detail technika alkalmazását. Ennek a technikának a célja, hogy a 3D modellek részletességét a kamera távolságától függően automatikusan szabályozza. Ha a kamera távolabb van, egyszerűbb, kevesebb poligonból álló változat jelenik meg, míg közelebről részletesebb modell töltődik be.

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 [2]. 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.

2.3. Háromdimenziós vizualizáció egy bútor webáruházban

Ez a cikk [3] 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. A MERN egy népszerű webfejlesztési technológiai stack, amely a MongoDB (adatbázis), Express.js (backend keretrendszer), React (frontend könyvtár) és Node.js (JavaScript futtatókörnyezet) összességét jelenti. Ez a kombináció lehetővé teszi a teljes körű JavaScript-alapú fejlesztést a kliens- és szerveroldalon egyaránt. Ennek következménye, hogy a fejlesztés során létrejövő architektúra skálázhatóvá és modulárisává válik, mind frontend, backend és adatbázis szinten is. Ezzel részben elérhető, hogy a teljesítmény jobbá váljon kevesebb erőforrás igénybevételével. Ez is egy fontos szempont, hogy az ügyfélelégedettség növekedésnek induljon.

A kutatás kimutatta a fentebb említett technikák alkalmazásával 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.

2.4. Fényhasználat optimalizálása háromdimenziós megjelenítésben

Ebben a cikkben [\[4\]](#) a fényhasználat optimalizálását vizsgálják. Ez az én kutatásomban szintén egy fontos támaszpont, ugyanis a fénybeállítások általában több erőforrást tudnak igénybe venni, mert minden egyes fényforrás további számításokat követel a jelenet minden objektumára vonatkozóan. Továbbá számítást végez arra is, hogy milyen intenzitással, szögben és színnel éri a felületet, és árnyékot vet-e. Dinamikus fények esetén pedig minden képkockánál újraszámítás szükséges, ami jelentősen növeli a processzor- és memóriahasználatot, valamint a GPU terhelését. Ezen információk birtokában nagyobb optimalizációra van szükségünk a fényforrások tekintetében, 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 [\[4\]](#).

A fényforrások helyes beállítása elengedhetetlen a valósághű árnyékok, tükröződések és fényvisszaverődések létrehozásához [16]. 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 fényterheléseket 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.

2.5. 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 [5] 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 erőforrás szempontjából történtek a mérések: GPU, CPU és RAM használat.

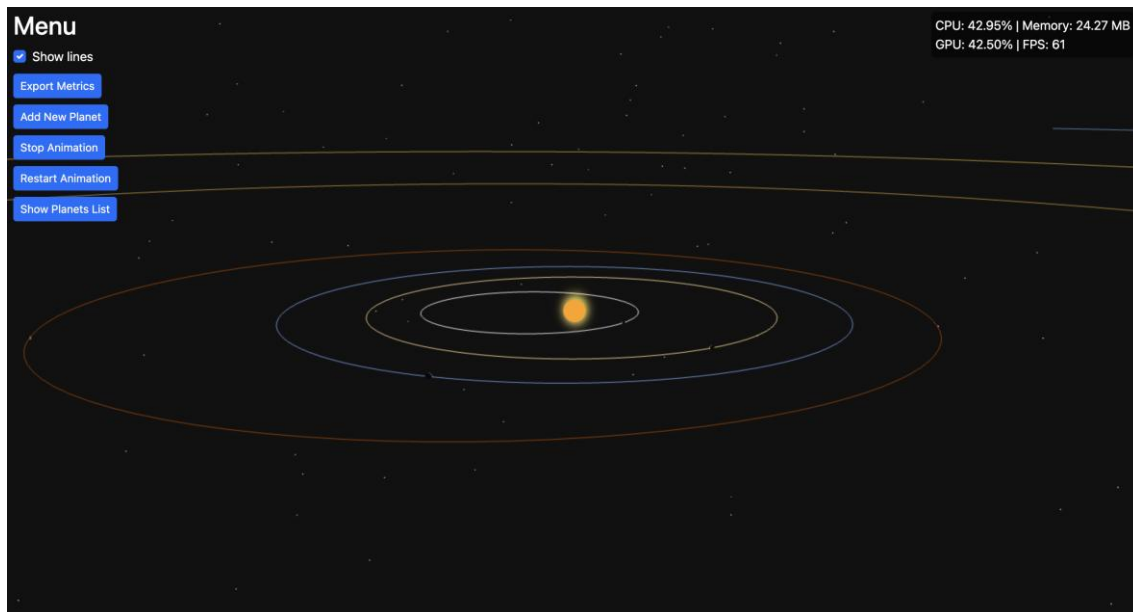
Módszertant figyelembe véve a tesztek konfiguráció tekintetében, Intel Core i7-6500U 2.5GHz x 4 CPU-val, Intel HD Graphics 520 és 8 GB RAM igénybevitelével készültek. Ubuntu rendszeren futott és NodeJS-en implementálták az automatizált

teszteket. A teszt során 10 különböző méretű és komplexitású térképet ugyanazzal a jelenettel 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 [\[14\]](#) alkalmaztak a különbségek szignifikanciájának vizsgálatára.

Eredmények tekintetében érdekes adatok kerültek napvilágra. GPU-terhelés szempontjából a ThreeJS sokkal jobb eredményeket mutatott, átlagosan 20-30%-kal kevesebbet a BabylonJS-hez képest. CPU-terhelésnél is a ThreeJS mutatott előnyt, viszont itt „csak” 15%-kal. 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 jobb futási időket tud produkálni szemben a BabylonJS-el. Utóbbi előnyére írható, hogy a fejlesztők nagy többségének véleménye alapján könnyebben kezelhető, egyszerűbb implementáció szempontjából, de a teljesítményhátránya nem elhanyagolható.

3. Felhasználói dokumentáció

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



1. ábra: felhasználói felület

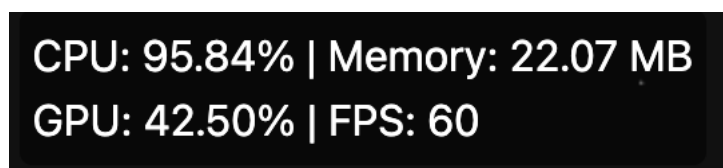
A webapplikációba belépve a felhasználó a fenti ábrán megjelenített oldalt találhatja. Itt rögtön több funkció is elérhető, ezeket a következő fejezetekben pontról pontra bemutatom. Az alap dolgok, amiket a felületen találhatunk, a menü a bal felső sarokban beállításokkal és funkciókkal, illetve a jobb 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 egérgommbbal, a program az adott bolygót kezdi el követni, valamint arra fókuszálni. Ez közelítésnél jó technika, amennyiben közelről szeretnénk megtekinteni az adott testet.

3.1. 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 név alatt a videokártya használatát jeleníti meg. 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.

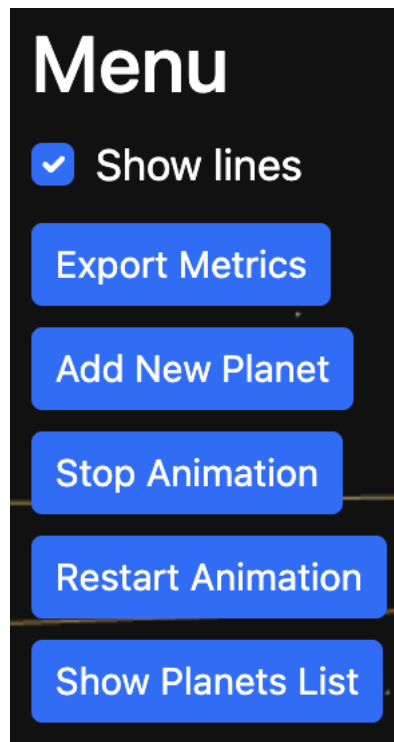


CPU: 95.84% | Memory: 22.07 MB
GPU: 42.50% | FPS: 60

2. ábra: mérési adatok

3.2. Menü

A program felső sarkában találhatjuk a főbb funkciókat, amelyekkel befolyásolhatjuk a Naprendszer modell működését, annak beállításait, hogy minél részletesebb képet kaphasson a felhasználó, valamint mérési adatokat is exportálhatunk a menüben található gombbal, hogy a ThreeJS-ben történő optimalizációval kapcsolatos kutatásokat elősegíthessük.



3. ábra: menü lehetőségei

3.2.1. Vonalak megjelenítése opció

A menürendszer egyik eleme a „Show lines” nevű jelölőnégyzet (checkbox), amely lehetőséget ad a felhasználónak arra, hogy tetszés szerint ki- vagy bekapcsolja a bolygók pályagörbéinek megjelenítését a szimulációban. Ez a funkció nem csupán egy vizuális beállítás, hanem egy tudatosan beépített eszköz, amely hozzájárul a felhasználói élmény és a kutatási szempontok közti egyensúly megteremtéséhez.

A pályavonalak megjelenítése – vagy más szóval az égitestek mozgási útjának vizualizálása – fontos szerepet tölt be a rendszer működésének megértésében, hiszen ezek segítségével egyértelműen láthatóvá válik, hogy az egyes bolygók milyen pályát követnek a Nap körül, milyen mértékben térnek el a tökéletes körpályától, és mi a relatív helyzetük más égitestekhez képest. Ugyanakkor a folyamatosan jelenlévő vonalak olykor zavaróvá válhatnak, különösen akkor, amikor a felhasználó közelebbről kívánja megvizsgálni egy-egy bolygó felszínét, felszínének textúráját vagy árnyékolási jellemzőit.

