Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra Informačních Technologií

Klíčování videa na GPU

Diplomová práce

Autor práce: MICHAL HVĚZDA

Studijní obor: Aplikovaná Informatika

Vedoucí práce: Ing. BRUNO JEŽEK, Ph.D.

Hradec Králové 2023

Prohlášení

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a s použitím uvedené literatury

. . . . . .

Michal Hvězda

13. srpna 2023

Poděkování

Děkuji vedoucímu své diplomové práce Ing. Bruno Ježkovi za ochotu, trpělivost a metodické vedení.

Anotace

Diplomová práce se zabývá metodami klíčování barevného pozadí. Zaměřuje se na problematiku zpracování videa v reálném čase s využitím technologie klíčování barevného pozadí. Implementuje dva algoritmy v programovacím jazyce C#. Jeden využívá shlukovou analýzu pro určení oblasti pro klíčování, zatím co druhý umožňuje nastavení klíčované oblasti zadat uživatelem. Oba algoritmy jsou optimalizovány, aby umožnily zpracovávat video i při vysokém rozlišení. V rámci varianty jsou představeny tři možné způsoby pro načítání videa. Následně jsou implementace s rozdílným načítáním videa mezi sebou porovnány v kvalitě a rychlosti zpracování, vůči verzi vzniklé v rámci bakalářské práci, na kterou tato práce navazuje a byla provedena v programovacím jazyce Java.

Anotation

Title: Chroma key on GPU

The diploma thesis is focused on method of chroma keying. Is focuses on issue of real-time video processing using chroma keying technology. It implements two algorithms in the C# programming language. First one uses cluster analysis to determine the region for keying, while the other allows the keyed region to be set by the user. Both algorithms are optimized to allow video processing even at high resolution. The implementation presents three possible ways to load the video. Subsequently, the implementations with different video loading are compared with each other in terms of quality and speed of processing, compared to the implementation created in the project of the bachelor's thesis, to which this work follows and was carried out in the Java programming language.

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Se stálým zlepšováním technologií, umožňujících pořizování video záznamu a jeho následného možného odesílání přes internet, je kladen požadavek na schopnost upravovat pořízený záznam v reálném čase. Při této úpravě se požaduje kvalita zpracování, a také jeho rychlost. V opačném případě je možné rozpoznat, že scéna byla upravována.

Jedna z častých úprava obrazu spočívá v odstranění pozadí určité barvy, která se nahradí jinou texturou, popřípadě se nechá pozadí průhledné. Tato problematika je řešitelná za pomoci mnoha přístupů. Lze využít rozdílné barevné modely a algoritmy pro výpočet požadované oblasti, která se má odstranit.

Diplomová práce navazuje na bakalářskou práci zabývající se klíčováním barevného pozadí. Práce se zabývá implementací dvou algoritmů za pomoci programovatelné grafické karty (GPU) s využitím knihovny OpenGL. Cílem práce je zpracovávat video v reálném čase, aniž by utrpěla kvalita zdrojového videa nebo kvalita zpracování. Cílem je zpracovat video o rozlišení 4K a se 60 snímky za sekundu.

Druhá kapitola se zabývá problematikou klíčování a zpracování videa v reálném čase. Kapitola obsahuje hlubší rozbor chyb klíčování a problém s nutností zpracovávat video v reálném čase s určeným snímkování a rozbor použitých barevných modelů.

Třetí kapitola je zaměřena na možnosti využití knihoven pro práci na GPU. Hlavní knihovna využitá pro tuto práci je OpenGL a kapitola se věnuje jejím vlastnostem a prvkům, které se dají pro implementaci využít.

Čtvrtá kapitola se věnuje návrhu implementace. Zde jsou stanoveny hlavní cíle celé práce a také jsou představeny testovací zdroje, na kterých se později bude testovat úspěšná implementace zvolených algoritmů.

Pátá kapitola popisuje nutné kroky pro implementaci zvolených algoritmů. Také jsou zde popsaná nutná vylepšení, rozdíly od bakalářského projektu a je představen způsob, kterým lze implementaci dále optimalizovat, aby bylo cíle práce možné dosáhnout.

V poslední kapitole jsou prezentovány výsledky implementace především z pohledu kvality. Jsou ukázány nedostatky daných implementací vzhledem k výsledkům bakalářské práce. Dále kapitola obsahuje výsledky z pohledu rychlosti zpracování videa jednotlivých implementací pro různá rozlišení testovacích videí.

2 Klíčování obrazu v reálném čase

Technologie klíčování spočívá v odstranění určité barvy pozadí a jejím nahrazením. Barva pozadí obvykle bývá zelená nebo modrá, kvůli snadnému odstranění těchto barev. Proces klíčování se sestává z mnoha dílčích částí, které je nutno provést pro možné zpracování snímku obrazu. Primárně je proveden algoritmus, který rozdělí scénu na popředí a pozadí. Tento algoritmus pouze vypočítá, která část scény má být odstraněna, popřípadě nahrazena.

Proces klíčování v reálném čase může způsobovat snižování kvality obrazu, popřípadě přeskakovat snímky ve videu, a tím snižovat počet jeho snímků za sekundu. Oddělení popředí a pozadí je prováděno sekvenčně, tak aby bylo zajištěno ve videu správné pořadí snímků, ačkoliv předzpracování je možné provádět paralelně.

U snímku je možné nejdříve provést down-scale (zmenšit jeho velikost), ta bude zachovávat dostatek informací o scéně, ale bude možné daný snímek dalšími kroky zpracovat rychleji. Druhou možností je snímky vynechávat, což umožní zpracovat více snímků za sekundu. V obou případech však klesá kvalita výsledného videa.

