Szegedi Tudományegyetem
Informatikai Intézet

SZAKDOLGOZAT

Pullai Szilárd

2024

Szegedi Tudományegyetem

Informatikai Intézet

Interaktív 3D Technológiák a Weben: Dinamikus Modell Megjelenítő Fejlesztése React Three Fiberrel

Szakdolgozat

Készítette:

Pullai Szilárd

Üzemmérnök-Informatikus Bprof
Szakos hallgató

Témavezető:

Dr. Iván Szabolcs

Tanszékvezető,
egyetemi tanár

Szeged

2024

Feladatkiírás

A kitűzött feladat egy webalkalmazás elkészítése amely képes részletes 3D modellek megjelenítésére. A projekt elkészítése során cél a Three.js, React.js könyvtárak alapos elsajátítása és átfogó elemzés készítése az alkalmazásban implementált optimalizálásokról.

A teljesítménynek skálázódik, mivel a programnak futnia kell mobil eszközökön, laptopokon és asztali számítógépeken, változó felbontásokat és képarányokat támogatva.

Tömörített 3D modelleket kell kezelnie az alkalmazásnak, mivel cél a gyors működés, mobilhálozatról is gyorsan és gördülékenyen kell történnie a betöltésnek. Az optimalizálási módokról részletes elemzést kell készíteni és a feltárt összefüggéseket ábrákon keresztül szemléltetni.

Az alkalmazásnak rendelkezni kell felhasználói felülettel, amin keresztül interakív módon változtatható a betöltött 3D modell megjelenése, a szintér környezete és a virtuális kamera forgatása.

A renderelésnél használt shader programok működésének megismerése, különböző fény számítási algoritmusok elemzése és összehasonlítása. Utófeldolgozás során alkalmazott algoritmusok megértése és ismertetése.

Tartalmi összefoglaló

- **Téma megnevezése:**
  Interaktív 3D Technológiák a Weben:
  Dinamikus Modell Megjelenítő Fejlesztése React Three Fiberrel

- **A megadott feladat megfogalmazása:**
  Egy webes környezetben működő, 3D modellek realisztikus megjelenítésére alkalmas alkalmazás fejlesztése, amely interaktív funckiókat tartalmaz és optimalizáltságnak köszönhetően mobilokon is jól működik.
  A fejletszési folyamat alatt használt techinkák elemzése, valamint az optimalizálási módok szemléltetése.

- **A megoldási mód:**
  Az alkalmazás React Three Fiber könyvtár felhasználásával készült JavaScript programozási nyelvben írva, a Visual Studio Code fejlesztői környezeten belül.

- **Alkalmazott eszközök, módszerek:**
  Az alkalmazás alapját a ***React Three Fiber*** könytár adja, továbbá a kamera és irányítás funkciók, illetve a 3D modell importáló modul a ***drei*** kollekcióból lett felhsaználva. Állapotok kezelését a ***Zustand*** rendszerével készült. ***Blender*** szoftverrel történt a modellek optimalizálása, szekesztése, tömörítése.

- **Elért eredmények:**
  Az elkészült program gyorsan és gördülékenyen fut, illetve betölt gyengébb eszközökön is, az optimalizálásoknak köszönhetően. A fejlesztések lehetőséget biztosítanak az eszköz továbbfejlesztésére. A dokumentált optimalizáslások pedig felhasználhatóak további 3D-s alkalmazásokban.

- **Kulcsszavak:**
  Interaktív, 3D, React, webalkalmazás, React Three Fiber, Drei, Zustand, Blender, optimalizáció, modell megjelenítő, Three.js, JavaScript, Web

Tartalomjegyzék

BEVEZETŐ

Azért szerettem volna 3D grafikához kapcsolódó szakdolgozatot készíteni, mert az egyetemi évek alatt a ***Számítógépes grafika*** kurzus keltette fel legjobban az érdeklődésemet, és tanulmányaim utolsó éveiben már dolgoztam is ezen a szakterületen.

Az utóbbi időben egyre jobban elterjedtek a 3D grafikát használó alkalmazások, a kis méretű és egyre erősebb hardvereknek köszönhetően. Gondolok itt mobil, AR és VR eszközökre amelyek mind rendelkeznek böngészővel és azon keresztül tudnak tartalmat megjeleníteni. Vannak felhasználási módok ahol különösen hasznos, például 3D animációk az oktatásban, épületek terve három dimenziós térben, körbeforgatható termékek megjelenítése weboldalon stb.

Szakdolgozatomban azt szeretném leírni, hogy milyen módszerekkel kell egy 3D-s alkalmazást elkészíteni, amely működik bármilyen modern böngészővel rendelkező eszközön, legyen az PC, mobil vagy VR eszköz. Ahogy a címben is látható a React függvénykönvtár és a Three.js segítségével fogom mindezt elérni.

Kiemelkedő szerepet fog kapni az optimalizálás, mivel az alkalmazás betöltésnél és a renderelésnél is sok lehetőségünk van elérni egy adott megjelenést állapotot, de ezek nem mindig a legköltséghatékonyabban történnek, ha a fejlesztő nem ismeri a pontos működést és a teljesítmény növelésére alkalmas módszereket.

A különböző megvalósítási módszerek összehasonlításával és fejlesztési eszközök jellemezésével az a célom, hogy a jövőbeli ilyen jellegű alkalmazások fejlesztése könyebb legyen és a megvalósított alkalmazások jobb teljesítménnyel fussanak gyengébb eszközökön is.

1. 3D WEBES ALKALMAZÁSOK

Webes alkalmazások fejlesztéséhez elengedhetetlen a HTML, CSS és JavaScript technológiák ismerete, viszont ha 3D-s webalkalmazást csinálunk tisztában kell lenni a WebGL működéséről is. Ezt leegyszeűsítik a különböző függvénykönyvtárak esetünkben a Three.js ami már előre megírt WebGL implementálásokat tartalmaz. Modern fejlesztéseknél keretrendszereket alkalmaznak, mint pl. Angular, Vue vagy React amik lehetővé teszik könnyen újrafelhasználható komponensekké szerveznünk a megírt kódot. Ebben a fejezetben ezek a technológiák lesznel bővebben kifejtve, hogy tisztában legyünk vele, mi is a története a szakdolgozat-projektemben felhasznált technológiáknak.

1.1 WEBGL TECHNOLÓGIA

Angolul Web Graphics Library, egy alacsony szintű, renderelő API, amely közvetlenül a böngészőben fut, és lehetővé teszi a 3D grafika megjelenítését egy HTML5 szabványból ismert Canvas elem segítségével. Ennek a technológia felhasználásával a fejlesztők készíthetnek olyan interaktív 3D alkalmazásokat, mint játékok, vizuális szimulációk, oktatási eszközök, és egyéb dolgok, amelyek elérhetőek bármilyen modern webböngészőn keresztül.

Több verziója van jelen, most a WebGL 2.0 a legnépszerűbb amely az elődje által lefektetett alapokon, és az OpenGL ES 3.0-ra épít. Ennek köszönhetően számos új funkciót és API-t kínál, amelyek jelentősen bővítik a grafikai lehetőségeket, mint például jobb textúrázási technikák és komplexebb árnyékolási modellek. Ezen felül a WebGL 2.0 garantálja az előző verzió sok opcionális kiterjesztésének elérhetőségét, ami nagyobb rugalmasságot és jobb teljesítményt biztosít a fejlesztőknek.