Ezt a kettősséget szem előtt tartva került bevezetésre a „Show lines” opció, amely lehetővé teszi, hogy a felhasználó maga dönthesse el, éppen mire szeretne fókuszálni. Ha a cél egy adott bolygó pályájának vagy mozgásának vizsgálata, akkor érdemes bekapcsolva hagyni a vonalakat. Ha viszont a cél inkább az adott égitest

részleteinek szemrevételezése, például a felszíni viszonyok, a világítás vagy a textúrák elemzése, akkor praktikusabb kikapcsolni őket, hogy a képernyőn ne jelenjenek meg zavaró vizuális elemek közel a célobjektumhoz.

3.2.2. Metrikák exportálása

A következőkben öt gomb lesz bemutatva, ebből az első az „Export Metrics”. A gomb 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.

```
Timestamp,CPU (%),Memory (MB),FPS,GPU (%)
1,43.27,23.97,0,90
2,44.4,24.59,30,49.99
3,46.09,25.53,60,40
4,79.76,22.35,60,40
5,80.69,22.62,60,40
6,81.66,22.89,60,40
7,83.96,22.27,60,40
```

4. ábra: generált CSV fájl tartalma

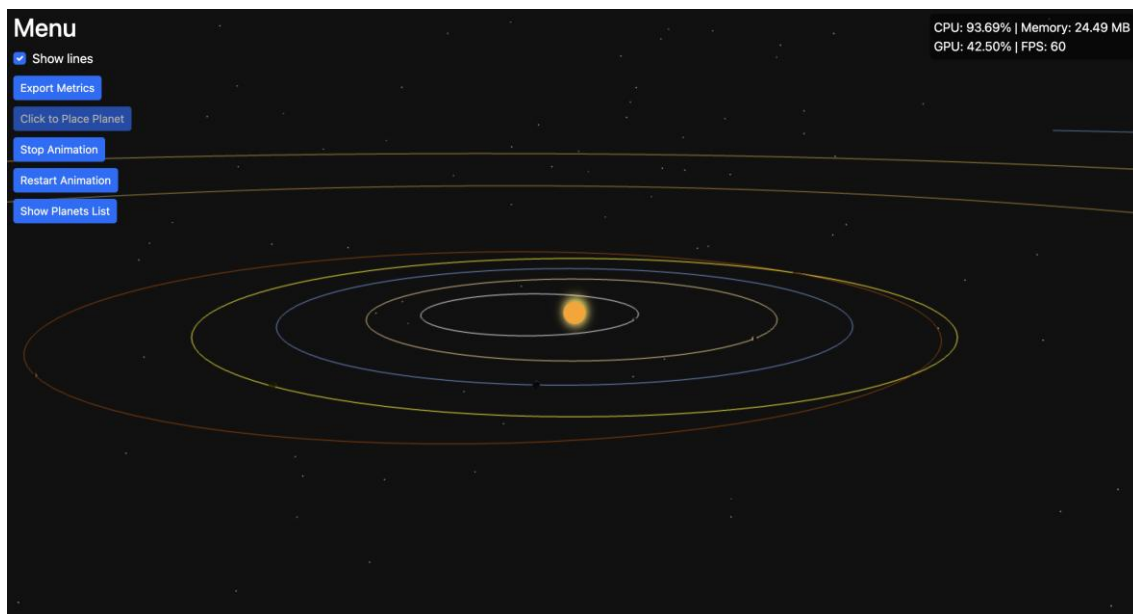
A fájl első sorában a fejléctet találhatjuk. A „timestamp” a mérés kezdetétől eltelt másodperceket mutatja, CPU a százalékos arányt, a memória MB-ot használ mutatóként. A negyedik oszlopban a képkockák száma találhatóak másodpercenként eltelve, míg az ötödik oszlopban a GPU használatot láthatjuk szintén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

3.2.3. Új bolygó hozzáadása

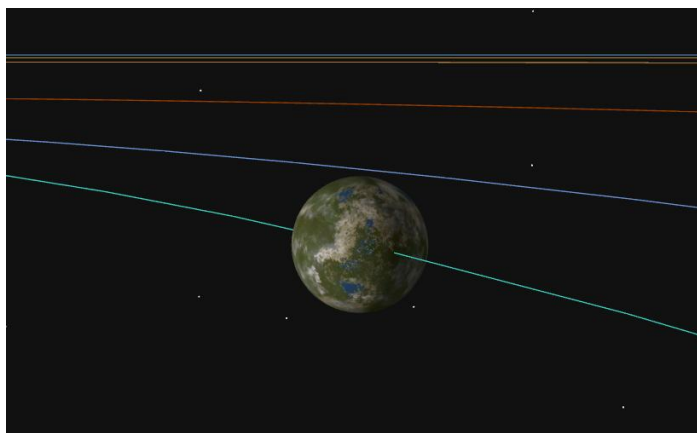
A második gomb „Add New Planet” névvel található meg. Ezzel adhatunk új exobolygót a modellhez. Ennek a gombnak a megnyomásával átkerülünk egy olyan fázisba, ahol már kötelező kiválasztanunk a bolygó pályáját az egér mozgásával. Amennyiben a bal egérgombra rányomunk, megbizonyosodva az általunk kiválasztott

pálya helyességéről, visszakerülünk a kiindulási fázisra a felhasználói kezelőfelület tekintetében és láthatóvá válik az új bolygónk a modellben.



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 – pontosan olyan távolságban, akkor kialakulhatnak rajta az exobolygók által uralt tulajdonságok, amelyek az élethez elengedhetetlenek. Ezt a projekt jelenlegi formájában csak a külső textúrája mutatja meg az objektumnak.



6. ábra: exobolygó generálása megfelelő környezetben

3.2.4. Animáció megállítása vagy elindítása

A szimuláció egyik funkciója az animáció futásának vezérlése, amelyet szintén a menüben helyeztem el egy egyszerűen kezelhető gomb formájában. A felhasználó ezzel a gombbal bármikor megállíthatja a bolygók mozgását, vagy újraindíthatja azt anélkül, hogy a program bármely más részébe bele kellene avatkoznia. Ez különösen hasznos akkor, ha valaki egy adott pillanatfelvételt szeretne alaposabban tanulmányozni, például megfigyelni egy bolygó adott helyzetét az elliptikus pályáján, vagy összehasonlítani több égitest pozícióját egy adott időpontban. A megállított állapotban lehetőség nyílik arra is, hogy a felhasználó szabadon mozogjon a 3D térben, és részletesebb vizsgálatokat végezzen. A vezérlés zökkenőmentes és azonnal reagál, így az animáció leállítása vagy elindítása nem töri meg a szimuláció folyamatosságát, hanem rugalmasan illeszkedik a felhasználói igényekhez.

3.2.5. Modell újragenerálása

Az animáció újraindítása egy olyan funkció, amely lehetővé teszi a felhasználó számára, hogy a szimulációt teljesen visszaállítsa a kezdeti állapotába, mintha újraindítaná az egész programot – de mindezt gyorsan, egyetlen kattintással. Ezt a lehetőséget azért építettem be, mert a Naprendszer mozgásának hosszabb távú megfigyelése során előfordulhat, hogy a bolygók elmozdulnak a kiindulási helyzetükből, és a felhasználó szeretné újratekinteni az elemzést, vagy egy adott vizsgálatot egy tiszta, rendezett indulóállapotból indítani.

Az újraindítás során minden égitest visszakerül az eredetileg definiált pozíciójába, a szimulációs időszámítás nullázódik, és a rendszer úgy viselkedik, mintha most indítottuk volna el először. Ez különösen hasznos például akkor, ha egy adott szituációt több különböző szempontból is szeretnénk elemezni: így nem kell manuálisan visszaállítani a bolygókat, vagy újra betölteni az alkalmazást, hanem egy gombnyomással megismételhető a teljes folyamat.

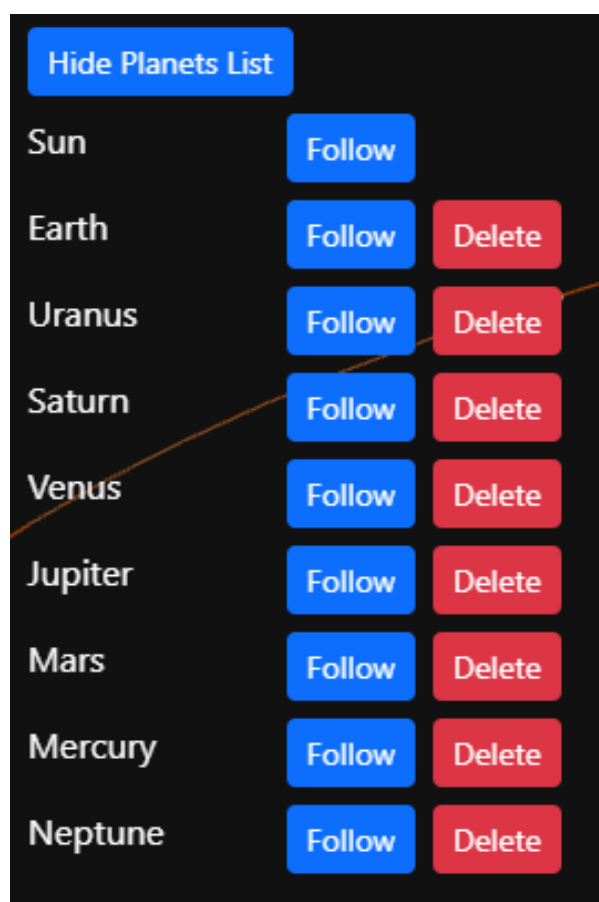
3.2.6. Bolygók listájának a megjelenítése

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 fókuszot 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 modellünkben. 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.



7. ábra: bolygók listája

4. Fejlesztői dokumentáció

A Naprendszer szimuláció megvalósítása során számos technológiai és architektúráis 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 [17], 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 kerülnek 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.

