ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

NEUMANN JÁNOS
INFORMATIKAI KAR

SZAKDOLGOZAT

OE-NIK Hallgató neve: **Mikhel Roland**

2022 Hallgató törzskönyvi száma: **T/006306/FI12904/N**

Óbudai Egyetem
Neumann János Informatikai Kar
Szoftvertervezés és -fejlesztés Intézet

SZAKDOLGOZAT FELADATLAP

Hallgató neve: **Mikhel Roland**
Törzskönyvi száma: T/006306/FI12904/N
Neptun kódja: W68OXV

A dolgozat címe:

Cross platform 3D Engine WebGPU API segítségével Rust nyelven
Cross platform 3D Engine using WebGPU and Rust

Intézményi konzulens: Dr. habil. Szénási Sándor
Külső konzulens:

Beadási határidő: 2022. május 15.

A záróvizsga tárgyai: Számítógép architektúrák
Szoftvertervezés és –fejlesztés
specializáció

A feladat

Tervezzen meg és implementáljon egy WebGPU API alapokon működő 3D megjelenítő motort! A rendszer nyújtson lehetőséget a platformfüggetlen és hatékony 3D és 2D megjelenítésre! Vizsgálja meg a már meglévő hasonló rendszereket és a rendelkezésre álló módszereket és eszközöket!
Az irodalomkutatás alapján válassza ki a használni kívánt eszközöket és módszereket! Ezek alapján tervezze meg az elkészítendő feladatot, majd implementálja azt a kiválasztott programozási nyelven! A megvalósítás során fordítson különös figyelmet a motor mások által történő felhasználásának elősegítésére is! A rendszer újrafelhasználhatóságát demonstrálandó, valósítson meg két egyszerű alkalmazást eltérő logikával! Továbbá végezzen teljesítménymérést a különböző platformokon a hatékonyság tesztelése céljából.

A dolgozatnak tartalmaznia kell:

- a feladat leírását,
- a meglévő rendszerek bemutatását,
- az alkalmazott eszközök, technológiák és eljárások bemutatását,
- a megvalósítandó feladat tervét,
- a tesztelés folyamatát és a teszteredményeket,
- eredmények bemutatását és értékelését,
- az architektúrában rejlő továbbfejlesztési lehetőségeket,
- a dokumentációt, valamint a rendszert bemutató prezentációt CD mellékleten.

Ph.

ÓBUDAI EGYETEM * Neumann János Informatikai Kar * 1. Budapest

.....
Dr. Vámossy Zoltán
intézetigazgató

A szakdolgozat elévülésének határideje: **2024. május 15.**
(OE TVSz 55.§ szerint)

A dolgozatot beadásra alkalmasnak tartom:

.....
külső konzulens

.....
intézményi konzulens

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar

HALLGATÓI NYILATKOZAT

Alulírott hallgató kijelentem, hogy a szakdolgozat / diplomamunka saját munkám eredménye, a felhasznált szakirodalmat és eszközöket azonosíthatóan közöltem. Az elkészült szakdolgozatomban / diplomamunkámban található eredményeket az egyetem és a feladatot kiíró intézmény saját céljára térítés nélkül felhasználhatja.

Budapest, 2022.05.13.

Mikhel Roland

hallgató aláírása

TARTALOMJEGYZÉK

1 BEVEZETÉS, PROBLÉMAFELVETÉS

Manapság számtalan eltérő platform létezik, legyen akár mobil vagy desktop, továbbá ezeken belül is léteznek alvariánsok, amelyek közül a legtöbb rendelkezik olyan egyedi aspektussal, ami megnehezíti az egységes fejlesztést. Ez a probléma különösen igaz, ha nagy teljesítményre lenne szükségünk akár CPU, akár GPU oldalon. Általában kompromisszumos megoldások születnek, ami az esetek nagy részében azt jelenti, hogy igazán sehol sem tudja a maximumát nyújtani az adott szoftver. Az elmúlt néhány évben viszont már jelentek meg vagy még jelenleg fejlesztés alatt állnak ígéretes technológiák, amelyek ezen problémakör egy-egy sarkalatos pontját próbálják megoldani / megkönnyíteni. Témámnak megjelenítő motor megvalósítását tűztem ki célul, amely egy 3D-s környezetet képes megjeleníteni, mindezt natív platformokon és böngészőben futtatható módon is képes elérni. Ezt a platformfüggetlen aspektust a kód komolyabb átírása nélkül valósítom meg, megspórolva sok időt, biztosítva emellett a fejlesztés egységes menetét. A legtöbb már létező hasonló megoldással szemben nálam elsődleges szempont a hatékonyság, mert ezen aspektus általában háttérbe szorul az ilyen típusú alkalmazásoknál.

1.1 Problémafelvetés

A megvalósítás során a leghangsúlyosabb megoldandó probléma az eltérő platformokon való egységes működés biztosítása, hogy az egyes platformok közt nagyjából azonos kódbázis és megközelítés legyen használható.

A legnagyobb kihívás a céleszközök különbségeiben mutatkozik. Nem biztos, hogy rendelkezik a céleszköz olyan technológiákkal, amit használni szeretnénk, ezért egy egységes környezet kialakítása nehézkes, vagy egyáltalán nem megoldható.

Példa egy ilyen sarkalatos pontra az ablakkezelés: ahány operációs rendszer, annyi eltérő ablakkezelő rendszer. Gyakran egy operációs rendszeren belül is több opció áll rendelkezésre (pl.: Linux alapú disztribúciók), attól függetlenül, hogy a felhasználó számára egységes képet mutat.

2 A MOTOR FELADATA

2.1 Fejlesztői oldal

Általánosságban kijelenthető, hogy egy motor feladata nem csupán a megjelenítendő jelenet kirajzolása, ennél manapság sokkal több funkcióval kell rendelkezzen. A megjelenítő rétegen túl a motor részét képezi egy olyan rendszer is, amely az egyes objektumok közötti fizikai hatásokat hivatott szimulálni, valamint egy animáció kezelő, amely szoros kapcsolatban áll ezzel. A legtöbb motor-megvalósításban jelen van valamilyen egységes mesterséges intelligencia rendszer, aminek feladatai közé tartozik például a nem játékos karaktereket (NPC) irányítása. Ezt a rendszert hasznos implementálni fejlesztőként, mivel lerövidíti az alkalmazáskészítés folyamatát, csökkenti a bonyolultabb kód folyamatos újra implementálását a felhasználók részéről. Nem elhanyagolható az optimalizáció kérdése sem, a motor "belsejében" történő általános megvalósítás sokszor hatékonyabb lehet, mint a felhasználók (motorra fejlesztők) megvalósítása.

További feladat az egyes eltérő formátumú forrásállományok (Assetek) kezelése. Manapság különböző importáló és exportáló modulok alkalmazására kerül sor annak érdekében, hogy az eltérő formátumú adatokat belül értelmezhető formára/formából oda/vissza alakítsa. Ezeknek a rendszereknek az összehangolt vezérlésére fontos, hogy biztosítsunk szkript környezetet, amely az egyes funkciókat a felhasználók számára elérhetővé teszi. További szerepe a szkript környezetnek, hogy betartassa a fejlesztők által meghatározott korlátokat, ilyen például az objektum hierarchia, amely manapság kétféle lehet. A tradicionális Objektum orientált megközelítés, vagy egy sokkal flexibilisebb rendszer, az Entity Component System (ECS)[1].

További megvalósítandó aspektus egy egységes könyvtár kialakítása a különböző matematikai műveletek elvégzésére. Ez hangsúlyos feladat, mivel komoly problémákat okozhat a nem egységes használat. Jó példa erre az elforgatás: ha radián helyett fokban adjuk meg az elforgatás mértékét, a rendszer pedig radiánra van felkészítve, helytelen működést kapunk. A lebegőpontos ábrázolás is problémás lehet, ha a szkript környezet másképp valósítja meg a számábrázolást, mint a motor belső komponense.

Találkozhatunk számos olyan fejlesztést segítő eszközzel, amit a kevesebb kód írása érdekében készítenek. A pályaszerkesztő egy ilyen eszköz, amellyel könnyen lehet az adott jelenetet módosítani kedvünkre. A motor gyakran ilyen eszközt is biztosít számunkra.

A motorok gyakran API-t is biztosítanak, amin keresztül akár saját eszközt is fejleszthetünk, bővíthetjük a motor képességeit.

2.2 Felhasználói oldal

Felhasználó alatt jelen esetben azt a személyt értjük, aki fejleszt a motorunkra. Ebből a szemszögből vizsgálva a motornak más hangsúlyos tulajdonságokkal is rendelkeznie kell.

A felhasználó minél gyorsabban, minél egyszerűbben szeretne kézzelfogható, látható eredményt elérni, a lehető legegyszerűbb munkafolyamaton keresztül.

A felhasználói oldal felé nyújtott képességek közé a szkript környezet felhasználóbarát mivolta, stabil megvalósítása. A felhasználó szkripteléssel valósítja meg az adott alkalmazás felhasználói felületét, logikai működését.

Ha a program olyan fájlokat szeretne kezelni, amik nem képzik az alkalmazás részét azt is a felhasználónak kell kezelnie, ehhez a motor biztosít API-t az alapszintű kezeléshez, de például a szerializáció / deszerializáció már a kliensre hárul.

Az egyes médiaelemek összeköttetése (animáció + modell + hang) is a felhasználót terheli, mivel előre nehéz megjósolni, hogy mi mivel áll kapcsolatban a készítendő termékben, a termék jellegétől függően változhat a felépítés.

A jelenetben megjelenítendő elemek manipulálása is a felhasználó feladata, ezt is a szkript környezeten keresztül végzi.

2.3 Motorral szemben támasztott követelmények

- Platformfüggetlen működés
    - Ne kelljen kompromisszumokat kötni annak érdekében, hogy egy adott platformon elérhető legyen az alkalmazás.
- Hatékony megjelenítés elsődlegesen 3D-s kontextusban, modern funkciók alkalmazásával
    - Minél jobban használja ki az adott hardverben rejlő teljesítményt és alkalmazza a manapság bevált optimalizációs technikákat.
    - Implementáljon olyan megjelenítési technikákat, amelyek maximalizálják az adott jelenet grafikai részletességét.
- Könnyen használható, de mégis funkciókban gazdag szkript környezet.
    - Anélkül lehessen az egyes paramétereket módosítani, hogy a motor forráskódját módosítanánk.
    - Próbáljon igazodni a manapság megszokott fejlesztési paradigmákhoz, mint például az eseményvezérelt programozás.

3 HASONLÓ MEGVALÓSÍTÁSOK

Unreal Engine

Az Unreal Engine egy fejlesztői eszközök tárházával felszerelt motor, amely valós idejű alkalmazások készítésére alkalmas, legyen szó film, virtuális valóság vagy tradicionális videójáték készítésről. Nagy előnye, hogy nem követelmény komolyabb programozási tudás, mivel rendelkezik egy ún. Blueprint rendszerrel, amely egyfajta vizuális programozást ad a felhasználó kezébe. Ezen rendszer segítségével a logikát és az adott jelenethez tartozó elemek vezérlését is meg tudja valósítani a felhasználó. Ha a maximális teljesítményt szeretnénk elérni akkor lehetőségünk van C++ alapú kódot írni és kedvünkre módosítani a motor egyes tulajdonságait.[2]

Az Unreal a játékiparban a 3D-s területen elterjedt, 2D-s alkalmazást nem célszerű benne kivitelezni, mivel hiányos a motor részéről megjelenő támogatottság. Az Unreal Engine használata ingyenes mindaddig, ameddig a termék nem ért el több, mint 1,000,000$ nettó bevételt, ezután 5%-os részesedést kell fizetni. Ingyen használható a motor azokban az esetekben is, amikor ingyenes alkalmazást fejlesztünk vagy egy belsős projektet szeretnénk megvalósítani (3.1 ábra).

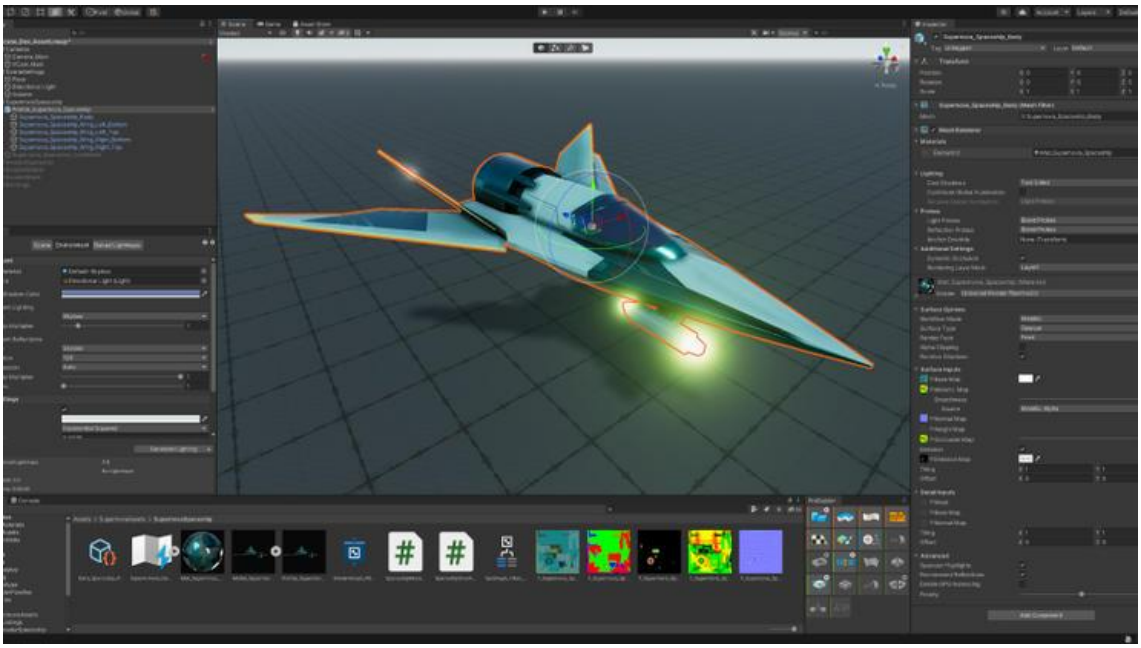
3.1. ábra - Unreal Engine [3]

Unity [4]

A Unity Technologies által fejlesztett motor, hasonlóan az Unreal Engine-hez, több, mint egy játékokhoz használt motor. A 2D-s játékkészítéshez is nyújt támogatást, ebben előnye van az Unreal Engine-hez képest. Rendelkezik egy ún. Asset-Store szolgáltatással, ahonnan mások által készített modellek, textúrák, hangok, kódok szerezhetők be. Ezek

között vannak fizetős termékek, de találunk ingyenesen, gyakran Creative Commons licensz alatt elérhető változatokat is.

A Unity a scriptelésre C# nyelvet használ, amit kezdetben a Mono[5] futtató környezet használatával tudtak platformfüggetlen (nem csak Windows) módon futtatni. Ezt manapság felváltotta egy ún. IL2CPP[6] fordítóval, ami a C# fordító által generált köztes kódot előbb C++ kóddá, majd az adott platformon elérhető C++ fordítóval natív kóddá alakítja. Ez a megoldás azért is előnyös, mert nem vonz maga után extra számításigényt, hiszen nincs szükség a .Net futtató környezetre, emellett eltűnik a Mono-tól való függés is. A Unity hasonló fizetési modellt alkalmaz, mint az Unreal Engine (3.2 ábra).



