

**Szegedi Tudományegyetem**

**Informatikai Intézet**

**OpenGL Shader programozó játék kezdő  
shader fejlesztőknek**

Szakdolgozat

*Készítette:*

**Rávai Adrián**

Programtervező informatikus BSc

hallgató

*Külső témavezető:*

**Gál Péter**

PhD hallgató

*Belső konzulens:*

**Dr. Kiss Ákos**

Egyetemi Adjunktus

Szeged

2021

# Tartalomjegyzék

Feladatkiírás . . . . .	4
Tartalmi összefoglaló . . . . .	5
Bevezetés . . . . .	6
<b>1. Háttér</b>	<b>8</b>
1.1. GLFW . . . . .	8
1.2. GLEW . . . . .	8
1.3. OpenGL . . . . .	9
1.3.1. Az OpenGL grafikus csővezeték . . . . .	10
1.4. Dear ImGui . . . . .	12
1.5. Kép összehasonlítás . . . . .	12
1.6. A GLSL shading nyelv . . . . .	13
1.7. Elvárások az alkalmazással szemben . . . . .	13
<b>2. Az alkalmazás felépítése</b>	<b>15</b>
2.1. OpenGL Renderelés . . . . .	15
2.1.1. SceneElement . . . . .	15
2.1.2. VertexBuffer . . . . .	16
2.1.3. VertexBufferLayout . . . . .	17
2.1.4. VertexArray . . . . .	18
2.1.5. IndexBuffer . . . . .	19
2.1.6. Shaderek . . . . .	19

2.1.7. Uniformok . . . . .	20
2.1.8. Textúrák . . . . .	20
2.2. Widgetek . . . . .	23
2.2.1. Widget interface . . . . .	23
2.2.2. Widget broker . . . . .	24
2.3. Kép összehasonlítás . . . . .	26
2.3.1. Chi square distance . . . . .	26
2.3.2. Structural similarity index measure . . . . .	27
<b>3. A kész alkalmazás bemutatása</b>	<b>30</b>
3.1. Az alkalmazás fordítása . . . . .	30
3.1.1. Függőségek . . . . .	30
3.1.2. Fordítás . . . . .	31
3.2. Az alkalmazás bemutatása . . . . .	33
3.2.1. Feladat elkészítése és exportálása . . . . .	33
3.2.2. Feladat betöltése . . . . .	38
3.2.3. Kép összehasonlítás . . . . .	38
Összefoglalás . . . . .	42
3.3. Elért eredmények . . . . .	42
3.4. Továbbfejlesztési lehetőségek . . . . .	42
Irodalomjegyzék . . . . .	44

# Feladatkiírás

## OpenGL/ES Shader programozó játék

Az alkalmazás célja, hogy a segítse a kezdő OpenGL(ES) programozókat és különféle feladatok során gyakoroltassa a shader programok írását, konfigurálását.

A programban megadott rajzolási feladatokat kell a felhasználónak megoldania.

Az alkalmazás a felhasználó által megadott shader és uniform adatok alapján OpenGL (ES) felhasználásával rajzol ki egy képet. Ezt a képet az éppen aktuális feladat által megkövetelt eredmény képhez hasonlítja a program. Amennyiben a felhasználó által "készített" kép megegyezik a feladatban elvárttal, akkor jöhet a következő feladat.

A felhasználói felület interaktívnak kell legyen, azaz ha pl.: a shader kód módosul, akkor az az által kirajzolandó képnek meg kell jelennie (akár automatikusan akár egy gomb lenyomásával) anélkül, hogy magát az alkalmazást újrafordítanánk.

A felhasználó számára biztosítani kell a következő lehetőségeket. A háttér szín megadását, (clear color), a vertex és fragment shaderek módosíthatóságát, vertex input attribútumok, megadását, tetszőleges uniformok hozzáadását (név, típus, érték), tetszőleges samplerek/képek megadását (név, típus, képfájl).

Adjunk lehetőséget a "sandbox" módra is. Lehetőség szerint a feladatok listájának bővíthetőnek kell lennie. Az alkalmazás újrafordítása nélkül lehessen további feladatokat hozzáadni.

A megvalósításnak modern OpenGL (ES)-ben kell történnie, célszerűen Windows és Linux platformon is működnie kell és nem lehet Web-es megvalósítású (nem WebGL).

# Tartalmi összefoglaló

A téma megnevezése, OpenGL Shader programozó játék, kezdő shader fejlesztőknek.

A feladat egy olyan desktop alkalmazás elkészítése, mely egy felületet biztosít shader programok forráskódjának a szerkesztéséhez, vertex, index, uniform, textúra adatok beviteléhez. Majd az adatok által generált képet összehasonlítani egy kitűzött cél képpel.

OpenGL, GLFW, GLEW könyvtárak felhasználása úgy, hogy az alkalmazás Windows és Linux rendszereken is működjön. Kép összehasonlítása SSIM és Chi square distance algoritmusok implementálásával.

Az alkalmazás fejlesztése C++ nyelven történt. Build rendszer gyanánt CMAKE-t, multiplatform ablak és bevitel kezelésre a GLFW könyvtárat, multiplatform extension betöltésre a GLEW könyvtárat, grafikus API-ként pedig az OpenGL-t használtam. Felhasználó felületet a DearImGui könyvtárral valósítottam meg, a képek beolvasását, exportálását pedig az stbi\_image könyvtár segítségével. A felhasználói felülethez használtam egyéb, már előre megvalósított, nyílt forráskódú widget-eket.

Az alkalmazás forráskódja bárki számára elérhető GitHub-on, így bárkihez eljuthat, akit érdekel a téma. A feladatkiírás által elvárt fő funkciókat sikeresen implementáltam és megfelelően működnek.

Kulcsszavak: Shader programozás, Kép összehasonlítás, OpenGL

# Bevezetés

A téma azért tetszett meg, mert mindig is érdekelt a GPU programozás és videójátékok révén a 3D grafika is. Szintúgy a C++ nyelv áll hozzám a legközelebb az összes közül. Célom egy olyan alkalmazás elkészítése volt mely biztosít mindent a felhasználó számára úgy, hogy a felhasználónak a shader kód írásán kívül más feladata ne legyen.

Mindenki aki shader programozásba akar fogni, belebotlik abba a problémába, hogy mennyi mindent kell megvalósítani és bekonfigurálnia mielőtt a programozáshoz kezdhetne. Olyan problémákba mint könyvtárak beszerzése, fordítása, build rendszerek konfigurálása. OpenGL-en belül vertex, index, uniform, textúra adatok bevitele, shader programok fordítása, feltöltése, OpenGL API hívások debugolása. Ezek az előkészületek akár napokba is telhetnek a felhasználó képzettségétől függően.

Az alkalmazás aktuális állapotát lehetséges egy .tsk kiterjesztésű fájlba exportálni. Ez a fájl importálható és ez képezi a feladat alapállapotát. Innentől a felhasználó számára biztosítva van egy kép, amit meg kell valósítania. A vertex és index adatokból az alkalmazás egy poligont rajzol. Ez a poligon festővászonként szolgál a felhasználó számára. A vertex shaderben ennek a poligonnak a tulajdonságait programozhatja a felhasználó, a fragmens shaderben pedig a raszterációs folyamatot, azaz hogy a poligon pixelei milyen színt vegyenek fel.

A rajzolt és a cél kép összehasonlításához több kép összehasonlító algoritmust is használok, hogy minnél pontosabb eredményt kapjak. Ezek az algoritmusok statisztikai módszereket alkalmaznak, nem pedig gépi tanulást a mai trendekkel ellentétben.

Az alkalmazás szintén használható képfeldolgozó algoritmusok gyors implementálására és vizualizálására.

Mintául a GLSL Sandbox nevű webalkalmazást vettem és annak az alapfunkcióit bővítettem ki. Ezek a plusz funkciók a programozható vertex shader, vertex, index, uniform, textúra adatok beolvasása, módosítása, generált kép exportálása. Szintén képes több objektum rajzolására így például color blendinget is fel lehet használni.

A kész program lightweight, a kész bináris fájl kevesebb mint 10MB helyet foglal és átlagos használat esetén 200MB memóriánál többre nincs szüksége. Minden Windows vagy Linux rendszeren működnie kell ahol biztosítva van egy integrált vagy dedikált GPU ami támogatja az OpenGL-t.

# 1. fejezet

## Háttér

A feladat egy grafikus C++ alkalmazás megvalósítása. Az alkalmazás függőségként a GLFW, GLEW könyvtárakat használja.

### 1.1. GLFW

A GLFW egy multiplatform ablak kezelő könyvtár. Operációs rendszertől függően implementálja az ablak megjelenítést és a bevitel kezelést. Windowson ez a Win32 API-n, Linuxon pedig a X.Org API-n keresztül történik.

### 1.2. GLEW

A GLEW egy multiplatform extension loader könyvtár. Ez a könyvtár dönti el, hogy operációs rendszertől és hardvertől függően melyik OpenGL extension-öket kell vagy lehet betölteni az alkalmazáshoz.