Popředí a pozadí (dvě základní části) lze obvykle snadno rozpoznat okem pozorovatele a zpravidla se nedají zaměnit. Tudíž nelze popředí zpracovat jako pozadí a opačně. Pokud by se tak stalo, výsledek by nedával smysl pro další zpracování nebo pro pouhou prezentaci. Využitý algoritmus musí rozpoznat oblast pozadí, aby bylo možné ho využít.

Popředím snímku se rozumí popředí scény, která by měla při klíčování zůstat zachována. Její vizuální barva je odlišná od barvy pozadí a většinou je blíže kameře nežli plátno nesoucí barvu pozadí. V některých případech jsou před popředím předměty nesoucí barvu pozadí a klíčují se s pozadím, ačkoliv jsou blíže kameře nežli samotné popředí. Takové případy potřebují následnou úpravu, kde je na vyklíčovanou scénu umístěno vybrané pozadí a textura předmětu, který stojí v popředí.

Pozadí snímku se rozumí pozadí scény, která je při klíčování odstraněna a je nahrazena jiným snímkem, se kterým je spojena v jeden. Pozadí je ideálně stejné

barvy, reálně má alespoň stejný odstín. Většinou se jedná o modrou nebo zelenou barvu, protože mezi nimi a lidskou kůží je velký kontrast.

2.1 Problematika zpracování v reálném čase

Problematika zpracování videa v reálném čase je velmi náročná. Náročnost se neodvíjí od počtu úkonů, které musí být vykonány při zpracování jednoho snímku, ale od časové náročnosti daných úkonů. Doba provedení všech potřebných úkonů určuje maximální možný počet snímků, které lze zpracovat. Tab. 1.

Pokud je požadavek zpracovat 30 snímků za jednu sekundu, je čas zpracování 33 milisekund. Za tento čas je nutno načíst daný snímek do paměti grafiky a provést všechny potřebné úkony. Následně je snímek ještě nutné vykreslit na obrazovku.

Běžné monitory jsou schopny vykreslit 144 až 240 snímků za sekundu. Což je velmi krátký časový úsek na přípravu snímku pro vykreslení. Požadavek na zpracování videa v reálném čase se odvíjí od počtu snímků za sekundu daného videa.

Tabulka 1: Přehled potřebného času při zpracování v reálném čase

Počet snímků za sekundu	Čas na zpracování jednoho snímku
24	41 ms
30	33 ms
60	16 ms
144	6,9 ms

Pokud je video zpracováno rychleji, nežli je jeho frekvence snímků, působí na pozorovatele video zrychleným dojmem. V opačném případě se zdá zpracovávané video zpomalené. Abychom tomuto jevu předešli, je důležité zpracovat během sekundy všechny snímky, které představují ve videu stejný časový úsek.

Synchronizace videa řeší problém vzniklý schopností zpracovat více snímků za sekundu, nežli má zpracovávané video. Pro synchronizaci lze využít prosté omezení snímkování v OpenGL. Druhou možností je využití systémového času nebo času videa [5]. Tato metoda je vhodná pro synchronizaci videa a zvuku z rozdílných zdrojů. Další

možností je vytvořit synchronizační signál, který bude určovat, kdy byl daný snímek pořízen a zpracovat ho [6]. Tato metoda je využitelná při více zdrojích zpracovávaného videa.

Běžné video má obvykle 60 snímků za sekundu, což na zpracování a vykreslení daného snímku dává pouze 16 milisekund. Jestliže byl předchozí snímek zpracováván déle, než je vymezený čas na jeden snímek, je možné jej zahodit. Daný snímek se přeskočí a začne se rovnou zpracovávat snímek následující. Touto technikou lze zpracovávat video s větší snímkovou frekvencí, aniž by pozorovatel poznal, že nejsou zpracovávány všechny snímky.

Požadavek na rychlost zpracování roste nejenom s frekvencí snímkování, ale zároveň s rostoucí velikostí zpracovávaného videa. Tab. 2.

Tabulka 2: Rozlišení s počtem barevných bodů

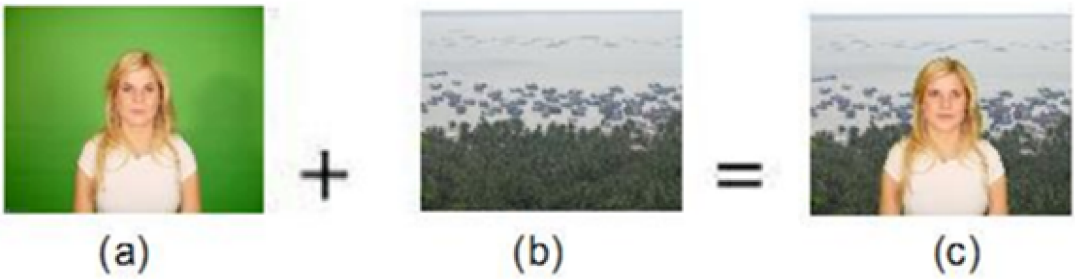
Název	Rozlišení	Počet barevných bodů	Počet pixelů oproti základu
Full HD	1920×1080	2 073 600	$100,0\%$
Quad HD	2560×1440	3 686 400	$177,7\%$
4K	3840×2160	8 294 400	$400,0\%$

Běžné zobrazení videa se v dnešní době pohybuje od rozlišení Full HD k rozlišení 4K. Toto zobrazení umožňuje porovnání rychlosti napříč rozlišeními a zajišťuje najít hranici, kde je možné ještě dané rozlišení zpracovávat a kde to již využívaný hardware nezvládne.

V případě malé textury je možnost zpracovávat ji za pomoci procesoru (CPU). Načtený snímek v paměti při malém rozlišení může být rychleji zpracován za pomoci CPU, pokud je požadovaná procedura dostatečně jednoduchá. Pomalejší zpracování CPU je v takovém případě vykompenzováno tím, že není třeba data přesouvat na GPU a provést zpracování tam.

