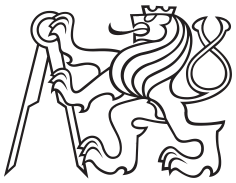
Bakalářská práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačové grafiky a interakce

Porovnání videofiltrů v GLSL a 2D Canvasu

Radek Pleticha

Školitel: RNDr. Ondřej Žára
Květen 2023

Poděkování

Chtěl bych poděkovat svému vedoucímu práce, RNDr. Ondřeji Žárovi, za jeho trpělivost, čas, dobré rady a zpětnou vazbu. Také bych chtěl poděkovat svým rodičům za jejich velkou podporu a trpělivost.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 7. 5. 2023

Abstrakt

Cílem práce je vytvořit JavaScriptovou knihovnu s použitím WebGL. Pomocí této knihovny bude možné provádět úpravy nad dvourozměrnými texturami pomocí jednoho či více shaderových programů v sérii. Tato knihovna bude testována na vzorové aplikaci, ve které se budou filtry, implementované pomocí shaderů, aplikovat na snímky videa v reálném čase. Posledně se bude měřit výkon aplikace s ohledem na snímky za sekundu upraveného videa a na časech strávených v shaderech. Tyto metriky budou měřeny v následujících kontextech: desktop vs. mobilní zařízení, integrovaná vs. dedikovaná grafická karta, v závislosti na rozlišení videa, v porovnání s čistě JavaScriptovou implementací.

Klíčová slova: WebGL, zpracování obrazu, GLSL, webová aplikace

Školitel: RNDr. Ondřej Žára

Abstract

The aim of this work is to create JavaScript library which uses WebGL. With this library it will be possible to edit a two dimensional texture using on or more shader programs in series. This library will be tested on demo application in which filters, implemented using shaders, will be used edit frames of a video in real time. Lastly, performance will be measured on this application in regards to frames per second and time spent in each filter. Those metrics will be measured in following contexts: desktop vs. mobile device, integrated vs. dedicated graphics card, depending on video resolution, in comparison to pure JavaScript implementation.

Keywords: WebGL, image processing, GLSL, web application

Title translation: Comparison of video filters in GLSL and 2D Canvas

Obsah

Obrázky

Tabulky

Kapitola 1
Úvod

Zpracovávání obrazu je důležité pro mnoho věcí. V robotice je možné použít obraz pro navigaci, pro řízení dopravy lze detekovat vozidla a měřit jejich rychlost, v lékařství se běžně využívá například pro rentgenové snímky a magnetickou rezonanci. Většina lidí běžně avšak používá zpracování obrazu pro méně podstatné záležitosti, jako jsou například úpravy fotografií nebo videí. Většina programů využívajících web kamery zařízení většinou mají implementované několik filtrů které upravují obraz získaný od zařízení. Otázka ale zní, jak by si jeden mohl vytvořit vlastní webovou aplikaci pro takové úpravy?

Pro tyto účely lze využít knihoven, které již implementují nějaké metody pro zpracování obrazu. Největší z nich je OpenCV, open-source knihovna stavěná na C++, která je dostupná i pro Python, Javascript (pomocí WebAssembly) a Javu. Tato knihovna je ve vývoji od roku 1999 a zaměřuje se obecně na počítačové vidění [1]. Další knihovna, podstatně menší, která také používá WebAssembly, je Web-DSP. Ta implementuje několik základních filtrů pro pole pixelů z videa či obrázku [2]. Poslední knihovna, která bude zmíněna, je Lena.js. Jedná se o malou knihovnu zaměřující se na úpravu obrázků. Je psána v čistém JavaScriptu a pracuje s pixely obrázku pomocí ImageData API [3].

Velkou výhodou OpenCV je, že je psaná v C++ a využívá OpenCL pro paralelní výpočty, čímž silně zrychluje zpracování obrazu. Problém avšak je, že pro přidání funkcí vlastních, by se od JavaScript vývojáře vyžadovala znalost nejen C++, ale i OpenCL. Web-DSP je také psané v C++. Funkce psané v C++ jsou ale jednodušší pro úpravu, protože nepoužívají paralelních výpočtů, což má ale negativní dopad na výkon. Dále v případě OpenCV i Web-DPS je nutné C++ kód po úpravách zkompilovat pomocí WebAssembly a to představuje další bariéru pro implementaci vlastních filtrů. Pro JavaScript vývojáře by se tedy zdálo, že nejpřístupnější bude Lena.js. Ta bohužel nemá podporu pro videa, a i kdyby měla, tak s čistým JavaScriptem by zpracování obrazu trvalo příliš dlouho pro úpravy v reálném čase.

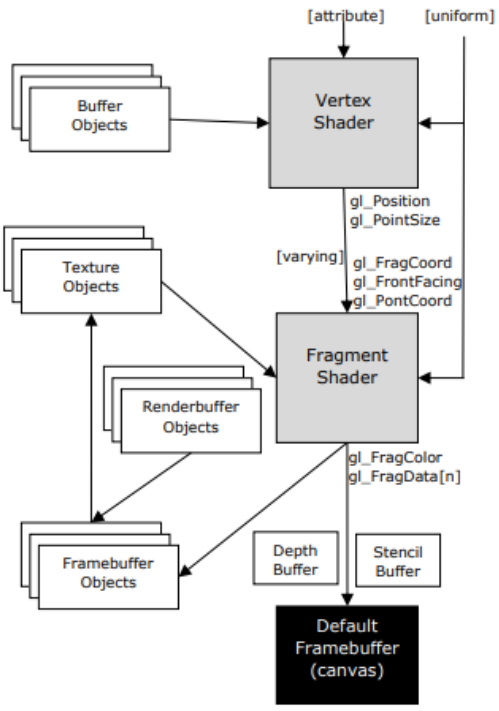
Cílem této práce je vytvořit vlastní JavaScriptovou knihovnu, která bude

používat knihovnu WebGL pro zrychlení výpočtů pomocí paralelizace na grafické kartě. Od této knihovny je požadováno, aby byla schopna aplikovat filtry na dvourozměrné textury (snímek videa nebo obrázek) pomocí shaderů. Těchto filtrů by mělo být možné aplikovat více v sérii a v definovaném pořadí. Funkce této knihovny by měla být demonstrována na vzorové aplikaci, která bude aplikovat jeden či více filtrů na záznam z web kamery v reálném čase. Dále bude implementovat alespoň tyto tři filtry: zesvětlení, zaostření a klíčování. Na této aplikaci se poté bude měřit počet snímků za sekundu videa upraveného těmito filtry a samotné časy strávené v jednotlivých filtrech.

Kapitola 2
Analýza

2.1 WebGL

WebGL je multiplatformní aplikační rozhraní pro vykreslování trojrozměrné grafiky určené pro web. WebGL je založen na OpenGL ES 2.0 a poskytuje podobné funkce pro vykreslování v HTML. WebGL je navržen jako kontext pro vykreslování do HTML Canvas elementu [9]. HTML Canvas je běžně schopen vykreslovat pouze dvourozměrnou grafiku a to pomocí kontextů CanvasRenderingContext2D a ImageBitmapRenderingContext. WebGL tedy dále rozšiřuje použití Canvasu pro trojrozměrnou grafiku. WebGL je vyvíjen a udržován neziskovým konsorciem Khronos Group. Aktuálně jsou k dispozici dvě verze WebGL, WebGL a WebGL2. V této sekci se tedy podíváme nejen, jak se obecně pracuje s WebGL, ale také na hlavní rozdíly mezi jejich verzemi.



Obrázek 2.1: Graphics pipeline, získáno z [10]

2.1.1 Úvod do WebGL

Pro použití WebGL je potřeba porozumění grafické pipeline a jejích komponent, viz. obr. 2.1.

Buffer Objects Jedná se pole binárních dat, které se posílají na grafickou kartu. Běžně obsahují informace jako pozice vrcholů, normály a texturové souřadnice geometrie. Pomocí atributů se specifikuje, jakým způsobem se mají data číst a kam je poté předat ve vertex shaderu [5].

Vertex shader Program psaný v jazyku GLSL, který má syntaxi podobnou C/C++. Vertex shader se volá pro každý vrchol geometrie a jeho hlavním úkolem je převést pozici vrcholu do ořezávacího prostoru a připravit data pro fragment shader.

Rasterization Jedná se o proces mezi vertex a fragment shaderem. Vstupem tohoto procesu jsou jednotlivé trojúhelníky geometrie. Jeho úkolem je převést tyto trojúhelníky na pixely, vyřadit ty, co jsou mimo ořezávací prostor a interpolovat data mezi vertex a fragment shadery.

Fragment shader Program psaný v jazyku GLSL. Jeho úkolem je vypočítat výslednou barvu pixelu.

Shader program Vertex a fragment shadery dohromady dávají shader program. Ten se přiřazuje v JavaScript kódu pro vykreslování.

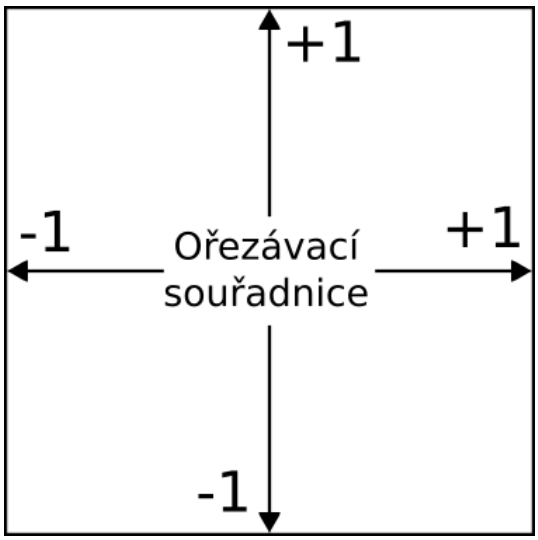
Texture Textury jsou pole dat, ke kterým lze přistoupit v shader programu. Nejčastěji se jedná o data obrázků, lze do nich ale dát i jiná data jako například informace o materiálu geometrie.

Framebuffer Pole dat do kterých se zapisuje výsledek vykreslování. Pokud není žádný framebuffer specifikován před vykreslováním, tak WebGL použije canvas. Data z framebufferu lze využít jako texturu při vykreslování.

2.1.2 WebGL pro 2D aplikace

WebGL samo o sobě pracuje s trojrozměrnou geometrií. Tato sekce je zaměřena na to, jak lze využít WebGL pro vykreslení dvourozměrné grafiky a jak pomocí textur získat obraz v JavaScriptu a použít ho pro vykreslování.