A shaderek, amelyek a WebGL-ben GLSL nyelven íródnak, kulcsfontosságú szerepet játszanak a grafikus renderelésben. Ezeket a shadereket szövegsorozatként adja át a fejlesztő a WebGL API-nak, amely fordítási folyamaton keresztül a grafikus feldolgozó egység gépi kód jává alakítja át őket. Ez a GPU kód végzi el a számításokat minden egyes 3D objektum csúcspontjára és a képernyőn megjelenített pixelek RBG értékét, ami lehetővé teszi a fejlesztők számára, hogy létrehozzanak

vizuális effekteket és részletesen kidolgozott 3D modelleket közvetlenül a webböngészőben.

A shader kódok az OpenGL ES Shading Language (GLSL ES) használatával készülnek, ami egy olyan nyelv, ami hasonlít a C vagy C++ programozási nyelvekre. Egy webalkalmazáson belül a JavaScript a WebGL API-t használva irányítja az alkalmazás logikáját és kezeli a bemeneteket a felhasználó és a weboldal között, míg a GLSL kódok, amelyek a grafikus műveletekért felelősek, közvetlenül a GPU-n futnak. A GPU-ra írt kódok lehetővé teszik a számítási és renderelési műveletek gyors végrehajtását, kihasználva a grafikus processzor párhuzamos számítási képességét, ami sokkal gyorsabb, mint a JavaScript kód, ami a fő processzorn futtatná az utasításokat.

A fejlesztők a WebGL-ben definiált két fő típusú shader kódot használhatnak: a vertex shader-eket, amelyek minden egyes 3D modell csúcspontjára lefutnak, és a fragment shader-eket, amelyek minden egyes pixel számításáért felelősek a képernyőn. Ezzel a két shader-rel lehetőség nyílik arra, hogy a fejlesztők részletesen kontrollálják a grafikus kimenet minden aspektusát, beleértve a fények és árnyékok, textúrák, és egyéb vizuális effektek kezelését.

WebGL programok esetében kritikus a teljesítmény optimalizálása, mivel a komplex 3D grafikák számítási igénye magas lehet. A fejlesztőknek gondoskodniuk kell arról, hogy a kódjaik hatékonyan használják a rendelkezésre álló erőforrásokat, minimalizálva ezzel a böngészők és a végfelhasználói eszközök terhelését. A WebGL kiterjedt eszköztárat és technikákat kínál a teljesítmény mérése és profilozására, segítve a fejlesztőket a hatékonyabb grafikai alkalmazások létrehozásában.

1.2 THREE.JS FÜGGVÉNYKÖNYVTÁR

A Three.js egy magas szintű, JavaScript-alapú 3D grafikai könyvtár, amelyet kifejezetten a WebGL API-ra építettek. A célja, hogy megkönnyítse a fejlesztők számára a 3D grafikák létrehozását a webes környezetben.

A Three.js egy absztrakciós réteget biztosít a WebGL fölött, így a fejlesztőknek nem kell közvetlenül a WebGL alacsony szintű API-jával foglalkozniuk. Ez a könyvtár egyszerűsíti a bonyolult grafikai programozási feladatokat. Előre definiált shaderek biztosítják az objektumok megjelenését, a fények kezelését és a színek tónusleképezését.

A Three.js rengeteg beépített funkciót és komponenst tartalmaz, mint például kamera típusok, fényforrások, geometriai alakzatok, animációs eszközöket és 3D modell beolvasó funkciókat. Ezek a komponensek előre megírtak és optimalizáltak, így jelentősen csökkentik a fejlesztési időt és erőfeszítést.

A Three.js könnyen integrálható más webes technológiákkal és keretrendszerekkel. Támogatja a különféle kimeneti formátumokat is, beleértve a WebGL mellett a Canvas 2D, SVG, és CSS3D rendereket. Ez a rugalmasság teszi lehetővé a fejlesztők számára, kiválasszák a projekthez szükséges technológiát. Ezekhez kapcsolódva jön ide a React függvénykönyvtár is amit elég effektíven tudunk használni a Three.js által előre definiált funckiókkal.

1.3 REACT.JS MINT KERETRENDSZER

A React egy JavaScript könyvtár, amelyet dinamikus adatokkal rendelkező, webalkalmazások felhasználói felületeinek létrehozására fejlesztettek ki. A Meta amerikai cég alkotta meg és tartja karban a nyílt forráslódú kódbázist, közreműködő felhasználók segítségével.

A React alkalmazások alapvető elemei a komponensek, amelyek újrafelhasználható, önálló elemek, és definiálják a felhasználói felület vizuális és interaktív aspektusait. Egy React alkalmazás több, egymásba ágyazott komponensből áll.

A React JSX-et használ ami egy JavaScript szintaxis kiterjesztés, ami HTML-re hasonlít. A JSX segítségével a fejlesztők HTML szerkezeteket írhatnak a JavaScript kód mellé, ami olvashatóbbá és kifejezőbbé teszi a kódot. A forrásfájlokba komponenseket hozhatunk létre, amelyek a React alkalmazásunk építőelemei, lehetnek osztály alapúak vagy funkcionálisak.

Az adatokat a szülő komponensektől a gyermek komponensek felé lehet átadni. Ezek olvasásra szántak és nem módosíthatók a gyermek komponensek által. React fejlesztők ezt „prop” –nak nevezik, mivel tudjuk ezzel tudjuk kezelni egy komponens viselkedését. Emellett még van állapota is komponenseknek, ami módosítható, és a React újragenerálja a komponenst az állapot változásakor.

A React működése egy virtuális DOM-on, Document Object Modell –en alapszik. Ez egy memóriában létrehozott másolat a valós DOM-ról, ahol a React először minden manipulációt ezen a virtuális DOM-on végez. Először is, a React kiszámítja az aktuális oldalszerkezet és az új szerkezet közötti különbségeket, majd csak a szükséges változásokat hajtja végre a valós DOM-on, ami hatékony frissítéseket tesz lehetővé. Másodszor, a React több frissítést összegyűjt a virtuális DOM-ban, majd egyetlen újrarendereléssel frissíti a valós DOM-ot, optimalizálva ezzel a teljesítményt. Amikor egy komponens állapota vagy tulajdonsága megváltozik, akkor a React összehasonlítja az újonnan visszaadott elemet az előzővel. Ha van különbség, akkor a React ennek megfelelően frissíti a DOM-ot. Ez az összehasonlítási folyamat egy speciális algoritmus segítségével történik, amely

feltételezi, hogy különböző típusú elemek különböző struktúrákat eredményeznek, így a DOM frissítése hatékonyan történik.

React egyszerűsíti az interaktív felhasználói felületek létrehozását deklaratív és komponens-alapú jellege révén. Láthattuk, hogy a React ökoszisztémája nemcsak hogy erős, de rendkívül kiterjeszthető is, ami lehetővé teszi innovatív könyvtárak létrehozását, amelyek kihasználják alapvető elveit a hagyományos 2D-s webfelületeken túl.

Ez vezet el minket a React Three Fiberhez, ami a React deklaratív képességeit ötvözi a 3D grafika összetett világával. React Three Fiber egy renderer a React számára, amely lehetővé teszi a Three.js használatát React fejlesztői rendszrek között. A következő fejezetben megvizsgáljuk, hogyan használja ki a React Three Fiber a React paradigmáját, hogy beilleszthesse a Three.js erőteljes 3D renderelési képességeit a React ökoszisztémába. Megnézzük, hogyan teszi lehetővé ez az integráció a fejlesztők számára, hogy ugyanolyan könnyedén és hatékonyan alkossanak 3D alkalmazásokat, mint ahogyan hagyományos webalkalmazásokat építenek. A React és a Three.js közötti hidat építve a React Three Fiber nemcsak kiszélesíti a Reacttal elérhető lehetőségek skáláját, hanem népszűsíti a 3D tartalom létrehozását is, elérhetővé téve azt szélesebb fejlesztői közönség számára.