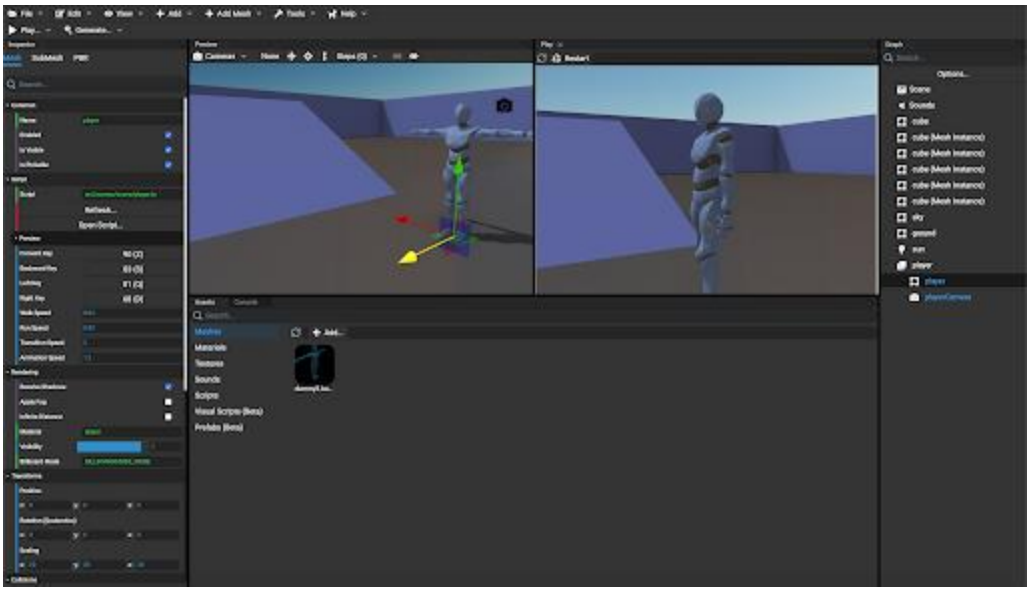
3.2. ábra -Unity3D[7]

Web – Babylon.JS[8]

A Bablyon.JS legfőbb célja, hogy egy teljesen ingyenes, egyszerű, Web alapokon nyugvó motort alkosson. JavaScript alapú scriptinget támogat, de hasonlóan a korábban bemutatott motorokhoz, itt is létezik vizuális szkripting. Többfajta web alapú megjelenítő technológiát is támogat, erre szükség is van, hiszen nem minden technológia érhető el minden böngészőben. A WebGL 1.0 általában elérhető, de léteznek bizonyos platformokon jobb, vagy még jelenleg is fejlesztés alatt álló alternatívák. Ezt a támogatást absztrakciókkal valósítja meg a motor: elrejti a felhasználó elől a technológia-specifikus kódrészeket, így a fejlesztőnek ezzel nem kell törődnie.

Ha nem csak böngészőben szeretnénk futtatni az programot akkor használhatjuk a Babylon Native projektet, ami natív környezetben képes futtatni a Bablyon.Js-ben készített alkalmazásunkat. Ez a projekt még erősen fejlesztés alatt áll, de vannak más, kiforrottabb projektek, amik már meglévő technológiák alkalmazásával érik el ugyanezt

a viselkedést. Ilyen például a Bablyon React-Native[9] projekt, amely React-Native integrációval ötvözi a Babylon.Js által adott lehetőségeket.(3.3 ábra)



3.3. ábra - Babylon.js[10]

Mobil - Corona SDK (manapság Solar2D)[11]

A Solar2D (régebben Corona SDK), ahogy a nevéből is adódik, egy teljes mértékben 2D-s motor, amely multiplatform alkalmazások készítésére alkalmas. Az eddigiekkel ellentétben itt az elsődleges programnyelv a Lua[12], amely biztosítja az egyes platformok közötti egységes kódbázis fenntartását, nem igényel kiterjedt programozási tudást. A könnyebb monetizációt elősegítendő, a fejlesztők számos opciót integráltak, mint például a banner hirdetések. Ez a motor is ingyenes annak ellenére, hogy nem nyílt forráskóddal rendelkezik. Folyamatosan növekvő felhasználóbázissal rendelkezik, amely segíti az újonnan megismerkedőket, fellelhetők közösségi fejlesztések, kiegészítések is a motorhoz.

4 TECHNOLÓGIALISTA

4.1 Felhasználói Felület

A motor több eltérő platformon fog futni, ezért célszerű megvizsgálni milyen létező keretrendszerek állnak rendelkezésre, amelyek segítségével egységes kinézetet és megegyező működést tudjunk biztosítani. Ügyelni kell a motor és a keretrendszer közötti problémamentes integrációra is.

4.1.1 Desktop keretrendszerek

A Qt az egyik legnépszerűbb és egyik legszéleskörben használt UI keretrendszer, amely mobilon és asztali környezetben is használható.

Több nyelven is lehet is benne fejleszteni, lehetőség van továbbá JavaScript alapú szkriptnyelv használatára is. Számos nyílt forráskódú szoftver használja, mint pl.: VLC media player, Krita, TeamViewer.

Az Uno Platform a másik népszerű keretrendszer**,** ami nyílt forráskódú, Universal Windows Platform és WinUI által XAML segítségével kerül megvalósításra. Több platformon elérhető, például iOS, Android, böngésző WebAssemblyn keresztül.

A fent említett opciók teljes keretrendszert foglalnak magukba, ezért a teljes megjelenítésért ők a felelősek. Ez számomra nem optimális, mivel elképzelhető, hogy az általam megvalósítani kívánt megjelenítő réteget nehézkesen vagy egyáltalán nem lehet integrálni ezekbe a keretrendszerekbe. Hangsúlyos szempont továbbá, hogy a webes környezetben való használatuk egyáltalán nem megoldott.

Teljes keretrendszerek helyett manapság bevált szokás, hogy egy böngésző kerül beágyazásra az adott alkalmazásba, ami megjeleníteni a tartalmakat. Ilyen megjelenítő a Chromium Embedded Framework, amely a Chromium böngészőn alapszik, és a manapság használt desktop operációs rendszerek mindegyikén használható. Jó példa erre napjaink egyik legtöbbet használt alkalmazása, a Spotify, ami ezt a megjelenítőt használja az egységes kinézet kialakítására az egyes asztali operációs rendszereken. A Chromium háttér miatt a különböző tartalmak megjelenítése webfejlesztői szemmel szinte ugyanaz, ezáltal a fejlesztés is egyidejűleg haladhat. Számomra ez a megoldás sem lenne optimális, mivel csak MacOS, Linux és Windows rendszereken elérhető, tehát a mobil és a webes környezetben más alternatívákat kellene használnom, ez pedig megnehezítené a fejlesztést.[13]

A fentebb említett hiányosságok miatt célszerűnek láttam megvizsgálni olyan függvénykönyvtárakat, amelyek a megjelenítési rétegtől függetlenül képesek működni, csak az egyes komponenseket és a hozzátartozó interakciós logikát definiálják, de a megjelenítést nem ezen könyvtárak végzik, azt a felhasználóra bízzák, hogy hogyan kerül a tartalom a képernyőre.

Az egyik legismertebb ilyen könyvtár a Dear ImGui, ami egy Immediate mode UI függvénykönyvtár. Eredetileg C++ nyelven íródott, fejlesztést segítő eszközök felületének megjelenítésére lett létrehozva, nem pedig tradicionális Felhasználói felületekre. Leggyakrabban valós-idejű 3D-s alkalmazásokba szokták integrálni. A fent említett pozitívumokkal rendelkezik: nem függ a megjelenítő rétegtől, valamint nincs külső függése más könyvtárak felé. Szüksége van backendre, amely felelős a megjelenítésért és az egyes inputokért, amiket továbbít az ImGui felé. Ezekből a backendekből létezik készítők által fenntartott változat, de fellelhető számos harmadik fél által készített megvalósítás is. Ezek alapján valószínűséggel az adott grafikai könyvtárhoz is található backend, amellett, hogy számos nyelvhez vannak összekötő komponensek is, biztosított a széleskörű nyelvi támogatás.[14]

4.1.2 Mobil keretrendszerek

A fent említett keretrendszerek között akadt olyan, ami képes mobil eszközöket megcélozni, de ez nem mindig kifizetődő, mert egy mobil platform merőben eltérő aspektusokkal rendelkezik, például irányítást, teljesítményt tekintve. Ebből következően előfordulhat, hogy ugyanazon alkalmazás mobilon futtatva nem fog megfelelően működni. Célszerű lehet dedikált mobil keretrendszer alkalmazása, ahol jobb optimalizáltságot lehet elérni, nem beszélve a különböző mobil specifikus jellemzőkről, mint például a különböző szenzorok, kamera használata, melyek leggyakrabban csak natív alkalmazásként elérhetőek.

Ilyen rendszer az Ionic, hasonlóan Web technológiákat alkalmaz az egyes elemek megjelenítésére, számos előre definiált elemet tartalmaz. iOS és Android rendszereken is jól használható, jó teljesítményt biztosít, valamint nagyfokú bővítmény készlettel rendelkezik, mert Cordova pluginokat használ például Kamera eléréshez.[15], [16]

A Xamarin a desktop keretrendszereknél megismert Uno Platformhoz nagy mértékben hasonlít, mivel ez a keretrendszer is XAML és .Net technológiákat használ a megjelenítéshez. További előnye a Xamarinnak, hogy minden natív funkcióhoz hozzáférést enged. Lehetőség van továbbá arra, hogy osztott kóddal dolgozzunk Xamarin.Forms-al, ami minden platformon megegyező UI-t tud biztosítani teljesítmény oltárán. Ezzel ellentétben lehet Natív kódot is írni, ahol a teljesítmény fontos és a fejlesztési idő kevésbé számít az eltérő platformok között.

A Xamarin.Forms továbbgondolása nyomán jelenleg is fejlesztés alatt áll a .Net Multiplatform App Ui (MAUI) amely számos fejlesztést tartalmaz. Ilyen például a közös kódbázis, amely a Xamarin.Forms nagy hátrányának tekinthető, mert minden platform különbözőn üzemel. Újítás továbbá a tradicionális MVVM modell mellett a MVU (Model-View-Update) modell támogatás is. A keretrendszer még nincs kész állapotban, a Microsoft úgy tervez, hogy 2022 második negyedévében fogják kiadni az első stabil verziót.[17], [18]

UI keretrendszerek tekintetében célszerű olyat választanunk, amellyel teljes behatásunk van a rajzolás folyamatára, a maximális teljesítmény elérése érdekében. Figyelembe kell vennünk azt is, hogy a használandó komponens lehetőleg egyáltalán ne, vagy csak minimálisan rendelkezzen külső függésekkel, ne függjön a választott grafikai API-tól. A Dear ImGui rendelkezik ezekkel a paraméterekkel, így erre esett a választásom.

4.2 Programnyelv

Grafikus megjelenítő alkalmazás fejlesztése során létszükséglet, hogy minél gyorsabban fusson az adott kód. Egy olyan programnyelv kiválasztása célszerű, ami nagyon jó teljesítményt biztosít túlságosan nagyon alacsony szintű nyelvek használata nélkül (pl.: Assembly), emellett minden platformon, amin szeretnénk használni az alkalmazást elérhető. Cél a böngészőben való futtathatóság is, ezért limitáltak a lehetőségek, hiszen a JavaScripten kívül csak a WebAssembly-t lehet használni, mint böngészőben futó „nyelv”.

A WebAssembly (rövidítve: WASM) standard-et a World Wide Web Consortium (W3C) kezdeményezte, valósította meg. Céljuk egy olyan hordozható bináris formátumú kód létrehozása volt, amely közel natív sebességgel tud futni a böngészőben. Ez a kód egy verem alapú virtuális gépen fut, ami közel áll például a Java Virtual Machine-hez.[19], [20],[21]

WebAssembly kódot nem szokásunk közvetlenül írni, mert meglévő nyelvekkel képesek vagyunk WebAssembly kódot generálni. Ilyen nyelvek például C/C++/Rust (LLVM nyelvek) Emscripten. Lehet C# kódot is WebAssembly-re fordítani (de ebben az esetben magát a .Net futtató környezetet is bele kell fordítani a végső állományba pl: Blazor), emellett lehet TypeScript alapú AssemblyScriptet is használni.[22], [23]

A WebAssembly kód ugyanúgy, mint a JavaScript, egy ún. „sandboxed” környezetben fut, így nem kell a biztonságot feláldozni a teljesítmény oltárán. Nem magára a JavaScript leváltásra találták ki, hanem egy alternatívának olyan esetekre, ahol a JavaScript nem elég hatékony. Van lehetőség JavaScript és WASM közötti kommunikációra: JavaScript oldalról lehet írni és olvasni magát a WASM memóriát, de ez a másik oldal felől nem lehetséges.

Ha gyors és memóriahatékony alkalmazást szeretnénk írni, a C++ szinte megkerülhetetlen, folyamatos fejlesztés alatt áll, támogat többféle paradigmát, nagy fejlesztői közösség övezi. Egy hátulütője, hogy nem platformfüggetlen: gyakran megesik, hogy valami fordul például Linux környezetben, de nem fordul Windows esetén. Az évek során hatalmas nyelvvé nőte ki magát, így sokszor átláthatatlan, ellentmondásos, nem naprakész információkkal találkozik a fejlesztő. A böngészőre történő fordítás WebAssembly-re csak Emscripten compiler használatával lehetséges, ami további problémákkal szembesít minket. Ilyen probléma például az a tény, hogy az Emscripten csak Linuxon elérhető stabilan. Léteznek módszerek, amelyekkel Windows operációs

rendszeren is „elérhetővé tehető” az Emscripten. Ilyen egy virtuális gép vagy a Microsoft Windows Subsystem for Linux (WSL) programja, ami egy teljes értékű Linuxot futtat Windowson belül, ezen kerülőutak viszont távol állnak a stabil elérhetőségtől. [24], [25]

Megfontolandó alternatíva lehet a Mozilla által fejlesztett multiparadigm systems-language, a Rust, amely hasonlóan a C++-hoz egy erősen típusos nyelv. Itt a teljesítmény mellett a biztonságon is nagy hangsúly van. Találunk egy dedikált csomagkezelőt, a Cargo-t, ami kezeli az esetleges függőségeket (Crate-eket), emellett megvalósítja a buildelés folyamatát. Összehasonlítva, a C++ esetében, nincs a nyelv fenntartói által kiadott csomagkezelő eszköz, harmadik féltől származó alternatívákkal lehet találkozni. Beépített teszt keretrendszerrel is találkozhatunk a Rust esetében, ami Cargo-n keresztül egyszerűen futtatható.[26], [27]

A Rust Garbage Collector nélkül éri el a biztonságos memóriakezelést, egy Borrow-checker alkalmazásával, ami fordítási időben vizsgálja az egyes referenciákat továbbá a túlcsordulásokat is figyeli. Ha memóriaszivárgás keletkezne a kódban, a fordító nem engedi a kódot lefordítani. A fordító képes a versenyhelyzet detektálásra az egyes szálak között, ami szintén Borrow-checker érdeme.[28]

Mivel a Rust is egy LLVM alapú nyelv, ezért itt is lehet a WebAssembly-t generálni a forráskódból. A különbség C++-al szemben az API hívások területén jelentkezik: egyszerűbben lehet meglévő API-kat hívni Rust kódból különböző csomagokon keresztül, amik böngészőben elérhető modulok (pl.: fetch API). A WebAssemblyre történő kódgenerálást a wasm-bindgen nevű eszköz biztosítja. Ez az eszköz nem csak a fordításért felelős: legenerálja a futtatáshoz szükséges állományokat és a különböző környezetek közötti megfelelő összekapcsolásokat is biztosítja.

A Rust egy megfelelő nyelv a motor megvalósítására, mivel natív kódra és WebAssemblyre is képes fordítani, közel azonos teljesítményt kínál, mint a C++, a beépített védelmek sokat segítenek memory-leak mentes kód írásában, emellett jó eszközparkkal rendelkezik (Csomagok és fejlesztést segítő eszközök terén is), ami meggyorsítja a fejlesztés folyamatát.

4.3 Szkript környezet

Egy modern motor könnyű kezelésére manapság gyakran használnak más nyelveket, szkriptnyelveket, melyeknek pozitívuma, hogy gyors fejlesztést és átláthatóbb alkalmazáskódot eredményeznek.

Ezek a szkript kódok nem képezik a motor központi részét csak az motor egyes elemeinek vezérlését látják el, ezért gyakran csak futás időben kerül sor az utasítások értelmezésre. Ha mégis fordítanánk a szkript kódot, lehetséges ezt valós időben (Just In Time Compiler - JIT segítségével) megtenni, így nem szükséges, hogy a fordításkor belekerüljenek a végső állományba. További előny, hogy az apróbb módosítások során nem kell a teljes alkalmazást újra fordítani, ezzel sok időt spórolunk meg. A több platformos cél

megvalósítása érdekében célszerű olyan nyelvet választani, ami könnyen integrálható a motor központi elemeihez.