Geometrie. K vykreslení dvourozměrných objektů je třeba pochopit co se vykresluje ve WebGL. Do canvasu se vykreslí vše, co je obsaženo v ořezávacím prostoru. To je prostor v rozsahu $[-1, +1]^3$, kde kamera hledí ve směru osy z, viz. obr. 2.2. Pro vykreslování dvourozměrných objektů tedy stačí vytvořit jednoduché plochy, které po zpracování ve vertex shaderu budou kolmé na osu z a umístěny v ořezávacím prostoru. V rámci tohoto projektu postačí jeden obdélník s levým dolním rohem v $(-1, -1, 0)$ a pravým horním rohem v $(1, 1, 0)$. Tento obdélník je kolmý na osu z a vyplní prostor v osách x a y.



Obrázek 2.2: Ořezávací prostor

Textury. Pro vykreslení obrazu na geometrii se používají textury. Textury lze vytvořit z několika HTML elementů a JavaScriptových objektů. Pro vytvoření textury lze využít polí dat různých typů, jako např. `Byte`, `Int` a `Float`. WebGL má také implementovaný převod na textury z javascriptových objeků ImageData a ImageBitmap, a také z HTML prvků HTMLImageElement, HTMLCanvasElement a HTMLVideoElement. Ve WebGL2 jsou specifikovány čtyři druhy textur: 2D, cube map, 3D, 2D array. Textury mají také mnoho formátů dat jako např. RGB, RGBA a luminance. V této práci se pracuje se snímky videa a obrázky, popis práce s texturami bude tedy zaměřen na dvourozměrné textury ve formátu RGBA.

Základy práce s texturami. Pro čtení dat z textur se používají texturové souřadnice, ty jsou na intervalu $[0, 1]^2$ a jednotlivé složky se ve WebGL značí s, t (další časté značení je u, v). Texturové souřadnice definují, odkud z textury se má číst hodnota pixelu. Souřadnice $(0, 0)$ odpovídá levému spodnímu rohu a $(1, 1)$ pravému hornímu rohu textury. V případě výše definované geometrie by měl tedy levý dolní roh texturové souřadnice $(0, 0)$ a pravý horní roh $(1, 1)$.

Čtení z textur taky ovlivňují parametry *Wrap* a *Mag/Min filter*.

Wrap Definuje, co se má dít při pokusu o čtení mimo interval $[0, 1]^2$. Při nastavení na *clamp* se bude číst z okraje textury. Při nastavení na *repeat*, se bude textura opakovat. Tyto hodnoty lze nastavit zvlášť pro obě souřadnice. V této práci se používá konvoluce, takže jsou obě souřadnice nastaveny na *clamp*.

Mag/Min filter Tvoří větší a menší kopie textury, kterým se říká mipmapy. Ty se používají pro zmírnění aliasing efektu, který může vzniknout, když je objekt příliš blízko či daleko od kamery. Tento parametr lze nastavit na interpolování mezi dvěma nejbližšími velikostmi textury nebo na použití

hodnoty nejbližší textury. V této práci je vzdálenost geometrie od kamery konstantní, používá se tedy jenom jedna velikost textury.

Tvorba a použití textur. Vytvoření textury a její použití se skládá z následujících kroků: tvorba objektu textury, připojení objektu textury, nastavení parametrů textury, nastavení zdroje dat a způsobu čtení, nastavení lokace textury v shaderu. V ústřižku kódu (2.1) lze vidět celý tento proces.

```
// Tvorba objektu textury
const texture = gl.createTexture();
// Pripojeni objektu textury
gl.bindTexture(gl.TEXTURE_2D, texture);
// Nastaveni parametru wrap a mag/min filtru
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S,
↪  gl.CLAMP_TO_EDGE);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T,
↪  gl.CLAMP_TO_EDGE);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
↪  gl.NEAREST);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
↪  gl.NEAREST);
// Nastaveni dat pro cteni a jejich formatu
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, width, height, border,
↪  gl.RGBA, gl.UNSIGNED_BYTE, image);
// Nastaveni aktivniho shader programu a lokace textury
gl.useProgram(program)
gl.uniform1i(texLocation, 0);
// Vykresleni
gl.drawArrays(gl.TRIANGLES, 0, nIndices);
```

Listing 2.1: Příklad použití textury

2.1.3 Shadery

Cílem této sekce je seznámí s shadery. Co jsou shadery, jejich použití, jak se píší a jak se do nich posílají data.

Shader a jeho použití. Shader je obecně program, který se vykonává na grafické kartě. Ve WebGL jsou dva druhy shaderů, vertex a fragment. Dohromady je jejich úkolem vykreslit geometrii do framebufferu (canvasu).

Psaní shaderů. Shadery se píší v jazyce GLSL. GLSL je syntakticky podobný C/C++ s několika rozdíly. Stejně jako C/C++ obsahuje tradiční typy jako např. `int` a `float`, ale také rozšiřuje o typy jako vektory (např. `ivec2`, `vec3`) a matice (např. `mat3`, `mat2x4`). GLSL má také implementované užitečné matematické funkce jako například lineární interpolaci `mix`, odraz světla `reflect` a lom světla `refract`.

Vertex shader. Vertex shader má za úkol vypočítat ořezávací souřadnice vrcholu, ty se ukládají do proměnné `vec4 gl_Position`. Vertex shader se volá jednou pro každý vrchol. Pro vypočítání souřadnic jsou potřeba dodatečné informace, ty lze poslat do vertex shaderu třemi způsoby.

Attribute. Atributy jsou data z bufferů (Buffer Objects). Tímto způsobem je možné každému vrcholu přiřadit specifická data. Do atributů se dávají informace jako například pozice vrcholu a barva vrcholu. Použití atributů probíhá následovně: tvorba buffer objektu, připojení buffer objektu, přiřazení dat z pole, nastavení způsobu čtení a lokace atributu v shaderu. Ve WebGL2 atributy používají klíčové slovo `in`.

Uniform. Uniformy jsou v podstatě konstanty, jsou stejné pro každý vrchol. Běžně se používají například pro transformační matici geometrie, materiál geometrie a zdroje osvětlení. Pro nastavení uniformu je třeba znát jeho lokaci a nahrát na ní hodnotu příslušného typu.

Texture. Vertex shadery jsou schopny přistupovat pomocí ST souřadnic k datům z textur, detailnější popis je v sekci 2.1.2

Fragment shader. Fragment shader má za úkol vypočítat výslednou barvu pixelu. Ta se v případě WebGL ukládá do `vec4 gl_FragColor`, ve WebGL2 lze použít jakkoliv pojmenovanou proměnnou typu `out vec4`. Fragment shader se volá jednou pro každý fragment (pixel) geometrie. Pro vypočítání barvy pixelu je často potřeba více informací, ty lze předat do fragment shaderu třemi způsoby.

Uniform. Stejně jako uniformy ve vertex shaderu měli stejnou hodnotu pro každý vrchol, tak ve fragment shaderu mají stejnou hodnotu pro každý pixel.

Texture. Fragment shadery jsou schopny přistupovat pomocí interpolovaných ST souřadnic z vertex shaderu k datům z textur, detailnější popis je v sekci 2.1.2.

Varying. Varying slouží k posílání dat z vertex shaderu. K tomu je potřeba mít proměnnou se stejným typem a jménem ve vertex i ve fragment shaderu s klíčovým slovem `varying`. Ve WebGL2 se místo varying používá `out` (vertex shader) a `in` (fragment shader). Ve varying se běžně posílají např. texturové souřadnice, souřadnice geometrie nebo barva fragmentu.

Tvorba shader programu. Pro tvorbu shader programu jsou třeba kódy vertex a fragment shaderů, ty budou značeny dále jako `vertSource` a `fragSource`. Kompilace se skládá ze čtyř částí: vytvoření objektů vertex a fragment shaderů, kompilace vertex a fragment shaderů, připojení vertex a fragment shaderů a jejich spojení do shader programu. Tyto kroky jsou vidět v tomto ústřižku kódu (2.2).

```
// Vytvoreni vertex shader objektu
const vertShader = gl.createShader(gl.VERTEX_SHADER);
```

```
// Pripojeni kodu vertex shaderu
gl.shaderSource(vertShader, vertSource);
// Kompilace kodu vertex shaderu
gl.compileShader(vertShader);
// Stejny proces, ale pro fragment shader
const fragShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragShader, fragSource);
gl.compileShader(fragShader);
// Vytvoreni program objektu
const program = gl.createProgram();
// Pripojeni vertex a fragment shader objektu
gl.attachShader(program, vertShader);
gl.attachShader(program, fragShader);
// Propojeni vertex a fragment shaderu
gl.linkProgram(program);
// 'program' ted lze pouzit pro vykreslovani
```

Listing 2.2: Vytvoření shader programu

2.1.4 Rozdíly mezi WebGL a WebGL2

Hlavní rozdíl mezi těmito verzemi je, že WebGL je postaven na OpenGL ES 2.0, zatímco WebGL2 je postaven na OpenGL ES 3.0. Díky tomu má WebGL2 mnoho nových funkcí. V této sekci jsou některé z nich ukázány.

Podpora Vertex Array Object (VAO). VAO je objekt, který je schopen držet více různých bufferů i s informacemi k jejich přístupu.[6]

Možnost získat rozměry textur z kódu. Ve WebGL bylo potřeba poslat rozměry textury manuálně v uniformu. Ve WebGL2 lze v shaderu zavolat `textureSize(sampler, lod)` a získáme `ivec2` obsahující výšku a šířku textury.[6]

Přímý přístup k texelům textury. Ve WebGL2 lze přistoupit k hodnotě pixelu textury přímo souřadnicemi pixelu. WebGL podporuje přístup pouze přes texturové souřadnice. Toto je více užitečné když se texture používá jako velké pole, spíše než obrázek.[6]

Více formátů textur. Rozšíření o více specifické formáty jako např. `RGBA32I`, `RGBA16UI`, `RGB8` a `R8`.

Pole textur. Ve WebGL se pro použití více textur musela napojovat každá zvlášť. Pole textur umožňuje napojit větší množství textur jednou funkcí a s jednoduchým přístupem z shaderů.

2.2 Specifikace projektu

Tato část je věnována pro návrh řešení rozhraní pro filtry. Hlavní požadavky pro toto rozhraní jsou: použití čistého JavaScriptu a WebGL, jednoduchost importu a použití, možnost použití více filtrů v definovaném pořadí, podpora HTML tagů `<video>` a `<image>`. Další požadavky pro aplikaci se týkají spíše filtrů, pro ně jsou následující požadavky: jednoduchost vytvoření a použití, možnost předávání parametrů od uživatele (např. ostrost a světlost) a umožnit uživateli použít vlastní filtry. Součást projektu je také implementace vzorové aplikace, kde bude ukázáno, jak se implementované rozhraní dá využít pro real-time filtrování z web kamery zařízení.

2.2.1 Rozhraní

Zastřešující třída rozhraní projektu je VideoContext. V této části bude popsáno jak by tato třída měla vypadat, aby odrážela výše uvedené požadavky.

Použití čistého Javascriptu a WebGL. Toto rozhraní nesmí používat žádné dodatečné knihovny jako např. jQuery nebo React. VideoContext tedy bude psán pouze v JavaScriptu a grafické záležitosti budou vykonávány ve WebGL.