Ezeket felhasználva az alkalmazás már képes egy üres ablakot megjeleníteni és egér illetve billentyűzet bevitelt kezelni. Tovább haladva biztosítsunk egy vizuális felületet a felhasználó számára az ImGui könyvtárral.



## 1.3. OpenGL

Az OpenGL [1] (Open Graphics Library) egy szabvány, melyet a Silicon Graphics nevű amerikai cég fejlesztett ki 1992-ben. Segítségével egy egyszerű, szabványos felületen keresztül megvalósítható a grafikus kártya kezelése. Ezek segítségével olyan alkalmazásokat hozhatunk létre melyek képesek gyorsabb teljesítményt elérni az által, hogy nem a processzor végez minden számítási feladatot hanem egy részét kiosztja a videokártya számára.

Széleskörűen használják számítógép által támogatott tervezésben, gyártásban (CAD/CAE), virtuális és augmentált valóság megteremtésében. Ezenkívül használják videojátékokban, Windows, Linux platformokon illetve mobiltelefonokon, konzolokon és emulátorokon is. Beágyazott rendszereken az OpenGL ES változata a legelterjedtebb.

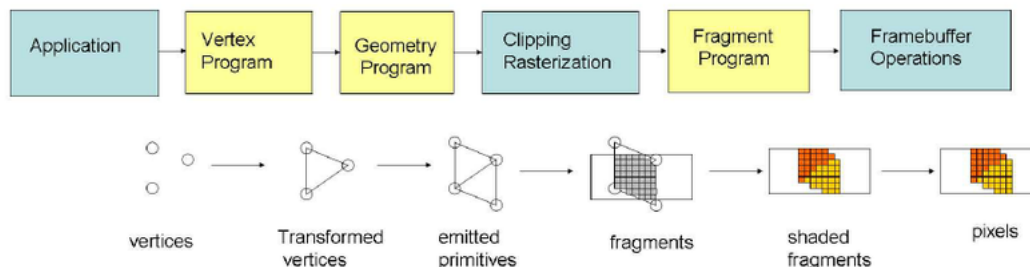
A grafikus kártyák legnagyobb erőssége abban rejlik, hogy képesek bizonyos számítási műveleteket egyszerre párhuzamosan több adaton is elvégezni (Single data, multiple instructions SIMD). Ezek tipikusan mátrixokkal és vektorokkal kapcsolatos számítási műveletek.

A nyílt forráskódú szoftverfejlesztés közösségében kifejezetten elterjedt, hiszen az OpenGL rendelkezik a legtöbb nyíltforráskódú implementációval.

Érdemes még megemlíteni az OpenGL vetélytársait: a zárt forráskódú DirectX-et melyet a Microsoft fejleszt és a feltörekvő szintén nyíltforráskódú Vulkan-t.

### 1.3.1. Az OpenGL grafikus csővezeték

Akárhányszor egy rajzolási műveletet végzünk, ez a 1.1 ábrán látható grafikus csővezetékön keresztül történik. A grafikus csővezeték öt lépésből épül fel.



1.1. ábra. Az OpenGL grafikus csővezeték[4]

#### Vertex feldolgozás

1. A vertexeket egyesével beolvassuk a video memóriába, úgy nevezett VAOk-ból (Vertex Array Object), melyek a memóriában léteznek.
2. Opcionális, primitív tesszelálása, bonyolult geometriai alakzatok felbontása egyszerűbb alakzatokká.
3. Opcionális, primitívek feldolgozása geometria shaderrel.

#### Vertex utófeldolgozás

Ebben a fázisban az előző fázis kimenetén igazítunk vagy módosítunk.

1. Úgynevezet transform feedback, ahol a vertex feldolgozás kimenetét buffer objektumokba mentjük. Ezáltal képesek vagyunk megőrizni a transzformáció utáni állapotot és ezt az adatot ezután többször is felhasználni.
2. Primitív Assembly, ahol a vertexek összekötésével valamilyen alakzatot képzünk.

3. Primitívek fedésének feloldása, perspektíva felosztása, és a nézetablak transzformáció az ablak térbe.

## Raszterizáció

A raszterizáció az a folyamat amikor egy alakzatokból álló vektor képet, pixelekből felépülő raszter képpé alakítunk. A raszterizáció az egyik leggyakoribb technika amit a számítógépes grafikában használunk, nagyon gyors folyamat ezért a grafikus motorok valós időben alkalmazzák. A raszterizáció szimplán csak annak a kiszámítása, hogy a térben lévő geometriákat hogyan alakítsuk pixelekké. A pixelek színét pedig a fragmens shader határozza meg.

## Fragmens shader

A programozható fragmens shader ami minden fragmenst feldolgoz és már a monitoron megjeleníthető kimenetet készít.

## Mintánkénti feldolgozás

1. Scissor teszt, ami levágja azokat a fragmenseket amik nem férnek bele a megjelenítendő képbe.
2. Stencil teszt, ahol a stencil bufferben tárolt értékek alapján elhagyhatunk fragmenseket.
3. Depth teszt, mélység érzet keltése, minél távolabb van egy fragmens a térben, annál világosabb lesz a színe.
4. Blending teszt, ezáltal érhető el az átlátszóság vagy az üveg hatás keltése, alpha érték alapján az egymást fedő pixelek színe keveredik.
5. Logikai műveletek, melyeket a fragmens színeit ábrázoló biteken végezhetünk.
6. Write mask, az előző lépések használni kívánt kimeneteit egy maszkban egyesíti.

A grafikus pipeline-nak biztosítsuk az adatokat amiket a felhasználói felületen keresztül adhatunk meg. Ezek az adatok legyenek a következők: vertexenként kilenc darab float érték, ebből az első három darab a vertex kordináták, második három darab a szín értékek, a harmadik három darab pedig a textúra kordináták. Kelleni fog még indexenként egy unsigned integer, a vertex és fragmens programok forráskódja, illetve a használni kívánt textúrák elérési útja.

A rendereléshez minden szükséges adat megvan a textúra adatokon kívül, ezekhez jelenleg csak az elérési útvonalak vannak meg. Az útvonalon keresztül olvassuk be a kép adatokat az stb\_image könyvtár segítségével.

## 1.4. Dear ImGui

Az ImGui egy úgynevezett "Immediate mode graphical user interface", azaz jellemzően a felhasználói felület ugyanazon a szálon fut mint az alkalmazás, hogy a lokális adatokat is elérje. Függvény hívásokból épül fel ahelyett, hogy a UI felépítése valamilyen leíró nyelven lenne egy külön fájlban meghatározva. Leggyakoribban grafikus alkalmazások debugolására használják mert használata nagyon egyszerű, gyors, intuitív és nem különösebben befolyásolja az alkalmazás teljesítményét. A kinézet testreszabása terén viszont nem ad túl nagy szabadságot a fejlesztő számára.

Ezen a ponton van már egy felhasználói felületünk amin keresztül meg tudunk hívni függvényeket, képesek vagyunk változtatni változók értékén. Most már elkezdhetjük a grafikát implementálni az OpenGL segítségével.

## 1.5. Kép összehasonlítás

Most már minden adatunk megvan egy kívánt kép generálásához. Két kép összehasonlítására használjuk az SSIM és Chi square distance statisztikai módszereket alkalmazó algoritmusokat. Fontos, hogy a két összehasonlítandó kép azonos felbontású legyen, különben ezek a módszerek nem alkalmasak. Utolsó lépésként ezek eredményét jelenítsük

meg a felhasználó számára.

A chi square distance algoritmus[2] veszi két kép hisztogramját és egyesével összegzi színcsatornánként a rajzolt és a cél kép színértékeinek különbségének a négyzetét majd eloszja a rajzolt kép színértékével. Minnél kisebb az összeg, annál jobb az egyezés.

Az SSIM elődjét Universal quality Index (UQI)-mal hívták, melyet Zhou Wang és Alan Bovik fejlesztett ki 2001-ben. Ezt fejlesztették tovább Hamid Sheikh és Eero Simoncelli segítségével azzá, amit ma SSIM-ként ismerünk[3]. Az SSIM algoritmust általában arra használják, hogy megvizsgálják mekkora a képromlás az eredeti kép és egy tömörített kép között. A képfeldolgozó közösségben az egyik legtöbbször használt algoritmus, a Google scholar szerint a 2004-es tanulmányt több mint húszezren hivatkoztak rá azóta.

Itt is fontos, hogy a két képünk, ezek legyenek  $x$  és  $y$  képek, azonos felbontásúak legyenek, ugyanúgy mint a chi square distance esetén. Az SSIM minőségi indexet az alábbi képlet alapján kapjuk meg, melyet szintén minden csatornára ki kell számolnunk, majd összegeznünk.

## 1.6. A GLSL shading nyelv

Az OpenGL shading nyelv egy magas szintű shader nyelv aminek a szintaxisa leginkább a C programozási nyelvére hasonlít. Azzal a céllal készült, hogy a fejlesztőknek több irányítása legyen a grafikus csővezeték felett anélkül, hogy shader assembly kódot kellene írniuk vagy más egyéb hardver specifikus nyelvet használni. Mint a legtöbb programozási nyelvben a program belépési pontja a main függvény. A nyelv szintén tartalmaz vektor és mátrix típusokat is, hogy a velük kapcsolatos műveletek egyszerűbbek legyenek. A programozó munkáját továbbá könnyebbítik egyéb beépített matematikai függvények mint például trigonometrikus függvények, gyökvonás, hatványozás.