4.1. 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.

4.1.1. 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 újra felhasználását és könnyű karbantarthatóságát. Ez a keretrendszer általában két fontosabb részből tevődik össze. Az egyik a html oldal. Itt jelenítjük meg a formai dolgokat, DOM-mal kapcsolatos funkciókat, valamint jeleníthetünk meg a TypeScript oldalon létező adatokat interpoláció segítségével [9]. A másik oldal a TypeScript oldal, itt funkciók futtatása történik, vagy éppen változók értékeinek az eltárolása. 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, 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 készíti a TypeScript oldalt, hogy valamely adatot manipulálja. Ahhoz, hogy minél strukturáltabban nézzen ki a kód és kevés styling adatot kelljen hozzáadnom, a Bootstrap 5 hozzáadását tartottam észszerűnek az Angularhoz. Ezt a csomagot `npm install bootstrap` parancs segítségével kellett hozzáadni a programhoz, majd az „angular.json” fájlba bele kellett foglalni 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” fájlban van.

Az „app.component.ts” fájl szolgál arra, hogy az inicializálást elkezdje, elküldve az adatokat a worker felé és egyben betöltve az adatokat a DOM-ba, 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. A „startAnimation” flaget se szabad kifelejteni. Ez mutatja meg, hogy éppen fut-e az animáció a worker threaden vagy sem, illetve státusztól függően változhat a gombnak a kiírása is a html fájlban.

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()” metódus az Angular életciklusának egyik kulcsfontosságú része, amely az „OnInit” interfész implementálásával használható egy komponensben. Ez a metódus közvetlenül azután hívódik meg, hogy az Angular teljesen inicializálta a komponens összes adat-bound (azaz @Input() dekorátorral ellátott) tulajdonságát, de még azelőtt, hogy a felhasználó bármilyen módon interakcióba léphetne az oldallal. Gyakorlatilag ez az a pont, ahol a komponens már rendelkezik a szükséges bemeneti adatokkal, és így biztonságosan elkezdhetjük az inicializáláshoz szükséges logikát végrehajtani. A mi esetünkben ebben metódusban történik meg két alap folyamat, az egyik a „canvas”-t inicializálja, hogy a későbbiekben már egy kész képen dolgozhassanak a funkciók, a másik pedig a mérési adatokat készíti elő és inicializálja azokat a „monitorSystemStats” függvény keretein belül.

Ezek után fontos, hogy olyan dolgok is lefussanak, amelyeknek be kell várniuk a kezdeti inicializációs lépéseket 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 nyilakat, 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. Az adatok lekérései is itt futnak le, a „data.service.ts”-en keresztül minden egyes bolygót lekérve, amiket a rendszer a program elején létre is hoz. 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.

A privát hozzáférési szintű „monitorSystemStats” feladata, hogy kiszámolja és elmenti a fent említett változóba, hogy az adott másodpercben mekkora volt az egyes fizikai komponensek 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 előzőekhez 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 osztályszintű 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. Az „ngOnDestroy” [\[15\]](#), hasonlóan az inicializáláshoz az Angular egyik életciklus-metódusa, amely akkor hívódik meg, amikor a komponens vagy szolgáltatást elpusztítja az Angular, például amikor a felhasználó elnavigál egy másik oldalra, és az adott komponens kikerül a DOM-ból. A fő célja, hogy leállítsuk azokat a folyamatokat, amelyek a komponens élettartamához kötődnek. Amennyiben ezt a lépést kihagyjuk, akkor memóriaszivárgás léphet fel, mivel a komponens elpusztult, de bizonyos háttér folyamatok (pl. HTTP-polling, WebSocket kapcsolat vagy időzítő) tovább futnak.

4.1.2. 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 megkötvé, 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 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”, azaz Naptávolság és „aphelion”, azaz Naptávolság 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 „siderealOrbit” 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

$$mass.value \times 10^{mass.exponent}$$

A Föld esetében ez a két érték 5.972 és 24.

4.1.3. ThreeJS – WebGL alapú háromdimenziós renderelés

A projektben, ahogy azt már a bevezetőben bemutatam, 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 [\[18\]](#).

A fájl elején találhatjuk az importokat és az inicializációkat. Itt vesszük át a főszáltól az „OffscreenCanvas”-t „canvas” néven. Az „OffscreenCanvas” egy böngésző API, amely lehetővé teszi, hogy egy „canvas” elem rajzolását a főszálon kívül, például egy Web Workerben végezzük, ezzel csökkentve a főszál terhelését és javítva az alkalmazás teljesítményét. Ez különösen hasznos nagy számításigényű grafikus műveleteknél, mint például a háromdimenziós renderelés vagy a valós idejű vizualizáció.

Maga a renderer is itt kerül kialakításra a háttérszínével és az árnyékbeállításokkal együtt. A renderer egy olyan objektum (pl. WebGLRenderer a Three.js-ben), amelynek feladata, hogy a jelenetet (Scene) és a kamerát (Camera) összeolvasztva ténylegesen képpé alakítsa a háromdimenziós világot a képernyőn.

A scene-t is létrehozuk, majd ezután jönnek a kamerával kapcsolatos beállítások. Ez tulajdonképpen a Three.js-ben egy konténer, amelyben az összes háromdimenziós objektumot, mint például a világítást (lightot), elemeket és a kamerát is tárolni lehet.

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 nagyobbak. 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-tal oldottam meg és ennek az értékét minimálisra állítottam, hogy minél valóságosabbabb 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 felvértézve. 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 (`showLines`), vagy éppen fut-e az animáció az elvártnak megfelelően (`runningAnimation`), hogy éppen a felhasználó húzza-e az egeret, mert pásztázni akar a kamerával (`isDragging`). Olyan változó is van, ami azt jelöli ki, hogy épp mi az aktuális objektum, amit a kamerának követnie kell (`targetObject`), de vannak olyan konstans adatok is, amelyek a bolygóközi távolságok arányát (`DISTANCE_DIVIDER`), vagy épp az animáció sebességét hívatott meghatározni (`ANIMATION_SPEED`).

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 [\[21\]](#).

$$\begin{cases} x = r \sin(\phi) \cos(\theta) \\ y = r \sin(\phi) \sin(\theta) \\ z = r \cos(\phi) \end{cases}$$

ahol

r a gömb sugara.

ϕ a magassági szög $[0, \pi]$.

θ a horizontális szög $[0, 2\pi]$:

Ahhoz, hogy létrehozassunk új elliptikus vonalakat vissza kell adnunk egy ThreeJS vonalat, amit a „createOrbitLine” metódusban generálunk a következő egyenlettel [20]:

$$r(\theta) = \frac{a(1 - e^2)}{1 + e \cos(\theta)}$$

ahol

$r(\theta)$ az adott szögű távolságot jelenti.

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

„e” az excentricitás.

θ az aktuális szög, amely $0 \leq \theta \leq 2\pi$ 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 bolygónak a modellben. A biztonság kedvéért a lambda-ban 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 készítettem egy segédfüggvényt, ami meghatározza annak távolságát a Naptól. Így kialakítja a program a megfelelő körpályát a bolygó számára a Naptól való távolság és a távolsági arányosság szorzatában (DISTANC_DIVIDER). 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.

$(d_1, t_1) = (57475497.74, 167)$ – Merkúr

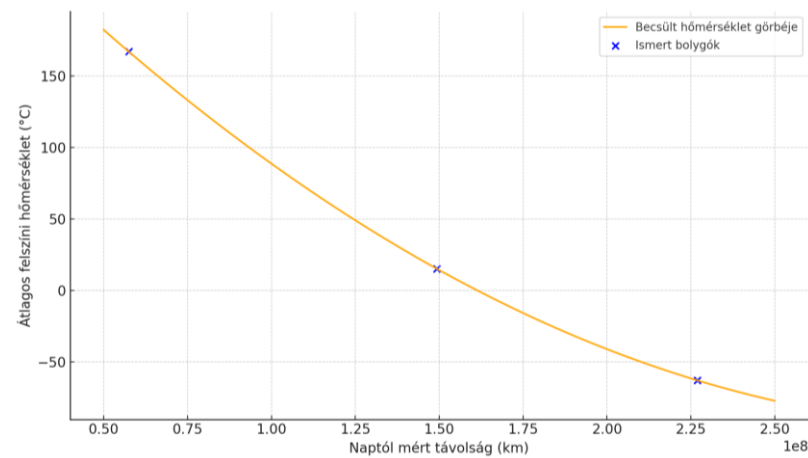
$(d_2, t_2) = (149339835.23, 15)$ – Föld

$(d_3, t_3) = (226914680.3, -63)$ – Mars

A távolsághoz tartozó értéket Lagrange-féle interpolációs polinom [\[10\]](#) formában lehet a legkönnyebben megkapni:

$$T(x) = t_1 \frac{(d - d_2)(d - d_3)}{(d_1 - d_2)(d_1 - d_3)} + t_2 \frac{(d - d_1)(d - d_3)}{(d_2 - d_1)(d_2 - d_3)} + t_3 \frac{(d - d_1)(d - d_2)}{(d_3 - d_1)(d_3 - d_2)}$$

A távolságot az x helyére beillesztve pedig könnyedén megkaphatjuk az újonnan generált bolygó hőmérsékletét. Polinomként ábrázolva ez a következőképpen néz ki:



8. ábra: 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 félnagy tengelyét. 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 bitmápekké 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 kód kiszervezésnek köszönhetően sikerült lerövidíteni a programkódot és gyorsabbá tenni a programot. Eddig minden bolygó tulajdonságát külön-külön kellett kialakítani, azonban most ez másképp történik. Az első sorokban a Holdnak állítjuk be a haladási irányát és a pozícióját annak „mesh” adattagján keresztül, majd a bolygók listáján is végig kell iterálni, hogy mindegyiknek újra állítsunk a pozícióját a „setPlanetPosition” függvény segítségével. Azonban ezek kódok csak akkor futnak le, ha az animáció futtatása folyamatban, azaz a „animationRunning” igaz értékkel rendelkezik. A többi kód minden esetben lefut, így a kamerakezelést, és az FPS számlálást gond nélkül tudja a program kezelni. Az „FPS” kiszámolásánál, a metódus azonnal is küldi vissza az adatot a főszálnak, hogy az adatokat elmentse és abból kalkulációkat végezzen. Majd a kijelölt bolygónkat is követnünk kell, erre camera.lookAt függvényét hívjuk meg. Finomítani kellett a kamerakezelésen a „lerp” technológia segítségével is.