A Lua manapság az egyik legelterjedtebb ilyen nyelv, többek között a Blizzard Entertainment használja a World Of Warcraft Felhasználói felület programozására, amit a felhasználók akár kedvükre alakíthatnak a nyelvnek köszönhetően. Nagy előnye, hogy egy virtuális gép formátumot alkalmaz, hasonlóan a Java-hoz, van Garbage Collectora, támogat procedurális és objektum orientált paradigmákat is. Egyszerű szintaxissal rendelkezik, ami megkönnyíti a tanulás folyamatát mivel hasonlít az angol nyelvhez, számos modern funkcióval rendelkezik, ilyen például az aszinkron kódfutás. A merőben eltérő szintaxisának hátránya is van: a nyelv manapság alapvetőnek vett absztrakt szintaktikai konvenciókkal megy szembe, esetenként teljesen mellőzi azok megvalósítását. Ilyen például a ternary operátor használata, vagy a tömb indexelés 1-től való indítása. [29], [30]

Egy másik opció lehet a JavaScript, amely az egyik leginkább elterjedt nyelv napjainkban. A böngészőkben csak ezt a nyelvet használhatjuk az egyes weboldalak módosítására. További érv mellette, hogy jól integrálható más nyelvekbe, például a C/C++/Rust-ba. Az egyik populáris JavaScript motor a Google által fejlesztett és a Chromium alapú böngészőben is üzemelő V8, de fellelhető más alternatíva is, például a SpiderMonkey, amit a Mozilla használ a Firefox böngészőjében. A nyelv sajátosságai közé tartozik, hogy egy gyengén típusos nyelv, ezáltal nem rendelkezik típusellenőrzéssel. Ha szeretnénk típusosságot bevezetni a nyelvbe, akkor a Microsoft által fejlesztett TypeScript nyelvet tudjuk használni, ami JavaScript kódot állít elő egy transpiler folyamaton keresztül, így nem keletkezik teljesítmény-beli hátrány a használatából. További eltérés a tradicionálisabb nyelvekkel ellentétben az Objektum Orientáltság támogatottsága, ami itt nem elsődleges, mert a JavaScript egy Prototype alapú megközelítést alkalmaz. Ez annyit tesz, hogy egy alap objektum prototípust készítünk, és ezt a prototípust felhasználva hozunk létre új objektumokat, ami szembe megy az objektum orientált szemlélettel. [31]–[33]

Mivel a böngészőkben elérhető a JavaScript, amely képes a WebAssembly kóddal kommunikálni és mindezt minimális költséggel teszi, ezért sok időt lehet spórolni, ha ezt a lehetőséget kihasználjuk és a scripting során a JavaScriptet alkalmazunk. További érv a JavaScript mellett, hogy széleskörben elérhetők fejlesztést segítő eszközök, valamint a nyelv popularitása miatt nagy tudásbázis érhető el hozzá.

4.4 Grafikus függvénykönyvtár

Egy hatékony megjelenítő motornál elengedhetetlen, hogy az egyes készülékekben lévő grafikus gyorsítókat (GPU) kihasználjuk, amennyire csak lehet. Ahhoz, hogy egy grafikus vezérlőnek utasításokat tudjunk kiadni, az adott vezérlő illesztőprogramját kell felhasználnunk. Mivel ez elég változatos és bonyolult lehet, számos magasabb absztrakciós szinten lévő programkönyvtár került létrehozásra az évek során. A bennük

található API-kon keresztül egyszerűbben és biztonságosabban lehet használni az egyes GPU-kat. Egy ilyen könyvtárnál nem csak a teljesítmény fontos, hanem az is, hogy milyen könnyen használható, valamint a támogatottsága is kritikus kérdés.

A Direct3D a Microsoft által létrehozott API, amely napjainkban az egyik legelterjedtebb és legnaprakészebb funkciókkal rendelkező csatolófelület. Egy ilyen funkció Ray Tracing. Előnye például, hogy számos olyan fejlesztői eszközt biztosít, amely megkönnyíti a fejlesztés folyamatát, a legújabb verzió (D3D12) pedig nagy szabadságot ad a programozónak, így hatékony és gyors végrehajtást eredményez.[34] Hátránya, hogy csak Windows és Xbox platformokon található meg, emellett komoly, API-specifikus szakértelmet igényel, ezért számomra ez nem járható út.[35]

Az egyik, ha nem a legszélesebb körben használt API, az OpenGL. Nyílt Standardként tették elérhetővé az 1980-as években. Ennek is köszönhető, hogy számos platformon a mai napig elérhető. Előnye, hogy könnyű használni, nem igényel olyan mértékű specifikus tudást, mint a Direct3D, de pont ebből fakad a hátránya is: nagyon kötött rendszer (pl.: az árnyékolás folyamatát sokáig nem lehetett módosítani), ezért teljesítményben manapság elmarad néhány riválisától.

Az OpenGL egyik legnagyobb pozitívuma, hogy nyelvfüggetlen, ezért a legtöbb nyelvben található hozzá olyan függvénykönyvtár, amellyel hozzá lehet férni az OpenGL függvényekhez. Készült egy beágyazott rendszerekre kiadott verzió is OpenGL ES néven, amelyet telefonokhoz, konzolokhoz fejlesztettek. Később ez lett az alapja a böngészőkben használható WebGL technológiának is.[36], [37], [38]

A OpenGL modernizálásának igényére jött létre a Vulkan, ami egy viszonylag új technológia. A Khronos Group hozta létre néhány évvel ezelőtt. Ez az könyvtár egy sokkal hardverközelibb, sokkal elosztottabb CPU/GPU felhasználásra törekszik, a mai többprocesszoros rendszerekre lett tervezve. Hasonlóan az OpenGL-hez, itt is nyílt Standard-ről beszélhetünk, sok platformon megjelenik, kezdve az egyszerű PC-től az okostelefonokig. Tervezéséből adódóan, mivel nagyon alacsony szintű API-ról beszélhetünk, itt is jelen van a nagy tudást igénylő API, de ez adja a teljesítményének jelentős részét is.[39], [40]

A legújabb fejlesztés grafikus könyvtárak terén a WebGPU, ami egy jelenleg fejlesztés alatt álló specifikáció. Nevével ellentétben nem csak Webes környezetekre, hanem mobil és asztali platformokra is készül, a W3C Working Group gondozásában.[41], [42]

Ez nem egy OpenGL-ES(WebGL)-hez hasonló átirat, hanem teljesen új technológia, de az alapja a modern API-kon nyugszik, mint például a Vulkan, Metal és a D3D12. Elsődleges célja, hogy a böngészőben is elérhető legyen a GPU intenzív számítások felgyorsítására. A böngészőben a JavaScripthez API-n keresztül lehet elérni, valamint maguk az egyes GPU példányok izoláltan futnak. Jelenlegi állapotában tesztjelleggel érhető el Chromium alapú és Firefox böngészőkben, de léteznek olyan harmadik fél által

készített könyvtárak, amelyek a specifikációt alapul véve tradicionális asztali környezetben is lehetőséget adnak WebGPU kód írására.

Grafikus függvénykönyvtárakat tekintve a WebGPU az optimális megoldás, mivel nincs eltérés böngésző és natív platformok implementációi között fejlesztési szempontból, emellett az esetleges teljesítmény deficitet natív platformokon ellensúlyozza, hogy itt ténylegesen egy kódbázist kell fejleszteni, és a weben ez a legjobb alternatíva. Natív környezetben harmadik helyre tenném mert a Vulkan teljesítményben jobb, az OpenGL-t pedig egyszerűbb használni.

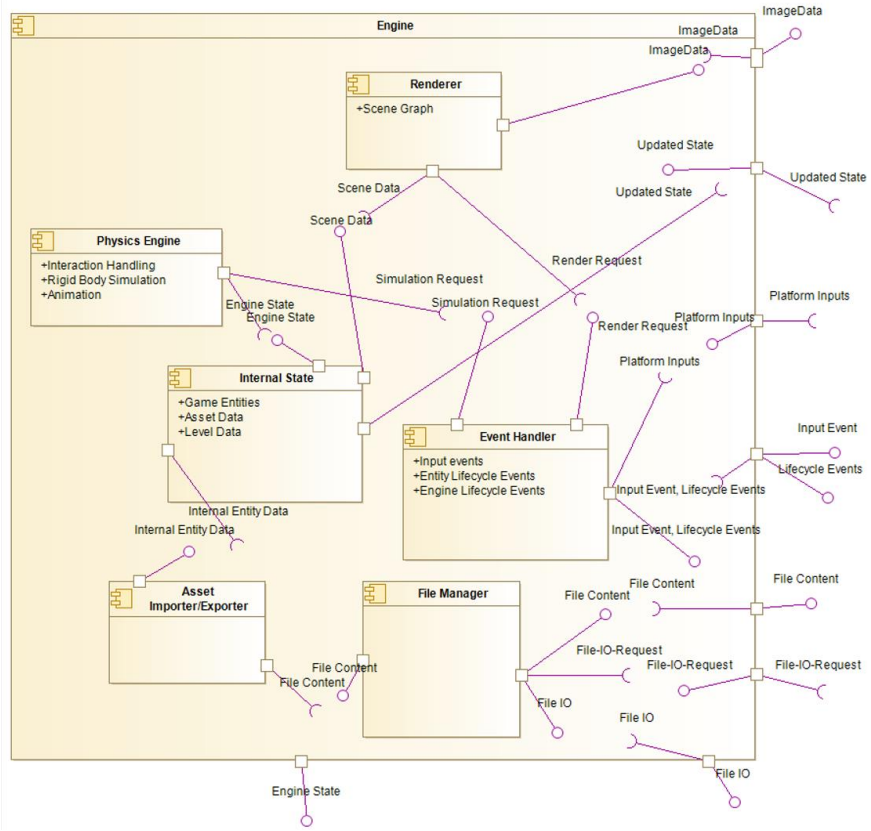
5 MÓDSZERTAN

Saját megvalósításom abban tér el a hasonló rendszerek áttekintése során felsorolt opcióktól, hogy próbál új lehetőségeket kiaknázni új technológiák alkalmazásával, ami a platformfüggetlen fejlesztést illeti. Ez a megközelítés olyan megjelenítő réteget biztosít, amely teljesen új kaput nyit a böngészőben használt nagy számításigényű alkalmazások felé azon felül, hogy natív környezetben is jó teljesítményt ígér.

A problémafelvetésben már említett hangsúlyos probléma a platformfüggő és platformfüggetlen kódrészek szeparációja. A problémát orvosolandó, célszerű a rendszer kialakítását úgy megtervezni, hogy a platformfüggetlen, univerzális részeket egyben, egy kódbázisként kezeljük, majd fordítás során összekombináljuk ezt az adott platformra jellemző, specifikus kódrészekkel. Ez a megközelítés már a fejlesztési folyamat kezdetén tudatos tervezést kíván.

5.1 Teljes alkalmazás

Alapvetően két fő egységre lehet bontani az alkalmazást logikailag: magára a motor központi részére, ami az alkalmazás futásáért felelős komponenseket foglalja magában és a szkript környezetre, amely a külső módosításokat továbbítja a központi egység felé. Az alábbi két komponens diagram ezen egységeket hivatott bemutatni részletesebben, demonstrálja, hogy milyen almodulok fogják felépíteni a nagyobb egységeket, valamint az ezek közötti kapcsolatokat is hivatott reprezentálni. (5.1, 5.5 ábra)

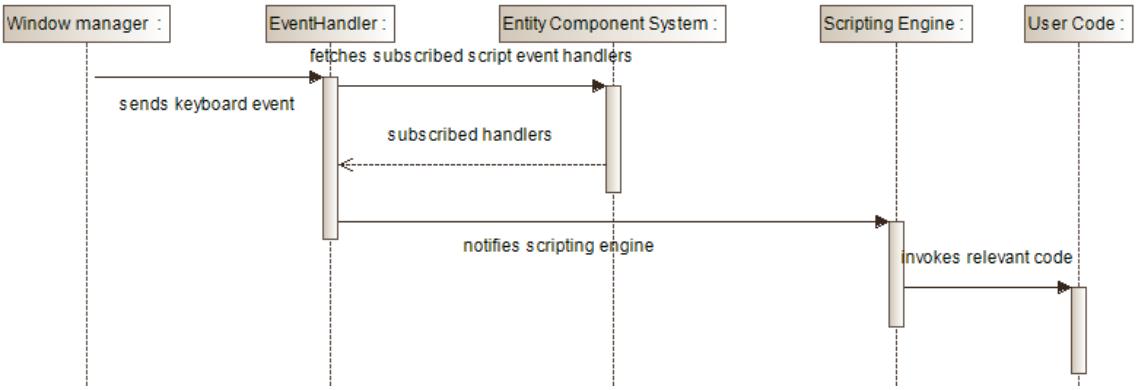


5.1. ábra - A motor és benne lévő modulok komponensdiagramjai
(saját készítésű ábra)

A központi egység feladata nem csak a megjelenítés, itt fog a fizikai szimuláció is végrehajtódni, emellett az egyes elemek aktuális állapotainak tárolása is itt valósul meg. A tárolást egy központi adattároló komponens fogja végezni. Ez a megoldás elsőre nem tűnhet túl optimálisnak, valamint a különböző adatok egységes tárolása sem triviális feladat. Ezen problémákat a korábban említett Entity Component System (ECS) tervezési minta megoldja: az egyes entitások nem konkrét típusokat reprezentálnak, csak az egyes komponensek között lévő kapcsolatot, ez végeredményében a Composition over Inheritance elvét valósítja meg. A különböző modulok elemeit Component-ként adjuk hozzá a rendszerhez, a különböző típusú Componentek tárolásának módját pedig én adhatom meg. Amikor szükség van rá az egyes System példányok csak a számukra releváns adatot kapják meg és dolgoznak vele.

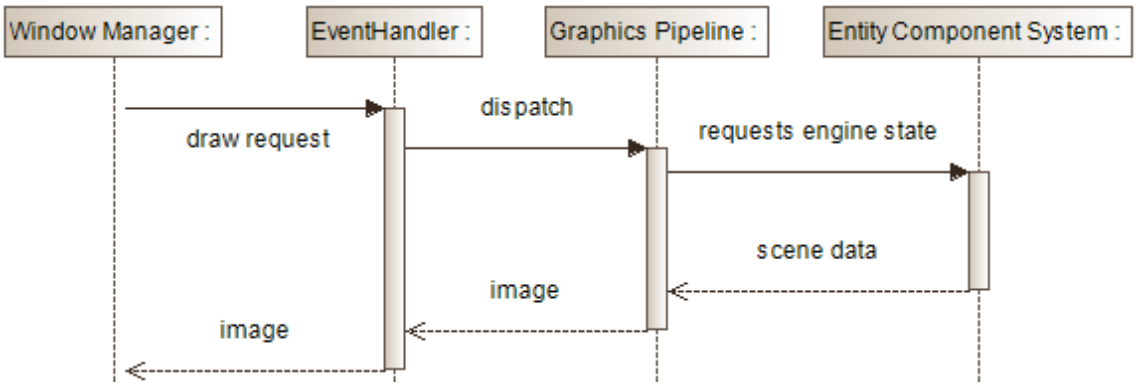
Ide fognak érkezni a futás során keletkezett események egy eseménykezelőn keresztül. Ezen komponens feladata lesz továbbítani a szkriptrendszer felé a releváns adatokat, mint például egy billentyűleütést.