## 1.7. Elvárások az alkalmazással szemben

- Clear color azaz a háttérszín állítása

- Konzol amin keresztül az alkalmazás információkat oszt meg a felhasználóval szöveges formában
- Konzolon keresztüli alkalmazás vezérlés szöveges parancsokkal
- Konzolon megjelenített szövegek törlése
- Kódszerkesztő widget, ahol a shader forráskódokat szerkeszthetjük
- Fájl böngésző widget, amin keresztül fájlok elérési útjait választhatjuk ki
- Scene szerkesztő widget, ahol a megjelenített objektumok adatait szerkeszthetjük
- Alkalmazás aktuális állapotának exportálása JSON fájlba
- Alkalmazás állapotának betöltése JSON fájlból
- Multiplatform, képes Windows és Linux operációs rendszereken is futni
- Alacsony gépigény
- Reszponzivitás, az alkalmazást kényelmes használni

## 2. fejezet

# Az alkalmazás felépítése

### 2.1. OpenGL Renderelés

A renderelés egy `Renderer` osztályban van megvalósítva. A konstruktora paraméterül vár egy `Context` és egy `UniformManager` objektumot. A `Context` objektum egy referenciát tárol egy scene-hez, a scene egy lista ami `SceneElement` objektumokat tárol, egy `SceneElement` tárol minden szükséges adatot és funkcionalitást, hogy a képernyőre rajzoljunk egy tetszőleges geometriai alakzatot.

#### 2.1.1. SceneElement

Egy `SceneElement`-nek tárolnia kell az alábbiakat:

- Egy nevet ami azonosítja az objektumot.
- Vertex és Fragmens shader forráskódokat.
- Vertex kordinátákat
- Vertex indexeket
- Használni kívánt textúrák elérési útvonalát.

A konstruktor beállítja az adattagokat, majd meghívja az `InitializeSceneElement()` függvényt. Ez a függvény létrehozza a következő objektumokat:

- `VertexArray`
- `VertexBuffer`
- `VertexBufferLayout`
- `IndexBuffer`
- `Shader`
- `Textures`

Minden OpenGL objektumot egy `uint32_t` `RendererID` azonosít. Mikor létrehozunk egy objektumot, ezt a változót adjuk át referenciaként és az OpenGL függvény beállít neki egy pozitív egész számot, mint azonosító.

### 2.1.2. VertexBuffer

Mikor létrejön egy `VertexBuffer` pontosan a fentebb említett történik, a referenciaként átadott változóba beleírja a buffer azonosítóját.

```
VertexBuffer::VertexBuffer(const void* data, uint32_t
size)
{
    glGenBuffers(1, &m_RendererID);
    glBindBuffer(GL_ARRAY_BUFFER, m_RendererID);
    glBufferData(GL_ARRAY_BUFFER, size, data,
GL_STATIC_DRAW);
}
```



A `void* data` egy pointer ami egy egydimenziós tömbre mutat, ami a vertex koordinátákat a memóriában lineárisan tárolja. Azáltal, hogy void típusú pointert adunk át, képesek vagyunk akármilyen típust átadni egy castolás segítségével, a `uint32_t size` pedig a tömb méretét adja meg, amit az alábbi képlet alapján számolhatunk ki.

$$(numberOfVertices/vertexLength) * vertexLength * sizeof(type)$$

A `glGenBuffers()` függvény létrehoz egy OpenGL buffer objektumot és beállítja egy azonosítót, majd a `glBindBuffer(GL_ARRAY_BUFFER, m_RendererID)` ehhez az azonosítóhoz hozzárendeli, hogy ez egy `GL_ARRAY_BUFFER` típusú objektumot jelöl. A `glBufferData(GL_ARRAY_BUFFER, size, data, GL_STATIC_DRAW)` majd ezt az `ArrayBuffer`-t feltölti a kapott adatokkal statikus módon.

Egy `ArrayBuffer`-t kétféle képpen tölthetünk fel adattal: statikusan, vagy dinamikusan. A statikus feltöltéssel azt jelezzük, hogy ezek az adatok csak egyszer lesznek módosítva majd többször felhasználva GL rajzolási parancsokhoz, a dinamikus feltöltéssel ellentétben többszöri módosításra és többszöri felhasználásra utal.

### 2.1.3. VertexBufferLayout

Szükséges, hogy meg tudjuk mondani, hogy egy Vertex milyen hosszú. Ezt egy `VertexBufferLayout` által tehetjük meg. A layout-ot az alábbi függvény által állíthatjuk be:

```
template<> inline
void VertexBufferLayout::Push<float>(uint32_t count)
{
    m_Elements.push_back({ GL_FLOAT, count, GL_FALSE });
    m_Stride += GetSizeOfType(GL_FLOAT) * count;
}
```

Minden hívással azt állíthatjuk be, hogy a vertex egyes attribútumai hány érték hosszúak, milyen típusúak és hogy kell-e őket normalizálni.

```
struct VertexBufferElement
{
    uint32_t type;
    uint32_t count;
    uint32_t normalized;
};
```

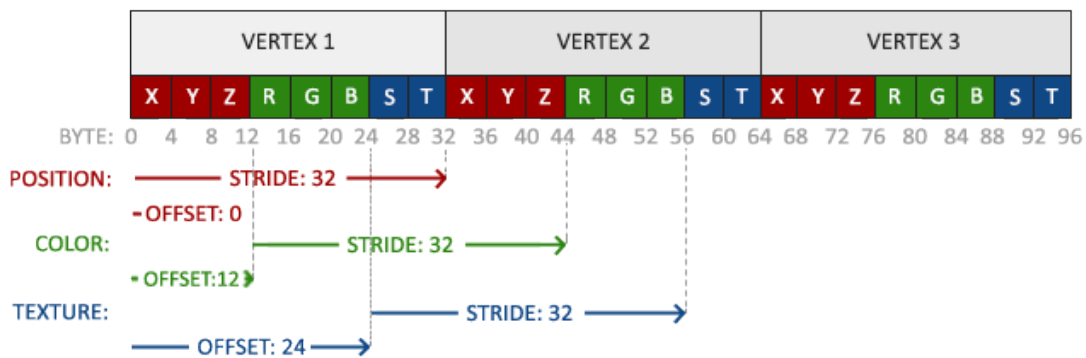
### 2.1.4. VertexArray

Most már, hogy van egy `VertexBuffer` és egy `VertexBufferLayout`, ezekből létrehozhatunk egy `VertexArray`-t. Először bind-oljuk a `VertexArray`-t, majd a `VertexBuffer`-t.

Ezt követően a layout-unkból lekérjük, hogy milyen elemeket tartalmaz, ezeknek az elemeknek pedig allokálnunk kell helyet. `glEnableVertexAttribArray(i)` függvény segítségével engedélyezzük az *i*-edik indexű vertex array-t.

A `glVertexAttribPointer()`-el végül beállítjuk, hogy egy attribútum, hány elem-ből áll, milyen típusokból, kell-e az értéket normalizálni, a stride-ot (lépésközt) ami megmondja, hogy a memóriában az egyes attribútumok között vertexenként hány bájtnyi lépés van, és az offset-et, ami meghatározza, hogy az egyes attribútumok hanyadik bájtól kezdődnek.

A 2.1 képen látható is, hogy minden érték négy bájt helyet foglal el a tömbben. A stride minden esetben harminckettő bájt lesz, mert vertexenként  $8 * 4$  bájtnyi memóriát allokálunk. Az offset pedig az első attribútum elem előtti értékek méreteinek az összege.



2.1. ábra. Vertex attribútumok elrendezése a memóriában[6]

### 2.1.5. IndexBuffer

A primitive assembly során minden vertexet összekötünk egy bizonyos sorrendben, ezáltal megadva az alakzatunkat. Erre használhatunk egy úgynevezett `IndexBuffer`-t, ami megadja, hogy ez a folyamat milyen sorrendben történjen. Szimplán csak egy tömbre van szükségünk ami pozitív egész számokat tárol amik megmondják, hogy a vertexek milyen sorrendben lesznek összekötve és annak a hosszára.

### 2.1.6. Shaderek

Sokan amikor a shader szót először meghallják, rögtön arra gondolnak, hogy megvilágítás vagy árnyékolás. Valójában egy shader szimplán csak egy program, ami a videokártyán fut. Egy shader program létrehozásához csupán a forráskódjára és a típusára van szükségünk. Első lépésben létrehozunk egy shader-t a típusa alapján a `glCreateShader()` metódussal, melyet egy pozitív egész szám fog azonosítani, OpenGL-ben ez az objektum azonosító. Beállítjuk neki a forráskódját a `glShaderSource()` függvénnyel, majd lefordítjuk a `glCompileShader()` metódussal. Ha a fordítás során valami hiba történik, akkor ezt ki tudjuk írni a felhasználó számára és ez alapján kijavíthatja a hibát a kódjában. Ha a shader sikeresen lefordult, akkor visszatérhetünk az azonosítójával.