2.2 Klíčování

Technologie klíčování je technika spočívající v úpravě obrazu, kdy je část odstraněna a nahrazena nějakou jinou scénou (Obr. 1). Celý proces spočívá ve dvou krocích - selekci a odstranění, popřípadě nahrazení vybrané části obrazu. [1]



Obrázek 1: Ukázka klíčování obrázků a) popředí b) pozadí c) výsledný obrázek [1]

Selekcí se rozumí proces, kdy je oblast, která nespadá do popředí, určena k odstranění a nahrazení. Výstupem selekce je maska obrazu, která slouží pro následovné odstranění vybraných částí obrazu. Odstranění lze provádět hned po selekci, tudíž výstupem je již vyklíčovaný obraz. Selekce může také spočívat ve vybrání jedné specifické barvy, případně určité oblasti kolem ní. V tomto případě je výsledek závislý na barevnosti použitého modelu, ve kterém je daná výseč definována. [1]

Práce [7] využívá pro klíčování právě definovanou oblast. Tato oblast je definována v barevném modelu YCbCr. Samotná oblast je tvořena výsečí určující hranice klíčované hranice. Metoda klíčování spočívá v určení, jestli zpracovávaný barevný bod leží v definované oblasti nebo ne. Pokud bod leží v oblasti, je odstraněn a nahrazen. V opačném případě je ponechán a začne se zpracovávat další bod.

Druhý způsob selekce lze nazvat shlukovou analýzou, za pomoci které se rozdělí obraz do K počtu oblastí dle barvy a následně se vyberou dané oblasti, které budou odstraněny. Tímto způsobem docílíme přesnějšího odstranění barvy pozadí nežli prvním způsobem. Vytvoření daného počtu oblastí pomocí shlukové analýzy, aby se tato výhoda projevila, může trvat příliš dlouho, pak je možné množství oblastí omezit na úkor přesnosti. Velká výhoda oproti prvnímu způsobu je využití autonomní selekce,

aniž by byla vyžadována jakákoliv interakce od uživatele, ačkoliv v takovém případě může poklesnou kvalita selekce. [1]

Práce [8] využívá shlukovou analýzu pro rozdělení popředí a pozadí, pro které využívá pouze dvě oblasti. Hraniční body následně rozřazuje při klíčování. Pro celý proces využívá specifický hardware, který je tomu uzpůsoben. Alternativa je práce [9], využívající také shlukovou analýzu implementovanou na programovatelném hardwaru. Po provedení shlukové analýzy je navíc provedeno hledání hran, pro co nejjemnější přechod u vyklíčovaného snímku.

V práci [10] se využívá shluková analýza pro klíčování dvoubarevného vzorovaného pozadí. Po výpočtu je následně kontrolováno, že se odstraňuje pouze oblast se vzorem a následně se scéna vyklíčuje. Výhodou této metody je schopnost pracovat s dobrými výsledky i pokud je jedna z barev obsažena v popředí, například na oblečení.

Poslední možný způsob je využití strojového učení, které bude určovat, co je barva popředí a co barva pozadí. Poté může vybranou barvu odstranit. Toto řešení nabízí možnost odstraňování samotného pozadí nehledě na jeho barvu. Barva pozadí nemusí být v takovém případě jednolitá. Dané řešení může být, ale podmíněno velkou náročností na hardware v podobě grafické karty. [1]

Po selekci lze přistoupit k odstranění, popřípadě nahrazení vybrané oblasti. V tomto kroku je již známé rozdělení snímku na popředí a pozadí, tudíž lze upravit původní snímek. Nastavíme průhlednost pozadí na nulovou hodnotu, aby nebylo pozadí nadále vidět, další možností je pozadí překrýt scénou, respektive částí snímku, který bude nově na pozadí. Překrytí probíhá dle souřadnic obou textur a při rozdílných rozměrech může docházet k deformacím obrazu v závislosti na tom, kam je který obraz vkládán. [1]

2.2.1 Využití klíčování

Klíčování slouží k odstranění, respektive vyčlenění pozadí. Výsledkem může být maska nebo zpracovaný obraz. Maska určuje oblast pozadí a lze ji využít k dalšímu zpracování. Zpracovaný obraz je bez pozadí. Typ využití lze rozdělit do dvou kategorií.

První kategorie využívá zpracovaný obraz jako výsledek a dále s ním již nepracuje. Pozadí je snížením průhlednosti odstraněno, popřípadě je nahrazeno jinou scénou a je použito jako výsledek. Do této kategorie, lze zahrnout výsledky této práce nebo aplikace, které umožňují odesílat video. Tyto aplikace jsou například Teams, OBS Studio nebo Nvidia Broadcast. Ačkoliv každá z aplikací nabízí při streamování videa i jiný typ úpravy daného videa, nežli je klíčování pozadí, není nutné použít žádnou úpravu zdrojového materiálu. [1]

Druhá kategorie využívá zpracovaný obraz a jeho masku pro další zpracování. To lze provést segmentací objektů ve scéně, popřípadě využitím v rozšířené nebo virtuální realitě. Samotné klíčování je pouze jeden z prvních kroků, které jsou na scénu aplikovány, aby bylo možné se dostat k samotnému cíli zpracování. Jako příklad lze uvést zpracování virtuálních efektů ve filmech, kde je celá scéna točena před barevným plátnem a poté je herec vyklíčován. Následně je přidáno pozadí, prvky popředí a běžně i další herci, kteří se mají v daném záběru objevit. Na výslednou scénu je možné ještě aplikovat korekturu barev, aby herci lépe pasovali do scény. [1]

Virtuální realita je plně počítačem generovaný svět, kde se lze interagovat s předměty a může obsahovat další virtuální osoby, řízené jinými lidmi, popřípadě samotnou virtuální realitou. Cílem virtuální reality je vytvořit svět, který bude budit iluzi skutečného světa, ačkoliv je plně generovaný počítačem. Rozšířená realita pouze přidává předměty a prvky do skutečného světa. Lze tím upravovat viditelnou skutečnost nebo ji dodat v případě popisných textů lepší srozumitelnost. [1]

2.2.2 Chyby klíčování

Chyby při klíčování vznikají vždy, pokud není zvolen správný algoritmus, respektive metoda klíčování nebo je zvolená metoda špatně nastavena na klíčovanou scéna.