Jednoduchý import. Import by neměl požadovat desítky `<script>` tagů v HTML. Pro import se použijí javaScriptové moduly. Jediné co bude třeba, je přidat uživatelský script jako modul a pomocí klíčového slova `import` z JavaScriptu přidat `VideoContext`.

Jednoduchost použití. Od uživatele by neměla být očekávána znalost WebGL. VideoContext by měl tedy dělat potřebné WebGL záležitosti za uživatele.

Možnost použití více filtrů. VideoContext musí být schopen přijmout několik filtrů pro sériové vykonání. Pro definici shaderů k vykonání a jejich pořadí bude použité pole `queue`, do kterého se vloží filtry k vykonání a o zbytek se postará VideoContext.

Podpora různých HTML tagů. Uživatel by měl mít možnost použít jak video, tak i fotky a obrázky pro filtrování. Je tedy potřeba aby VideoContext podporoval použití obou forem. VideoContext tedy musí implementovat funkce jako např. `drawFromImage(image) a drawFromVideo(video)`.

2.2.2 Filtry

Filtry budou implementovány jako zastřešení WebGL shaderů. To přináší ale určité problémy. Pro implementaci vlastních filtrů je potřeba základní znalost WebGL (kompilace, posílání dat) a GLSL (implementace shaderů), takže je potřeba udělat základní třídu, která zajistí geometrii a kompilaci shaderů.

Na té bude pak možno postavit konkrétní filtry s minimální znalostí WebGL, shadery avšak stále vyžadují znalost GLSL.

Jednoduchost vytvoření a použití. VideoContextem implementované shadery musí být jednoduše vytvořitelné. Proto by měl VideoContext mít funkce, které vytvoří příslušný filtr. Pro použití filtru ho bude stačit vložit do pole `queue` VideoContextu.

Předávání parametrů. Předávání parametrů filtrům by mělo být možné bez WebGL volání ze strany uživatele. K tomu budou sloužit proměnné definované v jednotlivých filtr třídách.

Možnost uživatelské implementace filtrů. Jak už bylo výše nastíněno, uživatelské filtry jsou problematické. VideoContextem implementované filtry dělají dobrou šablonu, která se dá zkopírovat a použít pro tvorbu filtrů vlastních. VideoContext má také možnost vytvořit shadery pouze pomocí GLSL kódu, bez implementace celé třídy. Takový filtr, ale nedokáže přijímat parametry.

Kapitola 3
Implementace

Tato kapitola se zaměřuje na vlastní implementaci rozhraní. Každá sekce bude věnována popisu jednotlivých komponent. Každá komponenta bude mít popis jak jí použít, jaké metody byly použity při tvorbě a jejich celkové struktury.

3.1 VideoContext

VideoContext je zastřešující třída rozhraní. Tato třída má za úkol řídit vykreslování a fungovat se jako továrna pro filtry. Od této třídy je vyžadováno, aby si předělala `<video>` a `<image>` HTML tagy na textury a vykreslila je s aplikovanými filtry získanými z pole `queue` do canvasu.

3.1.1 Import a použití

V této části se podíváme na VideoContext čistě z pohledu uživatele. Jak ho importovat, jak si vyžádat filtry a jak vykreslit do canvasu.

Import. Pro import jsou potřeba tři věci: uživatelský HTML `<script>` tag jako modul, HTML `<div>`, který bude sloužit jako lokace pro canvas, přidání VideoContext třídy v JavaScript modulu pomocí `import`. Ústřižek HTML kódu (3.1) ukazuje jednoduchý příklad importu.

```
<!-- Div kontejner pro canvas -->
<div class="canvas-container" id="canvas-container"></div>
<!-- Skript jako modul (na konci HTML) -->
<script type="module">
        // Import VideoContextu
        import VideoContext from "./VideoContextAPI/videocontext.js";
</script>
```

Listing 3.1: Script uživatele: filters.js

Použití. Po importu VideoContextu s ním lze začít pracovat. Workflow s VideoContextem vypadá následovně: tvorba instance VideoContext, vložení VideoContextem vytvořeného canvasu do `<div>` kontejneru pro canvas, získání filtrů, založení pole filtrů a zavolání vykreslovací metody. Tyto kroky lze vidět v ústřižku kódu (3.2).

```
// Tvorba VideoContex instance
const videocontext = new VideoContext();
// Pridani canvasu z VideoContextu do kontejneru
canvasContainer.appendChild(videocontext.canvas);
// Tvorba filtru (zesvetleni)
const brightnessFilter = videocontext.createBrightnessFilter();
// Vytvoreni fronty filtru (pole), ktera se pak vlozi do
↪  videocontextu
var queue = [brightnessFilter];
videocontext.queue = queue;
// Volani vykreslovani (video je <video> tag ziskany z html)
videocontext.drawFromVideo(video);
// Pro kresleni z <image> tagu lze pouzit:
// videocontext.drawFromImage(image);
```

Listing 3.2: Příklad použití VideoContextu

3.1.2 Funkce a proměnné

Tato část je věnována popisům implementovaných veřejných funkcí a proměnných ve VideoContextu.

Funkce.

`constructor()` Vytvoří canvas, framebuffery a textury potřebné pro vykreslování, také inicializuje jejich velikosti.

`drawFromTexture(texture, width, height)` Vykreslování z textury. Pro kreslení z textury je bohužel potřeba šířka a výška, protože z JavaScriptu není tato informace dostupná.

`drawFromVideo(video)` Vykreslení z videa. Získá texturu, šířku i výšku z `video` parametru.

`drawFromImage(image)` Vykreslení z obrázku. Získá texturu, šířku i výšku z `image` parametru.

`create`*[filter-name]*`Filter()` Skupina funkcí pro tvorbu filtrů. Za *[filter-name]* lze napsat jeden z těchto filtrů: `Identity`, `Sharpen`, `EdgeSharpen`, `Brightness`, `Chroma`.

`createFilterFromSource(vertex, fragment)` Vytvoří základní filtr pomocí GLSL kódů pro vertex a fragment shadery, předaných jako `string` v parametrech.

`getMaxTextureSize()` Vrátí hodnotu maximálního rozměru textur `MAX`. Pro dané zařízení je možno zadat 2D texturu o maximální velikosti `MAX` x `MAX` (v případě dostatečné paměti).

Proměnné.

`canvas` Canvas na který bude VideoContext vykreslovat.

`gl` GL je WebGL2 kontext canvasu.

`queue` Fronta shaderů, která se bude vykreslovat po zavolání vykreslovací funkce.

3.1.3 Proces vykreslování

Vykreslování je vykonáváno postupným aplikováním filtrů z fronty (`queue`). Samotný proces se ale liší podle množství aplikovaných filtrů. Uživatel nemusí zadat žádný filtr k vykreslení, pak VideoContext vykreslí obraz přijat z příslušné vykreslovací funkce. Pokud zadá jeden filtr, tak s ním stačí vykreslit přímo do canvasu. Problém avšak nastane při vykreslování více filtrů, protože pokud vykreslíme do canvasu s jedním filtrem a chtěli bychom zároveň číst a psát do canvasu, tak není jisté, zda čteme nová či stará data. V této sekci bude ukázáno jak tento problém vyřešit.

Struktury pro vykreslování. Pro vykreslování z HTML tagů jsou potřeba textury. Dále se používají framebuffery, to jsou zásobníky dat, do kterých může WebGL vykreslovat. Ty na sebe mají vázané textury, které lze použít pro čtení. K vytvoření framebufferu je potřeba: vytvořit texturu, vytvořit framebuffer objekt, připojit framebuffer objekt, připojit texturu k framebufferu.

Ústřižek kódu (3.3) ukazuje, jak se tvoří framebuffer s jeho texturou. Pro vytvoření textury je potřeba zadat velikost textury. Bohužel jeden z problémů VideoContextu je, že velikost obrazu není známá před zavoláním vykreslování uživatelem. Velikosti `canvas`, `framebuffer` a `texture` jsou tedy inicializovány na 1x1 px. Po zavolání vykreslování se reinicializuje velikost těchto objektů na velikost přijatého obrazu. Pokud se velikost obrazu nezmění mezi dvěma vykresleními, tak se velikosti nebudou reinicializovat.

```
// Tvorba textury
const texture = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, texture);
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, width, height, 0,
↪  gl.RGBA, gl.UNSIGNED_BYTE, null);
// Nastaveni parametru textury
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S,
↪  gl.CLAMP_TO_EDGE);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T,
↪  gl.CLAMP_TO_EDGE);
```

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
↪  gl.NEAREST);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
↪  gl.NEAREST);
// Tvorba framebufferu
const fb = gl.createFramebuffer();
// Pripojeni framebufferu
gl.bindFramebuffer(gl.FRAMEBUFFER, fb);
// Pripojeni textury k framebufferu
gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0,
↪  gl.TEXTURE_2D, texture, 0);
```

Listing 3.3: Tvorba framebufferu

Získávání textur. Při získávání textur záleží, co je jejím zdrojem. VideoContext implementuje tři zdroje: texturu, `<video>` tag a `<image>` tag. Velikost textury není možné zjistit z WebGL, musíme tedy požadovat zadání velikosti od uživatele. Rozměry `video` tagu lze zjistit pomocí `video.videoHeight` a `video.videoWidth`. Pro velikost `image` tagu lze použít `image.height` a `image.width`. Proto jsou použity tři různé funkce pro vykreslování. V kódu (3.4) je ukázka získání dat z videa pro předem vytvořenou texturu [4].