Még nem vagyunk teljesen kész, a lefordított shadereket hozzá kell csatolnunk egy

programhoz, majd ezt a programot linkelnünk. Itt is ugyanúgy kiírathatjuk a hibaüzenetet, ha valami akadályba ütközik az API a linkelés során. Végző lépésben a programot validálhatjuk, majd törölhetjük a shadereket mert már nincs rájuk szükségünk és visszatérünk a program azonosítójával, ami az objektum `RendererID`-ja lesz. Ezt az azonosítót átadva a `glUseProgram(m_RendererID)` függvénynek a programot aktiválhatjuk.

### 2.1.7. Uniformok

A uniformok olyan globális GLSL változók, melyeket kézzel állíthatunk be kívülről és akár minden képkocka során frissíthetjük őket. Uniformok segítségével tudunk olyan információkat feltölteni a shaderbe, mint a képernyő felbontását, a kurzor pozícióját, az eltelt időt, és textúrákat samplerok formájában az azonosítójukon keresztül. Egy uniform beállításához három dologra van szükségünk, a shader program azonosítójára amihez szeretnénk a uniformot csatolni, egy string-re, amivel majd tudunk hivatkozni a uniformra a forráskódon belül, és az értékre, amit a uniformnak adni akarunk. Az alkalmazásban minden uniformot egy `u_` prefixel jelölök.

```
m_UniformManager.SetUniform1f(rendererId,
                                "u_" + name + "Time",
                                glfwGetTime());
```

### 2.1.8. Textúrák

Képesek vagyunk minden shaderhez több darab textúrát is rendelni. A `Textures` konstruktor egy listát vár paraméterül, ami tartalmazza a használni kívánt textúrák elérési útját. A képek beolvasására használjuk az `stb_image` kép kezelő könyvtárat, aminek a kép beolvasásához csak egy elérési útvonalra, néhány paraméterre és egy változóra van szüksége, amiben majd el tudja tárolni a kép adatait. Minden textúrát beolvasunk az `stbi_image` könyvtár `stbi_load()` függvényével, a textúrák lehetnek például `.jpg`, `.png`, `.bmp` formátumú képek. Ez paraméterül vár egy elérési utat, két változót amibe beállítja a beolvasott

textúra szélességét és magasságát, még egy változót, amibe a fájl színcsatornáinak a száma kerül és végül, hogy ebből hány csatornát szeretnénk megtartani.

Minden textúrához kérnünk kell egy azonosítót az API-tól, majd beállítani, hogy ez az azonosító egy textúrát jelöl.

```
glGenTextures(1, &m_TextureIds[counter]);
```

A textúráknak megadhatunk különböző paramétereket, hogy egyes esetekben miként kezelje a textúrák rajzolását. Például, hogy végezzen a textúrán akármilyen filterezést vagy átméretezés során inkább széthúzza a textúrát vagy megismételje azt.

Ezután pedig fel kell töltenünk a textúra adatait a grafikus memóriába.

```
glTexImage2D(GL_TEXTURE_2D,  
             0, GL_RGBA8,  
             m_Width,  
             m_Height,  
             0, GL_RGBA,  
             GL_UNSIGNED_BYTE,  
             m_LocalBuffer);
```

Utolsó lépés pedig, hogy a textúrához rendeljünk egy sampler típusú uniformot, amin keresztül majd a forráskódban tudunk rá hivatkozni.

```
m_UniformManager.SetUniformli(  
    rendererId,  
    samplerName.str(),  
    counter);
```

### subsectionAz objektumok kapcsolatai

Ezeket az objektumokat lényegében azért hoztuk létre, hogy a shaderekben adatokat tudjunk elérni. A vertex tömbökön keresztül a vertex koordinátákat, a textúrákon keresztül

kép adatokat. Az indexek sorrendje szerint összekötjük a vertexeket, a vertex shader alapján transzformáljuk őket, a fragmens shader szerint pedig színezzük a rajzolt alakzatokat. Az uniformokon pedig egyéb adatokat érhetünk el a shaderek forráskódjaiban.

## 2.2. Widgetek

A widgetek, másnéven window objectek, olyan objektumok amik a felhasználói felületet reprezentálják. A widgetek felelősek az adatok megjelenítéséért, a felhasználói interakció kezeléséért.

### 2.2.1. Widget interface

A widgeteket egy interface-ként valósítottam meg. C++-ban az interface egy olyan osztály, ami csak virtual függvényekből áll.

```
class Widget {  
    public:  
        virtual void OnUpdate(float deltaTime) {};  
        virtual void OnRender() {};  
        virtual void OnImGuiRender() {};  
        virtual void RenderWidget() {};  
};
```

Az alkalmazásban minden widget ezt az interfacet implementálja. Ez segít abban, hogy a widgeteket generikusan tudjuk majd példányosítani upcastolás segítségével. Az `OnUpdate()` függvényben történik a változók képkockánkénti frissítése, az `OnRender()`-ben pedig az OpenGL render hívások, majd az `OnImGuiRender()`-ben az ImGui render hívások, ez a függvény mindig egy ImGui ablak context-ba van ágyazva.

```
void TaskWidget::RenderWidget() {
    OnUpdate(0.0f);
    OnRender();
    ImGui::Begin("windowName");
    OnImGuiRender();
    ImGui::End();
}
```

### 2.2.2. Widget broker

A widgetek példányosításáért egy `WidgetBroker` osztály felel. Az osztály `MakeWidget()` metódusa template paraméterként várja, hogy milyen típusú Widgetet készítsen, függvény paraméterként pedig egy `WidgetType` enum értéket kap, ami a widget azonosítója lesz, ezután pedig a példányosítani kívánt widget konstruktorának a paramétereit adhajtuk át. Végül eltároljuk egy map-ben a widgetet és visszatérünk egy pointerral ami egy unique smart pointer-ra mutat ami maga a widget.

```
template <typename WidgetT, typename... Arguments>
Widget* MakeWidget(WidgetType id, Arguments&&...
ArgumentValues) {
    assert(m_Widgets.count(id) != 1);
    m_Widgets[id] = std::make_unique<WidgetT>(
        std::forward<Arguments>(ArgumentValues)...);
    return m_Widgets[id].get();
}
```

Az alkalmazás különböző pontjain tudnunk kell az egyes widgetek-et manipulálni, például ha valamilyen funkcionalist egy gyorsbillentyűhöz akarunk kötni akkor a GLFW bevitel kezelő callback függvényeiben meg kell tudnunk hívni egy widget publikus metó-



dusát. Ezt a `GetWidget()` függvénnyel tehetjük meg, ami azonosító alapján megkeresi a map-ben a használni kívánt widget-et és visszaad egy arra mutató pointert, amit upcastolunk a generikusként megkapott widget típusra.

```
template <typename WidgetT>
WidgetT* GetWidget(WidgetType id) {
    return (WidgetT*)m_Widgets[id].get();
}
```

Példa a használatára.

```
context->widgetBroker
    .GetWidget<TextEditorWidget>("TextEditor")->Save();
```

Az alkalmazásban szereplő widgetek:

- Text editor widget: shader forrásfájlok szerkesztése
- Scene editor widget: scene elementek létrehozása, törlése, clear color beállítása és a scene element adatainak beállítása
- Console widget: oda-vissza kommunikáció az alkalmazás és a felhasználó között
- File browser widget: fájl böngésző amin keresztül fájlok elérési útjait tudjuk megadni
- Task widget: cél és a rajzolt kép közlése a felhasználóval, szintén ezen a widgeten keresztül tudunk feladatot beállítani és a képeket összehasonlítani.

## 2.3. Kép összehasonlítás

Az alkalmazásnak képesnek kell lennie egyszerű kép összehasonlításra, hogy a felhasználó kapjon valamilyen visszajelzést arról, hogy elégszült-e a feladattal. Természetesen száz százalék pontosságot nem lehet elvárni se a felhasználótól, se az algoritmusoktól, így a hibahatárt harminc százalékra állítottam. Hetven százalékos egyezés esetén a megoldást már elfogadottnak tekintjük. Fontos kikötés, hogy az összehasonlítás során mindkettő képnek azonos felbontásúnak kell lennie, különben a képet át kellene méretezni, ami artifactek-kel járhat. Ez megint csak megnehezítené a kép összehasonlítás már alpból nehéz feladatát.