Pokud primární účel je klíčování, lze snadno odhadnout do jaké míry bude fungovat na potřebnou situaci. Jediný problém by mohl nastat, pokud zvolená metoda nepodporuje klíčování v potřebné barvě, případně odstínu. Další problém by mohl nastat při špatné nastavitelnosti nebo úpravě výsledku, popřípadě formě výsledku.

Zvolená metoda nemůže být jakkoliv nastavena. Nastavení nevyhovuje požadavkům nebo výsledný formát, který metoda vrací nevyhovuje dalším požadavkům.

Převážné množství chyb pramení ze snahy klíčovat scénu, která není ideální, popřípadě je velmi špatné kvality. V případě neoptimálně vyklíčované scény (Obr. 2) vznikají přímé nebo nepřímé chyby, scény se pak velmi špatně klíčují. Nejčastější chyby jsou spjaté s nasvícením scény, které klíčování značně ztěžují nebo až znemožňují.

a) b)

Obrázek 2: Příklad chyb při klíčování a) klíčování přechodu popředí a pozadí, příliš tmavá část v pozadí b) klíčování oblečení blízko klíčované barvě [1]

Nasvícení scény může způsobovat lehce tmavší části v podobě stínů na pozadí. Stíny nezpůsobují při klíčování velké obtíže, ale algoritmus musí být o něco více agresivní, což může způsobovat nepřímé chyby. Tyto chyby jsou způsobeny nutností klíčovat tmavší části v pozadí, tak i v popředí.

Problém nastává, pokud je scéna nasvícena tak špatně, že barva pozadí přechází přes několik odstínů barvy. Například je zřetelný přechod ze světle žluté přes žlutou a zelenou do tmavě zelené. V takovém případě je klíčování v podstatě nemožné, jelikož je odstraňovaná barevná oblast tak velká, že je téměř nemožné ji oddělit od popředí.

Velkým problémem pro klíčování je situace, kdy barva popředí a pozadí velmi úzce sousedí v barevném modelu. V takovém případě je pravděpodobné, že zvolená metoda bude odstraňovat i daný objekt patřící do popředí. Do této kategorie patří například světlé vlasy na zeleném pozadí. Velmi problematický moment nastává v případě přesvětleného pozadí, kdy začíná zelené pozadí pomalu přecházet do žluté.

V takovém případě se nedá klíčování popředí předejít, pouze nastavit jemnost klíčování na co nejvyšší úroveň, aby bylo popředí co nejméně zasaženo a zároveň byla odstraněna co největší oblast pozadí.

Obdobný problém je podobnost barvy popředí a pozadí z pohledu tmavosti pozadí. Oblast popředí je vizuálně odlišná od pozadí, ale v používaném barevném modelu jsou od sebe velmi blízko. Jediná odlišnost je jejich tmavost, popřípadě světlost. V tomto případě je možností ignorovat všechny části scény, které mají požadovanou úroveň světlosti a tmavosti.

Poslední kategorie problémů jsou ty, které spadají do problematické části popředí. Například se zeleným tričkem se klíčuje zelené pozadí. Za této situace je nemožné bez manuální úpravy zamezit vyklíčování dané oblasti. Stejný případ nastává při klíčování průhledných předmětů, které z pohledu scény nesou stejnou barvu jako je barva pozadí (Obr. 3). Pokud je zdrojová scéna velmi kvalitní, metoda klíčování dostatečně jemná a způsob jakým se klíčuje to podporuje, klíčování lze provést. V opačném případě je nutností výsledek upravit manuálně.

Obrázek 3: a) klíčovaný obrázek b) problémová oblast patřící do popředí a nesoucí barvu pozadí c) výsledek klíčování [1]

2.2.3 K-means

Seskupující nebo také klastrovací algoritmy jsou běžně používané pro uspořádání dat podle daných podobností. Algoritmus má pouze informace o datech a jejich vlastnostech, na jejichž základě je rozdělí do K skupin. Pokud je zachováno rozložení dat při

inicializaci, algoritmus vždy skončí se stejným výsledkem. Každá skupina bude mít stejné členy a stejný počet daných členů jako v jiných inicializacích. Kód 1

V prvním kroku se přiřadí hodnota ke každé centroidě, která představuje střed celého seskupení. Prvotní hodnotu lze generovat, což zapříčiní možnost, že výsledek bude se stejnými daty vždy nepatrně jiný, nebo nastavit centroidy na hodnotu nultou, tudíž všechny centroidy, středy skupin, budou v bodě nula. Toto řešení je časově delší na vypočítávání, protože trvá několik iterací, aby se nulté prvotní hodnoty vyrovnaly náhodným nebo výchozím hodnotám. [1]

```
Metoda K-Means( )
   Přiřazení hodnot k centroidě
   while (předchozí centroidy ≠ momentální centroidy)
      Rozřazení bodů do nejbližší centroidy
      Přepočítání hodnot centriod
   Ukončení
```

Kód 1: K-Means algoritmus

V následujícím kroku jsou zpracovány jednotlivé prvky, které jsou přiřazeny ke skupině centroidy, ke které mají nejblíže. Centroida je následně přepočítána (Rovnice 1), aby stále reprezentovala střed bodů dané skupiny. Tento proces se opakuje, dokud se nepřiřadí všechny hodnoty a centroidy se přestanou mezi iteracemi měnit. K tomuto stavu nemusí vždy dojít a místo ustálení se algoritmus přeruší po přesném počtu iterací, tak aby bylo možné s výsledky pracovat. [1]