Egy ilyen folyamatot mutat be az egyes komponensek között az alábbi diagram. (6.2 ábra)



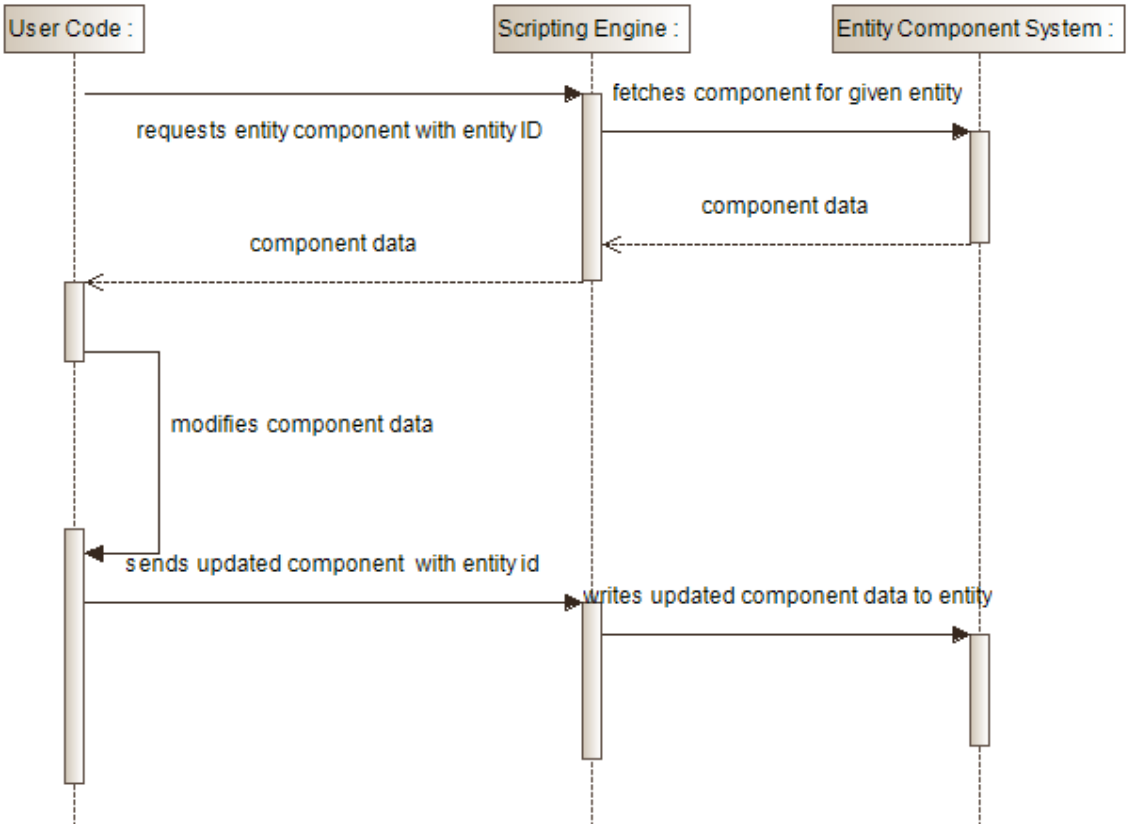
5.2. ábra – Billentyűlenyomás kezelésének folyamata
(saját készítésű ábra)

A rajzolás folyamata is hasonlóképpen fog megtörténni. Az adott ablakkezelő fogja kezdeményezni egy új frame kirajzolását, majd ezt az eseménykezelő továbbítja a megjelenítő felé, ami az ECS-ben tárolt információk alapján fogja megrajzolni a következő képkockát. (6.3 ábra)

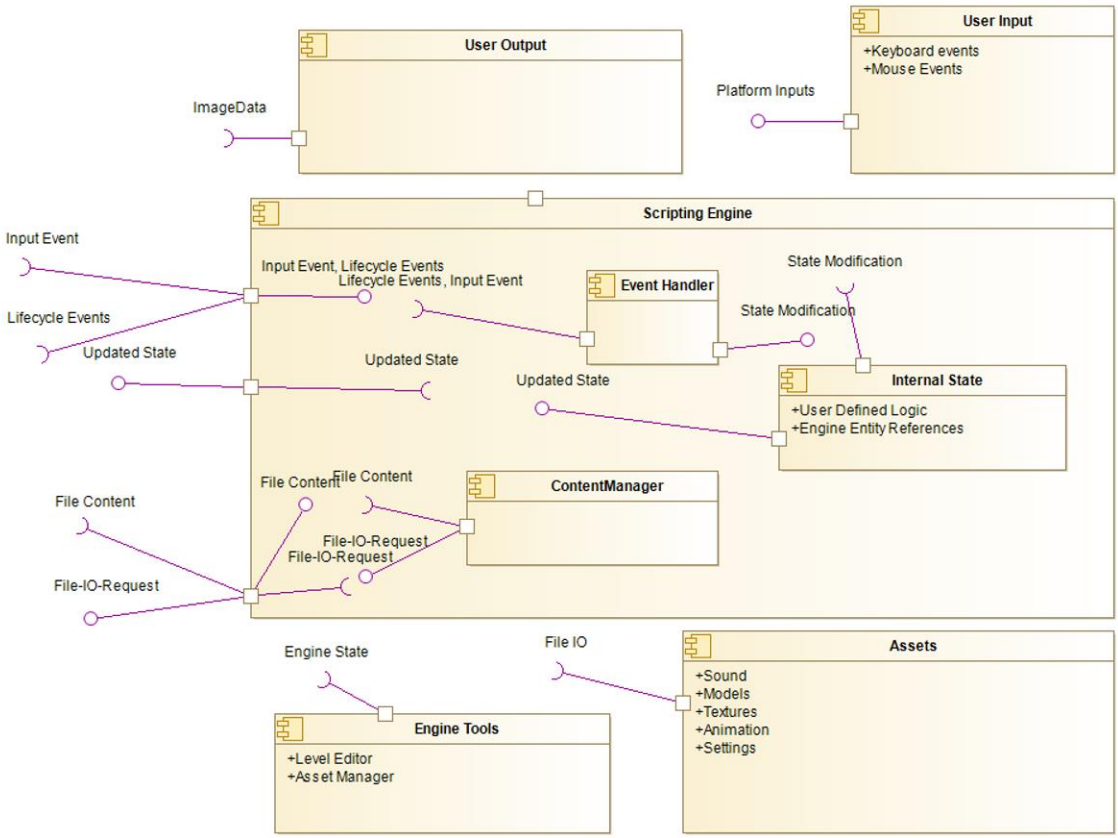


5.3. ábra - Megrajzolás esemény folyamata
(saját készítésű ábra)

A szkript rendszer elsősorban a saját működéséhez szükséges adatokat fogja tárolni, a belső objektumokat csak hivatkozás szinten fogja kezelni. Így elkerülhető a több helyen lekezelendő változások problémája. Az egyes értékek felülírással fognak az új állapotukba kerülni, például egy objektum pozícióját lekérve, így az adott komponensen történő adatmódosítás nem fog végbe menni automatikusan, csak ha a módosított értékekkel explicit módon felülírjuk. (6.4-6.5 ábra)



5.4. ábra - Komponens módosítás
(saját készítésű ábra)



5.5. ábra - Szkript környezet és egyéb modulok komponens diagramjai
(saját készítésű ábra)

5.2 Fejlesztői eszközök

A megszokott fejlesztés során használt eszközkészleten kívül szükségesnek láttam grafikus kártyára írt kód (Shader) hibakeresésére is képes programot, amely lényegesen megkönnyíti a fejlesztés folyamatát azáltal, hogy futás közben tudjuk vizsgálni az egyes Framebufferek tartalmát, valamint a különböző Pipeline-ok állapotait. Egy ilyen szoftver a RenderDoc, amely ipari szinten is nagyfokú felhasználtságnak örvend, kellő mennyiségű funkcióval rendelkezik. Nem rendelkezik elsődleges támogatottsággal a WebGPU API-hoz, de az általam felhasználni kívánt függvénykönyvtár rendelkezik natív futtatási lehetőséggel, ezt elsősorban már meglévő API-kra alapozva teszi, így lehet a Vulkan API sokrétű validációs rétegét használva a RenderDoc-ot alkalmazni. [43]

Fejlesztői környezet tekintetében nem feltétlenül szükséges komplex IDE-t használni, nem is létezik kimondottan Rust fejlesztésre készített típus. Egy egyszerű Text-Editor is tökéletesen megfelel, mert a buildelés folyamata parancssor segítségével is könnyen kivitelezhető a csomagkezelőnek köszönhetően. Léteznek bővítmények meglévő eszközökhöz, amelyek a más nyelvekben megszokott fejlesztést segítő jellemzőkkel bírnak, mint pl.: automatikus kódkiegészítés vagy kód navigáció. A lefordított CPU kód hibakeresése is egyszerűen kivitelezhető, mivel az LLVM eszköztárat használja a kód generálás során, így használható az LLDB hibakereső.

5.3 Szoftver és hardverigény

Specifikus hardverre nincs igény a fejlesztés/tesztelés során: egy modern funkciókészlettel bíró grafikus gyorsító az elmúlt pár évből megfelelő. Rendelkezem eltérő konfigurációkkal, ezért szélesebb körben tudom tesztelni az alkalmazásokat, ennek segítségével pedig jobb képet kapok arról, hogy az egyes összeállításokon milyen teljesítményt várhatunk.

Az asztali platformokon a futtatás során kelleni fog egy olyan megjelenítő backend, amire a WebGPU API-nak létezik absztrakciója, a legtöbb manapság használt modern operációs rendszer rendelkezik legalább egy ilyen backend-el. A fejlesztés során kelleni fog egy Rust fordító, amely futtatható állományt készít. A GPU által futatott kód hibakeresése miatt kelleni fog a VulkanSDK, ami tartalmazza a validációs rétegeket. Ezt az SDK-t alapvetően nem mellékelik a videokártya illesztőprogramok mellé.

6 IMPLEMENTÁCIÓ

Ezen fejezet célja bemutatni a fejlesztés során a kritikusabb komponensek fejlesztésének menetét, valamint az egyes felmerülő problémákat és azok megoldásait.

6.1 Ablakkezelés / Megjelenítő réteg

Grafikus alkalmazásról lévén szó szükségem van egy megjelenítő felületre, ahova rajzolni lehet. Az ablakkezelés és a hozzá tartozó események kezelése viszont elég körülményes feladat önmagában, mivel platformonként eltérő módon lehet ablakot létrehozni, de pl.: Linuxon két eltérő Megjelenítő Szerver protokoll van használatban, ami tovább bonyolítja a helyzetet és a webes környezetbe való integrálás még nem került említésre. Itt célszerűnek láttam már meglévő megoldások után keresni, mivel egy saját implementáció sok időt venne igénybe, valamint komplexitásában is jelentős.

A Winit Window Manager a három fő asztali platform mellett képes Androidon, és weben az ablakok létrehozására és kezelésére. A böngészőben alapvetően nem új ablakot hozunk létre amikor megjelenítünk valamit, hanem egy canvas elementet kell használni. Ez automatikusan viszont nem kerül létrehozásra, ezt explicit módon kell létrehozni [44]

Az ablaklétrehozás után ahhoz, hogy eseményeket kapjunk el kell indítani az event-loop-ot. Ennek a specialitása az, hogy hijack-eli azt a szálat, ami hívta és maga az event-loop-ot kezelő metódus sose fogja a hívónak visszaadni a végrehajtást. A weben ezt a működést a fejlesztők egy kivétel segítségével valósítják meg. Ezen megvalósítás a fejlesztés későbbi szakaszában jelentett problémát, mivel ez egy kezeletlen kivételnek minősül a böngésző szerint ezért további Promise-ok nem kerülnek létrehozásra, ami számos funkció működésképtelenségéhez vezet pl.: modell betöltés. A Promise-on belül futtatást nehéz elkerülni mivel a megjelenítő adapterhez való csatlakozás aszinkron módon történik. A probléma megoldására két módszert találtam alapvetően. Az egyik megoldás, hogy elkapjuk magát a kivételt, vagy pedig az eseménykezelő metódust az aszinkron metóduson kívül futtatjuk és egy saját esemény segítségével kapja meg az ablakkezelő.

Maga a WebGPU alapvetően egy standard, amely egy API-t biztosít, amelynek segítségével tudunk számításokat végezni a grafikus kártyán. Alapvetően egy Web Standardról beszélhetünk, de léteznek olyan könyvtárak, amelyek meglévő Graphics API-okra építkeznek.

Egy ilyen függvénykönyvtár a wgpu, amely a Mozilla gondozásában készül alapvetően Rust-nyelven íródott, de léteznek más nyelveken írt változat is pl.: wgpu-native ami kifejezetten C/C++ felhasználásra készült. Natív környezetben képes Vulkan, Metal, D3D12, D3D11 OpenGL, és WebGL megjelenítő backend-eken futni, valamint WebAssemblyn keresztül eléri a böngészők által biztosított WebGPU implementációt-t is.[45][46]

Mielőtt használni tudnánk a GPU-t, inicializálnunk kell azt. Első körön meg kell adni, hogy milyen grafikus backend-en keresztül szeretnénk futtatni a kódunkat, milyen extra funkciók kellenek nekünk, amik nem feltétlen találhatók meg az összes grafikus kártyán.

Ezt követően lehet lekérni egy számunkra megfelelő adaptert, amely a konkrét GPU-val lévő kapcsolatot hivatott reprezentálni. Ezt követően a megfelelő adaptert felhasználva lehet magát a logikai eszközt lekérni, valamint azt az utasítás sort, ahova az egyes utasítások fognak érkezni és továbbításra kerülnek a GPU felé.

Egy kép kirajzolásához első körben el kell látni a GPU-t adattal. Ezek rendszerint különböző Bufferek-be kerülnek. Egy Buffer egy lineáris memória blokk, amit az egyes GPU-n végzett műveletek során használni lehet. Minden ilyen tároló előre meghatározott mérettel kell, hogy rendelkezzen (bájtban megadva), valamint meg kell adni a felhasználási módját, amely az optimalizáción felül validációs célt is szolgál, tehát nem fordulhat elő az-az eset, hogy nem megfelelő Buffer kerül felhasználásra.

Továbbá lehet még feltölteni különböző textúrákat és hozzájuk tartozó TextureView-kat valamint a különböző mintavételezéshez használt Sampler-eket is. Egy textúra létrehozásakor is hasonló adatokat kell megadni, mint egy Buffer esetén, de itt meg kell adni magának a textúrának a formátumát, és a textúra dimenzióit. Egy TextureView-nak az a szerepe, hogy egy textúrának egy részére (vagy akár az egészére) hivatkozzunk, például különböző elemekre egy TextureArray-en belül. Egy Sampler-nek a szerepe, hogy meghatározza miként történik a mintavételezés egy adott textúrából, amely többféle változat is létezik.

Ahhoz, hogy az egyes Resource elemeket használni lehessen, egy úgynevezett BindGroupba kell őket tenni. Egy BindGroup szerepe, hogy egységben kezelje ezeket a Resource-okat, amiket az egyes ComputePass és RenderPass-ok során lehet felhasználni. Ez könnyebb kezelést eredményez és kevesebb hibát is mivel az egyes BindGroupok egy BindGroupLayout alapján kerülnek validálásra, amit explicit módon kell meghatározni. Az egyes LayoutEntryk szabják meg, hogy egy erőforrást, milyen Shader fázisban lesz elérhető és hogy milyen konkrét típust fogad maga az adott BindGroup adott indexű eleme.

A RenderPipeline határozza meg azon utasítások sorozatát, amely a bementi adatokat a különböző fázisokon keresztül módosítva a meghatározott kimenet adják. Hasonlóan egy BindGroup-hoz itt is szükséges egy Layout objektum, amiben az egyes BindGroupLayout-okat kell megadnunk. Ezen BindGroupLayout-ok sorrendje dönti el, hogy az adott Group milyen indexen lesz elérhető a Shaderekben. A Pipeline létrehozásakor kell megadni, hogy az egyes állomások során, hogy kell az adatokat kezelni. Pl.: a VertexState határozza meg hogy a megkapott bemeneti adatokat milyen módon kell értelmeznie VertexBuffer-ben ezen felül pedig, hogy maga a Vertex Shader-nek mi az belépési pontja, és hogy milyen modulban található.

Egy RenderPipeline tartalmaz Primitive Statet, ami a Raszterizáció és az egyes primitívek összeállításához szükséges paramétereket tárolja, valamint egy MultiSampleState-et ami az egyes élsimítások módját határozza meg. Ezen felül opcionálisan elérhető egy FragmentState, ahol a Fragment Shader kimeneteit lehet konfigurálni, és egy szintén opcionális DepthStencilState amely a mélységteszt és körvonalazáshoz tartozó attribútumokat tárolja.

Létezik a megjelenítésen kívül egy kimondottan csak számításra használt Pipeline is, ahol az egyes általános célú Compute Shadereket lehet futtatni. Itt az eltérés ott mutatkozik meg hogy nincsennek különböző állomások.