### 2.3.1. Chi square distance

A chi square distance algoritmust a [2.1](#) képlet szemlélteti.

$$CSD(x, y) = \sum_{i=1}^n \frac{(x_i - y_i)^2}{y_i} [2] \quad (2.1)$$

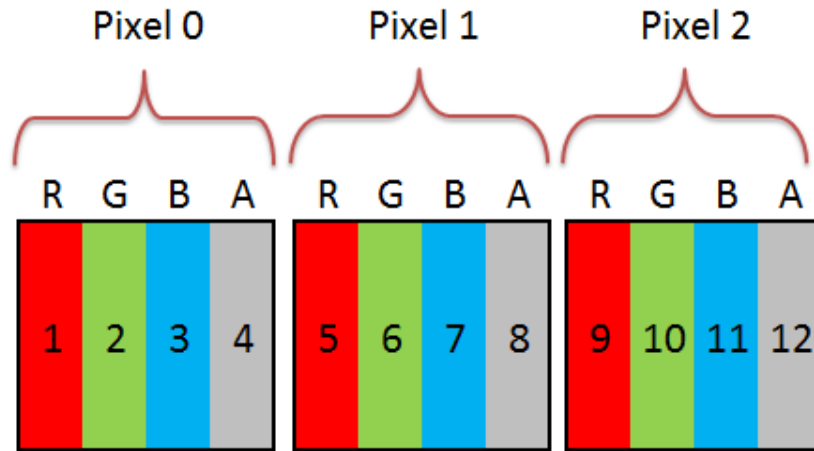
ami egy RGBA komponensű képre a [2.2](#) képlet.

$$RGBACSD(x, y) = CSD(x_r, y_r) + CSD(x_g, y_g) + CSD(x_b, y_b) + CSD(x_a, y_a) \quad (2.2)$$

Sajnos az algoritmus olyan rosszul teljesít, amennyire egyszerű. Forgatásra, eltolásra, tükrözésre nem érzékeny és kicsi változtatásokra is nagy eltérést tud mutatni.

Az algoritmust egy `ImageComparator` osztályban implementáltam. Az implementáció első része, hogy a kép minden színcsatornájából egy hisztogramot készítek, melyet egy struktúrával valósítottam meg. Esetemben minden kép RGBA komponensekből áll, tehát minden pixel négy bájt és minden bájt az RGBA komponens egyik színértékét ábrázolja mint ahogy ez látható a [2.2](#) ábrán. Így csak létrehozok minden színcsatornának egy  $2^8 - 1 = 255$  elemű tömböt aminek minden eleme nullára van inicializálva és a színértéknek megfelelő tömb indexen lévő értéket inkrementálok eggyel. Minden csatorna esetén csak a kinyerni kívánt csatorna első értékének a helyétől kell a ciklust indítani és minden

iterációban a ciklus változót a komponens hosszával növelni.



2.2. ábra. RGBA komponens felépítése[5]

Miután megvannak a hisztogram struktúráink, csak a fentebbi képlet alapján kiszámolunk egy értéket minden csatornára, majd összegezzük az összes színcsatornára kapott eredményt és visszatérünk vele.

A kapott érték egy  $[0, width * height * maxColorValue * componentLength]$  intervallum közötti lebegőpontos érték. Két kép akkor számít azonosnak ha a kapott érték nyolc tized vagy annál kisebb.

### 2.3.2. Structural similarity index measure

Az SSIM algoritmust a 2.3 képlet definiálja.

$$SSIM(x, y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)} [3] \quad (2.3)$$

ahol

- $\mu_x$  az x kép egyik színcsatornájának összes értékének az átlaga.
- $\mu_y$  az y kép egyik színcsatornájának összes értékének az átlaga.

- $\sigma_x^2$  az x kép egyik színcsatornájának összes értékének a varianciája.
- $\sigma_y^2$  az y kép egyik színcsatornájának összes értékének a varianciája.
- $\sigma_{xy}$  az x és az y kép egyik színcsatornájának a kovarianciája.
- $c_1 = (k_1 L)^2, c_2 = (k_2 L)^2, c_3 = (c_2/2)$  három változó ami stabilizálja az osztást gyenge nevező esetén.
- $L$  a legnagyobb érték, amit egy komponens felvehet, mi esetünkben ez  $2^8 - 1 = 255$
- $k_1 = 0.01$  és  $k_2 = 0.03$  alapból.

A varianciát a 2.4, kovarianciát pedig a 2.5 képletek alapján számoltam.

$$Var(x) = \sum_{i=0}^n \frac{(x_i - X)^2}{n} [3] \quad (2.4)$$

$$Cov(x, y) = Cov(X - k_x, Y - k_y) = \frac{\sum_{i=0}^n (x_i - k_x)(y_i - k_y) - (\sum_{i=0}^n (x_i - k_x))(\sum_{i=0}^n (y_i - k_y))/n}{n} [3] \quad (2.5)$$

Az SSIM formula három összehasonlító függvényből áll össze, ezek a fényerő(2.6), a kontraszt(2.7) és a struktúra(2.8). A különböző összehasonlító függvények a következők:

$$l(x, y) = \frac{2\mu_x\mu_y + c_1}{\mu_x^2 + \mu_y^2 + c_1} [3] \quad (2.6)$$

$$c(x, y) = \frac{2\sigma_x\sigma_y + c_2}{\sigma_x^2 + \sigma_y^2 + c_2} [3] \quad (2.7)$$

$$s(x, y) = \frac{\sigma_{xy} + c_3}{\sigma_x\sigma_y + c_3} [3] \quad (2.8)$$

az SSIM index pedig a 2.9 képlet szerinti összehasonlító méréseknek a súlyozott szorzata. A mi esetünkben az  $\alpha, \beta, \delta$  súlyokat egyre állítjuk.

$$SSIM(x, y) = [l(x, y)^\alpha * c(c, y)^\beta * s(x, y)^\delta][3] \quad (2.9)$$

Egy RGBA komponensű képre pedig az SSIM index a [2.10](#) képlet alapján számítható.

$$RGBASSIM(x, y) = SSIM(x_r, y_r) + SSIM(x_g, y_g) + SSIM(x_b, y_b) + SSIM(x_a, y_a) \quad (2.10)$$

Az így kapott SSIM index egy  $[0, 4]$  közötti érték, hozzuk százalékos alakra a [2.11](#) képlettel. Amiben  $N$  a kép komponens hossza.

$$\frac{SSIM(x, y)}{N} * 100 \quad (2.11)$$

Két képet akkor tekintünk azonosnak ha az SSIM szerint hetven százalékban egyeznek.

## 3. fejezet

# A kész alkalmazás bemutatása

Az alkalmazás forráskódja elérhető a <https://github.com/CptNero/SlimShady> oldalon.

### 3.1. Az alkalmazás fordítása

#### 3.1.1. Függőségek

Az alkalmazás fordítása előtt a felhasználónak be kell szerezni a GLEW<sup>1</sup> és GLFW<sup>2</sup> könyvtárakat, mint függőségek.

Windows rendszereken ajánlott az előrefordított binárisokat beszerezni, majd elhelyezni őket a Dependencies mappában és a mellékelt CMAKE fájlokat használni a fordítás során.

Linux rendszereken a könyvtárakat magunknak kell telepíteni, majd a CMAKE scriptben a `find_packages()` paranccsal megkeresni.

A projekt mindkét rendszerhez külön CMAKE build scriptet tartalmaz és a használt header only könyvtárak pedig a Vendor mappában találhatóak.

---

<sup>1</sup><https://github.com/nigels-com/glew>

<sup>2</sup><https://github.com/glwf/glwf>

A használt könyvtárak listája:

- [imgui](#)<sup>3</sup>
- [imfilebrowser](#)<sup>4</sup>
- [ImGuiColorTextEdit](#)<sup>5</sup>
- [GLM](#)<sup>6</sup>
- [stb\\_image/stb\\_image\\_write](#)<sup>7</sup>
- [Cereal](#)<sup>8</sup>
- [date](#)<sup>9</sup>

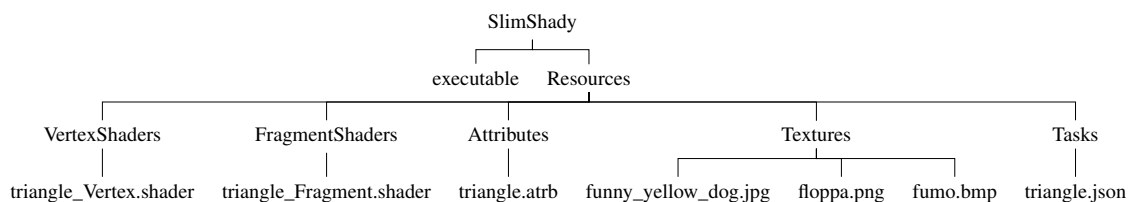
### 3.1.2. Fordítás

Az alkalmazás a következő parancsokkal fordítható:

```
cmake ~Slimshady
```

```
cmake --build ~Slimshady/build/bin
```

Az alkalmazás könyvtár struktúrája a [3.1](#) ábrán látható:



3.1. ábra. Az alkalmazás könyvtár struktúrája

---

<sup>3</sup><https://github.com/ocornut/imgui>

<sup>4</sup><https://github.com/AirGuanZ/imgui-filebrowser>

<sup>5</sup><https://github.com/BalazsJako/ImGuiColorTextEdit>

<sup>6</sup><https://github.com/g-truc/glm>

<sup>7</sup><https://github.com/nothings/stb>

<sup>8</sup><https://uscilab.github.io/cereal/>

<sup>9</sup><https://github.com/HowardHinnant/date>

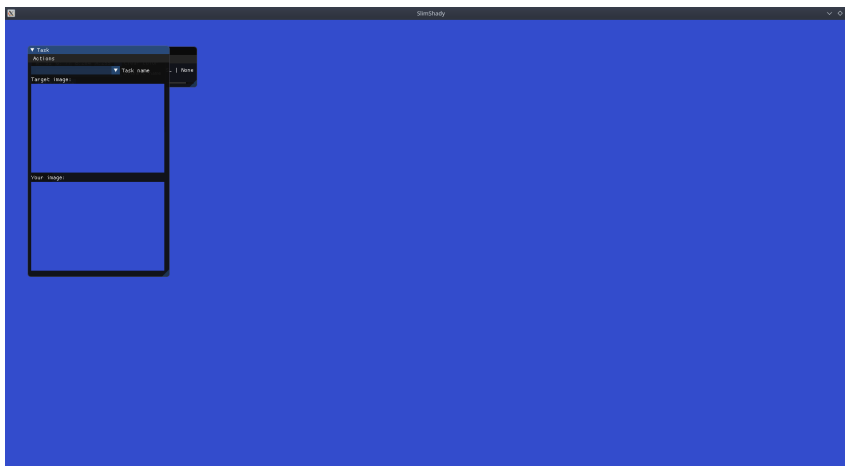
A VertexShaders és a FragmentShaders mappákban helyezkednek el a vertex és a fragmens shaderek forráskódjai. Az attributes mappában helyezkednek el az attribútum fájlok, amik a vertexeket, indexeket és a textúrak elérési útvonalát tartalmazzák. A textures mappában kell elhelyezni a használni kívánt textúrákat. A Tasks mappában pedig az exportált feladatok találhatók meg JSON formátumban.



## 3.2. Az alkalmazás bemutatása

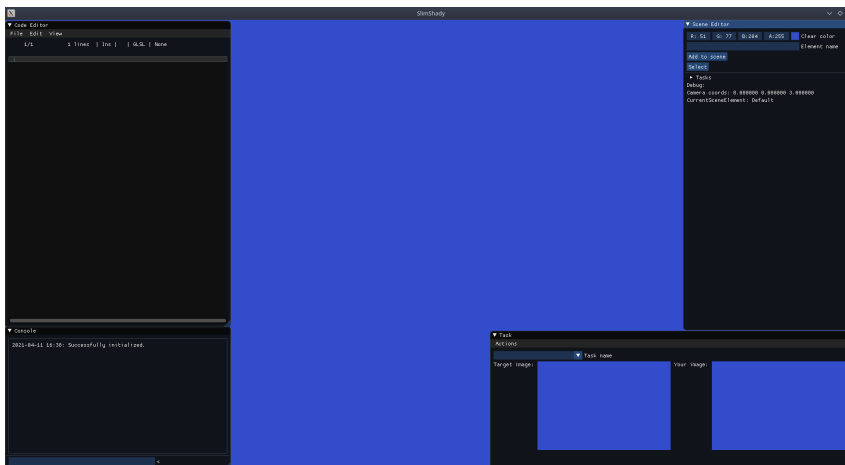
### 3.2.1. Feladat elkészítése és exportálása

Amikor a felhasználó elindítja az alkalmazást a 3.2 ábra látványával találkozik.



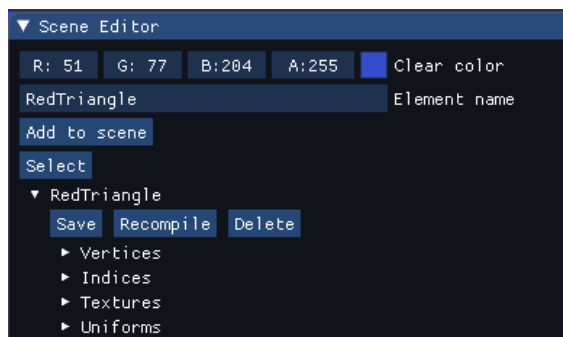
3.2. ábra. Az alkalmazás állapota az első indítás során

A widgeteket célszerű úgy elrendezni ahogy a felhasználó számára kényelmes, ezt csak egyszer kell megtenni, az alkalmazás a widget pozíciókat megtartja legközelebb is. Egy szépen elrendezett példa látható a 3.3 ábrán.



3.3. ábra. Az alkalmazás rendezett UI-al

Hozzunk létre egy új objektumot a 3.4 ábrán látható SceneEditor widget segítségével. Az "Element name" mezőben adjunk meg egy nevet az objektumunknak, majd nyomjuk meg az "Add to scene" gombot.



3.4. ábra. Objektum létrehozása

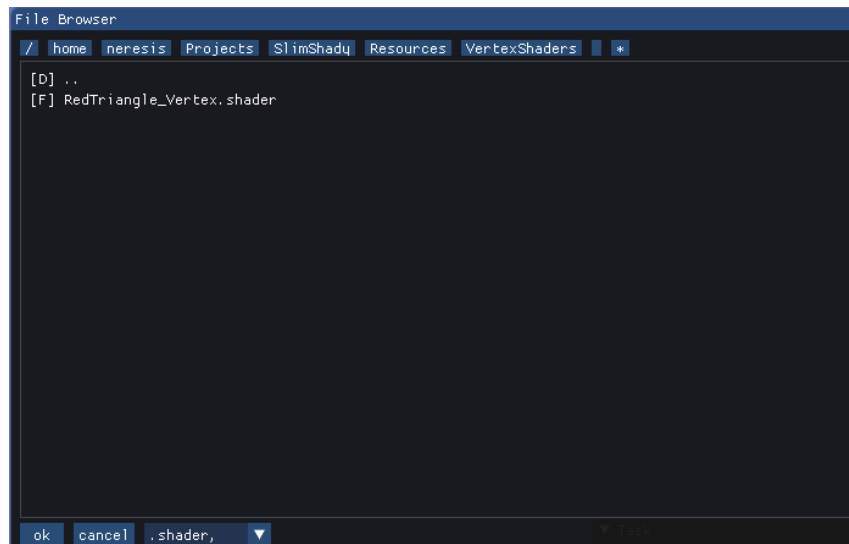
A "vertices" fület lenyitva adhatunk meg új vertexeket és hasonlóan indexeket mint ahogy a 3.5 ábra mutatja. Adjuk meg egy háromszög vertexeit és indexeit, a "Save" gombbal pedig mentjük el őket.



3.5. ábra. Vertex koordináták beállítása

A háromszögünk még nem fog megjelenni, hiszen még nem adtuk meg a vertex és

fragmens shadereket. Nyissuk meg az objektumunk vertex shader fájlját a szövegszerkesztőben, melyet a "File" fül alatt az "Open Vertex shaders" opcióval tehetünk meg. A 3.6 ábrán látható FileBrowser widget jelenik meg előttünk, ahonnan válasszuk ki a vertex shader fájlunkat.



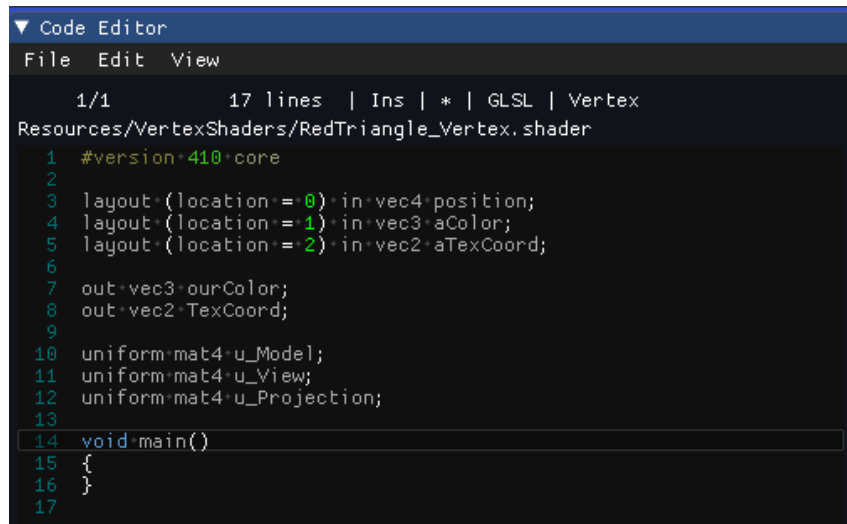
3.6. ábra. Fájl böngésző

A fájlt kiválasztva a shader kód jelenik meg előttünk a szövegszerkesztőben melyet kedvünkre szerkeszthetünk.