1.4 REACT THREE FIBER

A React Three Fiber (R3F) ötvözi a React deklaratív felhasználói felületi paradigmáit a Three.js erőteljes 3D renderelési képességeivel. Ennek a keretrendszernek a megértése azt jelenti, hogy értékeljük, hogyan használja ki mind a React, mind a Three.js alapvető erejét, miközben egy innovatív megközelítést vezet be a 3D alkalmazások és vizualizációk építéséhez egy React környezetben.

A React Three Fiber lényege, hogy egy React renderelő a Three.js számára. Ez azt jelenti, hogy lehetővé teszi a 3D jelenet struktúrájának kifejezését React komponensek segítségével, amelyek belsőleg kezelik a megfelelő Three.js objektumokat. Ahogy a React DOM leképezi a React komponenseket DOM csomópontokra, úgy képezi le az R3F a React komponenseket a Three.js objektumokra, mint amilyenek a meshek, geometriák, anyagok és fények.

A React deklaratív jellege megkönnyíti az állapot és az effektusok kezelését, biztosítva, hogy a UI összhangban maradjon az alapadatokkal. Ez kiterjed a 3D grafikára is, ahol a jelenetgráf struktúrájának állapotát intuitívabban lehet kezelni React komponenseken keresztül. Továbbá R3F használatával a fejlesztők zökkenőmentesen integrálhatják 3D tartalmaikat más React komponensekkel és hook-okkal. Ez az integráció biztosítja, hogy a fejlesztők kezelni tudják a bonyolult állapot-interakciókat, a routing, sőt az állapotkezelési eszközöket is, mint a Redux vagy a Context API egy 3D kontextusban.

A React Three Fiber optimalizálja a renderelési teljesítményt a React egyeztetési algoritmusa révén ami csak akkor frissíti az objektumokat, ha azok tulajdonságai megváltoztak, ahelyett, hogy újrarendezné az egész jelenetet, ami jelentős teljesítménybeli akadály lehet a hagyományos Three.js alkalmazásokban.

Az R3F támogatja a React hook-okat, lehetővé téve a mellékhatásokkal rendelkező függvénykomponensek használatát, amelyek elengedhetetlenek az animációk, interakciók és aszinkron adatbetöltések kezeléséhez a 3D jeleneten belül. A mellékhatás olyan műveleteket jelent, amelyek befolyásolják más komponenseket vagy rendszereket.

A <Canvas> komponens az R3F alkalmazás belépési pontja. Létrehoz egy Three.js renderert, és beállít egy kamerát és egy jelenetet. Ezen a vásznon helyezhetők el a 3D objektumok React komponensekként.

Az R3F számos szabványos komponenst tesz elérhetővé, amelyek a Three.js alapvető fogalmait képviselik, mint például <mesh>, <geometry>, <material>. Emellett létrehozhatók egyedi komponensek is, amelyek összetett viselkedést vagy kompozit objektumokat foglalnak magukban.

A React Three Fiber több rendszert is biztosít az események kezelésére. Mivel a Three.js objektumok nem valódi DOM elemek, az R3F leképezi a felhasználói interakciókat, mint a kattintások vagy pointer mozgások a 3D objektumokra, a Three.js ből ismert raycast segítségével.

A Suspend Mode Reactben általában azt a funkciót biztosítja, hogy a renderelés szüneteltethető addig, amíg bizonyos feltételek teljesülnek, vagy amíg adatokat nem töltünk be. Ezt elsősorban a React Suspense komponense kezeli. A Suspense lehetővé teszi a komponensek számára, hogy jelezzék: még nem készültek fel a renderelésre, mert valamilyen aszinkron adatra várnak, pl. modell vagy textúra betöltésére. Amikor egy komponens "szüneteltetésre" kerül, a React továbbra is megjelenítheti a jelenlegi felhasználói felületet, amíg a komponens által kért adatok elérhetővé válnak. Ez rendkívül előnyös az adatok aszinkron betöltésére, a kód felosztása vagy bármely aszinkron feladat esetén, amelynek befejeződése szükséges a renderelés folytatása előtt.

R3F alapból kezeli a PBR (physically based rendering) anyagokat és textúrákat, rugalmasságot is nyújt az egyedi árnyékolók és utófeldolgozási effektek implementálásához. Ezek integrálhatók React komponensek segítségével, megőrizve a deklaratív megközelítést, miközben kihasználják a GLSL árnyékolók előnyeit.

Interaktívabb 3D jelenetek esetén az R3F integrálható fizikai motorokkal, mint például a Cannon.js fizikai motor hook-alapú impelentációja vagy a Rapier ami egy újabb fizikai motor ami használható az alkamazás fejlesztéséhez. Ez az integráció lehetővé teszi a realisztikus fizikai interakciók kezelését a React ökoszisztémában.

Szerveroldali Renderelés (SSR): Az R3F képes a jeleneteket a szerveren renderelni, elküldve a kezdeti jelenet pillanatképét a felhasználónak, ahol interaktívvá válhat. Ez hasznos a kereső motorok számára és a kezdeti betöltési teljesítmény előnyeinek biztosítására.

A React Three Fiber jelentős fejlődést képvisel abban, hogy hogyan lehet a 3D tartalmakat webes alkalmazásokba integrálni. Összhangban van a modern webfejlesztési gyakorlatokkal, összehozva a Three.js funkcióit, a React fejlesztési előnyeivel. Akár interaktív játékot, VR alkalmazást vagy bonyolult vizualizációkat fejleszthetünk, az rugalmas és hatékony keretrendszert nyújt a munkához.

1.5 DREI

A Drei egy olyan eszközkészlet, amely további funkcionalitást és komponenseket nyújt a React Three Fiber számára, és célja, hogy megkönnyítse a 3D alkalmazások fejlesztését a Three.js használatával egy React környezetben. A Drei magában foglal számos magasabb szintű absztrakciót, hasznos eszközt és előre elkészített komponenst, amelyek egyszerűsítik a 3D jelenetek fejlesztését.

A Drei számos gyakori geometriát és kameramozgatást segítő vezérlőt, valamint más, gyakran szükséges eszközöket kínál, amelyek nem részei közvetlenül a Three.js-nek vagy a React Three Fiber-nek.

Egyszerűen használható komponenseket kínál shaderek és utófeldolgozási effektek implementálásához, amelyek beállítása gyakran összetett feladat. Az EffectComposer használatával gyorsan és egyszerűen tudunk post-processing effekteket beállítani, a paraméterek könnyen kezelhetők és a kód is nagyon átlátható.

A Drei olyan komponenseket tartalmaz, amelyek közös feladatokat kezelnek, mint például modellek vagy textúrák betöltése, környezetek beállítása és hibakereső segédprogramok hozzáadása. Alkalmazás fejlesztése alatt hasznát vettem a legtöbb ilyen komponensnek, például a leva, amivel egyszerű GUI ablakhoz hozzá tudjuk rendelni különböző paramétereinket és futásidő alatt lehetőségünk van finomhangolni a beállításokat.

A React Three Fiber alapvető React renderert biztosít, amely összeköti a Reactet a Three.js-sel, lehetővé téve, hogy a JSX szintaxis használatával definiálja 3D objektumait és jeleneteit. A Drei ezen épít tovább, olyan komponenseket biztosítva, amelyek már rendelkeznek a szükséges Three.js boilerplate kóddal, így még könnyebbé téve az összetett 3D objektumok és viselkedések React alkalmazásba való beillesztését.

A Drei csökkenti az írandó és kezelendő boilerplate kód mennyiségét, kibővített komponenseket biztosítva, amelyek közvetlenül használhatók a React komponenseiben.