2.3 Barevné modely

Hlavním cílem při klíčování je odstranění určité barvy představující pozadí. Pro tuto úlohu je nutné použít barevný model, který umožní plnit zadání co nejefektivněji a nejpřesněji. Určení hranice barvy je velmi komplikované dosáhnout v barevném modelu RGB a je lepší barvu transformovat do jiného barevného modelu, který se lépe hodí pro tyto účely. [1]

2.3.1 RGB barevný model

Barevný model RGB je běžně používán pro ukládání a práci s barvami v digitálním prostoru. Využívá se pro zobrazování v monitorech, při práci s grafickými kartami a shadery. Samotný barevný model se skládá ze tří složek. Každá složka reprezentuje barvu a její úroveň. Jednotlivé základní barvy jsou červená, zelená a modrá. Různou kombinací základních barev modelu lze vytvořit ostatní barvy. Existuje i podobný model s názvem RGBA, který je v podstatě stejný jako základní model RGB, ale má o jednu složku navíc, která nese údaj o průhlednosti. Všechny části modelu jsou v rozsahu od 0 do 255 nebo je jejich rozsah normalizovaný a je v rozsahu od 0 do 1. [1]

2.3.2 HSB barevný model

Jedním z barevných modelů, který je vhodný pro klíčování je barevný model HSB. Tento barevný model nabízí snadné rozpoznání tmavého a světlého odstínu dané barvy a jednoduché filtrování daného odstínu. Hlavní složka je v rozsahu od 0 do 360 a přestavuje jednotlivé barevné tóny. Zbylé barevné složky nabývají hodnot od 0 do 100. První udává sytost daného odstínu, respektive množství barvy daného odstínu, tudíž při malých hodnotách je výsledná barva skoro bílá. Druhá složka je jas. Tudíž při malých hodnotách je výsledná barva tmavá až černá. [1]

Tento barevný model je vhodný pro klíčování, kvůli snadné filtraci světlých a tmavých odstínů a hranice barev určuje pouze jeden prvek. Také tento barevný model je blízko k tomu, jak lidské oko vnímá barvy.

Transformace do barevného modelu HSB probíhá z normalizovaného modelu RGB. V prvním kroku se vypočítá největší a nejmenší prvek z RGB modelu a přiřadí se do proměnné Max a Min. Následně se aplikují vzorce, které vypočítají výsledný HSB model. [1]

Pokud $Max = R_{RGB}$, pak $H_{HSB} = 60(\frac{G_{RGB} - B_{RGB}}{Max - Min}) + 360 \mod 360$

Pokud $Max = G_{RGB}$, pak $H_{HSB} = (60(\frac{B_{RGB} - R_{RGB}}{Max - Min}) + 120) + 360 \mod 360$

Pokud $Max = B_{RGB}$, pak $H_{HSB} = (60(\frac{R_{RGB} - G_{RGB}}{Max - Min}) + 240) + 360 \mod 360$

Pokud $Max = 0$, pak $H_{HSB} = 0$

Pokud $Max = 0$, pak $S_{HSB} = 0$

Pokud $Max \neq 0$, pak $S_{HSB} = \frac{Max - Min}{Max}$

3 Možnosti využití grafické karty pro zpracování obrazu

Kvůli požadavku na rychlost je vhodné využít při zpracování grafického obsahu hardware GPU. GPU poskytuje dostatečně výkonný hardware pro zpracování grafického obsahu za pomoci shaderů.

Technologie GPU využívající hardwarovou akceleraci a programovatelné shadery by měla umožnit dostatečně rychlé zpracovaní jednotlivých snímků a docílit tak klíčovaní videa v reálném čase i pro vysoká rozlišení a vyšší snímkovou frekvenci. Je ale nutné navrhnout algoritmy a datové struktury vhodné pro shaderovou technologii a pro předávání dat do GPU. Jednotlivé snímky budou ukládány jako textury, ke kterým je možné na GPU přistupovat.

3.1 OpenGL

OpenGL je nejrozšířenější $2D$ a $3D$ grafický aplikační programovatelný interface (API), které je nezávislé na systému a umožňuje svou definicí snadnou čitelnost prováděných operací. Umožňuje vývoj softwaru bez ohledu na platformu a cílený operační systém. OpenGL je pouze specifikace názvů funkcí a parametrů přebírajících možný budoucí výsledek. Každá implementace této specifikace se může nepatrně lišit v různých maličkostech. Například jak se volají funkce, které OpenGL specifikuje nebo v jakém programovacím jazyce se daná implementace vyvíjí. [11]

3.1.1 Mipmap

Mipmap je sekvence textur, u kterých se postupně snižuje rozlišení dané textury. Šířka a výška textury je dvakrát menší nežli v předešlé textuře. Výsledek dělení je vždy zaokrouhlen dolů. Úvodní textura nemusí mít stejnou šířku a výšku, aby bylo možné vytvořit mipmapy. Mipmapy jsou reprezentovány jako řetězec. Hlavní textura je první v pořadí, následující úrovně jsou postupně vygenerovány jako navazující. Příklad vygenerovaných mipmap textur je na Obr. 4. Každá úroveň mipmap textury

navazuje na předešlou a následující úroveň navazuje až na nejmenší vygenerované rozlišení. [12]

Obrázek 4: Mipmap textura s hlavní texturou a vygenerovanými navazujícími texturami [2]

Nejvyšší úroveň textury se zobrazuje, když je pozorovací vzdálenost malá a s rostoucí vzdáleností se využívají stále nižší úrovně dané textury. Tato metoda je efektivnější na rozdíl od využití více textur se stále nižším rozlišením. [12]