```
// Ziskani rozmeru videa
var width = video.videoWidth;
var height = video.videoHeight;
// Funkce pro reinicializaci velikosti objektu pro vykreslovani
this.setSize(width, height);
// Pripojeni a ziskani dat z videa (video)
this.gl.bindTexture(this.gl.TEXTURE_2D, this.texture);
this.gl.texImage2D(this.gl.TEXTURE_2D, 0, this.gl.RGBA,
↪  this.gl.RGBA, this.gl.UNSIGNED_BYTE, video);
// Pak je mozne zacit vykreslovani z this.texture
```

Listing 3.4: Získání textury z videa

Vykreslování jednoho filtru. Pokud uživatel nezadá žádný filtr před vykreslením, tak bude použit `identity` filtr, ten vykreslí nezměněný obraz od uživatele. Postup vykreslení jednoho filtru se skládá z těchto kroků: získání filtru, připojení canvasu jako framebufferu, připojení textury od uživatele, nastavení parametrů shaderů filtru, zavolání vykreslení geometrie. Tento postup je vidět v ústřižku kódu (3.5)

```
1 // Definice filteru podle poctu zadanych filtru
2 var filter = this.queue.lenght == 0 ? this.identity :
  ↪  this.queue[0];
3 // Nastaveni pripojeneho framebufferu na null (vykresluje do
  ↪  canvas)
```

```
4  this.gl.bindFramebuffer(this.gl.FRAMEBUFFER, null);
5  // Pripojeni textury ziskane drive od uzivatele
6  this.gl.activeTexture(this.gl.TEXTURE0);
7  this.gl.bindTexture(this.gl.TEXTURE_2D, texture);
8  // Funkce filtru slozici k nastaveni uniformu a atributu jeho
   ↪  shaderu
9  filter.setAtribsAndUniforms();
10 // Vycisteni canvasu
11 this.gl.clear(this.gl.COLOR_BUFFER_BIT);
12 // Nastaveni velikosti viewportu (canvasu) pro WebGL
13 this.gl.viewport(0, 0, this.gl.canvas.width,
   ↪  this.gl.canvas.height);
14 // Nastaveni shaderu pro vykreslovani
15 this.gl.useProgram(filter.program);
16 // Vykresleni geometrie
17 this.gl.drawArrays(this.gl.TRIANGLES, 0, 6);
18 // Odpojeni programu
19 this.gl.useProgram(null);
```

Listing 3.5: Vykreslení jednoho filtru

Vykreslování více filtrů. Pro vykreslování více filtrů se používají dva framebuffery. Během vykreslování jednotlivých filtrů pak používáme střídavě jeden framebuffer jako zdroj textury, a druhý jako cíl vykreslování. Poslední filtr se pak vykresluje do canvasu [7]. V následujících ukázkách kódů budou framebuffery v poli `fbs[]` a jejich textury v poli `fbTexs[]`. V ukázce kódu (3.6) je vidět jak lze nastavit `fbs[0]` jako zdroj a `fbs[1]` jako canvas.

```
// Pripojeni framebufferu '1' vystup vykreslovani
this.gl.bindFramebuffer(this.gl.FRAMEBUFFER, this.fbs[1]);
// Pripojeni textury framebufferu '0' pro pouziti framebuffer
↪  '0' jako zdroje obrazu
this.gl.activeTexture(this.gl.TEXTURE0);
this.gl.bindTexture(this.gl.TEXTURE_2D, this.fbTexs[0]);
```

Listing 3.6: Nastavování framebufferů

Ústřižek (3.7) ukazuje, jak lze vykreslit z fronty pomocí střídání framebufferů. V tomto kódu bude vykreslení individuálního filtru nahrazeno funkcí `render(filter)`. Tato funkce obsahuje řádky `9 - 19` z ústřižku (3.5).

```
// Nejprve musime vykreslit z textury od uzivatele
// Pripojeni framebufferu '0' a uzivatelske textury
this.gl.bindFramebuffer(this.gl.FRAMEBUFFER, this.fbs[0]);
this.gl.activeTexture(this.gl.TEXTURE0);
this.gl.bindTexture(this.gl.TEXTURE_2D, userTexture]);
this.render(this.queue[0]);
```

```
// fbIdx je index framebufferu do ktereho budeme vykreslovat
var fbIdx = 1;
const nFilters = this.queue.length;
// Vykreslovat se bude od queue[1] do queue[nFilters - 2]
for(var i = 1; i < (nFilters - 1); i++) {
  // Vyber filtru
  var filter = this.queue[i];
  // Pripojeni framebufferu 'fbIdx' jako cil pro vykresleni
  this.gl.bindFramebuffer(this.gl.FRAMEBUFFER,
  ↪  this.fbs[fbIdx]);
  // Pripojeni textury framebufferu 'fbIdx - 1' jako zdroj
  ↪  obrazu
  this.gl.activeTexture(this.gl.TEXTURE0);
  this.gl.bindTexture(this.gl.TEXTURE_2D, this.fbTexs[1 -
  ↪  fbIdx]);
  // Funkce pro vykresleni obrazu
  this.render(filter);
  // Prohozeni framebufferu
  fbIdx = 1 - fbIdx;
}
// Vykresleni posledniho filtru do canvasu
this.gl.bindFramebuffer(this.gl.FRAMEBUFFER, null);
this.gl.bindTexture(this.gl.TEXTURE_2D, this.fbTexs[1-fbIdx]);
this.render(this.queue[nFilters - 1]);
```

Listing 3.7: Vykreslení z fronty pomocí střídání framebufferů

3.2 Filter

Tato třída slouží jako rozhraní pro shadery. Jejím úkolem je kompilace shaderu filtru, tvorba geometrie, přijímání parametrů od uživatele a posílání dat do shaderu. V této sekci bude popsána základní třída `Filter`, jak probíhá konstrukce (tvorba shader programů a inicializace geometrie) a jak si vytvořit vlastní filtr.

3.2.1 Popis třídy

Filter třída nemá příliš přesně definovaných proměnných a funkcí. Slouží spíše jako základ pro tvoření dalších filtrů, takže v této sekci bude ukázáno co, je potřeba ke splnění této funkce.

Funkce. Pro použití ve `VideoContext` musí `Filter` implementovat tyto dvě funkce: `constructor(gl)` a `setAttribsAndUniforms()`.

`constructor(gl, vert, frag)` Slouží k vytvoření instance. Má za úkol tvorbu shader programu, získání lokací parametrů, inicializaci geometrie

a parametrů. Parametry: `gl` je WebGL2 kontext `VideoContext` canvasu, `vert` a `frag` jsou vertex a fragment shader kódy jako `string` (vert a frag nejsou povinné).

setAttribsAndUniforms() Slouží k nastavení parametrů shaderu. Jediné co technicky vzato **musí** udělat, je nastavit index textury na `0`.

Proměnné. Proměnné, které jsou napsány `tímto fontem`, jsou povinné pro použití ve `VideoContext`. Ty, co jsou napsány *kurzívou*, jsou proměnné, které je možné dodatečně implementovat pro rozšíření funkce filtru.

`gl` Je WebGL2 kontext canvasu z `VideoContext`-u.

`program` Je zkompilovaný shader program.

`success` Slouží ke zjištění, jestli kompilace shaderu proběhla úspěšně.

Lokace parametrů Umístění parametrů v shaderech. Slouží jako adresa, na kterou budou poslány data pro shadery.

Data pro shadery Slouží k přijímání dat od uživatele. Ty se poté posílají na lokace parametrů do shaderu.

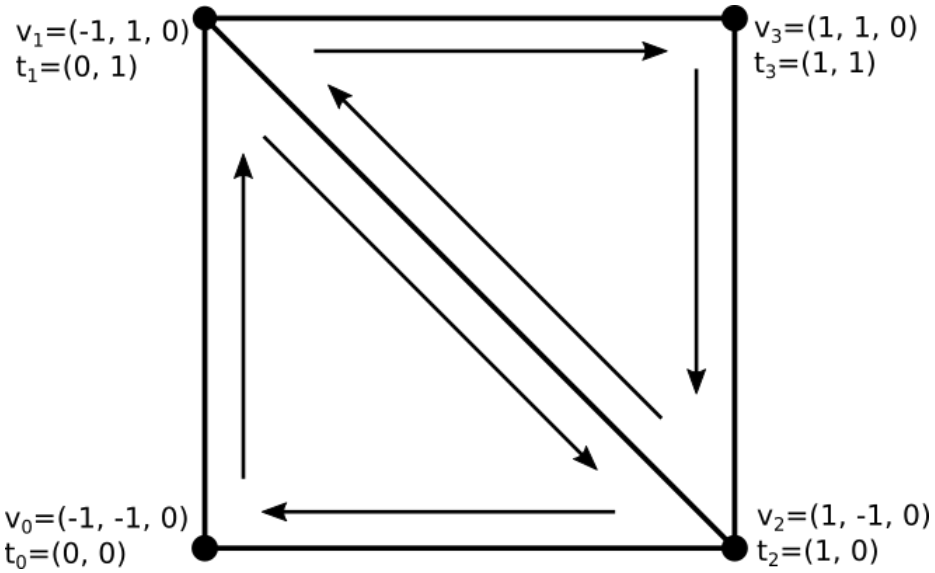
3.2.2 Konstrukce třídy

Tato část vysvětluje, co se děje během tvorby instance třídy. Pro zkrácení kódu bude použita pomocná třída `createProgram(gl, vert, frag)`, která provede samotnou tvorbu shader programu (viz. ústřižek kódu 2.2). Během tvorby instance se musí vytvořit shader program, inicializovat geometrie a její texturové souřadnice viz. kód (3.8). Geometrie se skládá z trojúhelníků. Z nich poskládáme čtverec s levým dolním rohem v souřadnicích $(-1, -1, 0)$ a pravým horním v $(1, 1, 0)$. Texturové souřadnice levého dolního rohu jsou $(0, 0)$ a pravého horního jsou $(1, 1)$. Výsledná geometrie se skládá ze čtyř vrcholů v_0, v_1, v_2, v_3 a ze čtyř texturových souřadnic t_0, t_1, t_2, t_3 viz. obr. (3.1). Pak jsou z nich složeny dva trojúhelníky, jeden s vrcholy v pořadí (v_0, v_1, v_2) a druhý s vrcholy (v_3, v_2, v_1). Pořadí je definováno tak, aby normála trojúhelníků směřovala ke kameře. Odvrácené trojúhelníky (s normálou směřující od kamery) jsou během vykreslování vyhozeny.

```
this.success = false;
this.program = createProgram(gl, vert, frag);
// Kontrola uspechu vytvoreni programu
if (this.program == null) return;
this.success = true;
// Pripojeni programu
gl.useProgram(this.program);
// Ziskani lokace pro souradnice vrcholu
const vertPosAttribLoc = gl.getAttribLocation(this.program,
↪  "position");
```

```
// Ziskani lokace pro texturove souradnice
const texCoordAttribLoc = gl.getAttribLocation(this.program,
↪  "tex_coord");
// Vytvoreni zasobniku pro souradnice vrcholu
const positionBuffer = gl.createBuffer();
// Pripojeni zasobniku
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
// Aktivace pole atributu
gl.enableVertexAttribArray(positionAttributeLocation);
// Definice zpusobu cteni ze zasobniku
gl.vertexAttribPointer(positionAttributeLocation, 2, gl.FLOAT,
↪  false, 0, 0);
// Definice geometrie (dat v zasobniku)
gl.bufferData( gl.ARRAY_BUFFER, new Float32Array([-1, -1, -1,
↪  1, 1, -1, 1,  1, 1, -1, -1,  1]), gl.STATIC_DRAW);
// Stejny proces, ale pro texturove souradnice
const texCoordsBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, texCoordsBuffer);
gl.enableVertexAttribArray(texCoordAttributeLocation);
gl.vertexAttribPointer(texCoordAttributeLocation, 2, gl.FLOAT,
↪  false, 0, 0);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array([0, 0, 0, 1, 1,
↪  0, 1, 1, 1, 0, 0, 1]), gl.STATIC_DRAW);
// Zde lze ziskat dalsi lokace atributu
// Odpojeni programu
gl.useProgram(null);
```

Listing 3.8: Tvorba Filter třídy



Obrázek 3.1: Geometrie použitá pro filtry

3.2.3 Jak vytvořit vlastní filtr

Pro vytvoření vlastního filtru lze využít základní třídy `Filter`, od které je možné dědit. Ve vlastní implementaci konstruktoru stačí zavolat `super(gl, vertexString, fragmentString)`, který vytvoří program s shadery definovanými kódy `vertexString` a `fragmentString`. Tyto shader kódy musí být napsány uživatelem v GLSL 3.0. Funkce `super` také vytvoří geometrii, takže shadery musí mít atributy: `in vec2 position` a `in vec2 tex_coord` a uniform `uniform sampler2D u_texture`. Zde (3.9) je implementace shaderu zesvětlení pomocí dědění. Kódy shaderů jsou získány mimo třídu z `vertexString` a `fragmentString`.