A Drei számos komponense teljesítményre optimalizált, biztosítva, hogy a 3D jelenetek simán fussanak anélkül, hogy mélyreható ismeretekkel kellene rendelkezni a Three.js teljesítményoptimalizálásáról.

Mivel a Drei kifejezetten a React Three Fiber használatához készült, komponenseinek projektbe való integrálása egyértelmű. Importálhat egy Drei komponenst, és használhatja azt a React komponensfában, akárcsak bármely más React komponenst.

1.6 BLENDER

A Blender egy 3D modellek készítésére és szerkesztésére alkalmas nyílt forráskódú és ingyenes program. Nagyon sok feladatot el tud látni az animációs, tervező és filmipari területeken, egyre több stúdió használja és egy hatalmas online közösség van mellette rengeteg fórummal és oktató videóval, amin keresztül bárki megtanulhatja a program használatát. Esetünkben a modellezés, geometriákhoz anyag hozzárendelés, árnyékok textúrára égetése és exportálás / tömörítés feladatok elvégzésére lett használva a szoftver.

A kiexportált fájl GLTF formátumú, ami egy szabványos fájlformátum háromdimenziós jelenetek és modellek számára. Úgy tervezték, hogy hatékonyan továbbítsa és töltsön be 3D modelleket olyan alkalmazásokban, mint például játékok és más virtuális környezetek. A glTF minimalizálja a 3D eszközök méretét és az azok felhasználásához szükséges futásidejű feldolgozást. Támogatja a teljes 3D jelenet leírásokat, beleértve a csomópont hierarchiáját, kamerákat, anyagokat, textúrákat és animációkat, ami sokoldalúvá teszi a valós idejű alkalmazások számára. Egyik fő előnye, hogy gyorsan elemezhető különböző platformokon és eszközökön, amely növeli hasznosságát a webalapú 3D grafikában.

2.0 FEJLESZTÉSI FOLYAMAT

A fejlesztés első lépéseként létrehoztam a git repository-t Github-on, hogy a fejlesztési folyamat alatt biztonságban legyen a forráskód és bármelyik gépről elérjem a projektet, helytől függetlenül. Egy ***.gitignore*** fájl létrehozása szükséges volt mivel node.js package manager-en keresztül használtam a fejlesztői könyvtárakat és a projekten belül van pár mappa mint pl.: *'node_modules'* vagy a *'.env'* aminek a feltöltését mellőzni kell ha node.js fejlesztési elveit követjük.

Egy React alkalmazás projektjének létrehozásához csak a node.js szükséges.
A következő parancsra:

```
npx create-react-app
```

Minden szükséges React függőség bekerül a ***package.json*** fájlba, ami számontartja a projektünkben használt függőségeket, verzió szerint és projekt beüzemelésekor minden itt szereplő függőséget letölt a node package manager.

React projektünkhöz hozzá tudjuk adni a React Three Fiber csomagot, hogy minden meglegyen a fejlesztés megkezdéséhez.

```
npm install three @react-three/fiber
```

Ez egy újabb függőség ami a React Three Fiberhez kapcsolódó eszközöket tartalmazza.
Alkalmazás futtatása a lokális hálózaton keresztül történik, az

```
npm install start
```

parancsra, ilyenkor localhost-on is elérhető a hostolt projekt és a hálózaton keresztül más eszközök is elérhetik ha rendelkeznek a hostoló ip címével és a portal ahol az alkalmazás hostolva van.

Az utóbbi lehetőséget ad számunka, hogy az alkalmazásunkat kölünböző eszközökön, különböző böngészőkkel ki tudjuk próbálni, mivel egy olyan 3D webalkalmazást szeretnénk fejleszteni ami a lehető legtöbb eszközt támogatja és ehhez elengedhetetlen a folyamatos tesztelés, teljesítmény, betöltési sebesség és reszponzivitás terén.

2.1 3D SZINTÉR, KAMERA, RENDERER LÉTREHOZÁSA

A létrehozott projekt belépési pontja az ***index.jsx*** lesz, ami egy ***Canvas*** elemet ad vissza a böngészőnek amin elhelyezkedik a kamera és a renderer. Attribútomok megadásával kell módosítanunk a korábban említett objektumok beállításait.

```
<Canvas
    gl={{ preserveDrawingBuffer: true }}
    camera={{
        fov: 45,
        near: 0.1,
        far: 200,
        position: [ - 4, 3, 6 ]
    }}
>
    <Experience />
</Canvas>
```

2.1. ábra: Renderer és kamera beállítása

Ezek beállítások szükségesek voltak az alkalmazás funkcionalitásai és kinézeti tulajdonságai miatt. A renderer beállítások módosításához egy objektumot kell küldenünk a komponensnek, azzal a név: érték párossal amelyik attribútumot változatni akarjuk és az új érték et át kell adnunk vele együtt.

Mivel az alkalmazás támogatja a renderel által generált kép lementését, ezért a renderer beállításain módosítanunk kell, hogy egy generált képkocka után ne törölje ki egyből az elkészült képmátrixot, mert azt mi szeretnénk majd kiolvasni, ha a felhasználó megnyomja a fotó mód gombot, amivel lehetőség van lementeni a renderelt képet .png formátumban.

A kamerán is különböző beállításokat szükséges végezni, hogy elérjük a kívánt eredményt. Mivel azt szeretnénk, hogy betöltött modellt körbe lehessen nézegetni, ami az origón helyezkedik el. A kamera alapértelmezetten az origóra kerül, tehát módosítani kell a pozíciójan a már fentebb leírt módón. Csak ebben az esetben egy 3 elemű tömböt adunk az objektumon belül a position adattagnak. Továbbá még módosítani kell a kamera near, far clipping tulajdonságain is. Ezek azt eredményezik hogy a virtuális téren belül mi legyen a minimum és maxium távolság a kamera szemszögében, amin belül eső objektumok megjelenjenek a renderer által kigenerált

képen. A fov, adattag a field of view tulajdonságért felelős, ami megadja hogy a kamera mekkora szöget lásson be.

Ehhez a szintérhez úgy tudunk 3D objektumokat hozzáadni, hogy a ***Canvas*** komponens nyitő és záró tagjai közé helyezzük el a kívánt modelleket, fényeket és egyéb funkciókat ami szükséges a jelenetünkhöz. Ahogy a fentebbi képen is látható, egy komponens szerepel esetünkben ami ***Experience***-nek neveztem el. Ebben van az összes további komponens elhelyezve, amik működtetik az alkalmazást.

2.2 KAMERA VEZÉRLÉS

Az ***OrbitControls*** komponsenst használja az alkalmazás ami a ***drei*** kollekcióból elérhető. Ez egy külön csomag, amit a node package manager –en keresztül kell letöltenünk a projektbe, utána könnyen importálható a koponens az alábbi módón:

```
import { OrbitControls } from '@react-three/drei'
```

2.2. ábra: Új Komponens beimportálása a drei kollekciónól

A komponens használatához csak bele kell raknunk az ***Experience*** srtuktúra törzsébe, beállításainkat pedig paraméterben szükséges átadnunk neki. Lehetőségünk van korlátozni a körbeforgatható tér ívét, a fel-le döntögetés szögeit és a zoomolás sebességét. Amit a felhasználói interfészből ki-be kapcsolhatóvá szeretnénk tenni az az automata forgatás. Ehhez egy boolean típusú állapotot veszek fel a React ***useState*** hookja segítségével. Majd az ***OrbitControls*** paraméterében ezt a változót adom oda az ***autoRoate*** adattag értékének. Ez a komponens kezeli az érintő kijelzőkön bevitt zoomolás és forgatás bemeneteket is, szóval ez az egy komponens kiszolgál minden irányítással kapcsolatos bevitelt, az összes platformon.