Před využíváním mipmapů je nutnost je vygenerovat. Je možné si vybrat počet vygenerovaných úrovní mipmapů. Každá vygenerovaná textura má poloviční rozměry té předchozí, tudíž textura s rozměry 256 pixelů na 256, se zmenší na 128 x 128. Další následující budou mít rozměry 64 x 64 a tak dále. Nejmenší možný rozměr textury je 1 x 1 a následující mipmap textury již nelze vygenerovat. Také nelze vygenerovat mipmap textury, které přeskakují zmenšení. Vždy musí být vygenerovány všechny textury v pořadí, tudíž nelze vygenerovat například pouze nejmenší mipmap textura, bez všech předchozích textur. Mipmap se využívá pro snížení času při renderování scény a zvyšuje pocit reálného vykreslení scény, ačkoliv požadují při použití více alokované paměti. [12]

Potřebný počet úrovní mipmapu lze vypočítat za pomoci vzorce $log_2(max(w, h)) + 1$. Tento vzorec vypočítá počet úrovní, za předpokladu, že mi-

nimální počet textur je alespoň jedna, tudíž minimálně základní textura. V OpenGL je index základní textury 0, proto je nutné při vyhledání textury odečíst 1 nebo využít vzorec $log2(max(w, h))$. Výpočet maximálního počtu mipmapů není důležitý, jelikož základní hodnota u příkazu pro generování mipmapu je 1000, což je výrazně více nežli vyžaduje jakákoliv běžně využívaná textura. [13]

3.1.2 Textura

Barva textury v OpenGL je ukládána ve formátu RGBA. To znamená, že každá barva se skládá z červené (R), zelené (G), modré (B) a alfa kanálu. Alfa kanál řeší viditelnost nebo průhlednost dané barvy. Možnost je také uložit texturu obsahující pouze jeden barevný kanál nebo různé kombinace.

Vnitřní formát textury udává, jak jsou data na GPU uložena. Od této hodnoty se odvíjí přesnost ale i rozsah, ve kterém lze s daty se shadery pracovat.

Hodnoty jednotlivých prvků lze ukládat třemi způsoby - normalizovaný integer, float nebo integer (Tab. 3). Normalizovaný integer a float způsobí, že hodnota barvy na shaderu je ve vektoru floatů, zatímco integer nabývá hodnot ve vektoru integerů. [4]

Tabulka 3: Vnitřní formát textury a jeho rozsah [4]

Typ	Rozsah	Počet bitů v paměti
Normalizovaný integer (bez znaménka)	[0, 1]	2, 4, 8, 16
Normalizovaný integer (se znaménkem)	[-1, 1]	8, 16
Integer (bez znaménka)	[0, Max]	8, 16, 32
Integer (se znaménkem)	[-Max, Max]	8, 16, 32
Normalizovaný float	[-1, 1]	16, 32

Při práci s hodnotami barev na shaderu, lze přiřadit jednotlivým hodnotám jakýkoliv význam. Může se s hodnotami pracovat jako s coordináty textury, normálou, popřípadě vypočtenou hodnotou. Jedná se o čísla, která lze využít libovolným způsobem. [4]

Obraz nemusí obsahovat všechny složky dané barvy, ačkoliv textura se bude skládat ze čtyř složkového RGBA vektoru. Komponenty, které obraz neobsahuje jsou vyplněny automaticky. Nulová hodnota je nastavena pro chybějící R, G nebo B složku a maximální barevná hodnota se nastaví pro chybějící alfa kanál. [4]

3.1.3 Render Target

Framebuffer Objekt (FBO) je v OpenGL objekt, který umožňuje své využití jako Framebuffer (FB), do kterého lze renderovat mimo hlavní obrazový výstup, aniž by byl nějak změněn. Výsledek se zapíše do jiné scény mimo tu hlavní, následně s ní lze dále pracovat. Jako objekt tohoto typu může být využit buď obrázek (Image), popřípadě textura (Textura). Při vytváření FBO musí být stanoven typ a adresa, do které je následně daný objekt přiřazen. Jsou čtyři typy, pod kterými může být obrázek přiřazen. [14]

1. `GL_COLOR_ATTACHMENTi` - i je pořadí nebo adresa daného objektu. Minimální počet je 8 tudíž i nabývá hodnot od 0 do 7 a v závislosti na implementaci může být hodnot více. Tomuto typu může být přiřazen pouze objekt, který obsahuje možnost renderování barev.

2. `GL_DEPTH_ATTACHMENT` - Mohou být přiřazeny pouze objekty, které podporují renderování hloubky a přiřazením se objekt stává Depth Buffer pro dané FBO. Pokud tento typ není k FBO přiřazen, je při renderování automaticky vypnuto testování hloubky (Depth Testing).

3. `GL_STENCIL_ATTACHMENT` - Mohou být přiřazeny pouze objekty, které podporují renderování šablony a přiřazením se objekt stává Stencil Buffer pro dané FBO.

4. `GL_DEPTH_STENCIL_ATTACHMENT` - Podpora pro předchozí dva zároveň.

Po vytvoření FBO je doporučeno zkontrolovat, jestli byl FBO vytvořen úspěšně funkcí $glCheckFramebufferStatus(GLenumtarget)$. Pokud funkce nevrátí status o úspěšném vytvoření je FBO špatné a nelze s ním dále pracovat. [14]

FB je kolekce části paměti (buffer), která se využívá jako cíl pro render scény. OpenGL obsahuje dva druhy FB. [15]

1. Výchozí FB, který poskytuje OpenGL kontext a většinou reprezentuje okno popřípadě obrazovku zařízení, kde se daný FB zobrazuje.