### **A szerkesztő rendelkezzeik pár kényelmi funkcióval:**

- Sorok számozása
- Tabulálások jelzése
- Szintaxis kiemelés
- Jelenleg megnyitott fájl elérési utvonalának jelzése
- Szöveg statisztika
- Többféle szín paletta

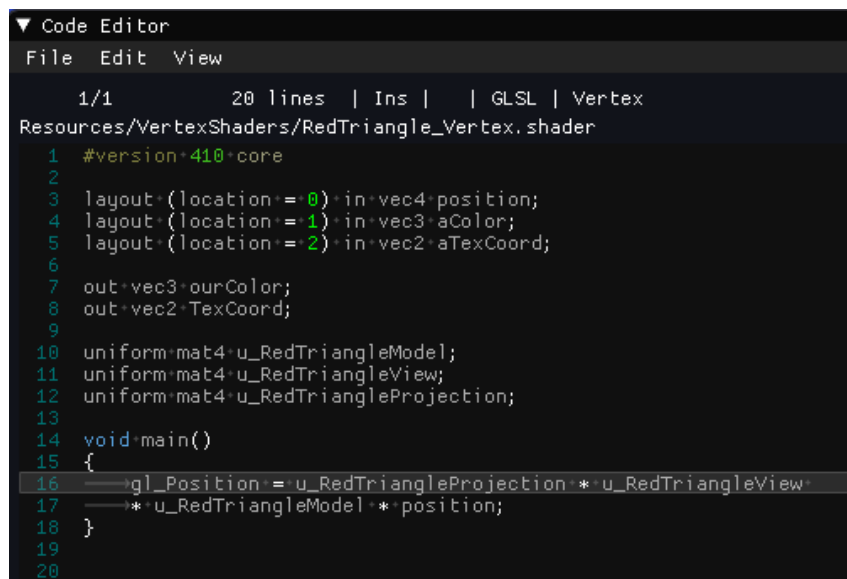
A kódszerkesztő widget a 3.7 ábrán látható.



```
1 #version 410 core
2
3 layout(location=0) in vec4 position;
4 layout(location=1) in vec3 aColor;
5 layout(location=2) in vec2 aTexCoord;
6
7 out vec3 ourColor;
8 out vec2 TexCoord;
9
10 uniform mat4 u_Model;
11 uniform mat4 u_View;
12 uniform mat4 u_Projection;
13
14 void main()
15 {
16 }
17
```

3.7. ábra. Sablon vertex forráskód

Szerkesszük meg úgy a kódot, hogy a vertex pozíciókat a model, nézet és projekciós mátrixokkal beszorozva értékül adjuk mint ahogy a 3.8 ábrán láthatjuk, majd a "File" tab alatt a "Save" opcióval mentjük el a fájlt.



```
1 #version 410 core
2
3 layout(location=0) in vec4 position;
4 layout(location=1) in vec3 aColor;
5 layout(location=2) in vec2 aTexCoord;
6
7 out vec3 ourColor;
8 out vec2 TexCoord;
9
10 uniform mat4 u_RedTriangleModel;
11 uniform mat4 u_RedTriangleView;
12 uniform mat4 u_RedTriangleProjection;
13
14 void main()
15 {
16 gl_Position = u_RedTriangleProjection * u_RedTriangleView *
17 u_RedTriangleModel * position;
18 }
19
20
```

3.8. ábra. Kész vertex forráskód

Hasonló módon nyissuk meg a fragmens shaderünket és a FragColor változót állítjuk be úgy, hogy a háromszögünket pirosra színezzük, szemléltetve a 3.9 ábra által.

Mentsük el a módosításokat és a SceneEditor widgeten nyomjuk meg a "Recom-

```

▼ Code Editor
File Edit View

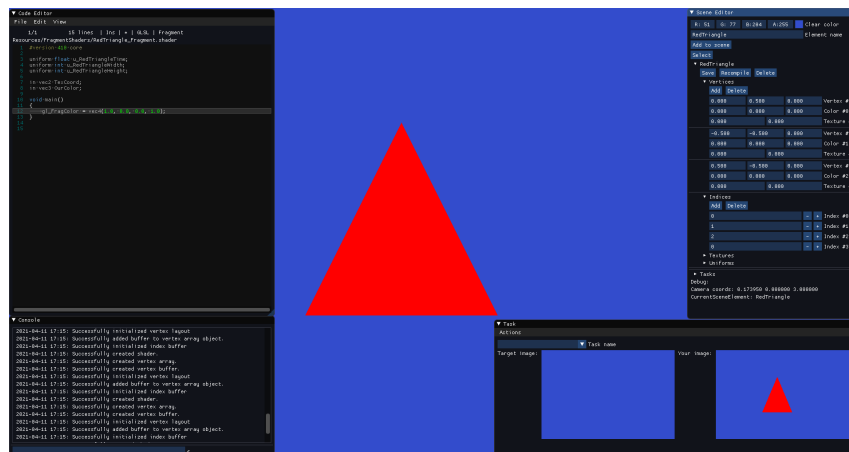
1/1 15 lines | Ins | * | GLSL | Fragment
Resources/FragmentShaders/RedTriangle_Fragment.shader

1 #version 410 core
2
3 uniform float u_RedTriangleTime;
4 uniform int u_RedTriangleWidth;
5 uniform int u_RedTriangleHeight;
6
7 in vec2 TexCoord;
8 in vec3 OurColor;
9
10 void main()
11 {
12     gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
13 }
14
15

```

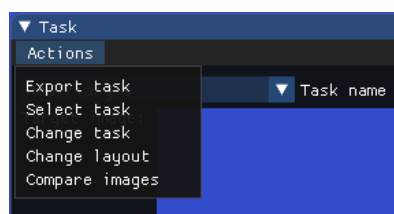
3.9. ábra. Fragmens forráskód

pile" gombot. Most már sikeresen megjelenik a piros háromszögünk mint ahogy a [3.10](#) ábrán látható.



3.10. ábra. A kész piros háromszög

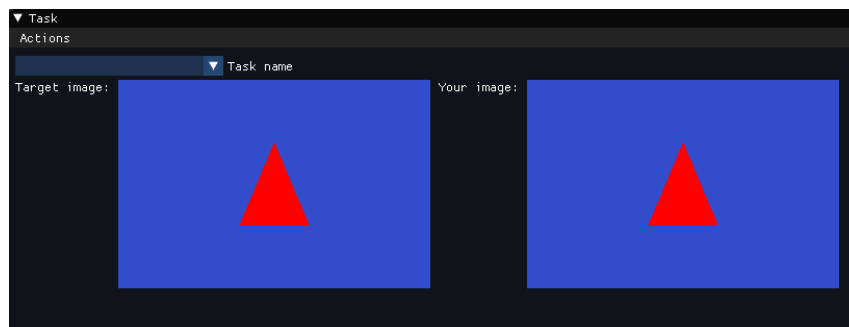
A kész objektumot pedig a [3.11](#) ábrán látható Task widgeten lévő "Actions" fül alatti "Export task" opcióval exportálhatjuk.



3.11. ábra. Feladat exportálása

### 3.2.2. Feladat betöltése

Töltsünk be az előbb exportált feladatot, a `Task` widgeten található "Actions" fül alatt navigáljunk a "Select task" opcióra, válasszuk ki a `RedTriangle.json` nevű fájlt, majd ugyanitt menjünk rá a "Change task" opcióra. Ezzel már sikeresen be is töltöttünk egy feladatot mint ahogy a 3.12 ábrán látható. Egy feladat betöltésekor létrejön egy objektum a megfelelő vertexekkel, indexekkel és textúrákkal, a felhasználónak csak a shader forráskódokat kell megírnia. Nekünk már ez kész van, így csak az eredményeket kell összehasonlítani.



3.12. ábra. Betöltött feladat

### 3.2.3. Kép összehasonlítás

Az "Actions" fül alatt válasszuk a "Compare images" opciót, majd az eredmény a 3.13 ábrán szemléltetett `Console` widgeten jelenik majd meg.

```
2021-04-13 13:44: Comparison result with SSIM: 99.9648% similarity
2021-04-13 13:44: Comparison result with chi square distance: 0 (Lower value is
2021-04-13 13:44: Successfully completed the task!
```

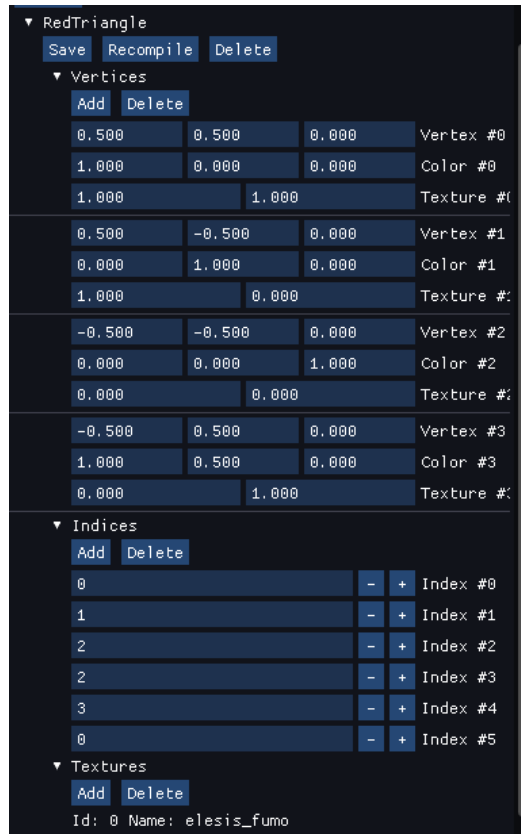
3.13. ábra. Összehasonlítás eredménye

A feladat akkor számít sikeresnek, ha az SSIM szerint a képek legalább hetven százalékban hasonlítanak vagy a CSD nyolc tizednél kisebb értéket ad vissza.