A végső szint a hierarchián belül, az egyes pipelineok futtatásáért felelős Pass-ok. Annak érdekében, hogy indítani tudjunk egy Compute vagy Render Pass-t szükség van egy CommandEncoderre, amely Kódolja a különböző Grafikus kártya műveleteket, valamint felelős a driver által kezelt erőforrások (Texturák, Bufferek, stb…) a GPU-ról/ra történő másolásáért. Amennyiben már nem kívánunk több utasítást rögzíteni, akkor kapunk egy CommandBuffer-t, amit elküldünk a GPU-nak, hogy hajtsa végre a tárolt utasításokat.

Egy RenderPass létrehozása során határozhatjuk meg mik lesznek az egyes kimenetek céltextúrái továbbá, hogy az egyes műveletek során az adott textúrát milyen módon kell kezelni. A létrehozást követően tudjuk megadni, hogy milyen Pipeline-t szeretnénk használni az adott RenderPassban, valamint, itt kell az egyes BindGroupokat is beállítani. Ezen BindGroupok validálása itt történik az adott Pipeline-ban megadott BindGroupLayout-nak megfelelően. Amennyiben a kettő nem egyezik hibát kapunk és nem fog végrehajtódni az adott parancs. Magát a kirajzolást különböző rajzoló parancsokkal (drawCall-okkal) lehet kiadni, de magát a kirajzolandó objektum adatait először fel kell vinni egy Vertex Buffer Object-be, amely az objektum diszkrét pontjait tartalmazza, és egy opcionális Index Buffer-be, ami a különböző pontok összetartozását hivatott reprezentálni.

A ComputePass ennél lényegesen egyszerűbb felépítését tekintve. Nem tartalmaznak explicit ki és bemenetek, hanem a BindGroupokon keresztül elérhető Bufferek és Texturákat tudják használni írásra-olvasásra. Itt is meg kell adni, hogy milyen Pipeline-t szeretnénk használni és a különböző Resource-okat is azonos módon tudjuk csatolni. Magát a GPU kernel indítását dispatch parancsokkal tudjuk, ahol meg tudjuk adni a futtatás során felhasználni kívánt szálak /blokkok számát.

A fent említett elemek felhasználásával került kialakításra a megjelenítő réteg. Az általam alkalmazott megoldás egy Deferred Rendering Pipeline-t használ, amely manapság elég elterjedt megoldás komplex jelenetek megjelenítésére. A legnagyobb eltérés itt az, hogy az egyes RenderPass-ok alatt nem magára a végső kimeneti textúrára rajzolunk, hanem Graphics buffer-eket (G-Buffer) alkalmazunk. Egy ilyen buffer képernyővel megegyező nagyságú textúra, ami olyan adatokat tárol, mint pl.: pozíció, textúrák vagy normál vektorok. Ehhez hasonló textúra a depth-buffer (z-buffer), ami mélységi információt

tárol, és az egyes elemek térbeli helyzetét tárolja, amit a későbbi fázisok során arra használ, hogy az elemek helyes sorrendjét nyomon kövesse. Amint ezek a buffer-ek, generálásra kerültek egy későbbi időpontban egy külön RenderPass kerül végrehajtásra, ami ezen textúrákat felhasználva hajtja végre a végső árnyékolás folyamatát. Ez több okból kifolyólag is teljesítménynövekedést eredményez. A legfőbb ok, hogy minimalizáljuk a fölösleges fragment shader futásokat. Míg egy tradicionális Forward Rendering megoldásnál olyan objektumokra is kiszámolnánk fényforrások kontribúcióját, amik a mélység teszt által eldobásra kerülnének, vagy a látótéren kívül vannak, ezáltal növelve a futásidőt. A megvalósítás során én további Render és Compute-pass-okat alkalmaztam, amely egy jelenet más-más részeit hivatott módosítani.[47]

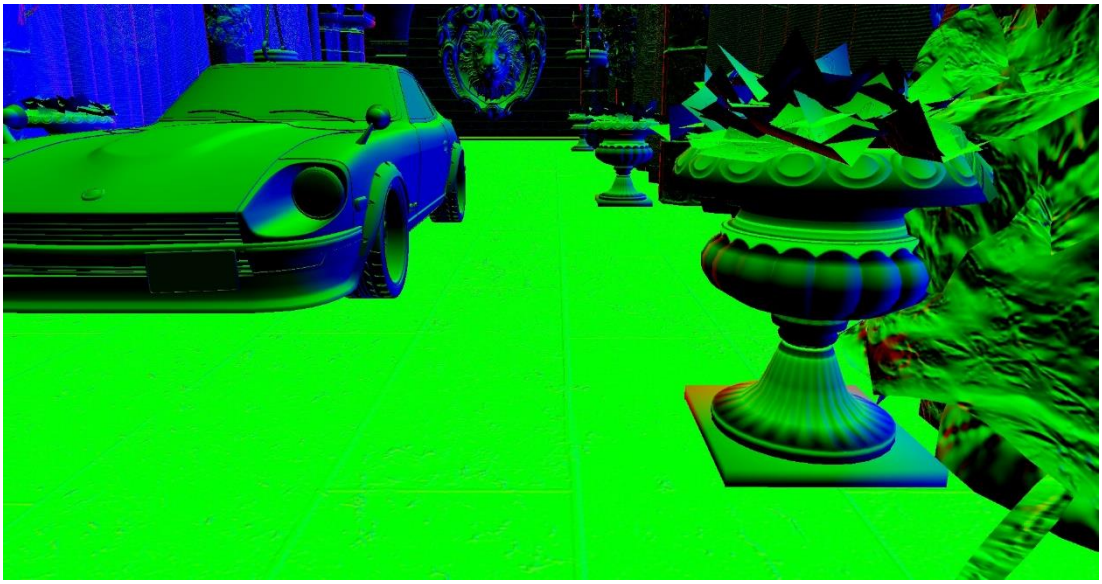
Első lépésként az árnyékok szimulálásához szükséges textúra generálása hajtódik végre. Ez úgy működik, hogy a jelentben lévő fényforrás (ami a napot szimulálja) „szemszögéből” kerül kirajzolásra az összes objektum. Itt a mélységteszt során generált adatok kerülnek eltárolásra(6.1 ábra).

6.1. ábra - Árényékgenerálási fázis
(saját készítésű ábra)

Ezt a lépést követte a Graphics Buffer generálása. A G-buffer 3 textúrából áll, melyek rendre az egyes objektumok pozícióit, a hozzájuk tartozó normálvektorokat, valamint a textúrákat tartalmazzák. Mivel az egyes bufferek 2D-s textúraként vannak létrehozva, ezért ezek könnyen megjeleníthetők és így egyszerűen lehet ellenőrizni, hogy mi kerül az egyes bufferekbe. (6.2-6.4 ábra)

6.2. ábra - Generált graphics buffer pozíciókat reprezentáló textúrája
(saját készítésű ábra)

6.3. ábra - Generált graphics buffer textúrainformációt tároló textúrája
(saját készítésű ábra)



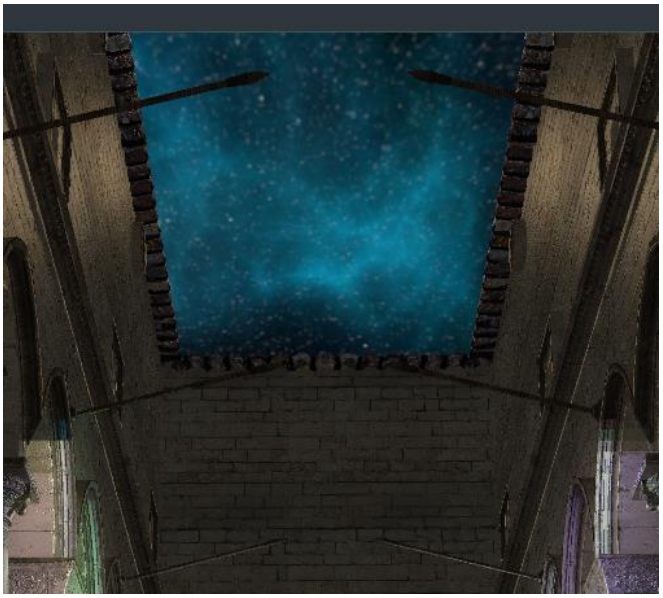
6.4. ábra - Generált graphics buffer normál vektorokat tartalmazó textúrája
(saját készítésű ábra)

Következő lépésként egy optimalizációs lépés megy végbe, amit Tiled Light Culling-nak hívnak, és a következőképpen működik: felosztjuk a képernyőt 16x16 pixeles blokkokra, majd ezen blokkokban meghatározzuk mely fényforrások vannak hatással az adott blokkra és csak az adott blokkon belüli fényforrásokat felhasználva fogunk fényelni. Ez a folyamat egy ComputePass segítségével zajlik.

Az optimalizációt követően a generált Graphics Buffer, és a csak az adott blokkhoz tartozó fényforrás információkat felhasználva zajlik a fényelés és az árnyékolás, ami már a kirajzolandó kép textúrájába fog kerülni. (6.5 ábra)

6.5. ábra - Árényékolás és fényelési fázis
(saját készítésű ábra)

A minél részletgazdagabb jelenet érdekében célszerűnek láttam implementálni dinamikusan módosítható hátteret, amelyet külön RenderPassként rajzolunk ki. Ennek az oka, hogy a már meglévő jelenet minden esetben előtérben van a háttérhez képest és a mélységi buffer-t felhasználva, csak azokra a helyekre kell rajzolni a hátteret, ahol még nincs előtte más objektum ezáltal is csökkentve a fölösleges műveletek számát. (6.6 ábra)



6.6. ábra – Háttér kirajzolás
(saját készítésű ábra)

A fizikai szimulációs rendszer könnyebb tesztelése érdekében bevezettem egy opcionális RenderPass-t, amely az egyes objektumok ütközőzónáit vizualizálja. Végső lépésként kerül a képre a felhasználói felület. (6.7 ábra)

6.7. ábra – Felhasználói felület megjelenítés
(saját készítésű ábra)

6.2 Entity Component System

Az alkalmazás maga számos eltérő típusú adatot kezel, a különböző Modelleketől a fent említett GPU erőforrásokon át, az egyes entitásokhoz tartozó komponensekkel. Maga a Rust nem támogatja a klasszikus értelemben vett Objektum Orientált paradigmát, ezért célszerűnek láttam olyan megoldások felé nézni, amelyek nem OOP alapokon fekszenek. Az irodalomkutatás során is említést tettem a manapság elég elterjedt az Entity Component System architekturális tervezési mintáról, amelynek az egyes entitásokhoz tartozó komponensek nem szorosan kapcsolódnak egymáshoz, valamint az egyes komponenseken végzet műveletek sem magában a komponensben van megvalósítva, hanem egy külön erre a célra létrehozott Systembe. Ez a megoldás amellett, hogy dinamikusan módosíthatóvá teszi az egyes entitásokat azáltal, hogy nem az egyes komponensek nem szorosan kapcsolódnak hozzájuk. Ha az adatok megfelelően szét vannak osztva és minimális függőség van közöttük, akkor az egyes műveletek jól párhuzamosíthatóak, ezáltal extra teljesítményhez juthatunk. Mivel ilyen rendszer megtervezése és implementálása, és tesztelése más fontosabb rendszerek fejlesztésétől venné el az időt már meglévő implementáció keresését láttam célravezetőbbnek.

A Specs ECS volt az a függvénykönyvtár, ami minden számomra fontos funkcióval rendelkezik. Többek között azért is erre esett a választás, mert magát a megjelenítést is bele lehet integrálni, ami mindig minden iteráció legvégén fut és a fő szálon. Nem csak Entitásokat és a hozzájuk tartozó Komponenseket lehet kezelni, létezik egy úgynevezett Resource típus is (amely különbözik a WGPU által definiált Resource-októl), amely egy Singleton-hoz hasonlóan az adott típusból 1 félét képes tárolni, ami hasznos például egy DeltaTime tároláshoz, amely elengedhetetlen az egyes rendszerek képkockafrissítés független működéséhez, mint például a Fizika.[48]

Jelenleg az alkalmazás által kezelt adatok nagy része az Entity Component Systemen keresztül érhető el. A különböző RenderPass-ok és ComputePass-ok 1-1 Systemben találhatók, az ezek futtatásához szükséges komponensek pedig mennyiségüktől függően dedikált ComponentStorage vagy Resource-on keresztül érhetőek el.

6.3 Szkript környezet

A szkript környezet megvalósítása JavaScript alapú, mivel ez minden böngészőben elérhető, valamint natív platformokon lehetőség van a böngészőkben használt JavaScript motorok integrálására. Az elérhető opciók közül a Google V8 a leggyakrabban használt. Ez adja az alapját a Node.JS, valamint a pár éve megjelent Deno futtatókörnyezetnek többek között.

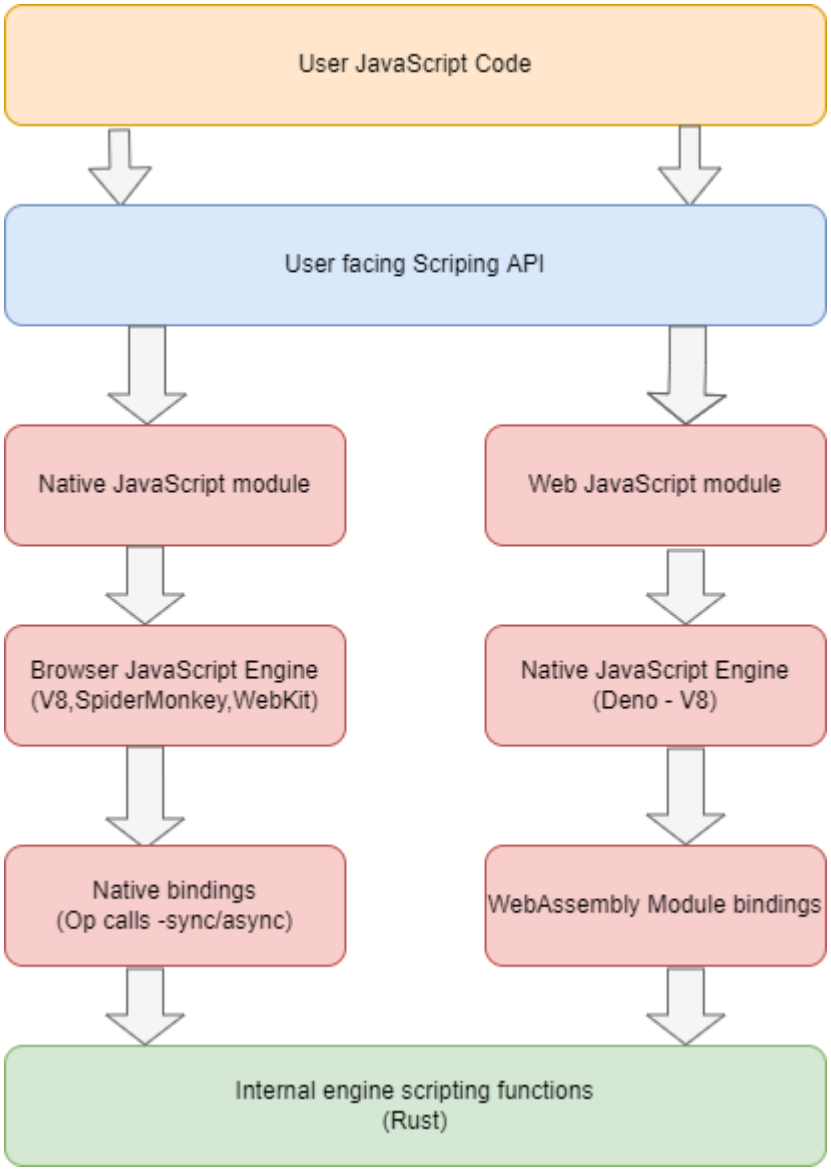
A Google biztosít dokumentációt, hogy hogyan kell meglévő alkalmazásba integrálni a v8 JavaScript Engine-t, viszont egy C++ kódbázisról beszélünk, ami komplikálja a helyzetet. A Rust képes a Foreign Function Interface-en keresztül kommunikálni C/C++ kóddal, de a memória kezelési megkötések miatt ezt unsafe kód mellett történik. A prototípus készítés során, elegendő volt egy olyan függvénykönyvtár, amely ezt a

problémát kezeli, viszont a fejlesztési későbbi szakaszában olyan JavaScript funkciók is felhasználásra kerültek, amelyekre alapvetően a V8 nem ad kész megoldást, hanem rábízza azt az integrálóra. Ilyen az Event Loop, amely többek között az kezeli a különböző micro- és macro- Taskokat, amik az egyes aszinkron kódok futtatását végzik, valamint a modern JavaScript alapját képező modulok használata is nehezen megoldható.