```
export default class BrightnessFilter extends Filter {
    // New locations
    brightnessFactorLoc;
    // New Data
    brightnessFactor = 1.0;

    constructor(gl) {
        super(gl, vertexString, fragmentString);
        // Ziskani lokace urovne zesvetleni
        this.brightnessFactorLoc =
        ↪  gl.getUniformLocation(this.program, "factor");
    }

    setAttribsAndUniforms() {
        this.gl.useProgram(this.program);
        // Posilani hodnot uniformum
        this.gl.uniform1i(this.textureLoc, this.texture);
        this.gl.uniform1f(this.brightnessFactorLoc,
        ↪  this.brightnessFactor);
        this.gl.useProgram(null);
    }
}
```

Listing 3.9: Implementace třídy filtru zesvětlení

3.3 Implementované filtry

Tato sekce bude zaměřena na popis implementovaných filtrů. VideoContext implementuje pět filtrů: identita, zesvětlení, zaostření přes detekci hran, zaostření pomocí unsharp a klíčování. Všechny tyto filtry používají stejný vertex shader. To je, protože není třeba nijak měnit geometrii ve vertex shaderu. Jediné co vertex shader dělá, je že nastavuje ořezávací souřadnice vrcholu (`gl_Position`) a texturové souřadnice vrcholu pro interpolaci (`v_tex_coord`), viz. kód (3.10). Veškeré filtry jsou tedy hlavně implementované ve fragment

shaderu, protože ty mají za úkol vrátit barvu pixelu. Funkce filtru bude popsána z pohledu jednotlivého pixelu. Tedy jaké operace se pro každý pixel musí vykonat. Všechny výsledky filtrů budou porovnávány na stejném obrázku (3.2).

```
#version 300 es
precision highp float;

in vec2 position;
in vec2 tex_coord;
out vec2 v_tex_coord;

void main() {
  gl_Position = vec4(position.x, position.y, 0, 1);
  v_tex_coord = vec2(tex_coord.x, tex_coord.y);
}
```

Listing 3.10: Kód vertex shaderu

Obrázek 3.2: Originální obrázek pro porovnání filtrů

3.3.1 Identita

Nejjednodušší filtr je identita. Jediné co se musí pro každý pixel udělat, je přidělit mu barvu z přijaté textury. Pro získání barvy pixelu z textury se používají texturové souřadnice z `in vec2 v_tex_coord` a textura přístupná z `uniform sampler2D u_texture`. Funkce pro získání barvy z textury ve WebGL2 je `texture(sampler2D texture, vec2 tex_coords)`. Celý kód fragment shaderu pak vypadá takto (3.11). Výsledek filtru bude stejný jako originální obrázek (3.2).

```
#version 300 es
precision highp float;

in vec2 v_tex_coord;
out vec4 outColor;

uniform sampler2D u_texture;

void main() {
    outColor = texture(u_texture, v_tex_coord);
}
```

Listing 3.11: Fragment shader filtru identity

3.3.2 Zesvětlení

Filtr zesvětlení zvětšuje jas přijatého obrazu. K tomu se použije parametr `float brightness`, který je přístupný ve třídě `BrightnessFilter`. Funkce `main()` fragment shaderu pak musí jen vynásobit barvu z textury faktorem zesvětlení. Ústřižek kódu (3.12) ukazuje funkci `main()` fragment shaderu. Výsledky filtru jsou vidět na obrázku (3.3).

```
void main() {
  // brightnessFactor je uniform float
  outColor = brightness * texture(u_texture, v_tex_coord);
}
```

Listing 3.12: Filtr zesvětlení

Obrázek 3.3: Výsledky zesvětlení

3.3.3 Zaostření pomocí detekce hran

Tento filtr je implementován ve třídě `EdgeSharpenFilter`. Jako parametr má `float sharpness`. Prvním krokem této metody zaostření je nalezení hran, pro to se použije konvoluce s maticí $M = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$ [15]. Konvoluce dělá sumu sousedních barev pixelů vynásobených příslušnými násobky z matice M. Tato matice zajistí, že pokud jsou sousední pixely matice podobné, tak jejich suma bude blízká černé. Při větších rozdílech mezi pixely se bude barva pixelu blížit bílé. Bílá barva tedy signalizuje hrany (viz. obr. 3.4).
Obrázek s detekovanými hranami se po vynásobení faktorem ostrosti přičte k originálnímu obrázku [15]. Výsledek zaostření s detekcí hran vypadá takto (3.5), kód funkce `main()` vypadá následovně (3.13).

```
// Ziskani velikosti textury
vec2 imageSize = vec2(textureSize(u_texture, 0));
// Jak daleko je dalsi pixel
vec2 onePixel = vec2(1, 1) / imageSize;
vec4 edges = vec4(0);
// Konvoluce
edges += 8.0 * texture(u_texture, v_tex_coord);
edges += -1.0 * texture(u_texture, v_tex_coord + vec2(-1, -1) *
↪  onePixel);
edges += -1.0 * texture(u_texture, v_tex_coord + vec2(-1, 0) *
↪  onePixel);
edges += -1.0 * texture(u_texture, v_tex_coord + vec2(0, -1) *
↪  onePixel);
edges += -1.0 * texture(u_texture, v_tex_coord + vec2(1, 0) *
↪  onePixel);
edges += -1.0 * texture(u_texture, v_tex_coord + vec2(0, 1) *
↪  onePixel);
edges += -1.0 * texture(u_texture, v_tex_coord + vec2(1, 1) *
↪  onePixel);
edges += -1.0 * texture(u_texture, v_tex_coord + vec2(-1, 1) *
↪  onePixel);
edges += -1.0 * texture(u_texture, v_tex_coord + vec2(1, -1) *
↪  onePixel);

// Original + ostrost * Hrany
outColor = texture(u_texture, v_tex_coord) + sharpness*edges;
```

Listing 3.13: Zaostření s detekcí hran

3.3.4 Zaostření pomocí unsharp

Unsharp je implementován ve třídě `SharpenFilter`. Jako parametr má `float sharpness`. Filtr se skládá ze tří kroků: rozostření, rozdíl originálu a rozostření,

přičtení rozdílu k originálu [15]. Pro rozostření je použita Gauss matice rozostření $G = \begin{bmatrix} 0.0625 & 0.125 & 0.0625 \\ 0.125 & 0.25 & 0.125 \\ 0.0625 & 0.125 & 0.0625 \end{bmatrix}$ [16]. Odečtení rozostřeného obrázku od originálu vrátí nalezené hrany, ale rozmazané. Tato metoda dává tedy menší zaostření obrazu. V ukázce kódu (3.14) bude rozostření provedeno funkcí `blur()`, protože je metoda stejná jako v kódu (3.13), pouze používá jiné konstanty.

```
// Ziskani velikosti textury
vec2 imageSize = vec2(textureSize(u_texture, 0));
// Jak daleko je dalsi pixel
vec2 onePixel = vec2(1, 1) / imageSize;
vec4 blurred = blur();
// Original - Rozostreni
vec4 edges = texture(u_texture, v_tex_coord) - blurred;
// Original + ostrost * Hrany
outColor = texture(u_texture, v_tex_coord) + sharpness*edges;
```

Listing 3.14: Kód zaostření metodou unsharp

3.3.5 Klíčování

Klíčování (Chroma Keying) je filtr, který dělá určitou barvu obrazu průhlednou. K tomu jsou třeba tři parametry. Barva, kterou chceme vyfiltrovat $\vec{C}$. Vzdálenost od $\vec{C}$, kde se bude barva plně vyfiltrovávat r_A. A vzdálenost od $\vec{C}$, kde se bude plně zachovávat barva originálního pixelu r_B (viz. obr. 3.8). Barvy v rozmezí $[r_A, r_B]$ jsou lineárně interpolované mezi $(0, 0, 0)$ a barvou originálního pixelu [14].

Tento filtr je implementovaný třídou `ChromaFilter` a pro jeho funkci používá tři parametry: `float innerTolerance` (r_A), `float outerTolerance` (r_B), `float[3] chromaColorRGB` ($\vec{C}$: mezi 0 a 1). Výsledek klíčování je vidět na obrázku (3.7), kde je znak v centru kontroléru průhledný (pozadí je černé). Ústřižek kódu (3.15) ukazuje `main()` funkci klíčování.