A lerp (linear interpolation, azaz lineáris interpoláció) Three.js-ben azt jelenti, hogy két érték (például pozíciók, színek vagy más vektorok) között simán, egyenletesen mozgatunk vagy váltunk egy köztes értéken keresztül. Mozgásnál például, ha van egy A és egy B pont, akkor a lerp a kettő között egy arányosan kiszámolt köztes pozíciót ad vissza – nem azonnal ugrik át B-re, hanem szépen, fokozatosan közelíti meg. Az [9. ábrán](#) látható kódrészletben 5%-kal közelíti meg a TargetPosition-t minden egyes frame-ben.

Az utolsó két sor a kódban kirajzolja a jelenetet a megadott kameranézetből a képernyőre és meghívja az „animate” függvényt a következő képkockánál, ezáltal létrehozva egy folyamatos és hatékony animációs ciklust.

```

function animate() {
  if (animationRunning) {
    const earth = getPlanetByName('earth');
    if (earth && moon) {
      moon.mesh.angle += moon.data!.speed;
      moon.mesh.position.x = earth.mesh.position.x + Math.sin(moon.mesh.angle) * moon.data!.distance!;
      moon.mesh.position.z = earth.mesh.position.z + Math.cos(moon.mesh.angle) * moon.data!.distance!;
    }

    planets.forEach(planet => {
      setPlanetPosition(planet);
    });
  }

  // fps counting
  const nowFps = performance.now();
  frameCount++;

  if (nowFps - lastFpsUpdate >= 1000) {
    fps = frameCount;
    frameCount = 0;
    lastFpsUpdate = nowFps;
    postMessage({ type: 'fps', fps: fps });
  }

  if (targetObject) {
    camera.lookAt(targetObject.position);
  }

  camera.position.lerp(cameraTargetPosition, 0.05);

  renderer.render(scene, camera);
  requestAnimationFrame(animate);
}

```

9. ábra: kódrészlet a Worker szálról

Az utolsó blokk fogadja a user interakciókat és a főszálról érkező adatokat is. A főszálról „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.

Egér baloldali gombjának elengedésére 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 főszárról új bolygó iránti kérés 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 következő 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.

Az utolsó eseménykezelés pedig a „toggleAnimation”. Ennek a szerepe, hogy az animációt a bolygók mozgását és forgását tekintve megálljanak. Ehhez átállítja az osztályszintű „animationRunning” adattagot a bejövő paraméter függvényében.

4.1.4. 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 terhelnék plusz funkciókkal.

Ehhez elsősorban létre kell hozni a „threejs.worker.ts” fájlt. Majd a létrehozott projektben egy másik fájlt is be kell konfigurálni „tsconfig.worker.ts” néven. Ebben a fájlban a [10. ábrán](#) látható kódot kell belefoglalni. A tsconfig.worker.json fájl célja, hogy külön konfigurációt biztosítson a Web Worker környezetre fordított TypeScript-kódhoz, így az elkülönülve kezelhető az Angular alkalmazás fő konfigurációjától.

Az „angular.json” fájlban belül hozzá kellett adnom a következő sort a „projects/{projectname}/architect/build/option”-ön belül: "webWorkerTsConfig": "tsconfig.worker.json".

```
{
  "extends": "./tsconfig.json",
  "compilerOptions": {
    "outDir": "./out-tsc/worker",
    "types": [
      "node"
    ],
    "lib": [
      "es2020",
      "webworker"
    ]
  },
  "include": [
    "src/**/*.worker.ts"
  ]
}
```

10. 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 [11. ábrán](#) láthatjuk a példakódot.

```

ngAfterViewInit() {
  this.worker = new Worker(new URL('src/app/_workers/threejs.worker.ts', import.meta.url)
  //...

  var hasOffscreenSupport = !!htmlCanvas.transferControlToOffscreen;
  if (hasOffscreenSupport) {
    var offscreen = htmlCanvas.transferControlToOffscreen() as any;

    this.worker.postMessage({ canvas: offscreen }, [offscreen]);

    // send random event

    htmlCanvas.addEventListener('mousemove', (event: any) => {
      if (this.worker) {
        this.worker.postMessage({
          type: 'mousemove',
          mouseX: event.clientX,
          mouseY: event.clientY
        });
      }
    });
  }
}

```

11. ábra: worker meghívása a főszál indításakor

A workerben `insideWorker`-t és a `THREE` csomagot inicializálni kell elsősorban, majd az `insideWorker`-en 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 [12. ábrán](#) látható.

```

const insideWorker = require("offscreen-canvas/inside-worker");
const THREE = require('three');

insideWorker((event: any) => {
  if (event.data.canvas) {
    const canvas = event.data.canvas;

    const renderer = new THREE.WebGLRenderer({ canvas: canvas });

    // threejs code...

    // example for event handling
    self.onmessage = function (event) {
      if (event.data.type === 'mousemove') {
        const mouseX = (event.data.mouseX / canvas.width) * 2 - 1;
        const mouseY = -(event.data.mouseY / canvas.height) * 2 + 1;

        object.position.x = mouseX * 5;
        object.position.y = mouseY * 5;
      }
    };
  }
});

```

12. ábra: esemény fogadása a mellékszálon

4.1.5. DataService és az API forrás

A Naprendszer szimulációs modelljének fejlesztése során kulcsfontosságú szempont volt, hogy a bolygók adatai valósághűek legyenek, hiszen ez adja a szimuláció tudományos hitelességének az alapját. Éppen ezért az alkalmazás futásának kezdetén szükségessé vált olyan adatforrásból dolgozni, amely megbízható és publikus módon szolgáltat reális, mért bolygóparamétereket.

Ezt az igényt kezdetben egy külső API segítségével valósítottam meg: a francia *Le Systeme Solaire* adatbázisából nyertem ki a szükséges adatokat. Az API végpontja – ami az alábbi url-en elérhető: <https://api.le-systeme-solaire.net/rest.php/bodies/{planetName}> – lehetőséget adott arra, hogy adott bolygónévre lekérve ("GET" metódussal) kapjak vissza részletes információkat, például a tömegről, átmérőről, átlagos naptávolságról, keringési időről stb. A lekérdezés során fontos volt figyelembe venni, hogy a bolygónevet francia nyelven kellett megadni a címben.

Az Angular keretrendszeren belül a „data.service.ts” fájl szolgált arra, hogy a webalkalmazás architektúráján belül elkülönített service layerként kezelje az adatlekérdezéseket. A „DataService” osztályban több különálló függvényt is implementáltam, amelyek mind egy adott bolygó adatait képesek visszaadni. Például a „getEarthData” metódus a Föld adatait szolgáltatja, a „getMarsData” pedig a Marsét. Ezáltal a komponensek egyszerűen kérhetnek bolygóadatokat a „DataService”-től anélkül, hogy közvetlenül az adatforrás ismeretére szükségük lenne, így az alkalmazás felépítése átláthatóbbá és karbantarthatóbbá vált.

A fejlesztés előrehaladtával viszont gyakorlati problémák merültek fel. A külső API-hoz való állandó hozzáférés fenntartása bizonyos kockázatokat hordozott magában: például az API esetleges elérhetetlensége, a szerverkarbantartások, hálózati hibák vagy a válaszidő növekedése mind rontották az alkalmazás stabilitását. Emellett a hosszú távú karbantarthatósági igényeket is figyelembe kellett venni: egy éles környezetben futó szimuláció nem függhet külső szolgáltatásoktól.

Ezért a döntés az lett, hogy a kezdeti API-használatot követően letöltöttem a szükséges bolygóadatokat egy statikus JSON fájlba (planets.json), amelyet az alkalmazás helyben tárol és onnan olvas be futáskor. Így a „DataService” most már a helyi adatfájltra (/data/planets.json) hivatkozik, és az egyes metódusok ebből a statikus állományból szolgáltatják ki az adatokat. Ennek az előnye, hogy a program teljesen

offline működőképes, nincs kitéve hálózati hibáknak, valamint gyorsabb és kiszámíthatóbb adatbetöltést eredményez.

A jövőbeni fejlesztések során lehetőség van arra is, hogy egy háttérfolyamat időnként (például havonta egyszer) újra lekérje a legfrissebb adatokat az eredeti API-forrásból, majd automatikusan frissítse a helyi „planets.json” állományt. Ez a megoldás kombinálná a statikus adathasználat stabilitását a dinamikusan frissített adatok naprakészségével anélkül, hogy folyamatosan élő kapcsolatot kellene fenntartania egy külső szolgáltatóval.

4.1.6. 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 adathégyesünket, 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 előrebb.