A Deno Runtime is csakúgy, mint a V8 integrálható más alkalmazásba a Deno-Core könyvtáron keresztül, amely a fent említett problémákat orvosolja, valamint egyszerűsíti a két nyelv közötti kommunikációt, de van olyan terület, ahol nem ad egyszerű megoldást. Ahhoz, hogy az eseményvezérelt architektúra implementálásához szükség van callback függvényekre, amiket a belső komponensek hívnak a megfelelő időpontban. A Deno nem ad egyszerű módot ennek a kezelésre, alapvetően csak fix méretű bináris buffereket képes kezelni az előre definiált kommunikációs rétegje, viszont a V8 képes függvényeket továbbítani, viszont a callback-et fogadó függvényt manuálisan kell implementálni, és elérhetővé tenni JavaScript oldalon. Kommunikáció a két réteg között alapvetően szinkron hívást és aszinkron hívást támogató függvényekkel történik. Aszinkron hívás esetén, meg kell jelölni a függvényt, hogy ne várjon az event-loop annak befejezéséig különben holtpontra fut az alkalmazás, mivel a motor belső event-loop-ján fut a Deno event-loop.[49], [50]

6.4 Közös kódbázis

A JavaScript oldalról maga a kommunikáció egységes módon történik, de ez az Engine oldalról eltérő módon kell kezelni a webes és natív környezetben is. Ezen részt nem lehet egységes szintre hozni, mivel más a beérkező adat struktúrája, és az egyes callback metódusok szignatúrája is merőben eltér. Ezt a problémát csak platform specifikus kóddal tudtam orvosolni, amely ugyanazon függvényeket az adott platformnak megfelelő módon implementálja, majd ezen függvények bemeneteit egy közös belső reprezentációra alakítja (JSON reprezentációra), majd ezt követően már közös metódusok végzik el az egyes műveleteket. Készítettem szimpla adattároláshoz olyan struktúrákat is, amelyek függetlenek a belső reprezentációtól és a feladatuk csak az egyes adatok továbbítása két kódbázis között (6.8 ábra).



6.8. ábra - Szkript környezet és a belső egység kommunikációja
(saját készítésű ábra)

A két környezet közötti kommunikáció eseményvezérelt alapon működik. JavaScript oldalon fel tudunk iratkozni bizonyos eseményekre, amelyeket majd az alkalmazás fog a megfelelő időben elsütni. Ez a megoldás nagyon közel áll a böngészőben és a Node.JS-ben alkalmazott modellhez. Az egyes callback function-ök az ECS-ben tároldónak és külön System felelős ezen események megfelelő hívásáért. Külső függvények tárolása is platform specifikus és a meghívásuk sem egységes ezért ezen részek is 1-1 platform specifikus implementációval rendelkeznek. Mindkét környezetnek szüksége van egy kompatibilitási rétegre, annak érdekében, hogy a felhasználói szinten ne kelljen kezelni, hogy a megfelelő függvényhívások menjenek végbe. Ezt a réteget Weben a WebAssembly modul automatikusan generálja külön JavaScript fájlként, natívon viszont manuálisan kell ezt létrehozni.

A legnagyobb problémát az implementáció során ezen a területen viszont az okozta, hogy a JavaScript kód bármilyen időpillanatban hívhat belső függvényeket, ami azért problémás mert könnyen versenyhelyzet alakulhat ki. Szinkron hívások esetén nem merül fel ez a probléma mivel ezek a hívások a stack-en hajtódnak végre, és addig nem kerül vissza a vezérlés a futtatókörnyezethez míg a stack nem üres. Aszinkron kontextusban viszont párhuzamosan futhat egy modell betöltés, valamint maga a Render kód, ez a helyzet Rust-ban mindig azonnali futásidejű hibával megállítja a program futását, mivel egyidőben vagy korlátlan számú olvasási referencia létezhet, vagy egy írási referencia a kettő egy időben sosem. Klasszikusan különböző szinkronizációs primitívek segítségével lehet kezelni, de a weben a kód alapvetően egy szálon fut ezért könnyen holtpontra lehetne futni, ha blokkolni próbálnánk, amit a böngésző a fő JavaScript szálon nem enged, csak Web Workereken belül. Az egyidejű hozzáférés megoldására a következő ötlet vált be: Az aszinkron függvény nem használ olyan változót, amit más is használhat vele egyidőben, végrehajtja a kívánt műveletet majd a függvény végén, egy egyirányú csatorna kerül létrehozásra azzal a céllal, hogy értesítse ezt a függvényt. A feldolgozott adatot és az értesítést küldő objektumot elküldjük az ablakkezelő event-loop-jára, mint egy speciális esemény, mert erre ad lehetőség az ablakkezelő könyvtár és addig ameddig nem kapunk választ addig várunk. Ez a várakozás aszinkron módon történik, tehát visszakerül a vezérlés a JavaScript Runtime-nak, befejeződik a JS event loop futása, így visszakerül a Rusthoz a végrehajtás, aki elkezdi feldolgozni a kapott eseményt. Ha megtörtént a feldolgozás, akkor az eseményben megkapott küldő objektum értesíti azt a folyamatot, aki ehhez a csatornához tartozik, és a JS event loop következő iterációjában tovább fut az aszinkron kód és befejeződik a függvényhívás (Ha van visszatérési érték akkor azzal tér vissza). A csatornák közötti kommunikáció során nem fordul elő versenyhelyzet mert olvasási referencián keresztül történik a küldés, illetve a fogadás, ezt egy úgynevezett Interior Mutability Pattern segítségével lehet kivitelezni, amit a problémát előidéző objektumnál nem lehet kivitelezni. [51], [52], [53]

6.5 WebAssembly

A legegyedibb aspektusa az alkalmazásnak a böngészőben való futtathatóság ugyanazzal a kódbázissal. A natív kód böngészőben való futtathatósága, egy pár évvel ezelőtt még egyáltalán nem volt megoldott. Maga a WebAssembly (Wasm) is egy Compilation Target hasonlóan, mint az x86_64 vagy aarm64. A Rust extra függőség nélkül képes Wasm kódot generálni, amely minden WebAssembly-t futtatni tudó környezet képes kezelni. Ha az alkalmazás használ más, nem platformfüggetlen funkcionalitást akkor ezt a kódot összekell kötni az adott platformon futó kód megfelelő részeivel (ezt glue-kódnak is szokás nevezni). Ez a működés viszont már nem alapvető Rust funkció, erre egy külön eszközt kell használni, amely a wasm-bindgen. Ezen eszköz nem csak a kódbázisok közötti kapcsolatot teremti meg, hanem generál automatikusan TypeScript fájlokat az egyes WebAssembly modul által exportált függvényeknek, és a böngészőben importálható kóddá készíti elő. A böngészőbe való importálást ECMAScript modul

segítségével lehet megtenni felhasználói oldalon, ami nem különbözik az egyszerű JavaScriptben írt modulok importálásától. Különbség ott mutatkozik meg, hogy a WebAssembly modult inicializálni kell mielőtt használni lehetne. A webes futtatáshoz szükséges állományok elérése egy mezei http webszerver használatával is megoldható, ha a szerver képes a megfelelő média típusokat kiszolgálni.[54]

6.6 Asset pipeline

Korábban megemlítettem a különböző Asset-ek betöltésének problémáját. A böngészőben nincs Fájlrendszer hozzáférés, ahol egyszerűen tudtam volna kezelni a fájlokat. A böngészőben lévő Fetch API-val töltöm be az alkalmazásba a futás során szükséges fájlokat, ami egy kérést küld a webszerver felé majd a kapott bájtsorozatot betöltve dolgozza fel. Ez a metódus hasonlóan a többi böngészőben lévő I/O művelethez aszinkron módon működik, elsősorban itt jött elő a versenyhelyzetből adódó probléma.[55]

Az egyes fájlok betöltésén felül a feldolgozás is említést érdemel. Az alkalmazás kezdeti stádiumában a Wavefront OBJ fájlformátumú modelleket kezelt, viszont ez több szempontból is problémás volt, például az egyes modellekhez tartozó material data és textúrák nem képezik a fájl részét, hanem külső hivatkozásként vannak megjelölve, valamint az egyes objektumok diszkrét pontjai sem a leghatékonyabban vannak tárolva, de a formátum legnagyobb hiányossága, hogy nem támogatja a manapság elterjedt Physically Based Rendering-hez használt Material property-ket és animációkat sem tárol. Ezen hiányosságokkal tisztában célszerűbbnek láttam egy olyan formátum utáni keresést, amely a fenti hiányosságokra megoldást ad. A glTF fájlformátum orvosolja ezen problémákat, mivel az egész modellt és a hozzá tartozó kiegészítő adatokat egy fájlként tárolja bináris formában, amit lehet tömöríteni is, amely a webes futtatás során is pozitívum, az adatok már a GPU kompatibilis formában vannak tárolva ezzel időt spórolva a feldolgozás során.[56], [57]

6.7 Fizika

Az Entity Component System miatt bármelyik entitást fel tudjuk ruházni fizikai tulajdonságokkal magáért a szimuláció futtatásáért a Rapier Physics Engine felelős. A különböző modellek esetében az egyes ütközőzónák felépítése a modellt leíró pontok által egy ConvexHull kerül létrehozásra, majd az egyes ütközőzónák egy CompositeShape-be kerülnek elkerülve azt, hogy a modell egyes alrészei ütközés eseményt generáljanak más ugyanazon modellhez tartozó ütközőzónákkal. A fizikai szimuláció futtatása is egy külön Systemben található, amely az egyes objektumok transzformációit módosítja a szimulációban kiszámított értékekre. Lehetőség van továbbá olyan ütközőzónák kialakítására is, amelyek nem egy modellhez tartoznak, hanem egyfajta triggerként viselkednek (6.9 ábra).[58]

6.9. ábra - Generált ütközőzónák vizualizálása
(saját készítésű ábra)

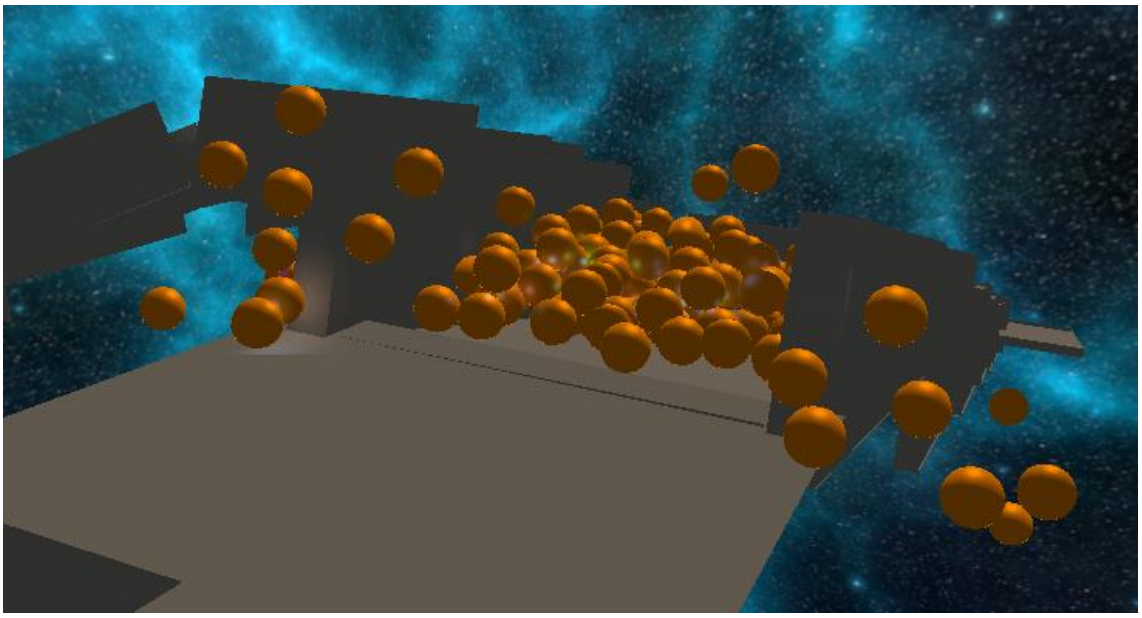
6.8 Felhasználói felület

Az irodalomkutatás során Dear ImGui tűnt a legjobb választásnak, mivel a Library-hez készült Rust bindig-okat terveztem használni, de weben nem megoldott a használata, ezért más alternatívák után kellett, hogy nézzek. Az egui bizonyult egy jobb alternatívának, tisztán Rust-ban íródott, van WebAssembly támogatás, és minimális függőségekkel rendelkezik. A felhasználói felület elsődleges funkciója az alkalmazás futásával kapcsolatos információk megjelenítése, erre többek között azért volt szükség, mivel a grafikus kártyán futó kód hibakeresése nem lehetséges a böngészőben így kellett egy olyan mód, ahol vizualizálni lehet a különböző buffer-ek tartalmait például.[59]

7 EREDMÉNYEK ÉRTÉKELÉSE

A követelményspecifikációban meghatározásra került a könnyű kezelhetőség, azon felül, hogy az egyes paramétereket úgy lehessen változtatni, hogy magába a belső rendszer kódját módosítani kellene. Ezen funkciók validálására 2 példa alkalmazás fejlesztését tűztem ki célul, ezzel bizonyítva, hogy a rendszer újrafelhasználható és a felhasználó által írt kód nincs szoros függésben a motorral.

Az első példaprogram a megjelenítő rétegtől független komponensek hatékony működését hivatott bemutatni, elsősorban a fizikai szimulációt, az Asset Pipeline-t, valamint a szkriptrendszert. Az alkalmazás működése az alábbi módon néz ki: Az inicializációs fázisban betöltésre kerülnek a felhasználni kívánt modellek, amely áll egy akadálypályából és az itt végig haladni próbáló golyók. Ezt követi az egyes entitások létrehozása, ahol a fizikai rendszer automatikusan kiszámolja az egyes entitásokhoz tartozó ütközőzónákat. Futás közben a pálya egyes elemei folyamatosan mozognak dinamikus akadályt képezve (7.1 ábra).



7.1. ábra - Első példaprogram futás közben
(saját készítésű ábra)

A másik tesztkörnyezet célja a megjelenítő réteg hatékonyságának tesztelése. Itt a program több komplex modellt tölt be, majd több azonos példányt hoz létre eltérő pozíciókban. Ezen felül létrehozásra kerül még nagy számú fényforrás, melyek folyamatos mozgásban vannak így növelik a számításigényt mivel folyamatosan vizsgálni kell a pozíciójukat, hogy megkapjuk az adott blokkban melyek a számunkra hasznosak (7.2 ábra).

7.2. ábra - Második példaprogram futás közben
(saját készítésű ábra)

7.1 Teljesítménymérés metodikája

Egy grafikus alkalmazás teljesítményét leggyakrabban az egy másodperc alatt megjelenített képkockák (Frames Per Second - FPS) számával azonosítjuk, minél többször jelenítünk meg valamit a képernyőn annál folyamatosabb a képek közötti váltás ezáltal megőrizve a mozgókép illúzióját. Ez az egy érték önmagában nem ad átfogó képet, mivel nem csak a képkockák száma fontos, hanem ezek eloszlása is többek között. Ha a program futása közben nagy az ingadozás az értékek között az ugyanolyan rossz, mintha fele annyi lenne az FPS. Esetemben további nehézséget okozott az a tény, hogy a böngészőkben ez az érték maximalizálva van az adott megjelenítő maximális képfrissítési rátájára, ami a monitorok nagytöbbségén 60Hz, ami azért kell, hogy az úgynevezett Visual Tearing jelenséget minimalizálja.