2.3 VIRTUÁLIS KÖRNYEZET

Az alkalmazásban több választható háttér és környezet közül is tudunk választani. Elsődeleges környezet egy stúdió, amit a legtöbb 3D szoftverből már megismertünk. A van egy rácsunk az X és Z tengelyek mentén, ami arra hivatott hogy szemléltesse, hogy van a 3D jelenetünk padlója, illetve legyen egy viszonyítási alapunk a modell méretét illetően, ugyanis a rácsok 1 méter x 1 méteresek. Ez egy izolált környezetet ad az alkalmazásunknak, ami arra jó hogy a 3D modelleket tudjuk szemügyre venni és ne terelje el semmi más a figyelmünket.

Ennek megvalósításához megint egy ***drei*** komponenst használtam fel, ami a ***Grid*** nevet viseli. Egy testreszabható rácsot ad hozzá a 3D-s szintérhez és ez ki is tudja elégíteni a studió környezethez a szükségletet. Még egy dolog ide tartozik az úgynevezett ***Environment*** komponens, amit be tudunk állítani a szintér hatterének, valamint a tükröződő modelleken lévő anyagokon milyen tükröződés jelenjen meg. Esetünkben csak egy tükröződésre van szükség, szóval az ***Environment*** komponens bacground adattagját false-ra állítjuk be, a tükröződéshez és a környezeti megvilágításhoz egy ***hdr*** 360 fokos panoráma képet fogok alkalmazni.
A hdr (High Dynamic Range) azt jelenti, hogy a kép több fényértéknyi információt is tartalmaz a kép egész területén, így alkalmas hogy a modellünk megvilágatására használjuk és tükröződéseket is számoljunk belőle.

A stúdió-n kívül még további környezetek is elérhetőek az alkalmazásunkban. Ezeknél hasonlóan mint korábban, az ***Environment*** komponenst használjuk fel, de most a hátteret is engedélyzzük a background kapcsoló beállításával.
Itt is hdr képeket használunk fel, viszont az ***Environment*** komponensnek van még egy jó funkciója, amivel a földre tudujk vetíteni a 360 fokos környezetünket. Ehhez paramétereket kell megadnunk a ground adattagnak, mivel minden 360 fokos képen más magasságban készült, más beállításokkal, különböző méretű terekről és ezt be kell állítanunk előre minden környezeti képhez amit szeretnénk majd felhasználni az alkalmazáshoz.

Ezek paraméterek tárolására egy konstans objektumot hozta létre, ahol az adattag egy string, a kiválasztott kép nevével. Ehhez tartozik egy másik objektum az adott képhez tartozó vetítési paraméterekkel.

```
export default function Background() {

    const background = useStore((state) => state.background)

    const { height, radius, scale } = useControls('background', {···
    })

    const params = {
        'sunrise': {height: 20, radius: 440, scale: 10},
        'road': {height: 6, radius: 128, scale: 8},
        'depot': {height: 12, radius: 64, scale: 10}
    }

    return <>
        <Environment
        background
        ground={ params[background] }
        files={'./environment/' + background + '.hdr'}
        />
    </>
}
```

2.3. ábra: Vetítési paraméterek tárolása és felhasználása

A kiválasztott környezet egy állapotban van letárolva, innen tudja a komponens hogy mit kell éppen megjelenítenie. A fájlneveket használom állapotnak, így a betöltés is könnyen megoldható az elérési út módosításával. Ugyanitt a ground adattagnak csak a params objektumot kell átadni, mert a beállításokhoz zartozó név szintén megegyezik az állapot nevével.

Több kép paramétereinek beállítása elég sok időt venne igénybe ha csak forráskódban mindig próbálgatnánk melyik érték milyen eredményt ad, de szerencsére erre is van egy könyvtár ami a segítségünkre van: ***leva***.

2.4 PARAMÉTEREK BEÁLLÍTÁSA LEVA HASZNÁLATÁVAL

A Leva egy JavaScript könyvtár, amelyet arra terveztek, hogy segítse a paraméterek finomhangolását webalkalmazásokon belül. Kiemelkedő tulajdonsága, hogy automatikusan generál felhasználói felületeket a fejlesztők által meghatározott paraméterek alapján, így nincs szükség kézzel készített UI vezérlők létrehozására a paraméterek módosításához. Ez az előny jelentős időmegtakarítást és a beállítási folyamatok bonyolultságának csökkenését jelenti.

A Leva paneljein keresztül végrehajtott módosítások az alkalmazásban azonnal frissülnek. Ez különösen hasznos 3D webalkalmazások esetén, ahol a világítás, a kamera szögei és az objektumok elhelyezkedésének módosításait azonnal láthatjuk és finomíthatjuk. Támogatja az egyedi komponensek és vezérlők hozzáadását, így alkalmazkodik a különböző projektigényekhez.

```
import { folder, useControls } from 'leva'

export default function Plastic(props) {
    const { color, roughness } = useControls('materials', {
        'plastic': folder({
            color: '#1D1D1D',
            roughness:{
                value: 0.9,
                min: 0,
                max: 1,
                step: 0.01
            }
        }, {
            collapsed: false
        })
    }, {
        collapsed: false
    })
```

2.4. ábra: Leva használata, szín és anyag tulajdonságok beállítására

A fenti kód az alkalmazásban egy anyagi tulajdonság beállításához lett használva, de ugyez az eljárás minden hasonló érték, szín és állapot hozzáadása a leva panelhez.

2.5 MODELLEK MEGJELENÍTÉSE

Sok lehetőség van 3D modelleket kiexportálni különböző formátumokban, de webes környezetben a legcélravezetőbb a glTF formátum. Egyszerűen tömöríthető, textúrák beágyazthatóak és animációkat is, a modellel együtt be tudjuk tölteni. Betöltéshez megint csak a ***drei*** csomagból a ***useGLTF*** hook-ot fogom használni.

A hook paraméterben csak egy elérési utat vár, ahol a gltf, vagy glb fájl helyezkedik el. Több objektumot is visszaad a függvény, de a legfontosabb az a nodes objektum, amiben megtaláljuk a kiexportált fájl teljes struktúráját. Esetünkben csak a geometria-ek betöltésére van szükségünk és ezt úgy érjük el, hogy hivatkozunk nevére a nodes objektumon belül, majd lekérjük a hozzá tartozó geometriát:

```
export default function Car(props) {
  const { nodes } = useGLTF('./glb/' + props.model + '/body.glb')

  return <group position={[0, .14, 0]}>
    {/* Body */}
    {nodes.body_carpaint && <mesh geometry={ nodes.body_carpaint.geometry }>
      <Carpaint envMapIntensity={props.envMapIntensity}/>
    </mesh>}
```

2.5. ábra: glTF fájl betöltése és geometria hozzárendelése egy anyaghoz

Itt van egy ellenőrzés is a && operátor használatával, ugyanis mielőtt lekérjük a geometriát és hozzárendelünk valamit, érdemes megnézni hogy az a nevű mesh, objektum létezik –e a beolvasott fájlban. Reactban így működik a feltételes végrehajtás illetve a null check is, hogy az és operátort használjuk. Ha a bal oldali operandus, esetunkben az ellenőrzés hamissal tér vissza, akkor a operátor jobb oldalán elhelyezkedő kódrészlet nem is lesz végrehajta így bebiztosítjuk hogy nem fog undefined object-el elszállni az alkalmazásunk. Erre egyébként is szükségünk van, mivel több autó modellt kezel az alkalmazás és előfordül olyan mesh, ami az egyikben szerepel, de a másikban nem.