2. Uživatelské FB objekty, které se využívají jako cíl renderu a nikdy nejsou přímo viditelné.

Stejně jako při vytváření jiných objektů v OpenGL je FB přiřazena adresa, odkaz na daný objekt. Před použitím se musí svázat příkazem (Bind). Posléze je možné s ním pracovat a slouží jako cíl renderu, kam se zapíše výsledek provedených operací. Pokud se využije výchozí FB, lze výsledek vidět v okně OpenGL. Při renderu do FB objektu není vidět žádná změna v okně a je nutno s daným výsledkem renderu dále pracovat, respektive přerenderovat obsah FB objektu do výchozího FB. [15]

Textura je OpenGL objekt, který obsahuje jeden nebo více obrázků (image), které jsou všechny stejného formátu. Texturu lze použít dvěma způsoby. Může být použita jako zdroj dat pro shader a operace v něm nebo jako cíl renderování. Textura obsahuje jeden nebo více snímků jako pole pixelů s určitými vlastnostmi jako jsou dimenze, velikost nebo formát. Všechny snímky v textuře musí mít stejné vlastnosti. [16]

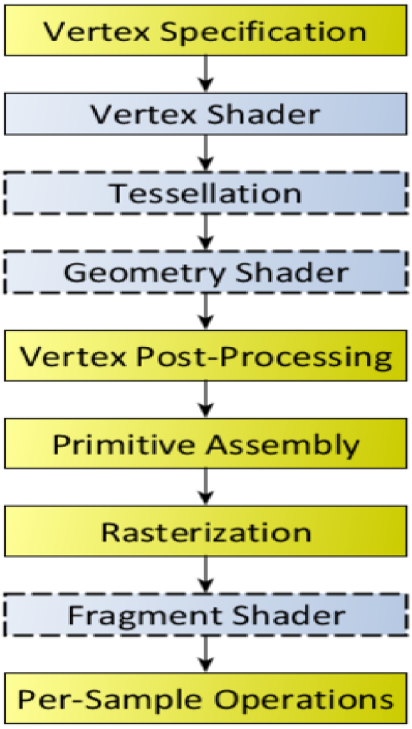
Každá textura má parametry, které definují její vlastnosti a upravují možnosti práce s danou texturou. Lze upravovat filtrování, velikost mipmap a nebo další vlastnosti. [16]

RenderBuffer Objekt (RBO) jsou vytvořeny a použity specificky pro FBO a jsou optimalizovány pro použití jako cíl renderování, zatímco textury tuto výhodu mít nemusí. RBO na rozdíl oproti textuře neumí vzorkování ze zpracovaného snímku, tudíž lze použít pouze pokud není při daném řešení vzorkování potřeba, v opačném případě lze použít texturu. [17]

3.1.4 Shader

Shader je uživatelsky definovaný program vytvořený pro běh na grafické kartě. Každý ze shaderů je využit v programovatelných částech v rendrovací posloupnosti. [18]

Posloupnost renderování definuje danou sekci, kterou lze shaderem programovat. Každá sekce má vstupy a výstupy, které se následně předávají do další sekce. Není podmínkou, aby byla následující sekce naprogramována. Příklad renderovací posloupnosti na Obr. 5. [3]



Obrázek 5: Vizualizace renderovací posloupnosti, jak jsou jednotlivé shadery postupně prováděny [3]

Každý shader v OpenGL lze programovat v jazyce OpenGL Shading Language (GLSL). Jedná se o hlavní shaderový jazyk, ačkoliv díky rozšíření OpenGL (OpenGL Extension) lze využít i jiné shaderové jazyky. Samotný GLSL připomíná svým stylem programovací jazyk C. Přebírá koncept podmínky a opakující se smyčky, ale má také mnoho rozdílů. [18]

Z hlediska GLSL je shader pouze kompilovaný textový řetězec pro danou programovanou sekci. Zatímco program se bere jako více naprogramovaných shader sekcí při sobě. GLSL má nepřeberné množství verzí, které lze využít při programování shaderů. Momentální verze GLSL je 4.60. [18]

Při vytváření shaderu je nutno uvést jeho typ, obvykle za pomoci enumu nebo konstanty. Renderovací posloupnost se skládá z následujících shaderů [18]:

- Vertex Shader (VS)
- Tessellation Control Shader (TCS) a Tessellation Evaluation Shader (TES)
- Geometry Shader (GS)
- Fragment Shader (FS)
- Compute Shader (CS)

VS je shader zpracovávající jednotlivé vrcholy scény. Shader obdrží jednotlivě vrchol, který může nějak upravit a následně ho uloží do výstupu. Během tohoto procesu nelze vytvářet nové vrcholy, přesněji aby během jednoho cyklu VS uložil dva vrcholy do výstupu. Popřípadě, aby VS do výstupu nic neuložil nebo již uložené výstupy smazal. [19]

TCS je shader přebírající vypočítané vrcholy za pomoci VS. Pracuje s takzvanými patches, které vytváří v závislosti na nastavení a vstupních datech. Výstup je ve formě dat pro jednotlivé vrcholy nebo pro jednotlivé cesty (patch) a je následně předán TES.

TES je shader, který zpracovává vstupní údaje cest a následně vypočítává finální vrcholy vygenerovaných primitiv reprezentující primitiva v celém procesu cest.[20]

GS je shader přebírající jednotlivá primitiva jako vstup. Výstup může být více nebo žádné primitivum. Typ primitiva je definován v shaderu a musí odpovídat příchozímu poli vrcholů. Pokud je teselace zapnutá, typ primitiv musí odpovídat typu, který je definován v TES. [21]

FS je shader, který zpracovává fragment vygenerovaný rasterizací. Rasterizace je proces, kdy jsou primitiva přetvořena na fragmenty. FS pracuje vždy s jedním fragmentem jako vstupem a výstupem je jeden fragment. Výjimkou je funkce Discard, kterou lze zpracovávaný fragment zahodit, tudíž se daný fragment neobarví. Výstup se skládá z vypočítané barvy výstupového fragmentu včetně průhlednosti a údaje o hloubce. Výsledný fragment lze zakreslit do více render targetů vždy s jinou barevnou hodnotou. [22]

CS je shader používaný pro výpočet libovolných informací, který není začleněn do renderovací posloupnosti. Shader samotný nemá výchozí vstup a výstup a je nutno ho uživatelsky definovat. [23]

3.2 Alternativy

Pro práci na GPU se využívá API. Nejznámější je implementace OpenGL, DirectX a Vulkan. Všechny zmíněné umožňuji jednoduchou práci s grafickou kartou a všechny nutné orchestrace.