```
vec4 color = texture(u_texture, v_tex_coord);
// Vzdalenost mezi barvou k vyfiltrovani a barvou originalu
float dist = length(color - chromaClr);
// Vypocet pruhlednosti pixelu (alpha)
float alpha;
if (dist < rA) {
  alpha = 0.0;
} else if (dist < rB) {
  alpha = (dist - rA)/(rB - rA);
} else {
  alpha = 1.0;
}
outColor = color*alpha;
```

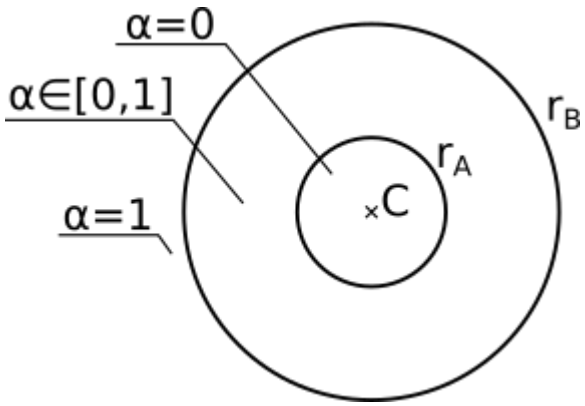
Obrázek 3.4: Detekce hran

Obrázek 3.5: Zaostření s detekcí hran

Obrázek 3.6: Zaostření metodou unsharp

Listing 3.15: Kód klíčování

Obrázek 3.7: Klíčování s černým pozadím

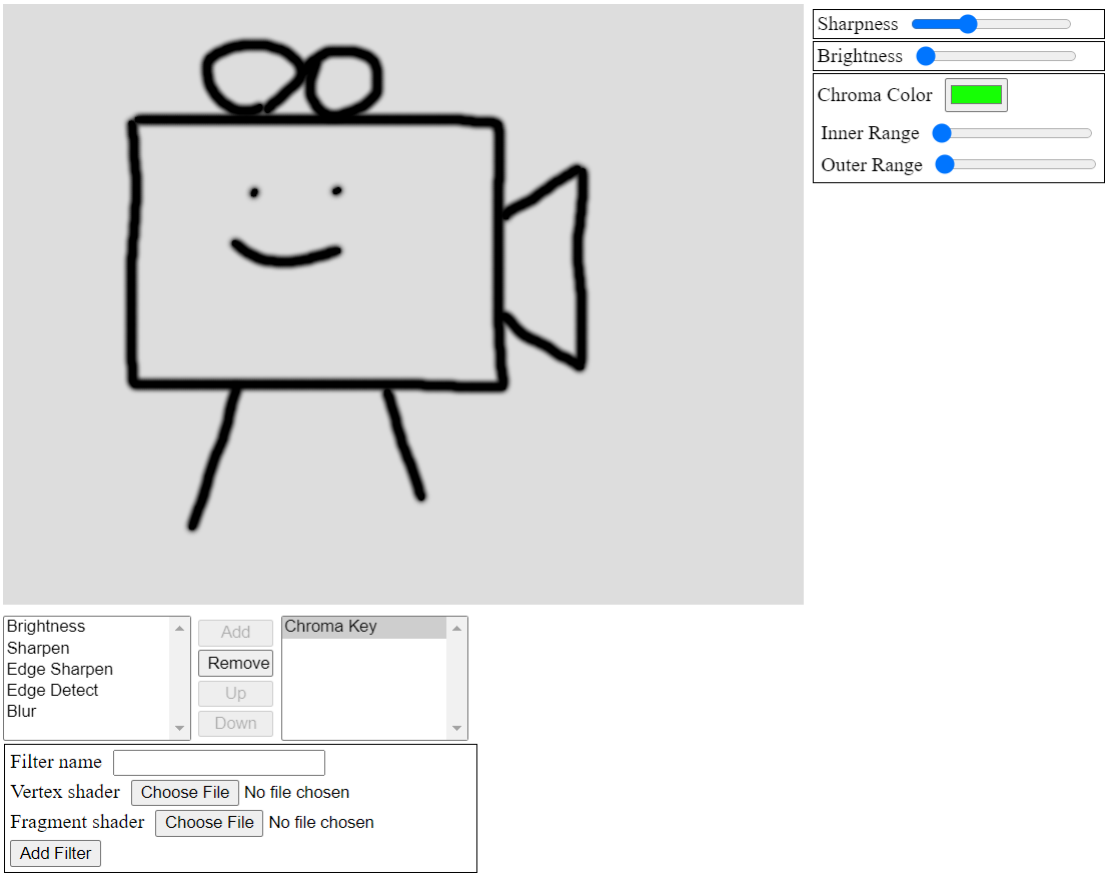


Obrázek 3.8: Viditelnost pixelu okolo barvy klíčování

Kapitola 4

Aplikace

Pro ukázku použití VideoContext rozhraní byla vytvořena demo aplikace. Tato aplikace přijímá obraz z web kamery a aplikuje na něj filtry zvolené uživatelem pomocí dvou seznamů. Aplikace také umožňuje vytvoření vlastního filtru pomocí souborů vertex a fragment shaderů (přípony `.vert` a `.frag`). Tyto soubory musí pracovat s atributy a uniformy definovaných v základní `Filter` třídě.



Obrázek 4.1: Screenshot aplikace

4.1 Popis aplikace

Aplikace (viz. obr. 4.1) se skládá ze čtyř hlavních částí: Canvasu (šedé pozadí s ilustrací kamery), vstupů pro parametry filtrů (vpravo od canvasu), výběru filtrů (pod canvasem), tvorby nových filtrů (dole).

Canvas Canvas VideoContextu. Jako textura je použita web kamera zařízení, ke které se přistupuje pomocí `<video>` tagu.

Parametry filtrů Jezdce a volba barvy pro filtry implementované VideoContextem. *Sharpness* pro zaostření, *Brightness* pro zesvětlení a *Chroma Color*, *Inner Range* a *Outer Range* pro klíčování.

Volba filtrů Pomocí dvou seznamů: levý seznam obsahuje aktuálně nevyužité filtry, pravý seznam obsahuje používané filtry. Filtry se vykonávají ze shora dolů a jejich pořadí lze měnit pomocí tlačítek *Up* a *Down*. Tlačítko *Remove* předá filtr z pravého seznamu do levého. *Add* předá filtr z levého seznamu do pravého.

Tvorba filtru Nový filtr lze vytvořit v aplikaci pomocí vertex a fragment shaderů ze souborů (přípony `.vert` a `.frag`) psaných v GLSL 3.0. Filtr pojmenovaný v kolonce *Filter name*, lze pak vytvořit pomocí *Add Filter* tlačítka.

4.1.1 Získání záznamu web kamery

Pro získání záznamu z web kamery se využívá `video` elementu, buď z HTML tagu nebo vytvořením v javascriptu (viz. kód 4.1).

```
<!-- V definici vzhledu stranky -->
<video id="video" playsinline></video>
<!-- Na konci HTML -->
<script type="module" >
  // Takto se vytvori novy video element
  const video = document.createElement("video");
  // Takto se pristupuje k <video> tagu z HTML
  const vidFromHTML = document.getElementById("video");
</script>
```

Listing 4.1: Video tag pro připojení kamery

Záznam z kamery se získá pomocí `getUserMedia(constraints)`. Objekt `constraints` slouží k nastavení parametrů pro video a zvuk zařízení, např.: hlasitost zvuku, rozměry videa, výběr z více kamer (přední/zadní u telefonu) [17]. V této aplikaci zvuk není potřeba a je použita přední kamera bez změny rozměrů. Objekt se záznamem z kamery se poté vloží do `video.src`, viz. kód (4.2).

```
async function initCamera() {
  // Ziskani zaznamu z predni kamery bez zvuku
    const stream = await
    ↪  window.navigator.mediaDevices.getUserMedia({
        audio: false,
        video: {
          facing: 'user'
        }
    });
    // Schovani videa, chce zobrazovat pouze canvas
    video.style.display = "none";
    window.devicePixelRatio = 1;
    // Nastaveni zdroje videa na zaznam z kamery
    video.srcObject = stream;
    video.play();
}
```

Listing 4.2: Inicializace kamery

4.1.2 Použití rozhraní

Rozhraní je použito uvnitř uživatelského skriptu `filters.js`, který je typu `module`. Pro zobrazení canvasu se používá `<div>` tag jako kontejner pro canvas, viz. kód 4.3.

```
<!-- Kontejner pro umisteni canvasu -->
<div id="canvas-container"></div>
<!-- JS import na konci HTML souboru -->
<script type="module" src="./filters.js"></script>
```

Listing 4.3: Import skriptu a kontejner pro canvas

Uvnitř `filters.js` je VideoContext importován pomocí klíčového slova `import`. Poté začíná inicializace rozhraní: získání kontejneru pro canvas, vytvoření VideoContextu, vložení canvasu do kontejneru, vytvoření filtrů a jejich přidání do seznamu filtrů, viz. kód 4.4.

```
// Import rozhrani
import VideoContext from "./VideoContextAPI/videocontext.js";
// Ziskani kontejneru pro canvas
const canvasContainer =
↪  document.getElementById("canvas-container");
// Vytvoreni VideoContextu
const videocontext = new VideoContext();
// Pridani canvasu do kontejneru
canvasContainer.appendChild(videocontext.canvas);
// Vytvoreni seznamu filtru
```

```
const filters = [];
// Vytvoreni filtru mimo list (lehci posilani parametru)
const brightFilter = videocontext.createBrightnessFilter();
const sharpenFilter = videocontext.createSharpenFilter();
const edgeSharpFilter = videocontext.createEdgeSharpenFilter();
const chromaFilter = videocontext.createChromaFilter();
// Vytvoreni vlastniho filtru z vertex a fragment shaderu
const blurFilter =
↪  videocontext.createFilterFromSources(vertShader, blurFrag);
// createOption(filter, filterName) prida do seznamu aplikace
↪  novy filtr
createOption(brightFilter, "Brightness");
/* createOption pro vsechny ostatni filtry */
createOption(blurFilter, "Blur");
```

Listing 4.4: Inicializace rozhraní

Po inicializaci programu a načtení web kamery se začne každý snímek volat funkce `render()`. Ta má za úkol: nastavit parametry filtrů, vytvořit frontu filtrů a zavolat vykreslení ve VideoContextu. Nastavování parametrů se dělá podle příslušných parametrů definovaných VideoContextem. Fronta filtrů se tvoří čtením seznamu filtrů vybraných uživatelem. Vykreslování je voláno pomocí `videocontext.drawFromVideo(video)`. Ústřižek kódu (4.5) ukazuje implementaci funkce `render()`.

```
// Nastaveni parametru filtru
setFilterParameters();
// Ziskani vybranych filtru (index + jmeno)
var selFilters = rightSelect.options;
var nFilters = selFilters.length;
var queue = [];
// Tvorba fronty
for (var i = 0; i < nShaders; i++) {
  // Ziskani indexu filtru v listu filters[]
  var index = selFilters.item(i).value;
  // Ziskani filtru z listu a pridani do fronty
  filter = filters[index];
  queue.push(filter);
}
// Prirazeni fronty
videocontext.queue = queue;
// Zavolani vykresleni
videocontext.drawFromVideo(video);
// Registrace funkce pro vykresleni dalsiho snimku
window.requestAnimationFrame(render);
```

Listing 4.5: Vykreslování v filters.js

Kapitola 5

Výsledky

Pro použití filtrů v reálném čase je důležité zajistit dostatečnou rychlost vykreslování. V demo aplikaci se pro vykreslovací smyčku používá metoda `requestAnimationFrame(func)`. Ta se běžně volá 60 krát za sekundu. Proto by čas strávený při vykreslení měl být méně než 16 ms. V této sekci bude ukázáno, jak si VideoContext vede při: vykreslování videí s různými rozlišeními, v závislosti na grafické kartě (integrovaná vs. dedikovaná), na mobilním zařízení a v porovnání s čistě JavaScriptovou implementací pomocí ImageData API (bez delegace na GPU). Pro tyto porovnání se bude měřit čas strávený v samotných shaderech a počet snímků za sekundu (FPS) na filtrech implementovaných VideoContextem.

5.1 Metody měření

Tato část se bude zabývat metodami, které byly použity pro měření výše zmíněných metrik. Pro měření byl použit následující (poměrně zastaralý) hardware.

PC CPU Intel Core i7-7700HQ @ 2.80GHz

PC Integrovaná GPU Intel HD Graphics 630

PC Dedikovaná GPU NVIDIA GeForce GTX 1050 Ti

Mobilní telefon Huawei Nova 5T, CPU HiSilicon Kirin 980, GPU Mali G76 MP10

Pro měření byly použity dvě video s různými rozlišeními. Video s menším rozlišením má 0.921 MPx (Quicktime HD 720p), video s větším rozlišením má 8.85 MPx (4K TV).

5.1.1 FPS

Pro měření snímků za sekundu byly použity funkce `console.time(name)` a `console.timeEnd(name)`. Na začátku funkce `render()` se zastaví časování z minulého volání `render()` a poté se započne nové, viz. kód 5.1. Měří se čas tak vykreslení a i čas mezi jednotlivými `render()` voláními. Jedná se tedy o

celou periodu mezi vykresleními T. Z těchto časů se pak získá FPS pomocí $FPS = 1/T$.