A React Three Fiber-ben a mesh tag, egy geometriát vár paraméterben, és egy anyagi objektumot gyerekként. Így lesz teljes egy modellünk megjelenése és a hozzárendelt anyagi objektummal, amit mi készítünk el külön komponensként, biztosítjuk hogy alkalmazáson belül teljes hozzáférésünk van a paraméterek módosításához,

optimalizáláshoz, ellenkező esetben a glTF fájl minden eleméhez egy ***meshStandardMaterial*** lenne hozzárendelve, ami számunka pazarás és optimalizálás hiányát jelentené.

Ezeket a materialokat, magyarul anyagi tualjdonságokat külön komponensként hozom létre. Így jól struktórált marad az alkalmazás minden fájlja és egyszerűen használható például a fentebb említette ***leva*** eszköz is, ami segítségével egyedileg tudunk módosítani mindent anyagi tulajdonság paraméterén.

2.6 ANYAGI TULAJDONSÁGOK

Three.js által hozzáférünk rengeteg előre megírt anyag osztályhoz amelyek a megjelenést hivatottak szolgálni. Egy materialnak vannak bemenetei amik felhasználásával egy úgynevezett shader programon keresztül kiszámolja a grafikus processzor hogy az adott területen megjelenő objektumnak milyen megjelenése legyen, figyelembe véve a fényeket, beállított anyagi tulajdonságokat és a renderer beállításait.

Ezek az anyagok különböző módon reagálnak a fényekre és árnyékokra, és eltérő hatással vannak a renderelési teljesítményre. Íme egy áttekintés néhány gyakori three.js anyagról, összehasonlítva azok teljesítményét és vizuális minőségét.

MeshBasicMaterial: Ez az anyag nem veszi figyelembe a jelenet fényeit, ezért nagyon könnyű és gyorsan renderelhető. Ideális fény nélküli jelenetekhez, mint például drótvázak vagy egyszínű objektumok.

MeshLambertMaterial: Ez az anyag Lambert-féle visszaverődést használ, így teljesítményben jobb, mint a Phong-alapú anyagok, mert a fényhatásokat csúcspontoknál számítja, nem pedig pixelenként. Jó választás alapvető diffúz visszaverődésre szoruló objektumokhoz, nagy részletesség nélkül.

MeshPhongMaterial: Részletesebb tükröződési kiemeléseket kínál, mint a Lambert, de nagyobb számítási igényű, mivel pixelenként számolja a fény kölcsönhatásokat. Fényes felületekhez megfelelő.

MeshStandardMaterial: Ez az anyag fizikailag alapú renderelési modellt használ, biztosítva realisztikusabb fényeket és árnyékokat. Nagyobb teljesítményigényű, mint a Lambert vagy Phong, de sokkal magasabb minőségű eredményeket nyújt.

MeshPhysicalMaterial: A MeshStandardMaterial kiterjesztése, további tulajdonságokat, mint például tiszta bevonatot biztosít, ami még realisztikusabbá és vizuálisan vonzóbbá teszi, de növeli a számítási terhelést.

Teljesítmény és felhasználási mód közötti különbségeket a következő táblázatba foglaltam össze:

Anyag	**Teljesítmény Költsége**	**Kimeneti Minőség**	**Alkalmazási terület**
MeshBasicMaterial	Nagyon Alacsony	Alacsony	Fény nélküli jelenetek, alapszínek
MeshLambertMaterial	Alacsony	Mérsékelt	Egyszerű világítás, gyorsabb renderelés
MeshPhongMaterial	Közepes	Magas	Részletes világítás, fényes felületek
MeshStandardMaterial	Magas	Nagyon Magas	Realisztikus renderelés, fejlett világítás
MeshPhysicalMaterial	Nagyon Magas	Legmagasabb	Speciális realisztikus felületek

React Three Fiberen belül érdemes komponensekre bonatni a különböző materialokat. Példának vegyük az autó fényezésének az anyagát, az alkalmazásban ennek tudjuk módosítani a színét, és ***MeshPhysicalMaterial*** lett felhasználva erre a célra, mert ez az egyetlen anyag ami tudja számolni lakkozott felületek megvilágítását is, továbbá mivel az alkalmazásban autók szerepelnek ezért a képernyő nagy részét ez az tulajdonság fogja kitenni, szóval érdemes jó megjelenésű shadert használni.

```
import { folder, useControls } from 'leva'
import useStore from '../../stores/useStore.jsx'

export default function Carpaint(props) {

    const colors = useStore((state) => state.colors)
    const colorIndex = useStore((state) => state.colorIndex)

    const { color, roughness, metalness, clearcoat, clearcoatRoughness } = useControls('materials', { ···
    })

    return <meshPhysicalMaterial
        color={ colors[colorIndex].color }
        roughness={ roughness }
        metalness={ metalness }
        clearcoat={ clearcoat }
        clearcoatRoughness={ clearcoatRoughness }
        envMapIntensity={props.envMapIntensity}
        />
}
```

2.6. ábra: Kódrészlet egy fényezés anyagi tulajdonságot tartalmazó komponensről

A színek egy állapotba vannak lementve, és paraméterként átadva az anyag komponensnek. Így ha a felhasználó módosítja a színt az alkalmazásban, az anyag komponens újra fog futni a React-on belül és frissíti a változást.

2.7 ÁLLAPOTKEZELÉS

Amikor a React Three Fibert használunk, az állapotkezelés és annak propson keresztüli átadása nehézkes és negatív hatása lehet a teljesítményre. Ez a hagyományos módszer, az állapotot olyan komponenseken keresztül adja át, amelyeknek nem feltétlenül kell tudniuk az állapotról, csak hogy elérjék azokat a komponenseket, amelyeknek szükségük van rá. Ez felesleges újrarajzolásokhoz vezethet, és bonyolulttá és nehezen kezelhetővé teheti a komponens hierarchiát.

Ezért van szükségünk egy állapotkezelőre, amiből le tudják kérni a komponensek az őket érintő álapotokat. A Zustand lehetővé teszi, hogy egyszerű API-val hozzon létre globális állapotot, ehhez egyéni hookokat használ, így könnyen integrálható és használható a React komponensekben. A Zustand nem támaszkodik a React kontextusra, ami azt jelenti, hogy az állapotfrissítések nem eredményeznek újrarajzolásokat a komponensfában. Ez hatékonyabbá teszi, mivel csak azok a komponensek frissülnek újra, amelyek az állapot bizonyos részeire iratkoztak fel. A Zustand megváltoztathatatlan állapotfrissítéseket támogat, amelyek könnyebben kezelhetők és hibakeresésük egyszerűbb. Ha szükséges, az állapotlogikát több tárolóra oszthatja, ami megkönnyíti a nagyméretű alkalmazások kezelését.

Zustand könyvtárat az npm segítségével kell letöltenünk és importálás után használhatóvá is válik. Állapotok létrehozásához a **create** objektumot kell felhasználnunk ami egy call back function-on keresztül várja az állapotok listáját, és értékeit. Ezeket később bármelyik komponensben fel tudjuk használni a **useStore** hook-on keresztül. Állapotok listáját függvényekkel is tudjuk bővíteni, így például megoldható az is hogy egy komponens meghív egy függvényt egy tőle teljesen független másik komponensből.

```
import { create } from 'zustand'

export default create((set) => ({
    car: 's2000', // Default car state
    setCar: (type) => set(() => ({car: type})),
    rotate: false,
    setRotate: (value) => set(() => ({rotate: value})),
```

2.7. ábra: Állapotok létrehozása Zustand create segítségével

Minden deklarált állapothoz kell még egy setter függvény is amit referencián keresztül meg tudunk hívni az adott állapotot változtató komponensből. A fenti képen láthatjuk az aktuálisan kiválasztott modellt, ami string értékként tárolja le az állapotot. Utána meg egy boolean értéket a kamera forgatásához kapcsolódóan. Mind a két adattaghoz van setter is amit az ***Interface*** komponensből hívok meg event hatására. Ilyenkor a setter függvényt hívom meg az új állapotot átadva a függvénynek.