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

4.2. Telepítés

Ebben a fejezetben bemutatom, hogy hogyan kell a Node alapú Angular webapplikációt telepíteni és elindítani. Fontos ehhez, hogy a gépen telepítve legyen a NodeJS. Ezekről az információkat a <https://nodejs.org/en/download> linken találjuk. Ha

ezzel megvagyunk, akkor a <https://angular.dev/installation> linken láthatjuk az Angular keretrendszer telepítésével kapcsolatos tennivalókat. Ha ezeket a lépéseket megtettük a célszámítógépen, akkor a telepítéshez be kell lépni a program mappájába. Megnyitva a konzolt az adott mappában ki kell adni a következő két parancsot: `npm install` majd az alapsomagok telepítésével futtatható a program az `npm run start` paranccsal. Amennyiben a 4200-as port nem foglalt, abban az esetben a `http://localhost:4200` link alatt találhatjuk a webapplikációt.

5. Eredmények

Mint az a szakirodalmi összefoglalóban olvasható volt fontos, hogy a program minél jobban optimalizált legyen, a kutatásom is erre helyeződött ki. 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 optimalizációban. Olyan eszközök kellettek, mint például a számítások külön szálra való kivezetése, vagy éppen a kódstrukturáltság és kódrendezettség, ami azt eredményezi, hogy felhasználói interakció esetén is szakadozásmentes marad a program. Mindezek után vizsgálatokkal és kutatási módszerekkel be kellett bizonyítani, hogy a webapplikáció teljesítménye javult, míg erőforrásigénye fordítottnan arányosan csökkent.

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

A tesztelések minden esetben 60 másodpercig futottak 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 fókuszot váltottam egy másik bolygóra és pásztáztam a kamerával. Négy teljesítménybeli tényezőt mértem a kutatás során: a képkockák számát másodpercenként (FPS), memória-, videokártya-, illetve processzorhasználatot. Ezeket fogom most a következő fejezetekben részletesebben is elemezni. Minden számítógép esetén öt worker thread változatú, majd öt worker thread nélküli változatú tesztet futtattam. Így összesen harminc teszt készült. Minden számítógép esetén átlagoltam a worker nélküli öt tesztet minden másodpercben azt összezsúroltam az átlagolt worker változatú tesztel [7], majd azokból alkottam meg a diagramokat a Flourish nevezetű diagramkészítő segítségével [8]. Egy diagramon egy számítógép átlagértékei láthatóak mind worker thread, mind worker nélküli változatban. A számítógépek specifikációit a táblázatban foglaltam össze. A következő fejezetben ezen teszteknek az eredményei kerülnek elemzésre.

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

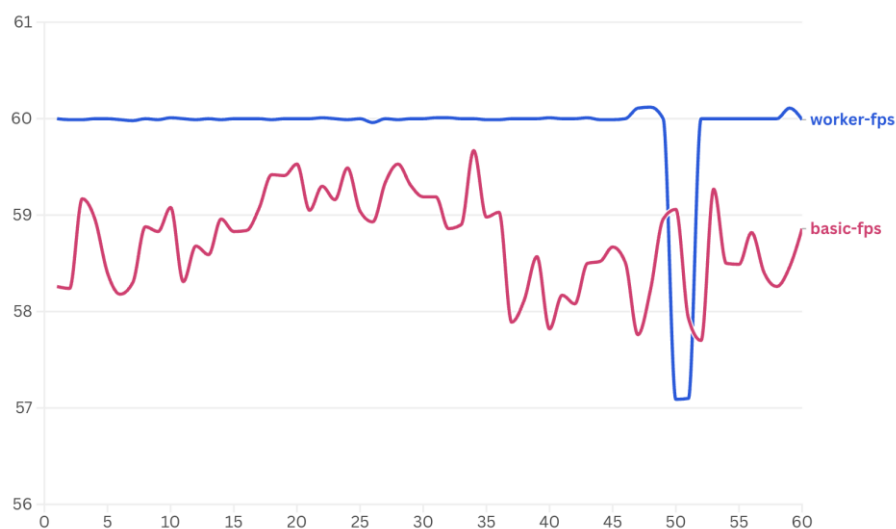
13. ábra: futtatott számítógépek adatai

5.1. 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.

5.1.1. Apple MacBook Air M1

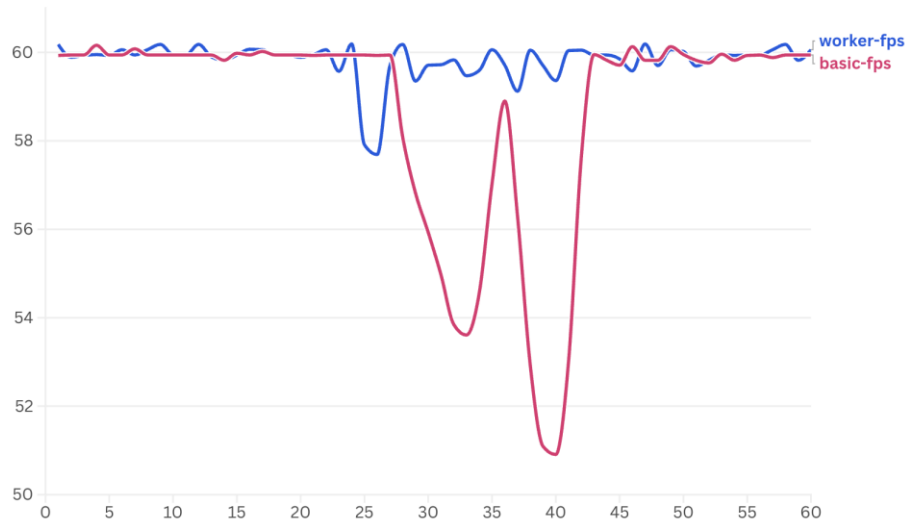
Ahogy azt az első eredmények is mutatják, egy mellékszál létrehozása és a sok optimalizációs művelet rengeteget tud segíteni az elosztáson és egy stabilabb, jobb teljesítménnyel rendelkező webapplikáció tud megjelenni a felhasználó előtt. Worker szál nélkül az FPS számosságának a volatilitása elég magas volt, de sose érte el a stabil hatvanas értéket. Ezzel szemben a Worker-rel - az ötvenedik másodperc környékét kivéve – stabilan hatvan FPS-el működött a modell megjelenítése. Ez egy ennél nagyobb és még komplexebb modell működtetése esetén még inkább szembetűnő különbség lenne.



14. ábra: Átlagolt FPS Apple Macbook Air-en

5.1.2. Windows számítógép 1

A második tesztnél a képkockák száma worker esetén stabilan tartották magukat az 58-60-as tartományban. Azonban a worker nélküli esetben gyakori kilengések történtek negatív irányban. Történt ez akkor, mikor új bolygó elhelyezését intézte a felhasználó vagy épp a kamerával nagymértékben pásztázott. Ennek a kilengésnek a fő oka, hogy egy szálon történik minden a DOM rendereléstől a háromdimenziós számításokig.



15. ábra: Átlagolt FPS az első Windows számítógépen

5.1.3. Windows számítógép 2

A harmadik átlagolt teszt elején is megfigyelhető, hogy mindkét esetben megvolt a közel hatvanas FPS szám, azonban voltak nagyobb és hosszabb idejű kilengései a worker nélküli optimalizálatlan változatnak. Mivel ez egy lassabb számítógép, így láthatjuk, hogy a workerrel implementált változat is benézett 55 FPS alá. A worker nélküli változat azonban több esetben is érezhetően lement az átlagosan 43 FPS körüli értékre.



16. ábra: FPS a második Windows számítógépen

Összesítve az átlagolt eredményeket, ezek alapján kijelenthető, hogy a worker beépítése egy szükséges lépés volt a teljesítmény javítása érdekében.

5.2. 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.

5.2.1. Apple MacBook Air M1

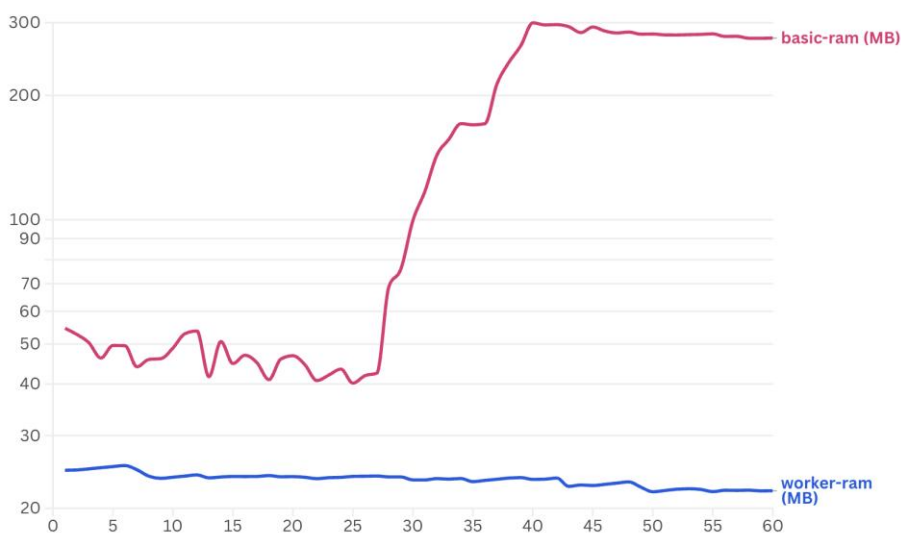
Ahogy az első grafikonon látható is, olyan magas volt az átlagos memóriaigénye a worker nélküli változatnak, hogy kénytelen voltam logaritmikus léptékű grafikonon ábrázolni mindezt. Ebben az esetben a memóriahasználat elérte a közel 600 MB értéket, ami rendkívül soknak számít egy ilyen programnál. Worker thread használata esetén ez az érték az elején 35 MB-ról leesett 26 MB-ra és tartotta mindvégig ezt az értéket. Köszönhető mindez a hatékony memóriahasználatnak.