A fent említett indokok miatt célszerűbbnek láttam ezen metrika helyett inkább egy képkocka megjelenítésénél alkalmazott Render és Compute Pass-ok futásidejét használni, mivel így jobb képek kapok az egyes tesztesetek milyen módon befolyásolják a megjelenítő réteg egyes részeit. Ennek a metodikának egyedüli negatívuma, hogy jelenleg csak Vulkan backend-en használható, mivel a WebGPU API még erősen fejlesztési fázisban van és az egyes funkciók más-más implementációs szinten vannak. Azokon a platformokon, ahol nincs hozzáférés ezen adatokhoz, ott a teljes Frame megjelenítéséhez szükséges időt vizsgáltam.

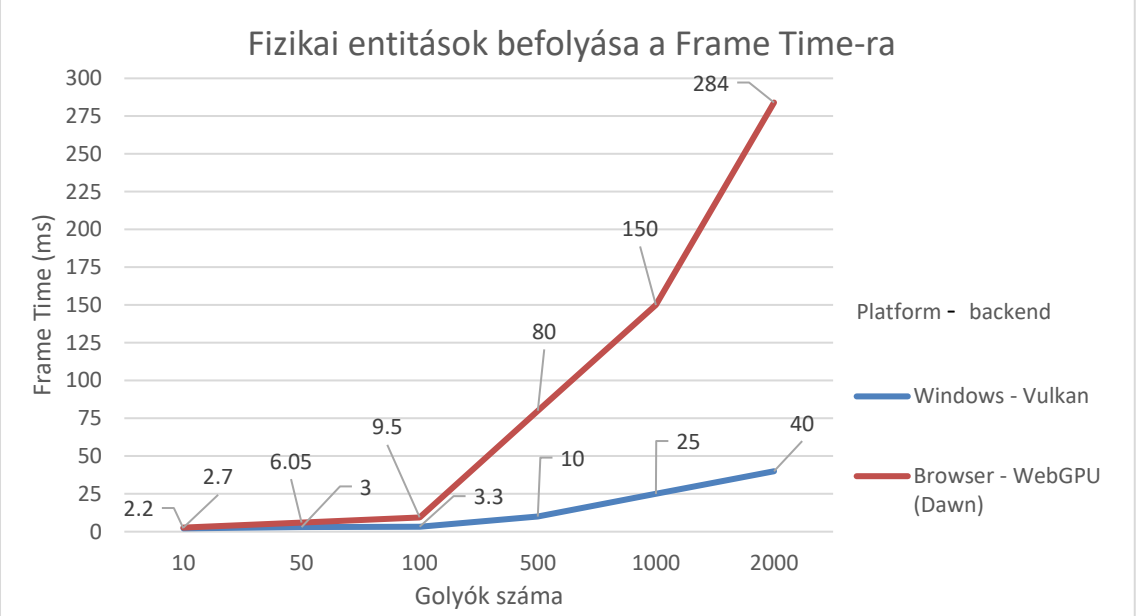
A tesztelés az alábbi hardver és szoftver konfiguráción zajlott:

- CPU - Intel core i5 8600k @ 4.3 GHz
- GPU - Nvidia GTX 1060 6 GB (Driver verzió: 512.15)
- RAM - DDR 4 16 GB @3200 Mhz
- OS – Windows 11 Version 21H2
- Browser – Google Chrome Canary 103.0.5011

A teljesítménymérés eredményei során a szinteket a manapság elfogadott 16.67 és 33.33 milliszekundumos egy képkocka kirajzolásához szükséges átlagidőket tekintem. (=33.33 az elégséges, <=16.67 az optimális).

7.2 Mérések eredményei

Az első példaprogram elsődleges funkciója az volt, hogy a rendszer könnyű kezelhetőségét mutassa be, ezen felül pedig a fizikai szimuláció hatását a rendszer egészére. Az egyes tesztesetek során folyamatosan növekvő entitás számokkal figyeltem meg, hogy a megnövekedett számításigény miként befolyásolja a rendszer egészét. Mivel a szimuláció alapvetően a CPU-n fut ezért elsősorban itt nem tartottam hangsúlyosnak az egyes Pass-ok idejének vizsgálatát (7.3 ábra).

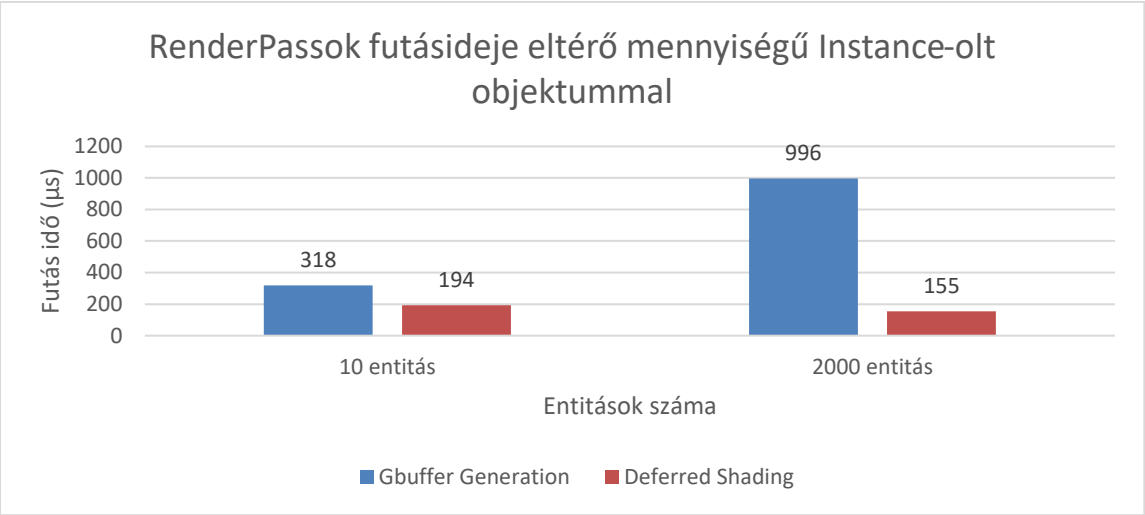


7.3. ábra - Fizikai entitások befolyása a Frame Time-ra
(saját készítésű ábra)

Megállapítható, hogy viszonylag nagy entitás számnál is még elfogadható időt kapunk natív környezetben, viszont a böngészőnél CPU limitbe ütközünk és már az elégségesnek mondható időt is túllépjük relatív kevés entitásnál. Ez a teszt alapvetően a legrosszabb helyzet vázolja fel, amikor nagy számú kontaktus megy végbe minden iterációkor. A szimuláció előrehaladtával ez a számításigény mérséklődik, mivel a fizikai rendszer csak

úgynevezett aktív objektumokat szimulál, akik már egy ideje nyugalmi állapotban vannak őket nem fogja figyelembe venni, kivéve, ha ütközik mással vagy explicit módon „felébresztjük” valamilyen behatással.

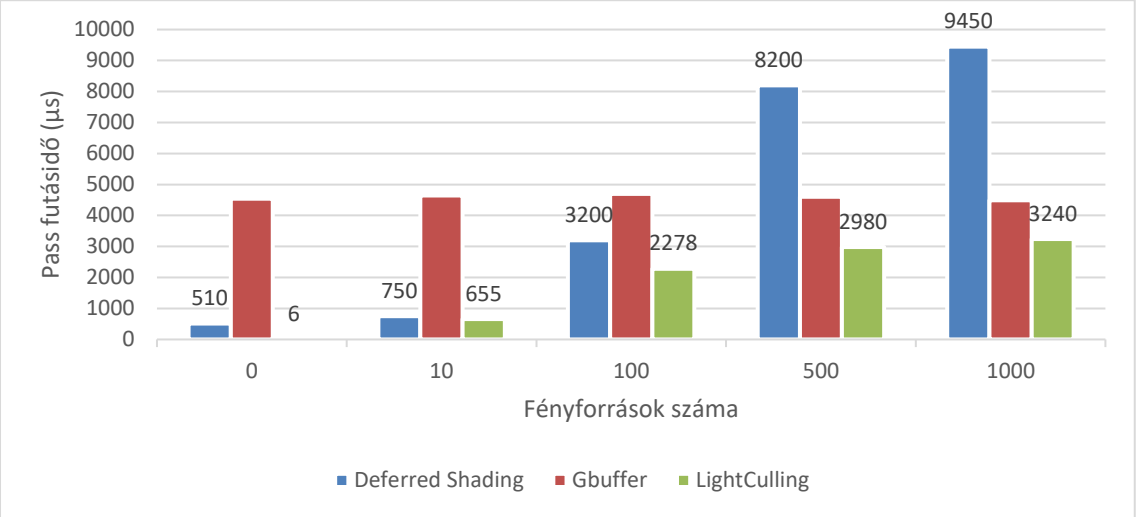
Ugyanezen környezet a megjelenítő réteg egy optimalizációját is jól tudja szemléltetni, amiről még nem tettem külön említést ez az Instancing technika. Az alapötlet a következő: ugyanolyan modellel rendelkező entitásokat egy draw hívással jeleníti meg, a példányonkénti eltérő adatok (transzformációk, eltérő material property-k) külön buffer-ből kerülnek kiolvasásra, amiket a kirajzolás során a példány azonosítójával kérünk le (7.4 ábra).



7.4. ábra – RenderPass-ok futásideje Instance-olt objektumokkal
(saját készítésűt ábra)

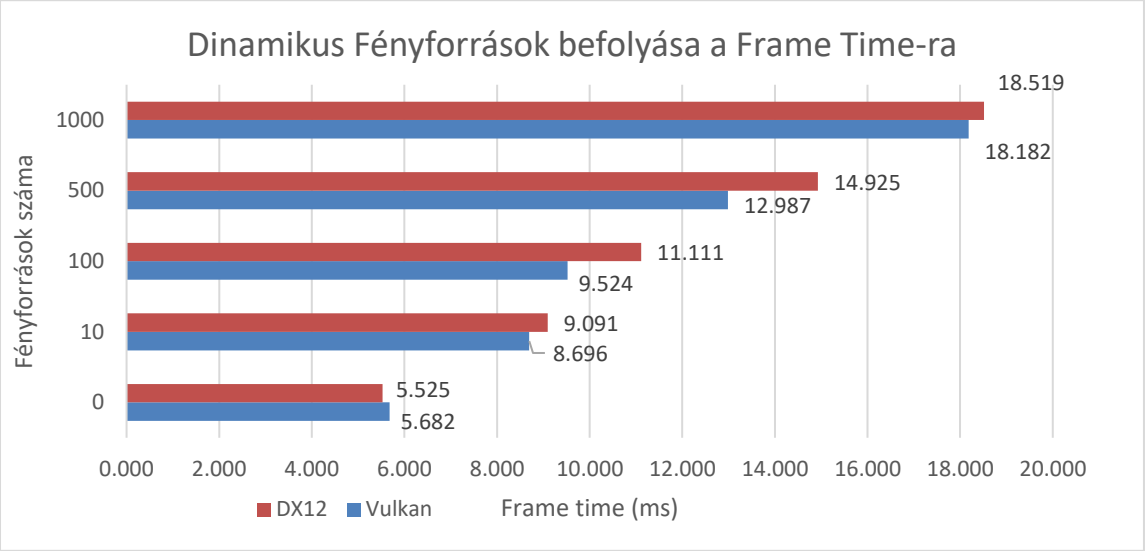
Megfigyelhető, hogy még egy ilyen nagy mértékű (200x) növekedés esetén is nem nő szignifikánsan a futásidő, ez annak tudható be, hogy minimalizálva van a GPU-ra történtő adatmásolás, ami sok időt venne igénybe, ha pl.: A objektum után egy B objektumot majd újra egy A objektumot kellene kirajzolni. Ez komplex modellek esetén nagy overhead-el járhat. A többi RenderPass / ComputePass-nál nem volt észrevehető különbség, ezért nem kerültek fel a diagramra.

A második alkalmazás az implementáció során ismertetett Deferred Rendering megvalósítás előnyeit hivatott bemutatni, valamint a Tile Based Light Culling optimalizáció hatását. Tradicionális megközelítésben jelentősen nő a számításigény, mivel minden egyes új objektum kirajzolásakor az összes fényforrást figyelembe kell venni, az egyes számításoknál. Első körön a különböző Render és Compute Pass-okra gyakorolt hatását vizsgáltam (7.5 ábra).



7.5. ábra - Dinamikus fényforrások befolyása a különböző fázisokra
(saját készítésű ábra)

Megállapítható, hogy a GBuffer generálást nem befolyásolja, hogy mennyi dinamikus lámpa van az adott jelenetben, ez volt az elvárt működés mert ebben a fázisban nem történik fényekkel kapcsolatos számítás. Az optimalizációt végző Light Culling Pass futása arányosan növekszik a lámpák számával, de lesz egy pont, ahol ez kisebb mértékben fog történni, mivel egy adott 16x16 pixeles blokkot jelenleg maximum 128 db fényforrás befolyásolhat. Végezetül a Deferred Shading fázis, ami a tényleges fényeléshez szükséges számításokat végzi egyenesen arányosan növekszik. A korai prototípusnál, ahol nem volt külön optimalizációs lépés fordított arányosság volt a Frame Time és fények számával, pedig már akkor is Deferred megoldás volt alkalmazva, de önmagában annak megvannak a korlátjai. Ennél a tesztnél a Webes backend nem volt használható, így DirectX12-vel helyettesítettem, de a különbség marginális volt ezesetben (7.6 ábra).



7.6. ábra - Dinamikus fényforrások befolyása a Frame Time-ra
(saját készítésű ábra)

Megállapítható, hogy még jelenetős számításigény növekedés esetén is, közel vagyunk az optimális időintervallumhoz, valamint az egyes megjelenítő Backend-ek sem jelentenek számottevő különbséget.

8 ÖSSZEFOGLALÁS

8.1 Elért eredmények

A megoldásom során sikerült implementálnom olyan funkciókat, amelyek önmagukban is számos kihívást rejtettek, de ezek összehangolt működése volt talán a legnagyobb megoldandó feladat. Az alkalmazás legegyedibb aspektusa az a platformokon között hordozható GPU kód, amelyre eddig nem volt példa korábban, különösen nem a böngésző – natív kontextusban, ebben nyújt újat az általam elkészített alkalmazás a már meglévő hasonló megoldásokkal szemben. Ehhez szorosan kapcsolódik a CPU oldali kód böngészőben való futtatása a WebAssembly segítségével oly módon, hogy ne kelljen külön kódbázist fenntartani a különböző architektúrával rendelkező platformokon, és ezen megkötés ne menjen a teljesítmény rovására. A motort felhasználó és fejlesztő szemszögéből is lényeges szempont a különböző Asset-ek formátuma, de más-más okból kifolyólag. A glTF formátum kezelése mindkét fél számra előnyös, az egyik legnagyobb pozitívum, hogy a formátum kezeli a Physically Based Rendering-hez szükséges adatokat, amellyel az egyes jelenetek élethűsége jelentősen megnő. A felhasználó szemszögét figyelembe véve lett gondosan megtervezve és kivitelezve a szkript rendszer, amely egységes módon vezérli a motor egyes elemeit a különböző JavaScript környezetek között. A megjelenítő réteg különös figyelmet kapott, mivel az egyik fő követelmény a hatékony megjelenítés volt 3D-s kontextusban. A Deferred Rendering model mellett további optimalizációs technikák is felhasználásra kerültek pl.: Tile based Light Culling, Reversed Depth Buffer[60], vagy az Inifinite Perspective Projection[61]. Az egyes részegységek modularitása érdekében, valamint az esetleges későbbi fejlesztés segítésében került implementálásra az Entity Component System architekturális tervezési minta, ami laza csatolást valósít meg az egyes komponensek között, valamint elkülöníti az adatokat az üzleti logikától. A minél széleskörűbb felhasználás érdekében került integrálásra egy 3D-s fizikai szimulációs függvénykönyvtár. A megvalósításról összeségében elmondható, hogy a követelményspecifikációban foglaltakat teljesíti és jó alapot biztosítana a további fejlesztésre.

8.2 Limitációk

Az eredmények mellett persze vannak olyan limitációik is, amiket orvosolni kellene. A legkritikusabb ezek közül a fájlok betöltése és feldolgozása natív platformon, ami jelenleg szinkron módon történik, így, ha már a program futása közben indítanánk egy modell betöltés kérelmet az blokkolná az egész alkalmazást ameddig nem menne végbe. Ezen a vonalon haladva az Entity Component System is jelenleg csak 1 szálon fut, ami nem túl előnyös mert maga a rendszer alapvetően úgy van kitalálva, hogy Fork-Join alapon képes legyen párhuzamos működésre ezáltal gyorsítva az egyes CPU-n futó kódot. A GPU kód futtatása nem okozna itt problémát, mivel alapvetően a WGPU szálbiztos módon van implementálva.