2.8 FELHASZNÁLÓI FELÜLET

A funkciók irányításához egy felhasználói felületet implementáltam, ahonnan gombok és kiválaszható menük segítik az autó, háttér és a fényezés kiválasztását, valmint a kamera forgatás ki – be kacsolását és a fotó mód gombját.

```jsx
export default function Interface(){

    const setCar = useStore((state) => state.setCar)
    const setRotate = useStore((state) => state.setRotate)
    const rotate = useStore((state) => state.rotate)
    const colors = useStore((state) => state.colors)
    const colorIndex = useStore((state) => state.colorIndex)
    const setColorIndex = useStore((state) => state.setColorIndex)
    const background = useStore((state) => state.background)
    const setBackground = useStore((state) => state.setBackground)
    const saveImage = useStore((state) => state.saveImage);

    return <div className="interface">
        <div className="button-container">
            <div className="dropdown-container">
                <select onChange={(e) => setCar(e.target.value)} defaultValue="s2000">
                    <option value="s2000">S2000</option>
                    <option value="civic">Civic</option>
                </select>
                <select onChange={(e) => setColorIndex(e.target.value)} defaultValue={colorIndex}>
                    {colors.map((item, index) => (
                        <option key={index} value={index}>{item.name}</option>
                    ))}
                </select>
                <select onChange={(e) => setBackground(e.target.value)} defaultValue={background}>
                    <option value="studio">Studio</option>
                    <option value="sunrise">Beach</option>
                    <option value="road">Highway</option>
                    <option value="depot">Building</option>
                </select>
                <button className="button-40" onClick={(e) => setRotate(!rotate)}>Rotate {rotate ? 'OFF':'ON'}</button>
                <button className="button-40" onClick={saveImage}>Capture</button>
            </div>
        </div>
```

2.8. ábra: Felhasználói interfész komponens

Az állapotok változtatására, ***Zustand*** segítségével lekértem a settereket és az event funkciókat, majd a html elemeknél meghívom a megfeleő paraméterekkel.

3.0 OPTIMALIZÁLÁSOK

A React Three Fiber és a Three.js alkalmazások optimalizálása a webes és mobil eszközökön különösen fontos több kulcsfontosságú okból. Ezek a keretrendszerek lehetővé teszik a bonyolult és vizuálisan vonzó 3D grafikák létrehozását webes platformokon, amelyek alapvetően nagy erőforrás-igényűek. Figyelembe véve a különféle eszközképességeket, különösen a mobiltechnológiában, létfontosságú, hogy ezek az alkalmazások zökkenőmentesen működjenek minden platformon az elérhetőség és a felhasználói megtartás érdekében.

A teljesítmény elsődleges szempont, mivel a 3D renderelés jelentős számítási teljesítményt és memória használatot igényel, amely megterhelheti az eszköz erőforrásait, ami lassú betöltési időket, akadozó animációkat és akár alkalmazás összeomlását eredményezheti. Ez különösen problémás lehet a mobil eszközökön, amelyek általában kevesebb feldolgozási kapacitással és memóriával rendelkeznek, mint az asztali gépek. Egy rosszul optimalizált 3D jelenet kimerítheti az akkumulátor élettartamát, túlzott adatfelhasználást eredményezhet és hőt termelhet a kézben tartott eszközökön.

Ezenkívül az alkalmazás válaszkészsége közvetlenül kapcsolódik a felhasználói élményhez. A felhasználók gyorsan betöltődő és zökkenőmentesen működő webes és mobilalkalmazásokat várnak el. A nehéz 3D grafika miatt lassú vagy nem válaszképes felületek frusztrációt okozhatnak, és a felhasználók feladhatják az alkalmazást. A React és annak ökoszisztémája kontextusában a magas teljesítményű szabvány fenntartása azt is jelenti, hogy biztosítani kell, hogy a virtuális DOM újrarenderelési folyamata ne váljon akadállyá a 3D jelenet gyakori frissítései miatt.

3.1 HDRI KÉPEK OPTIMALIZÁLÁSA

Alkalmazásunk hátterét nagy felbontású és nagy dinamikájú ***hdr*** formátumú képek adják. Ezek a képek viszont elég nagy méretűek és ez lassú betöltést okozhat.

Egy átalgos 2k felbontású, 32 bites hdr fotó 7 mb helyet foglal. Mivel a legtöbb kijelző 8 bites színtérrel dolgozik, levehetjük ezt a színmélységet 8 bitre is, de fontos megemlíteni hogy a 32 bites színmélységre is szükségünk van, ugyanis a környezeti megvilágítás minősége ezen múlik, továbbá a modellen lévő tükröződések is felhasználják ezt a színmélységet és ettől függ a tükröződések minősége is.

Megoldás erre a problémára hogy háttérnek egy 2k felbontású, de 8 bites tömörített ***webp*** formátumú képet használunk. Környezeti megvilágításhoz és tükröződésekhez pedig egy alacsony felbontású, de 32 bites színmélységű ***hdr*** képet használunk.

Név	Formátum	Színmélység	Felbontás	Méret
chinese_garden	hdr	32 bit	2048 x 1024	6.68 mb
chinese_garden (environment)	hdr	32 bit	256 x 128	111 kb
chinese_garden (background)	webp	8 bit	2048 x 1024	470 kb

Tehát a betöltendő fájlméret 92% -al csökkent, látványbeli minőségvesztés nélkül. Ez jelentősen növeli a betöltési időt és a mobil eszközöket használó felhasználók adatforgalma is sokkal jobban kímélve van ennek az optimalizálásnak köszönhetően. Ingyenes online eszközök vannak képek optimalizálására, ilyen például a Google által fejleszett, nyílt forráskódú ***squoosh*** alkalmazás amivel a webp-be konvertálást végeztem.

3.2 GLTF MODELLEK OPTIMALIZÁLÁSA

Mivel elég komplexek tudnak lenni a 3D modellek, nagyon oda kell figyelni hogy exportáljuk ki őket, mit mellékelünk a lementett fájlba, és milyen használatra. 3D objektumok pivot pontjai, méret és orientációk, vertex color adatok, UV koordináták, csak hogy megemlítsek párat. A legnagyobb tényező amelyik befolyásolja a méretet egy gltf fájlban, az textúrák mennyisége és formátuna. glTF készítői egy alkalmazás készítettek, amivel könnyen és egyszerűen lehet fájlokat elemezni, optimalizálni. Ezt ***glTF Report*** –nak hívják.

Név	Tömörítés	Textúrák formátuma	Modell formátuma	Méret
spinning_wheel	nincs	tga	gltf	5.89 mb
spinning_wheel (optimalizált)	draco	webp	glb	491 kb

glTF report eszköz használatával, a modell méretét 91% -al sikerült lecsökkenteni. Textúrák webp-be konvertálása, felesleges adatok kitakarítása a hierarhiából, duplikált vertexek törlése és a draco tömörítés a kis méret titka.