17. ábra: Átlagos RAM használat Apple Macbook Air-en

5.2.2. Windows számítógép 1

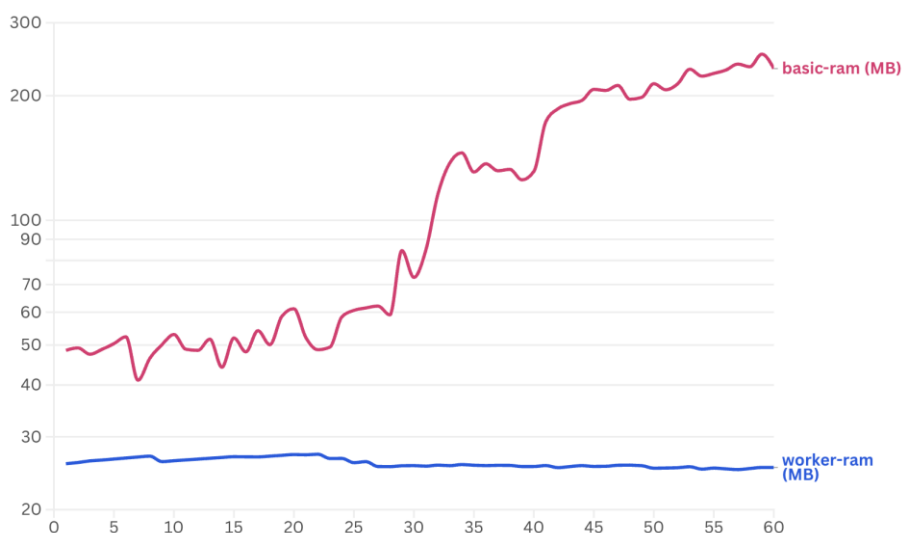
A két Windows operációs rendszerű gép használatakor is hasonló eredmények jöttek ki. Worker szál segítségével az értékek maradtak 30 MB alatt, azonban az értékek borzasztó adatot mutattak, amikor worker nélkül teszteltem a programot. Ahogy az látszódik is, az első Windows számítógép esetén majdnem elérte a 300 MB-os memóriahasználatot, ami egy ilyen program esetén, ha az nagyobb volumenű lenne, drámaian lelassítaná a számítógép teljesítményét.



18. ábra: Átlagos RAM használat az első Windows számítógépen

5.2.3. Windows számítógép 2

A harmadik esetben hasonló átlagértékek keletkeztek, mint a két fenti tesztnél. Itt a worker nélküli verzió átlaga elérte a 250 MB-ot a futtatás 55 másodpercében, míg a worker változat az elején 27 MB-os értékről lement 25 MB-ra és szépen tartotta ezt a memóriahasználatot másodpercenként.



19. ábra: RAM használat a második Windows számítógépen

A fenti három tesztről kijelenthetjük, hogy ha sokáig futtatnánk a programot, akkor a worker nélküli változat esetében valószínűleg fokozatos memóriahasználat-növekedést tapasztalnánk, mivel a memóriafelszabadítás nem tűnik olyan hatékornak, mint a worker thread-del megvalósított verzióban. Ez azonban nem feltétlenül jelent memóriaszivárgást. Ahhoz, hogy biztosan kijelenthessük, az egy perces tesztek helyett több órás stressztesztre lenne szükség, amely során a memóriahasználat trendjét folyamatosan monitoroznánk.

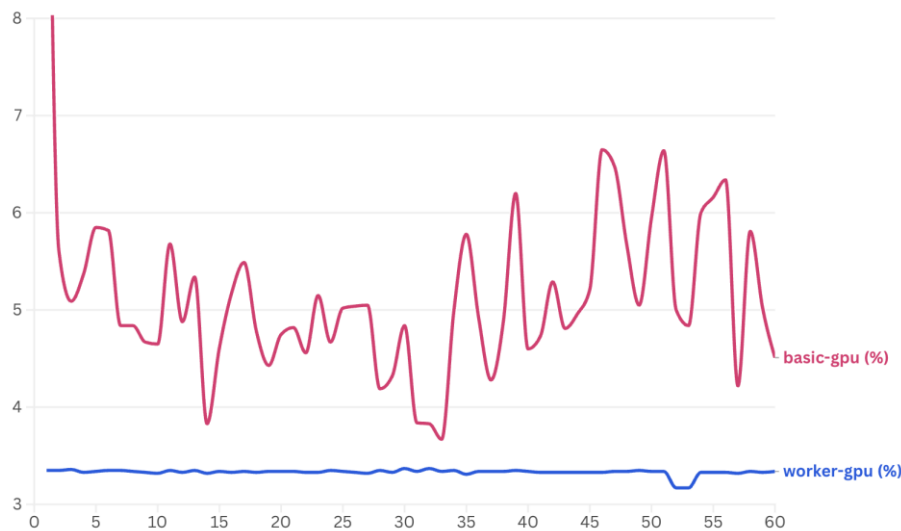
A worker thread-es változat stabil memóriahasználata azt sugallja, hogy a külön szálon való futtatás jobban elkülöníti az erőforrásokat, és lehetővé teszi a Garbage Collector számára a gyorsabb és hatékonyabb memóriakezelést. Ez hosszabb távon jobb skálázhatóságot és kisebb kockázatot jelent az esetleges memória-túlterheléssel szemben.

5.3. GPU-használat

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

5.3.1. Apple MacBook Air M1

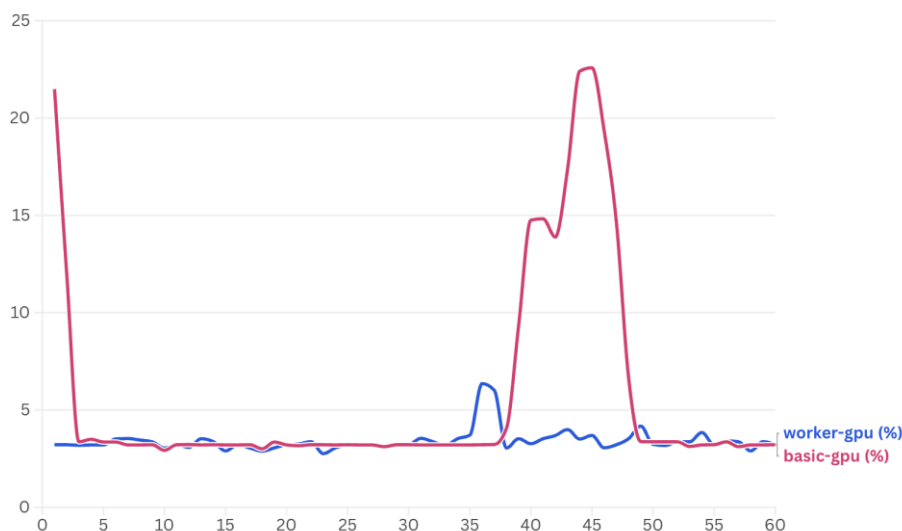
Worker thread nélkül az első átlagolt teszt esetén az érték 10%-ról indul, azonban később ez az érték a 3.5-6.5%-os arányban mozog eléggé erős volatilitással, míg a worker változat sokkal alacsonyabb videokártyahasználattal, stabilan tartja a 3.5% alatti szintet.



20. ábra: GPU-használat Apple Macbook Air esetén

5.3.2. Windows számítógép 1

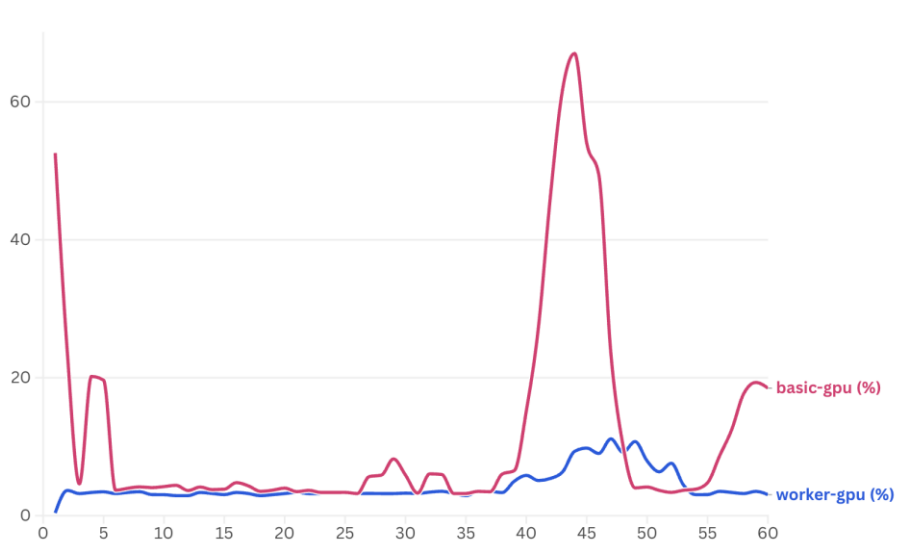
A második tesztben elérhető adatok alapján is látható, hogy a worker thread nélküli eset ezen a számítógépen is kirívó videokártyahasználattal kapcsolatos átlagokat ért el. Volt, hogy a 23%-ot is átlépte. Az Apple számítógépen elért eredményekkel ellentétben itt azonban egy ideig (5-30. másodpercig) együtt mozgott a két érték 3% környékén, majd csak utána történtek az erősebb GPU terhelések az új bolygók létrehozásának következményeként. Ezért érte el két másodperc elejéig az átlagosan 7%-os terhelést is a worker thread-el tesztelt változat.