A GPU-val történő kommunikáció során jelenleg az összes Buffer tartalma teljes mértékben felülírásra kerül, abban az esetben is, ha az adott elem nem módosult semmilyen szempontból. Ennek a problémának a megoldására, az Entity Component System tud megoldást adni, mivel képes tárolni melyek voltak azok az entitások, akik módosításra kerültek a legutolsó iteráció óta bármilyen formában.

8.3 Továbbfejlesztési lehetőségek

Továbbfejlesztési lehetőségeket tekintve, az alábbi területeken lehetne szignifikáns fejlődést elérni:

- Megjelenítő réteg: Itt lehetne a legtöbb új funkciót bevezetni mert számtalan olyan technika létezik, amellyel egy jelenet részletességét lehetne bővíteni, mint például: Particle System, Screen Space Ambient Occlusion, Screen Space Reflections, és további optimalizációs technikákat is lehetne még használni
- Asset pipeline: Jelenleg egy glTF fájlnak csak a modellhez tartozó adatait használtam fel, de komplett jelenetek tárolására is alkalmas, ahol meg lehet adni az egyes modellek transzformációit, különböző fényforrásokat, kamera állásokat, Animációkhoz tartozó adatokat stb.
- Szkript környezet: Ezen a területen leginkább több funkció elérhetővé tétele lehet egy esetleges továbbfejlesztési lehetőség, mivel a JavaScript kód futtathatósági szempontból minden mai modern követelménynek megfelel.
- Fizika: Jelenleg dinamikusan kerülnek kiszámításra az egyes ütközőzónák, ami nem túl hatékony különösen komplex modellek esetén. Itt célszerű lenne egy olyan tool-t fejleszteni, ami ezt a műveletet előre elvégzi, és vagy külön fájlként vagy a már meglévő modell mellé integrálva kerül be az alkalmazásra.
- Felhasználói felület: Hasonlóan a megjelenítési réteghez, itt is van még bőven kiaknázatlan terület a felhasznált UI libraryben. A szkriptrendszerrel összhangban lehetne például egy Ui Builder-t elérhetővé tenni, ahol felhasználói szinten lehetne kezelni az egyes elemeket.

A rendszer olyan irányelvek alapján készült, hogy az egyes modulok további komponensekkel egyszerűen bővíthetőek legyenek. Ennek megfelelően az összes felsorolt fejlesztés megvalósítható és a már meglévő motorba illeszthető.

9 TARTALMI ÖSSZEFOGLALÓ

Dolgozatomban ismertetésre került a platformfüggetlen fejlesztés jelenlegi helyzete és a problémafelvetés részben egy 3D-s megjelenítő motor megvalósítása során felmerülő kritikus aspektusok megemlítése is.

Az irodalomkutatás során először megvizsgáltam mi is a feladata egy ilyen alkalmazásnak, valamint milyen problémákat kell fejlesztői és felhasználói oldalról figyelembe venni. Ezt követően a már meglévő hasonló rendszereket hasonlítottam össze, ahol a közös szempontokat kerestem, amikkel minden ilyen alkalmazásnak rendelkeznie kell, valamint az egyes megvalósítások miben nyújtanak többet a konkurenciájukhoz képest. Ezután határoztam meg milyen követelményeknek kell megfelelnie egy ilyen alkalmazásnak, ami többek között a platformfüggetlen működést, hatékony megjelenítést és könnyen használható szkript környezetet foglalja magában.

Ezt követte a megvalósítás során felhasználni kívánt technológiák vizsgálata, névszerint a választott programnyelv, szkript környezet nyelve, a grafikus függvénykönyvtár, valamint a felhasználói felület. Ezen vizsgálat eredményeképpen kerültek kiválasztásra a követelményspecifikációban foglaltaknak leginkább megfelelő eszközök.

A felhasználni kívánt technológia lista bemutatása után ismertettem az architektúra, valamint a nagyobb alrendszerek felépítését. Két fő részre lehet bontani az rendszert, a belső központi egységre, valamint az ezt vezérlő szkript környezetre. A következő fejezetben az implementáció menetét dokumentáltam a főbb komponensek működésének bemutatásán keresztül, valamint a fejlesztés során előjött problémákat és azok megoldásait. Szó esik többek között a megjelenítő réteg, a szkript környezet és nagyobb belső komponensek működéséről.

A tesztelési fejezet során ismertettem a tesztelés metodikáját, valamint a az egyes tesztek eredményeit. A rendszer legnagyobb előnye az újrafelhasználhatóság ezért két példa alkalmazást valósítottam meg, amelyek eltérő logikával rendelkeznek. A rendszer hatékonyságát demonstrálandó különböző helyzetekben történő teljesítménymérés dokumentálása is itt kapott helyet.

A dolgozatomat a rendszer értékelésével zártam, amely tartalmazza többek között az egyes továbbfejlesztési lehetőségeket, továbbá a rendszer hiányosságait is. A fejlesztés eredményeképpen egy platformfüggetlen és hatékony megjelenítőmotort hoztam létre, amely a fentebb ismertetett követelményeknek megfelel.

10 SUMMARY

The aim of my thesis was to implement a 3D rendering engine that could work in a platform-independent manner, without sacrificing performance. In the introduction chapter I gave a brief overview of the current state of cross-platform development and highlighted some of the issues that need to be addressed to achieve what I set out to do.

During the literature research I examined what is the goal of an application of this kind and what problems are needed to be resolved on the users' and the developers' side respectively. I also compared the frameworks that already exist in this field and looked for the common aspects that define them, and which features make them stand-out among their competitors. Using this knowledge, I specified the requirements that my system must adhere to. These aspects include platform-independent operation, efficient rendering, and easy-to-use scripting environment.

In the following chapter I looked at the various technologies that I would be using during the implementation stage. These include but not limited to programming language, scripting environment language, graphics library, and user interface. Using this research, I chose the most appropriate technology that meets the needs described in the requirement specification.

After the technology selection I presented the core architecture for the application and some of its most significant components. The framework can be split up into two distinct parts, namely the internal unit, and the scripting environment that provides easy access to the former. In the implementation part of the thesis, I documented the most significant parts of the application and the issues that came up during this process and how I overcame these obstacles.

In the testing chapter I began by detailing the testing methodology and the results of these tests. The most significant aspect of the engine is reusability. To demonstrate this, I created two basic applications with different logic. The test cases are meant to show that the engine is capable of platform independent and efficient rendering in primarily 3D context.

I concluded my thesis with an overview of the complete engine. This chapter includes the limitations of the system and how it could be improved later down the line. As a result of the development, I managed to create a cross-platform engine with 3D rendering capabilities that meets the previously specified requirements.

IRODALOMJEGYZÉK

[1] F. E. Garcia and V. P. de Almeida Neris, “A Data-Driven Entity-Component Approach to Develop Universally Accessible Games,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8514 LNCS, no. PART 2, pp. 537–548, 2014, doi: 10.1007/978-3-319-07440-5_49.

[2] X. Chen, M. Wang, and Q. Wu, “Research and development of virtual reality game based on unreal engine 4,” *2017 4th International Conference on Systems and Informatics, ICSAI 2017*, vol. 2018-January, pp. 1388–1393, Jun. 2017, doi: 10.1109/ICSAI.2017.8248503.

[3] J. Lee, *Learning Unreal Engine Game Development*. 2016.

[4] P. P. Patil, P. P. Patil, and R. Alvares, “Cross-platform Application Development using Unity Game Engine,” *International Journal of Advance Research in Computer Science and Management Studies*, vol. 3, no. 4, 2015, Accessed: Apr. 21, 2022. [Online]. Available: https://www.researchgate.net/publication/312591645

[5] “Home | Mono.” https://www.mono-project.com/ (accessed Nov. 28, 2021).

[6] “Unity - Manual: IL2CPP Overview.” https://docs.unity3d.com/Manual/IL2CPP.html (accessed Nov. 28, 2021).

[7] “Unity Platform Roadmap.” https://unity.com/roadmap/unity-platform (accessed Nov. 28, 2021).

[8] “Babylon.js: Powerful, Beautiful, Simple, Open - Web-Based 3D At Its Best.” https://www.babylonjs.com/ (accessed Nov. 28, 2021).

[9] “Babylon React Native.” https://www.babylonjs.com/reactnative/ (accessed Nov. 28, 2021).

[10] “Babylon.JS Editor.” http://editor.babylonjs.com/ (accessed Nov. 28, 2021).

[11] “Solar2D Game Engine.” https://solar2d.com/ (accessed Nov. 28, 2021).

[12] “The Programming Language Lua.” https://www.lua.org/ (accessed Nov. 28, 2021).

[13] “chromiumembedded / cef — Bitbucket.” https://bitbucket.org/chromiumembedded/cef/src/master/ (accessed Nov. 30, 2021).

[14] “ocornut/imgui: Dear ImGui: Bloat-free Graphical User interface for C++ with minimal dependencies.” https://github.com/ocornut/imgui (accessed Nov. 30, 2021).

[15] “Apache Cordova.” https://cordova.apache.org/ (accessed Nov. 30, 2021).

[16] “Cross-Platform Mobile App Development: Ionic Framework.” https://ionicframework.com/ (accessed Nov. 30, 2021).

[17] “Xamarin | Open-source mobile app platform for .NET.” https://dotnet.microsoft.com/apps/xamarin (accessed Nov. 30, 2021).

[18] “dotnet/maui: .NET MAUI is the .NET Multi-platform App UI, a framework for building native device applications spanning mobile, tablet, and desktop.” https://github.com/dotnet/maui (accessed Nov. 30, 2021).

[19] “WebAssembly.” https://webassembly.org/ (accessed Nov. 30, 2021).

[20] “World Wide Web Consortium (W3C).” https://www.w3.org/ (accessed Nov. 30, 2021).

[21] A. Hilbig, D. Lehmann, and M. Pradel, “An empirical study of real-world webassembly binaries: Security, languages, use cases,” in *Proceedings of the Web Conference 2021*, 2021, pp. 2696–2708.

[22] “AssemblyScript.” https://www.assemblyscript.org/ (accessed Nov. 30, 2021).

[23] “Blazor | Build client web apps with C# | .NET.” https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor (accessed Nov. 30, 2021).

[24] “Main — Emscripten 3.0.1-git (dev) documentation.” https://emscripten.org/ (accessed Nov. 30, 2021).

[25] “What is Windows Subsystem for Linux | Microsoft Docs.” https://docs.microsoft.com/en-us/windows/wsl/about (accessed Nov. 30, 2021).

[26] “Rust Programming Language.” https://www.rust-lang.org/ (accessed Nov. 30, 2021).

[27] “Introduction - The Cargo Book.” https://doc.rust-lang.org/cargo/ (accessed Nov. 30, 2021).

[28] “References and Borrowing.” https://doc.rust-lang.org/1.8.0/book/references-and-borrowing.html (accessed Nov. 30, 2021).

[29] “Blizzard Entertainment.” https://www.blizzard.com/en-us/ (accessed Nov. 30, 2021).

[30] “Home Page - World of Warcraft Programming: A Guide and Reference for Creating WoW Addons.” https://wowprogramming.com/ (accessed Nov. 30, 2021).

[31] “Home | SpiderMonkey JavaScript/WebAssembly Engine.” https://spidermonkey.dev/ (accessed Nov. 30, 2021).

[32] “TypeScript: JavaScript With Syntax For Types.” https://www.typescriptlang.org/ (accessed Nov. 30, 2021).

[33] “JavaScript | MDN.” https://developer.mozilla.org/en-US/docs/Web/JavaScript (accessed Nov. 30, 2021).

[34] “Direct3D 12 programming guide - Win32 apps | Microsoft Docs.” https://docs.microsoft.com/en-us/windows/win32/direct3d12/directx-12-programming-guide (accessed Nov. 30, 2021).

[35] L. Wang, D. Shi, A. Zhao, C. Tan, and D. C. Liu, “Real-time scan conversion for Ultrasound Imaging based on CUDA with Direct3D display,” *5th International Conference on Bioinformatics and Biomedical Engineering, iCBBE 2011*, 2011, doi: 10.1109/ICBBE.2011.5780361.

[36] “OpenGL - The Industry Standard for High Performance Graphics.” https://www.opengl.org/ (accessed Dec. 01, 2021).

[37] “WebGL: 2D and 3D graphics for the web - Web APIs | MDN.” https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API (accessed Dec. 01, 2021).

[38] D. Wang and Z. Wang, “Research of real-time trajectory simulation module based on Qt and OpenGL,” *Proceedings - 2013 International Conference on Computational and Information Sciences, ICCIS 2013*, pp. 179–182, 2013, doi: 10.1109/ICCIS.2013.55.

[39] “Home | Vulkan | Cross platform 3D Graphics.” https://www.vulkan.org/ (accessed Dec. 01, 2021).

[40] D. Pankratz, T. Nowicki, A. Eltantawy, and J. N. Amaral, “Vulkan Vision: Ray Tracing Workload Characterization using Automatic Graphics Instrumentation,” *CGO 2021 - Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, pp. 137–149, Feb. 2021, doi: 10.1109/CGO51591.2021.9370320.

[41] “WebGPU.” https://www.w3.org/TR/webgpu/ (accessed Dec. 01, 2021).

[42] A. N. Sisyukov, V. Dobrynin, and O. S. Yulmetova, “Web Based GPU Acceleration in Embodied Agent Training Workflow,” *Proceedings of the 2021*

IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering, ElConRus 2021, pp. 686–689, Jan. 2021, doi: 10.1109/ELCONRUS51938.2021.9396663.

[43] "RenderDoc — RenderDoc documentation." https://renderdoc.org/docs/index.html (accessed Dec. 02, 2021).

[44] "rust-windowing/winit: Window handling library in pure Rust." https://github.com/rust-windowing/winit (accessed Dec. 09, 2021).

[45] "gfx-rs/wgpu: Safe and portable GPU abstraction in Rust, implementing WebGPU API." https://github.com/gfx-rs/wgpu (accessed Dec. 09, 2021).

[46] "wgpu - Rust." https://docs.rs/wgpu/latest/wgpu/ (accessed Apr. 17, 2022).

[47] A. Lauritzen, "Beyond Programmable Shading Course ACM SIGGRAPH 2010 Deferred Rendering for Current and Future Rendering Pipelines".

[48] "amethyst/specs: Specs - Parallel ECS." https://github.com/amethyst/specs (accessed Dec. 10, 2021).

[49] "V8 JavaScript engine." https://v8.dev/ (accessed Dec. 10, 2021).

[50] "Deno - A modern runtime for JavaScript and TypeScript." https://deno.land/ (accessed Apr. 18, 2022).

[51] "Interior mutability - The Rust Reference." https://doc.rust-lang.org/reference/interior-mutability.html (accessed Apr. 17, 2022).

[52] "Web Workers API - Web APIs | MDN." https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API (accessed Apr. 17, 2022).

[53] "The event loop - JavaScript | MDN." https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop (accessed Apr. 17, 2022).

[54] "Introduction - The `wasm-bindgen` Guide." https://rustwasm.github.io/wasm-bindgen/ (accessed Apr. 18, 2022).

[55] "Fetch API - Web APIs | MDN." https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API (accessed Apr. 17, 2022).

[56] "glTF Overview - The Khronos Group Inc." https://www.khronos.org/gltf/ (accessed Apr. 18, 2022).

[57] M. Pharr, W. Jakob, and G. Humphreys, *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016.

[58] "Rapier physics engine | Rapier." https://rapier.rs/ (accessed Apr. 18, 2022).

[59] “emilk/egui: egui: an easy-to-use immediate mode GUI in pure Rust.” https://github.com/emilk/egui (accessed Dec. 10, 2021).

[60] “Depth Precision Visualized | NVIDIA Developer.” https://developer.nvidia.com/content/depth-precision-visualized (accessed Apr. 21, 2022).

[61] E. Lengyel, “ Matrix Projection Matrix Tricks”, Accessed: Apr. 21, 2022. [Online]. Available: http://www.terathon.com/

ÁBRAJEGYZÉK