```
function render() {
    // Konec mereni z minuleho render()
    console.timeEnd("render-period");
    // Zacatek noveho mereni
    console.time("render-period");
    /*
      Proces vykreslovani
    */
    // Registrace vykreslovaci funkce
    window.requestAnimationFrame(render);
}
```

Listing 5.1: Měření FPS

5.1.2 Časy strávené v shaderu

K měření času stráveného v shaderu se používají rozdílné metody pro ImageData API a WebGL. Tato část se bude zaměřovat na WebGL. K tomu bylo použito rozšíření `EXT_disjoint_timer_query_webgl2`. Toto rozšíření měří čas strávený na GPU pomocí dotazů `Query`, viz. kód 5.2 [11].

```
// Ziskani rozsireni pri inicializaci VideoContext
let ext = gl.getExtension('EXT_disjoint_timer_query_webgl2');
/*
  Pri vykreslovani ve VideoContext
*/
// Vytvoreni dotazu
let query = gl.createQuery();
// Zacatek dotazu
gl.beginQuery(ext.TIME_ELAPSED_EXT, query);
// Vykreslovani
gl.drawArrays(...);
// Konec dotazu
gl.endQuery(ext.TIME_ELAPSED_EXT);
/*
  Pozdeji, az bude vysledek k dispozici, ziskani vysledku
*/
// Kontrola jestli dotaz existuje
if (query) {
  // Zjisteni dostupnosti dotazu
  let available = gl.getQueryParameter(query,
  ↪  gl.QUERY_RESULT_AVAILABLE);
  let disjoint = gl.getParameter(ext.GPU_DISJOINT_EXT);
  if (available && !disjoint) {
```

```
    // Ziskani casu v nanosekundach a vypsani do konzole v ms
    let timeElapsed = gl.getQueryParameter(query,
    ↪  gl.QUERY_RESULT);
    console.log(timeElapsed / 1000000.0);
  }
  if (available || disjoint) {
    // Odstraneni dotazu
    gl.deleteQuery(query);
    query = null;
  }
}
```

Listing 5.2: Měření času na GPU

5.1.3 Integrovaná vs. dedikovaná grafická karta

Tyto měření se týkají WebGL. Pro nastavení grafické karty se musí v nastavení Windows v sekci *Nastavení Grafiky* přidat používaný prohlížeč a nastavit pro něj požadovanou grafiku. Toto nastavení může být také potřeba udělat v ovladači dedikované grafiky. Metody měření FPS i času v shaderu jsou pro oba případy identické.

5.1.4 Na mobilním zařízení

Pro měření na mobilním telefonu, s operačním systémem Android, lze použít vzdálené ladění přes USB pomocí Google Chrome. To umožňuje v prohlížeči počítače ovládat prohlížeč v telefonu a číst výpisy do konzole, HTML, CSS i JS otevřené záložky. K webové aplikace se lze na telefonu připojit pomocí přesměrování portů z počítače, kde je webová aplikace lokálně hostována [18]. Měření FPS na telefonu probíhá stejným způsobem jako na počítači. Měření času v shaderu pomocí výše zmíněného rozšíření není možné kvůli chybějící podpoře. Tato metrika je tedy nahrazena měřením FPS s několika aplikovanými filtry pro odhad zátěže.

5.1.5 Pomocí ImageData API

Pro toto měření je nutné každý filtr implementovat pomocí ImageData API. To je rozhraní pro získání obsahu `canvas` elementu jako pole pixelů ve formátu `0-255 RGBA`, kde každá složka je uložena jako celé číslo [13]. Do canvasu se zapíší hodnoty z videa pomocí `CanvasRenderingContext2D`. 2D kontext canvasu má funkci `drawImage(source, posX, posY, width, height)`, pomocí které lze získat obraz z obrázku či videa (`image`/`video` elementu) [12]. Měření FPS a času ve filtru je možné udělat pomocí `console.time()` a `console.timeEnd()`. Měření obou metrik pro jeden filtr pak vypadá následovně (5.3).