21. ábra: GPU-használat az első Windows számítógépen

5.3.3. Windows számítógép 2

Ha az optimalizálatlan tesztet nézzük a második és harmadik eset között némi párhuzam vonható, azzal a különbséggel, hogy a második Windows számítógép egy fokkal gyengébb teljesítményű, ahogy az az FPS használatban is látható volt. Többnyire együtt mozgott a két érték 5% környékén az 5-25. másodpercben, azonban ezután a worker nélküli változat volt, hogy elérte a 65%-os videokártya használatot (ami nagyon magas érték egy ekkora programnál), addig a worker thread-el rendelkező változat maximum az átlagos 9-10%-os értéket érte el új bolygó létrehozásánál, ami egy jelentős érv a worker thread fontossága és használata mellett.



22. ábra: GPU-használat a második Windows számítógép esetén

Szemmel látható, hogy worker thread segítségével a GPU terhelése alacsonyabb és egyben stabilabb is mindhárom teszt esetén. Ami jelentősen elősegíti, hogy a program simábban és jobb teljesítménnyel futtasson.

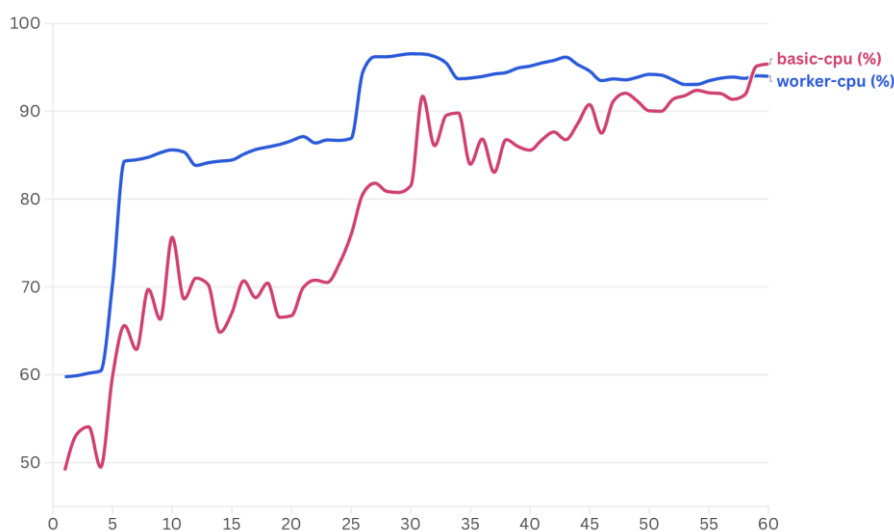
5.4. CPU-használat

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.

5.4.1. Apple MacBook Air M1

Mivel egy helyen történtek a számítások, így nem is volt akkora igénybevétele a CPU-nak, ellenben a mellékszál esettel, azonban néhol még így is majdnem meghaladta annak átlagolt értékeit már a kezdeti negyedik másodperc környékén. Ennek a változatnak az értékei kiegyensúlyozatlannak mutatkoztak, ellenben a workerrel rendelkező változattal.

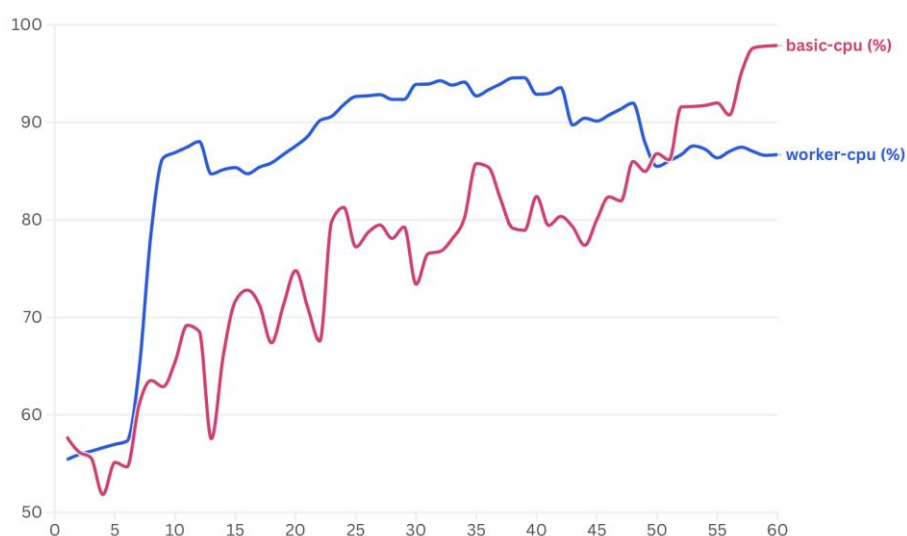
Mindkét érték egy folyamatosan növekvő tendenciát mutatott, annak függvényében, hogy jelentősebb felhasználói interakciók mind a tíz tesztetben megtörténtek Mellékszál igénybevétele esetén átlagosan először 60%-os majd később egy átlagos 80-95%-os CPU terhelés figyelhető meg.



23. ábra: CPU-használat Apple Macbook Air esetén

5.4.2. Windows számítógép 1

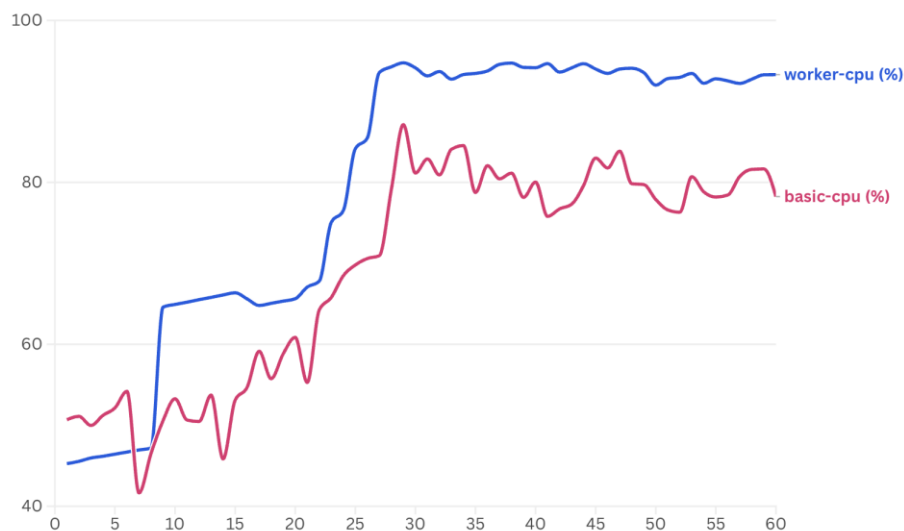
A második eset is egy izgalmas diagramot vázolt fel nekünk. Mindkét eset átlagosan 50-60% közötti értékkel kezdi el a processzorhasználatot. Worker használatával gyorsan eléri a program a 85% feletti processzor kihasználtságot, és ott stabilan tartja magát kisebb, kiegyensúlyozottabb átlagos ingadozásokkal. Ezzel szemben a worker nélküli változat kezdetben alacsonyabb kihasználtságot mutat, majd körülbelül az ötvenedik másodperc után hirtelen közel 98%-ra ugrik. Folyamatosan ingadozik az érték és folyamatosan éri el ezt a magas szintet stabil állapot nélkül.



24. ábra: CPU-használat az első Windows számítógépen

5.4.3. Windows számítógép 2

A második Windows számítógépen mért eredmények hasonlóan érdekes mintázatot mutatnak. A worker szállal futó megoldás eleinte kisebb CPU-használattal indul, de nagyon gyorsan, körülbelül 25. másodpercnél már 90% fölé ugrik, majd ott stabilan csak minimális kilengésekkel működik tovább. Az optimalizáció nélküli változat pedig már a kezdetektől követi a worker változat átlagszámait, de azokat sose éri el, sőt kevesebb (80% körüli) processzorhasználatot mutat. A grafikon tehát jól szemlélteti, hogy a worker alkalmazása itt is kiegyensúlyozottabb, de magasabb processzorterhelést ér el. Ez a stabilitás különösen fontos lehet nagy számítási igényű alkalmazások esetén.



25. ábra: CPU-használat a második Windows számítógépen

Összegezve tehát 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. A grafikonok alapján a worker szállal elkészített módszer gyorsabban és egyenletesebben terheli a CPU-t, míg az optimalizáció nélküli módszer lassabban, de nagyobb kilengésekkel éri el a magas terhelést és az első két teszt esetében az utolsó másodpercekben nagyobb átlagterhelést is mutatott, mint a worker változat.

6. Jövőbeli fejlesztések

A jelenlegi Naprendszer modell már egy komplex, több szempontból jól működő alkalmazás, amely képes több bolygó és egy központi csillag dinamikus megjelenítésére, azok pályakövetésére, valamint a felhasználó által történő interakciók kezelésére is. Ennek ellenére számos területen kínálkozik lehetőség a rendszer továbbfejlesztésére és elmélyítésére mind gyakorlati, mind elméleti szempontból.

6.1. Továbbfejlesztési lehetőségek

Az egyik fejlesztési irány a rezponzivitás kiterjesztése, vagyis az alkalmazás különböző képernyőméretekhez és ablakméretekhez való alkalmazkodóképessége. Jelenleg a képernyő átméretezése nem minden esetben működik tökéletesen, különösen akkor, ha a felhasználó átméretezi a böngészőablakot vagy mobil eszközön futtatja az alkalmazást. Az automatikus újra generálás és a kamera újra kalibrálásának implementálása jelentősen javítaná a felhasználói élményt, valamint lehetővé tenné a szoftver szélesebb körű használatát oktatási és tudományos célokra.

Egy másik fejlesztési lehetőség az elliptikus pályák módosításának lehetősége. Jelenleg a bolygók mozgása egy statikus elliptikus modell alapján történik, de ez továbbfejleszthető lenne azzal, hogy lehetővé tesszük a pályák excentricitásának, Napközelségének, Naptávolságának és pályasíkjának interaktív beállítását. Ez nemcsak vizuálisan lenne érdekes, hanem tudományos szempontból is releváns lehet, mivel egyes exobolygókat jellemző erősen excentrikus pályák szimulálását így pontosabban lehetne megvalósítani.