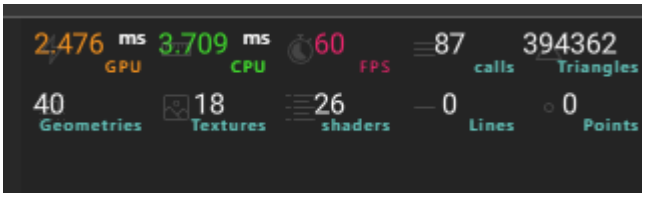
```
import {prune, dedup, draco, textureCompress } from '@gltf-transform/functions';
import sharp from 'sharp'; // Node.js only.

await document.transform(
    // Remove unused nodes, textures, or other data.
    prune(),
    // Remove duplicate vertex or texture data, if any.
    dedup(),
    // Compress mesh geometry with Draco.
    draco(),
    // Convert textures to WebP (Requires glTF Transform v3 and Node.js).
    textureCompress({
        encoder: sharp,
        targetFormat: 'webp',
        resize: [1024, 2024],
    })
);
```

3.2. ábra: glTF Transform használata modellek optimalizálására

3.3 RENDERELÉS OPTIMALIZÁLÁSA

Amikor egy jelenetet renderelünk a képernyőre, a processzor végigmegy minden egyes objektumon és elküldi a rajzolás utasítást a grafikus processzornak a modellel, hozzá tartozó textúrákkal és a modellhez tartozó shader programmal együtt. Ez a legnagyobb „szűk keresztmetszet” a 3D grafikában, adatok átvitele a processzor és a videókártya között. Minél kevesebb draw call-al rendelkezik a jelenetünk, annél jobb lesz a renderelési teljesítmény. Tehát arra kell törekednünk 3D alkalmazás készítésekor, hogy minél kevesebb draw call-al dolgozzunk, ezt úgy érjük el hogy az azonos shadert és textárát felhasználó objektumokat egyesítjük, így csökkentve a draw call-ok számát.

React Three Fiber-ben van lehetőség monitorozni a teljesítményt a ***Perf*** nevű komponensel amit a ***r3f-perf*** csomagból érhetünk el. Használathoz csak létre kell hozni a komponenst a React Three Fiber alkalmazásunk **Canvas** komponensünk alatt.



3.3. ábra: Teljesítmény monitorozás perf komponens használatával

A komponens használata után, alkalmazásunk jobb felső sarkában megjelennek a statisztikák amiből fontos adatokat olvashatunk le. A mellékelt képen látszik, hogy monitorozni tudjuk az aktuális képkocka renderelésének processzor és videokártya idejét milliszekundumban, képkockák számát másodpercenként, draw callok számát, a jelenetben lévő háromszögek száma. Továbbá a második sorban találhatóak az objektumok, textúrák és felhasznált shaderek száma. A visszajelzések segítségével optimalizálhatjuk a jelenetünk renderelését és ezzel az alkalmazásunk teljesítményét.

3.4 REACT THREE FIBER OPTIMALIZÁLÁSA

Hatékony React Three Fiber (R3F) alkalmazások létrehozása a React általános optimalizálási technikáinak és a three.js rendereléshez kapcsolódó speciális stratégiáknak a kombinációját igényli.

A komponensek újrarenderelésének optimalizálásához a ***useMemo*** hook-ot a felesleges újrarenderelések elkerülésére. Továbbá a ***useCallback*** és ***useMemo*** függvényeket a függvények és értékek memoizálására, amelyeket propsként ad át komponenseknek, így elkerülve a felesleges újrarendereléseket.

A futás alatti modell, textúra betöltéshez a ***Suspense*** komponenst kell használni, így csak akkor tölti be őket, amikor szükséges. Emellett dinamikusan importál könyvtárakat vagy komponenseket.

Állapotok változása ***Zustand*** –en keresztül történik és csak a változást lekezelő komponensek frissülnek ha valami változik.

4.0 ELÉRT EREDMÉNY

Az elkészült program gyorsan és gördülékenyen fut, illetve betölt gyengébb eszközökön is, az optimalizálásoknak köszönhetően. A fejlesztések lehetőséget biztosítanak az eszköz továbbfejlesztésére. A dokumentált optimalizáslások pedig felhasználhatóak további 3D-s alkalmazásokban.

Betöltéskor kevesebb mint 10 mb adatot tölt be a weboldal, sciptekkel együtt és mobil eszközön, valamint régi PC-n is 60 kékocka / másodperc sebességen megy a 3D jelenet renderelése.

Az UI html elemekkel lett megoldva, és mobil eszközökön, különböző orientációk mellett is működőképes. Az állapotok ***Zustand*** állapotkezelőből egyenesen az érintett komponensekhez lesznek felhasználva, ezzel mellőzve a felesleges újra rendereléseket, számításokat a ***React*** virtuális hirearchiáján belül.

4.1 TOVÁBBFEJLESZTÉSI LEHETŐSÉGEK

Az alkalmazás bővítése és felhasználó barátabbá tétele benne van a rövid távú célaim között. A bővítési és fejlesztési lehetőségek magukba foglalják a következőket.

- **Grafikai beállítások a felhasználó számára:** Mivel most csak egy alapértelmezetten beállított vizuális megjelenés van, célszerű lenne legalább 3 minőségbeli beállítást elérhetővé tenni. Alacsony beállítással élsimítás kikapcsolása, normál beállításoknál a jelenlegi beállítások használata, valamint egy magas opciónál különböző utófeldolgozási effektek is belekerülnének az alkalmazásba.
- **Utófeldolgozási effektek:** Bloom, depth of field, screen space ambient occlusion post processing számítások hozzáadása az alkalmazáshoz a még realisztikus megvilágítás és valósághű megjelenés elérése érdekében.
- **Interaktív elemek:** Raycast segítségével a React Three Fiber objektumok képesek érzékelni, hogy ha a felhasználó kijelölt egy elemet a 3D térben. Ezt az ***onClick*** attribútum felhasználásával meg lehetne tenni és például a modell valamelyik része megváltozhatna, elindulhatna egy animáció stb.

IRODALOMJEGYZÉK

1. React Three Fiber (2023.04.10): https://docs.pmnd.rs/react-three-fiber/getting-started/introduction
2. Drei (2023.03.16): https://github.com/pmndrs/drei#readme
3. Zustand (2023.04.01): https://docs.pmnd.rs/zustand/getting-started/introduction
4. Blender (2023.05.06): https://www.blender.org/about/
5. React (2024.05.02): https://react.dev/
6. Three.js (2024.04.22): https://threejs.org/docs/index.html#manual/en/introduction/Installation
7. W3schools (2024.03.21): https://www.w3schools.com/
8. Three.js Journey (2024.04.16): https://threejs-journey.com/
9. Coursera – Meta React Kurzus (2023.11.12): https://www.coursera.org/professional-certificates/meta-front-end-developer
10. WebGL Wiki (2024.02.03): https://www.khronos.org/webgl/wiki/Main_Page
11. OpenGL (2024.03.15): https://www.opengl.org/
12. Rapier (2023.12.10): https://rapier.rs/docs/user_guides/javascript/getting_started_js/
13. EffectComposer (2024.03.10): https://docs.pmnd.rs/react-postprocessing/effect-composer
14. Node Package Manager (2024.03.20): https://www.npmjs.com/
15. Poly Haven (2024.03.12): https://polyhaven.com/hdris
16. leva (2024.04.12): https://github.com/pmndrs/leva
17. draco (2024.04.10): https://google.github.io/draco/
18. squoosh (2024.05.10): https://squoosh.app/
19. glTF Transform (2024.05.02): https://gltf-transform.dev/
20. glTF Report (2024.05.02): https://gltf.report/
21. p3f-perf (2024.05.11): https://www.npmjs.com/package/r3f-perf
22. Suspense (2024.05.10): https://react.dev/reference/react/Suspense
23. WebGPU (2024.05.10): https://en.wikipedia.org/wiki/WebGPU

NYILATKOZAT

Alulírott Pullai Szilárd Üzemmérnök-Informatikus szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Szegedi Tudományegyetem, Informatikai Intézet Szoftverfejlesztés Tanszékén készítettem, Üzemmérnök-Informatikus BProf diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel. Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Diplomamunka Repozitóriumában tárolja.

Szeged, 2024.04.20.

aláírás