```
// Mereni FPS
console.timeEnd("render-period");
console.time("render-period");
// Skryty canvas ktery prijme data z videa
bCtx.drawImage(video, 0, 0, width, height);
// Ziskani ImageData ze skryteho canvasu (s vykreslenym videem)
const srcImg = bCtx.getImageData(0, 0, width, height);
const destImg = new ImageData(width, height);
// Ziskani poli pixelu z ImageData (0-255 RGBA)
const srcData = srcImg.data;
const destData = destImg.data;
// Mereni casu na filtru zesvetleni (brighten)
console.time("filter-time");
brighten(srcData, destData);
console.timeEnd("filter-time");
// Vlozeni ImageData do viditelneho canvasu
fCtx.putImageData(destImg, 0, 0);
window.requestAnimationFrame(render);
```

Listing 5.3: Měření na CPU pomocí ImageData API

5.2 Naměřené hodnoty a grafy

Tato sekce je věnována naměřeným hodnotám z jednotlivých metrik. Pro zkrácení tabulek je: měření pomocí ImageData API značeno jako CPU, telefon zkrácen na tel. a měření na grafické kartě s WebGL značeno jako GPU. Časy strávené ve filtrech jsou vidět v tabulkách (5.1) (rozlišení 0.921 MPx) a (5.2) (rozlišení 8.85 MPx). Tyto časy jsou také vidět na grafu (5.1), kde jsou porovnány na filtru klíčování.

Kvůli chybějící podpoře WebGL rozšíření pro měření času na telefonu chybí časy na grafické kartě telefonu. Tato metrika je nahrazena měřením FPS s několika filtry. S rozlišením videa 8.85 MPx a při použití dvou filtrů zaostření, vykreslování poklesne na 55 FPS, při aplikování tří filtrů zaostření poklesne dále na 33 FPS. S rozlišením 0.921 MPx bylo avšak možné použít přes deset filtrů zaostření a vykreslení pořád zůstalo nad 60 FPS. Je tedy vidět že pro videa s menším rozlišením (720p, 1080p) je WebGL dostatečně výkonné pro vykreslování s více filtry v reálném čase.

Měření snímků za sekundu bylo rozděleno do tří tabulek: FPS na telefonu (5.3), FPS na rozlišení 0.921 MPx na PC (5.4) a s rozlišením 8.85 MPx na PC (5.5). Počet FPS lze také vidět na grafu (5.2), kde je použit filtr klíčování. Nejlepšího výkonu dosahuje dedikovaná grafická karta počítače. Druhý nejlepší výkon má překvapivě grafická karta telefonu s třetím místem patřícím integrované kartě počítače. Pro FPS proběhlo ještě jedno testování

na integrované grafické kartě počítače s videem o rozlišení 2 MPx (1080p), kde i při použití pěti filtrů zaostření video zůstalo na 60 FPS.

Z měření je vidět, že použití ImageData API není vhodné pro filtrování v reálném čase. S použitím WebGL na integrované grafické kartě se pro filtr zaostření (konvoluce 3x3) a s rozlišením videa 8.85 MPx zlepší čas vykreslování 66 krát. S rozlišením videa 0.921 MPx je vykreslení 86 krát rychlejší. Také si lze povšimnout, že při vykreslování videí s velkým rozlišením na integrované grafické kartě, FPS klesne pod 60. Pro větší rozlišení by tedy bylo lepší použít dedikovanou grafickou kartu. VideoContext, zdá se, je vhodný k použití pro aplikování několika filtrů na videa s běžnými rozlišeními (720p, 1080p) nebo na záznam z web kamery v reálném čase.

Filter	Tel. CPU	PC CPU	PC GPU: Intel	PC GPU: NVidia
Zesvětlení	15.038 ms	9.355 ms	1.094 ms	0.701 ms
Klíčování	29.34 ms	14.897 ms	1.088 ms	0.765 ms
Zaostření	177.26 ms	110.1 ms	1.288 ms	0.724 ms

Tabulka 5.1: Rozlišení 0.921 MPx: Čas strávený ve filtru

Filter	Tel. CPU	PC CPU	PC GPU: Intel	PC GPU: NVidia
Zesvětlení	135.13 ms	66.33 ms	11.482 ms	2.288 ms
Klíčování	202.75 ms	118.56 ms	11.359 ms	3.018 ms
Zaostření	1237.5 ms	992.78 ms	14.9 ms	3.145 ms

Tabulka 5.2: Rozlišení 8.85 MPx: Čas strávený ve filtru

Filter	0.921MPx CPU	0.921MPx GPU	8.85MPx CPU	8.85MPx GPU
Zesvětlení	15.93	68.5	3.33	62.3
Klíčování	12.79	69.6	2.73	62.6
Zaostření	4.41	64.3	0.586	62.1

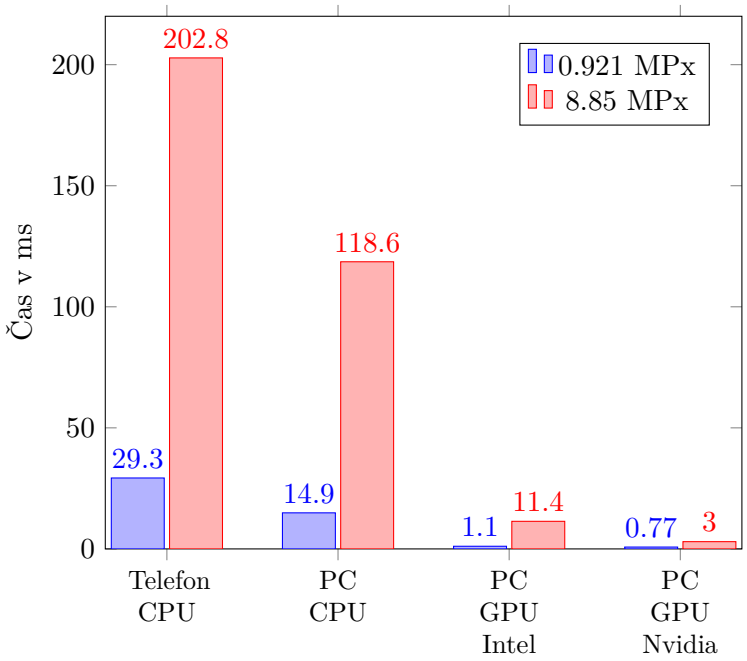
Tabulka 5.3: FPS na telefonu

Filter	CPU	GPU Intel	GPU NVidia
Zesvětlení	32.76	66.2	64.6
Klíčování	27.93	66.4	65.4
Zaostření	7.69	64.1	62.8

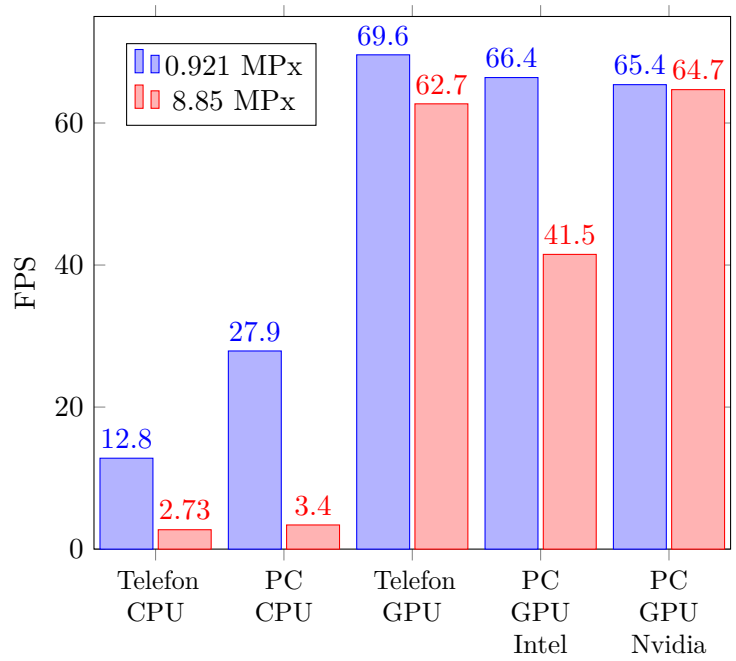
Tabulka 5.4: Rozlišení 0.921 MPx: FPS na PC

Filter	CPU	GPU Intel	GPU NVidia
Zesvětlení	3.953	42.39	64.4
Klíčování	3.4	41.45	64.7
Zaostření	0.925	41.16	64.4

Tabulka 5.5: Rozlišení 8.85 MPx: FPS na PC



Obrázek 5.1: Časy strávené ve filtru klíčování



Obrázek 5.2: FPS filtru klíčování

Kapitola 6

Závěr

Cílem této práce bylo vytvořit rozhraní v JavaScriptu za pomoci WebGL pro provádění úprav nad dvourozměrnými texturami pomocí shaderů. Toto rozhraní mělo být schopné vytvořit shadery a aplikovat jeden či více shaderů v definovaném pořadí. Poté se měla vytvořit aplikace na základě tohoto rozhraní, která tyto shadery, ve formě filtrů, aplikovala na video či záznam z web kamery. Posledně se měl změřit výkon této aplikace (FPS a čas strávený v shaderu).

Jako rozhraní pro úpravu dvourozměrných textur bylo vytvořeno VideoContext API, se kterým lze pomocí třídy `Filter` aplikovat shadery na textury a video/image elementy HTML. Pomocí pole lze také definovat jejich pořadí pro vykonávání. Tvorbu shaderů je možné udělat pomocí dědění z `Filter` třídy nebo pomocí GLSL kódů vertex a fragment shaderů s pomocí příslušné funkce VideoContextu. V rámci VideoContextu byli vytvořeny čtyři filtry: zesvětlení, zaostření (dvěma způsoby) a klíčování.

Pro testování rozhraní byla vytvořena demo aplikace, ve které lze pomocí dvou seznamů volit, které filtry budou použity při vykreslování a také upravit jejich pořadí vykonání. Pro filtry implementované VideoContextem je také možné měnit parametry ovlivňující jejich chování. Vlastní filtr je uživatel schopen vytvořit přidáním vlastních GLSL kódů vertex a fragment shaderů přímo v aplikaci.

V rámci měření výkonu se hledělo na časy strávené v shaderech a na snímky za sekundu výsledného obrazu upraveného VideoContextem. Tyto metriky se měřily v následujících kontextech: počítač vs. mobilní telefon, integrovaná vs. dedikovaná grafická karta, v závislosti na rozlišení videa, v porovnání s čistě JavaScriptovou implementací (ImageData API). Z měření vyšlo najevo, že použití WebGL na integrované grafické kartě samotné zlepšilo výkon až 80 krát oproti ImageData API. Dedikovaná grafická karta (i zastaralá) je schopna pracovat i s 4K videem a zůstat na 60 FPS. Mobilní zařízení je schopné pracovat s videi o rozlišení 720p a 1080p bez problémů i s více filtry.

6.1 Možné pokračování

Velkým nedostatkem rozhraní a aplikace je relativně velká bariéra pro tvorbu vlastních filtrů (základní znalost GLSL a WebGL). Pro rozhraní by pak bylo vhodné navrhnout obecné filtry pro jednodušší operace jako například konvoluce pomocí matic různých velikostí nebo násobení barvy vstupu jinou barvou. V případě aplikace by bylo dobré umožnit úpravu kódů shaderů za běhu pro lehčí psaní vlastních filtrů.

Další věc, kterou by bylo možné udělat, je přidání více filtrů jako např. deformace obrazu (zrcadlení, vlnění obrazu), Gauss rozostření, mapování barev. VideoContext by taky byl možný rozšířit o možnost vykreslení textu či obrázku přes původní obraz, což je často používaná operace na internetu. Text je avšak poměrně složitá úloha sama o sobě.

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Pleticha** Jméno: **Radek** Osobní číslo: **492065**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Specializace: **Počítačové hry a grafika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Porovnání videofiltrů v GLSL a 2D Canvasu

Název bakalářské práce anglicky:

Comparison of video filters in GLSL and 2D Canvas

Pokyny pro vypracování:

Seznamte se s rozhraním WebGL, koncepy OpenGL a základy jazyka GLSL.
Následně navrhněte JavaScriptovou infrastrukturu, pomocí které bude možné provádět úpravy nad dvourozměrnými texturami pomocí Fragment shaderů. Popište vzniklé API pro zadávání jednotlivých shaderů a definici pořadí jejích zpracování.
Naimplementujte vzorovou aplikaci (prototyp), která bude v reálném čase aplikovat jeden či více těchto shaderů na snímky videa, přehrávaného v HTML značce <video>. Aplikace ať disponuje alespoň těmito efekty/shadery: změna světlosti, klíčování, zaostření/rozostření. Transformovaná videodata následně zobrazujte v HTML Canvasu.
Navrhněte UI, pomocí kterého by mohl uživatel zadávat shadery vlastní - buď ze souborů na lokálním disku, nebo v interaktivním formulářovém poli.
Otestujte výkon (celkové FPS i čas strávený v jednotlivých shaderech) takovéto aplikace v následujících kontextech:
- na desktopu vs. na mobilním zařízení;
- integrovaná vs. dedikovaná grafická karta;
- v závislosti na rozlišení videa;
- v porovnání s čistě JavaScriptovou implementací (kterou taktéž naimplementujete pomocí 2D Canvas - ImageData API).

Seznam doporučené literatury:

https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API
https://thebookofshaders.com/00/
https://webglfundamentals.org/webgl/lessons/webgl-render-to-texture.html

Jméno a pracoviště vedoucí(ho) bakalářské práce:

RNDr. Ondřej Žára Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **08.03.2023** Termín odevzdání bakalářské práce: **26.05.2023**

Platnost zadání bakalářské práce: **16.02.2025**

RNDr. Ondřej Žára	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D.
podpis vedoucí(ho) práce		podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. _____ _____

Datum převzetí zadání Podpis studenta

Literatura

[1] OpenCV, *OpenCV modules* [online]: https://docs.opencv.org/4.x/index.html [cit. 2023-04-13].

[2] shamadee, *WebDPS - GitHub repozitář* [online]: https://github.com/shamadee/web-dsp [cit. 2023-04-13].

[3] Davidson Fellipe, *Lena.js - GitHub repozitář* [online]: https://github.com/davidsonfellipe/lena.js [cit. 2023-04-13].

[4] Mozilla Developer Network, *Animating textures in WebGL - Web APIs | MDN* [online]: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Tutorial/Animating_textures_in_WebGL [cit. 2023-03-20].

[5] WebGLFundamentals, *WebGL Fundamentals* [online]: https://webglfundamentals.org/webgl/lessons/webgl-fundamentals.html [cit. 2023-03-20].

[6] WebGLFundamentals, *WebGL2 What's New* [online]: https://webgl2fundamentals.org/webgl/lessons/webgl2-whats-new.html [cit. 2023-03-20].

[7] WebGLFundamentals, *WebGL Rendering to a Texture* [online]: https://webglfundamentals.org/webgl/lessons/webgl-render-to-texture.html [cit. 2023-03-20].

[8] Khronos, *WebGL 2.0 Specification* [online]: https://registry.khronos.org/webgl/specs/latest/2.0/ [cit. 2023-03-20].

[9] Khronos, *WebGL Specification* [online]: https://registry.khronos.org/webgl/specs/latest/1.0/ [cit. 2023-03-21].

[10] Anonymní, *webgl - Overview Diagram* [online]: https://webcodingcenter.com/webgl/WEBGL [cit. 2023-03-21].

[11] Khronos, *WebGL EXT_disjoint_timer_query_webgl2 Extension Specification* [online]: https://registry.khronos.org/webgl/extensions/EXT_disjoint_timer_query_webgl2/ [cit. 2023-03-20].

[12] Mozilla Developer Network, *Manipulating video using canvas - Web APIs | MDN* [online]: https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Manipulating_video_using_canvas [cit. 2023-04-05].

[13] Mozilla Developer Network, *ImageData - Web APIs | MDN* [online]: https://developer.mozilla.org/en-US/docs/Web/API/ImageData [cit. 2023-03-20].

[14] Edward Cannon, *Chroma Key* [online]: http://gc-films.com/chromakey.html [cit. 2023-03-20].

[15] Robert Fisher, Simon Perkins, Ashley Walker and Erik Wolfart, *Spatial Filters - Unsharp Filter* [online]: https://homepages.inf.ed.ac.uk/rbf/HIPR2/unsharp.htm [cit. 2023-03-20].

[16] Shermal Fernando, *Gaussian Blur - OpenCV Tutorial C++* [online]: https://www.opencv-srf.com/2018/03/gaussian-blur.html [cit. 2023-03-30].

[17] Mozilla Developer Network, *MediaDevices.getUserMedia() - Web APIs | MDN* [online]: https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia [cit. 2023-03-20].

[18] Kayce Basques, Sofia Emelianova, *Remote debug Android devices - Chrome Developers* [online]: https://developer.chrome.com/docs/devtools/remote-debugging/ [cit. 2023-03-20].