A hőmérsékleti adatok megjelenítése szintén fontos szerepet játszhat a szimuláció jövőbeli részletességében. A bolygó légköri tulajdonságai alapján becsült felszíni hőmérséklet vizualizálása színkódolással vagy információs megjegyzés formájában lehetővé tenné a lakhatósági zónák felismerését is.

6.2. További kutatási irányok

A rendszer továbbfejlesztése nemcsak technikai szinten lehetséges, hanem elméleti és kutatási irányban is. Az egyik ilyen kutatási irány lehet a bolygók közötti,

illetve csillagok közötti gravitációs kölcsönhatások szimulálása is. Az n-test probléma (n-body problem) pont ezt veti fel. A jelenlegi szimulációkban a bolygók pályái előre ki vannak számítva, és nem hatnak egymásra. Egy olyan modell, amely a tömegek közötti gravitációs erőket is figyelembe veszi és időben frissíti azokat, lehetőséget adna a rendszerek hosszú távú stabilitásának vizsgálatára, valamint kaotikus rendszereknek a szimulálására is.

Egy másik érdekes további kutatása lehet a témának az exobolygók éghajlati és légköri viszonyainak modellezése a tudomány által eddig megszerzett adatai alapján. Az égitestek forgási sebessége, tengelyferdesége, légköri összetétele és egyéb tényezők alapján akár egyszerűbb klímamodellek is készíthetők, amelyek hozzájárulhatnak a lakhatóság vizsgálatához, illetve annak bemutatásához, hogy milyen körülmények mellett lehetne élni egy adott exobolygón.

A fejlesztés és kutatás során sokat gondolkodtam azon, hogy a mesterséges intelligenciát és gépi tanulást miképpen lehetne integrálni a rendszerbe, ezzel új szintre emelve a felfedezést. Például olyan algoritmusok fejleszthetők, amelyek képesek stabil bolygórendszerek kialakítására, vagy éppenséggel a hosszú távon instabil rendszerek azonosítására. Ezzel nemcsak újfajta szimulációk hozhatók létre, de akár a tudományos kutatásokban is hasznosítható tapasztalatok születhetnek.

Mindezek a továbbfejlesztések a jövőben segíthetnének abban, hogy bizonyos forgatókönyvek mentén nem csak kutatásokat készítsenek a diákok vagy érdeklődők, hanem mélyebben megértsék a bolygó- és csillagrendszerek dinamikáját és a kozmikus fizika alapelveit.

7. Konklúzió

A kutatásom során a háromdimenziós grafika és a szimulációk területére fókuszáltam, a Three.js könyvtár használatával. A célom az volt, hogy mélyebb megértést nyerjek a számítógépes szimulációk és vizualizációk alkalmazásáról a csillagászat és a fizikai modellezés kontextusában. A projekt középpontjában a Naprendszer modell szimulálása és az exobolygó rendszerek generálása állt, amelyek lehetővé tették a csillagászati jelenségek, bolygórendszerek és azok dinamikájának vizsgálatát. A háromdimenziós ábrázolás segítségével nem csupán egy esztétikai élményt sikerül alkotni, hanem a fizikai valóság modellezése is lehetővé vált, miközben a Three.js optimalizálásával és az API által megszerzett adatok révén a modellek pontosságát is biztosíthattam.

A kutatómódszertani szempontokat alkalmazva a projekt során olyan tudományos megközelítést igyekeztem alkalmazni, amely lehetővé tette a részletes fizikai modellek implementálását. Az algoritmusok és a matematikai modellek precíz kidolgozása és azok helyes alkalmazása központi szerepet játszott a kutatásomban. Az egyes modellek és rendszerek kidolgozása során arra törekedtem, hogy minden egyes szimulált jelenséget pontos matematikai alapokra építsünk, a különböző interakciókat és dinamikai hatásokat megfelelően ábrázolhassak.

A kutatás során az is világossá vált, hogy a háromdimenziós modellek pontos és hatékony implementálása érdekében szükséges a modern grafikai rendszerek mélyebb megértése. A Three.js segítségével képes voltam az optimális teljesítmény biztosítására is, amely kulcsfontosságú tényező egy ilyen komplex rendszernél. A kódoptimalizálás, a textúrák kezelésének finomhangolása és a számítási erőforrások hatékony kihasználása mind hozzájárultak a projekt sikeréhez.

A projekt során alkalmazott kutatómódszertani megközelítések is hasznosak voltak, hiszen nemcsak az algoritmusok pontos megvalósítása volt a cél, hanem a kutatási folyamat alapos dokumentálása és elemzése is. Fontos cél volt, hogy az eredmények ne csupán technikai szempontból, hanem tudományos értelemben is értékelhetőek legyenek. Az alkalmazott kutatási módszerek, mint például a mérés, adatgyűjtés és az eredmények validálása, mind hozzájárultak a projekt megalapozottságához és tudományos megbízhatóságához.

A kutatás eredményei alapján levonható konzekvenciának tudhatjuk be, hogy a háromdimenziós grafika és a szimulációk fejlesztéséhez szükséges tudás nemcsak technikai, hanem elméleti szempontból is rendkívül fontos. Abból a szempontból is, hogy nem mindegy milyen módon implementáljuk a kutatáshoz szükséges dolgokat, nem elhanyagolható az, hogy mindezt milyen módon tesszük és milyen optimalizáció mellett. A projekt sikerességének tudható be, hogy ezt bizonyítottam és alá is tudtam támasztani pontos számokkal és diagramokkal is.

Összefoglalva, a kutatás és a projekt során elért eredmények megerősítettek abban, hogy a háromdimenziós szimulációk és a számítógépes grafika fejlődése lehetőséget biztosít a csillagászati és fizikai jelenségek pontosabb megértésére. Az elért eredmények nemcsak a technikai fejlődést szolgálták, hanem tudományos szempontból is hozzájárultak az asztronómiai rendszerek modellezéséhez. A következő lépésként a kutatások bővítése és a szimulációk további finomítása lehetne a cél, hogy még pontosabb, valósághűbb és komplexebb rendszereket hozhassunk létre.

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/#>, 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.
- [10] Lagrange Interpolating Polynomial
<https://mathworld.wolfram.com/LagrangeInterpolatingPolynomial.html>, 2025.04.23.
- [11] Angular. <https://angular.dev/>, 2025.
- [12] ThreeJS. <https://threejs.org/>, 2025.
- [13] Instanced model simplification using combined geometric and appearance-related metric. <https://api.semanticscholar.org/CorpusID:230799534>, 7 January 2021.
- [14] Understanding the Wilcoxon Signed Rank Test.
<https://www.statisticssolutions.com/free-resources/directory-of-statistical-analyses/how-to-conduct-the-wilcox-sign-test/>, 2025.
- [15] Angular OnDestroy. <https://angular.dev/api/core/OnDestroy>, 2025.
- [16] Research and Design of Toy Chinese Dragon Model based on OpenSCAD and Render based on Three.js. <https://wepub.org/index.php/TCSISR/article/view/2420>, 12 August 2024.
- [17] Dirksen, Jos: Learn Three.js Fourth Edition.
<https://books.google.hu/books?id=DGKoEAAAQBAJ&lpg=PP1&ots=k28R7fiAv2&dq>

[=learn%20three.js%20fourth%20edition&lr&hl=hu&pg=PP1#v=onepage&q=learn%20three.js%20fourth%20edition&f=false](#), February 2023.

[18] Interactive visualization of volumetric data with WebGL in real-time.

http://cadcamcae.eafit.edu.co/documents/non_final_paper_Web3D_2011.pdf, 20 June 2011.

[19] Restore Traditional Chinese Lanterns Based on Openscad and Threejs.

<https://www.atlantis-press.com/proceedings/dai-23/125998097>, 14 February 2024.

[20] Kepler's First Law: Planetary Orbits are Ellipses.

<http://burro.case.edu/Academics/Astr221/Gravity/kepler1.htm>, 2025.

[21] Spherical coordinates. https://mathinsight.org/spherical_coordinates, 2025.

Ábrajegyzék

1. ábra: felhasználói felület	12
2. ábra: mérési adatok	13
3. ábra: menü lehetőségei.....	14
4. ábra: generált CSV fájl tartalma	15
5. ábra: új bolygó hozzáadása	16
6. ábra: exobolygó generálása megfelelő környezetben	16
7. ábra: bolygók listája.....	18
8. ábra: Nap távolságától számított becsült felszíni hőmérséklet	29
9. ábra: kódrészlet a Worker szálról	31
10. ábra: worker importálása	33
11. ábra: worker meghívása a főszál indításakor	34
12. ábra: esemény fogadása a mellékszálon	34
13. ábra: futtatott számítógépek adatai	39
14. ábra: Átlagolt FPS Apple Macbook Air-en	39
15. ábra: Átlagolt FPS az első Windows számítógépen	40
16. ábra: FPS a második Windows számítógépen	41
17. ábra: Átlagos RAM használat Apple Macbook Air-en.....	42
18. ábra: Átlagos RAM használat az első Windows számítógépen.....	42
19. ábra: RAM használat a második Windows számítógépen.....	43
20. ábra: GPU-használat Apple Macbook Air esetén	44
21. ábra: GPU-használat az első Windows számítógépen.....	45
22. ábra: GPU-használat a második Windows számítógép esetén	45
23. ábra: CPU-használat Apple Macbook Air esetén	46
24. ábra: CPU-használat az első Windows számítógépen	47
25. ábra: CPU-használat a második Windows számítógépen.....	48