Módosítsunk a képünkön úgy, hogy az egy négyzet legyen, amire rárajzolunk egy textúrát. Először is vegyünk fel még egy vertexet, állítsuk be a textúra koordinátáknál a

negyedik vertexet és igazítsuk úgy az indexeket, hogy a negyedik vertexet is felhasználva egy négyzetet rajzoljunk. A beállított értékek a 3.14 ábrán látható.

A Scene editor widgeten a "Select" gombra kattintva választhatunk ki textúrát, amit a Textures fül alatt az "Add" gombbal adhatunk hozzá az objektumunkhoz.



3.14. ábra. Textúra koordináták beállítása

A vertex shaderben pedig adjuk értékül az `OurColor` és a `TexCoord` változóknak a `aColor` és a `aTexCoord` értékeit. Amik azok a koordináták lesznek amiket a `SceneEditor` widgetben adtunk meg. A kész vertex shader a [3.15](#) ábrán látható.

```
Resources/VertexShaders/RedTriangle_VerTEX.shader
1 #version 410 core
2
3 layout(location=0) in vec4 position;
4 layout(location=1) in vec3 aColor;
5 layout(location=2) in vec2 aTexCoord;
6
7 out vec3 OurColor;
8 out vec2 TexCoord;
9
10 uniform mat4 u_RedTriangleModel;
11 uniform mat4 u_RedTriangleView;
12 uniform mat4 u_RedTriangleProjection;
13
14 void main()
15 {
16     gl_Position = u_RedTriangleProjection * u_RedTriangleView *
17     * u_RedTriangleModel * position;
18     OurColor = aColor;
19     TexCoord = aTexCoord;
20 }
21
22
```

3.15. ábra. Textúra koordináták átadása a vertex shaderben

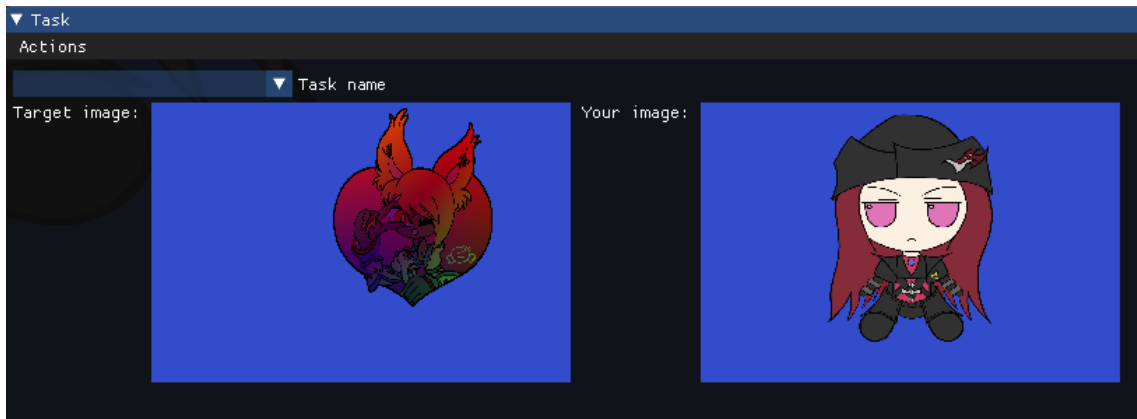
A fragmens shaderben pedig vegyünk fel egy `Sampler2D`-t uniformot a textúránknak, aminek a neve legyen `"u_{ObjektumNév}Texture{Id}"`, itt az `"ObjektumNév"` a shaderhez tartozó objektum neve az `"Id"` pedig a textúra azonosítója. A samplert állítsuk be textúrának. Ezt a texture függvénnyel tehetjük meg, ami paraméterül vár egy samplert és a textúra koordinátákat, melyeket az `in` változókból kapunk meg a vertex shaderen keresztül. A már textúrát rajzoló fragmens shadert a [3.16](#) ábrán láthatjuk.

```
Resources/FragmentShaders/RedTriangle_Fragment.shader
1 #version 410 core
2
3 uniform float u_RedTriangleTime;
4 uniform int u_RedTriangleWidth;
5 uniform int u_RedTriangleHeight;
6
7 uniform sampler2D u_RedTriangleTexture0;
8
9 in vec2 TexCoord;
10 in vec3 OurColor;
11
12 void main()
13 {
14     gl_FragColor = texture(u_RedTriangleTexture0, TexCoord);
15 }
16
```

3.16. ábra. Textúra beállítása fragmens shaderben

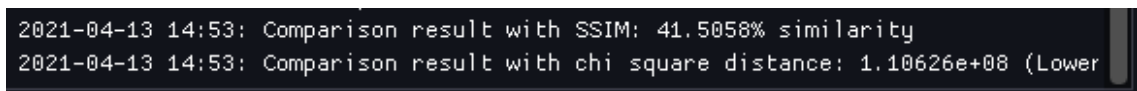
Mentsük el a shader forráskódokat és az attribútomkat, majd a `"Recompile"` gommbal fordítsuk újra az objektumot. Szintén töltsünk be egy másik feladatot is.





3.17. ábra. Két különböző kép összehasonlítása

A 3.17 ábrán láthatóan a kettő kép különböző, nézzük meg, hogy a képösszehasonlító funkció szerint is így van-e.



3.18. ábra. Különböző képek összehasonlításának eredménye

A 3.18 ábrán látjuk, hogy a két kép egyik metrika szerint sem egyezik, így mondhatjuk azt, hogy a képösszehasonlítás funkció megfelelően működik.

# Összefoglalás

## 3.3. Elért eredmények

A szakdolgozat írása során rengeteg új tudásra tettem szert. Megismerkedtem a CMAKE build rendszerrel és a C++ libraryk-el. Mélyebb belátást szereztem a C++ fordítók pre-processorának és linkerének működésébe. A C++ programozási készségeim bővültek a lambda függvények, templatek, fájlkezelés, memóriakezelés, debugging és más egyéb std könyvtár által biztosított funkciók ismerettségével. Implementáltam egy egyszerű grafikus motort az OpenGL grafikus API-n keresztül és shader kódokat írtam a GLSL shader nyelven. Megtanultam az ImGui felhasználói felület használatát és belátást szereztem a képfeldolgozás világába az SSIM és a CSD algoritmusok implementálásán keresztül.

Sikerült egy C++ alkalmazást készítenem ami működik Windows és Linux rendszereken is, alacsony a gépigénye, kevés erőforrást használ, a felhasználói felület pedig átlátható és reszponzív. Az alkalmazás indítása után pedig szinte rögtön futtatható shader kód.

Az alkalmazás minden része nyílt forráskódú és akárki számára elérhető, így bárki tovább fejlesztheti aki akarja vagy felhasználhatja saját céljaira.

## 3.4. Továbbfejlesztési lehetőségek

Utólag visszatekintve rengeteg mindent máshogy csinálnék és nem mindent sikerült a legszebben megvalósítanom. Ezért az alkalmazás egyes részeit célszerű lenne újratervezni és írni.

Az elvárt funkciókat mind sikeresen implementáltam, de rengeteg új funkcióval lehet-

ne felruházni az alkalmazást. Köztük modellek betöltése, VR támogatás, három dimenziós kép összehasonlítás, számítási shaderek támogatása, Vulkan API implementálása, feladatmegosztó webalkalmazás vagy akár egy teljes interaktív oktató jellegű játék steamworks integrációval.

# Irodalomjegyzék

- [1] A hivatalos OpenGL weboldal.  
<https://www.khronos.org/opengl/>
- [2] H. Sadeghi and A. Raie, "Approximated Chi-square distance for histogram matching in facial image analysis: Face and expression recognition," 2017 10th Iranian Conference on Machine Vision and Image Processing (MVIP), Isfahan, Iran, 2017, pp. 188-191, doi: 10.1109/IranianMVIP.2017.8342346.  
<https://ieeexplore.ieee.org/document/8342346>
- [3] Z. Wang, E. P. Simoncelli and A. C. Bovik, "Multiscale structural similarity for image quality assessment," The Thrity-Seventh Asilomar Conference on Signals, Systems & Computers, 2003, Pacific Grove, CA, USA, 2003, pp. 1398-1402 Vol.2, doi: 10.1109/ACSSC.2003.1292216.  
<https://ieeexplore.ieee.org/document/1292216>
- [4] Az OpenGL grafikus csővezeték  
<https://www.researchgate.net/profile/Christoph-Guetter/publication/235696712/figure/fig1/AS:299742132228097@1448475501091/The-graphics-pipeline-in-OpenGL-consists-of-these-5-steps-in-the-new-generation-of-cards.png>
- [5] Az RGBA komponens  
<https://olcovers2.blob.core.windows.net/coverswp/2016/11/rgba-pixel-model.png>

[6] Vertex attribútumok elrendezése a memóriában

[https://learnopengl.com/img/getting-started/vertex\\_attribute\\_pointer\\_interleaved\\_textures.png](https://learnopengl.com/img/getting-started/vertex_attribute_pointer_interleaved_textures.png)

[7] Joey De Vries. Learn OpenGL tutorials

<https://learnopengl.com>