Vulkan a DirectX umožňují v podstatě stejné operace jako OpenGL. Vulkan je otevřené GPU API, umožňující vývoj softwaru nezávisle na platformě. Na druhou stranu DirectX je uzavřená platforma poskytující dané API a je dostupná pouze pro operační systém Windows. [1]

4 Návrh algoritmů

Cílem diplomové práce je navázat na bakalářskou práci Segmentace objektů ve videu s využitím klíčování barevného pozadí [1]. Převzetí algoritmu na bázi K-Means a jeho implementaci na grafické kartě za pomoci shaderů. Cílem není praktické využití dané implementace, ale návrh GPU implementace a její odzkoušení. Hlavní implementace bude provedena na rozdíl od bakalářské práce v programovacím jazyce C#. Kvůli možnosti finálního porovnání bude implementace také vytvořena v jazyce Java, aby bylo možné porovnat implementaci a její výsledky z bakalářské práce.

Hlavní důraz je kladen na rychlost při zachování kvality, kterou nastavila bakalářská práce. Při měření rychlosti bude provedeno měření každého kroku v dané iteraci, aby bylo možné porovnat výsledky s bakalářskou prací, která výpočet prováděla jiným způsobem.

Cíl je dosáhnout zpracování videa o rozlišení Quad HD, což je 2560 x 1440. Toto rozlišení by mělo být možné klíčovat v reálném čase, při zachování standardních šedesáti snímků za sekundu. Snahou je dosáhnout stejného výsledku i se zdrojovým videem o rozlišení 4K (3840 x 2160). Cílem je zpracování všech snímků stejným algoritmem, bez nutnosti nějaké snímky přeskakovat, kvůli neschopnosti zpracovat šedesát snímků za sekundu.

Primární je celý výpočet shlukového algoritmu přesunout na GPU. Přesun by měl způsobit urychlení celého procesu, a tudíž možnost přepočítávat pozice centroid a následně za pomoci toho při zpracování každého snímku klíčovat.

Pokud nebude možné vypočítávat nové centroidy při zpracování každého snímku, budou provedeny potřebné optimalizační kroky, které umožní přepočet centroid v průběhu videa. Podmínkou implementace není výpočet centroid každý snímek, pouze provádět tento výpočet průběžně, například každý n-tý snímek.

Pro celou implementaci a její rychlost je nutnost analyzovat implementaci použitou v bakalářské práci. Zjistí se limity dané implementace a vyzkouší se možné optimalizace a přístupy za pomoci, kterých se značně urychlí proces klíčování, respektive proces rozhodnutí, která část patří do oblasti popředí nebo pozadí.

4.1 Návrh implementace

Inicializace algoritmu a jeho chod probíhá podle životního cyklu dané implementace OpenGL. Implementace je rozdělena na inicializační část (metoda *init*), ve které je možné načíst potřebné prostředky nebo textury a inicializovat například shadery a je spuštěna pouze jednou na začátku celého životního cyklu implementace. Výkonná část (metoda *display*) řeší samotné zobrazování a aktualizování scény. Tato funkce běží v nekonečném cyklu, dokud není onen cyklus buď z kódu nebo uživatelem ukončen. Poslední je uvolňovací část (metoda *dispose*) a probíhá na samém konci životního cyklu. Tato metoda má za úkol vyčistit operační paměť po renderování. Vymaže všechny dočasné soubory na disku, které mohly během procesu vzniknout. Dále uvolní všechny prostředky a zdroje, které byly před nebo během vykreslování alokovány a nachází se stále v paměti.

Běh celého algoritmu proběhne ve čtyřech krocích Kód 2. Načtení momentálního snímku ve videu, který se bude následně zpracovávat. Poté se rozdělí za pomoci shaderů na GPU snímek na 3 oblasti. Počet oblastí se odvíjí od počtu centroid. Tyto oblasti budou složeny vždy z nejbližších barevných bodů v daném barevném modelu k určité centroidě. Následně se vypočtou nové hodnoty centroid, jako průměrná hodnota všech bodů obsažených v dané skupině. Hodnoty centroidy budou nastaveny na výchozí na začátku zpracování a dále budou předávány mezi iteracemi zpracování.

```
Metoda RENDER( )
   Načtení snímku z videa
   Aplikace shaderů a rozdělení scény na oblasti dle centroid
   Vypočtení nových centroid dle rozdělení scény
   Odstranění scény za pomoci nových centroid
   Načtení dalšího snímku
```

Kód 2: Návrh implementace metody, zpracovávající jednotlivé snímky videa

V posledním kroku bude odstraněna oblast pozadí určená danou centroidou. Tento výsledek bude zobrazen na obrazovce. Vzápětí se začne zpracovávat další snímek.

V případě nutnosti bude celý proces upraven, aby lépe vyhovoval potřebám optimalizace. A hlavnímu cíli této práce, což je zpracování videa v reálném čase. Snaha je optimalizovat celý proces, aby byl hlavní cíl splněn, aniž by bylo nutno degradovat kvalitu videa zahazováním snímků nebo snižováním jejich počtu.

4.2 Testovací zdroje

Implementace a testování proběhne na stejném videu jako v bakalářské práci. Toto video je optimální pro testování a práci s ním, kvůli jeho vlastnostem. Vlastnosti scény se během celé délky videa výrazně nemění, přesto poskytuje mnoho možností na otestování správnosti a funkčnosti implementace.

Dané video je poměrně krátké. Obsahuje postavu, jejíž oděv je svou barvou velmi blízko barvě pozadí v barevném modelu HSB. Navíc se za postavou nachází různé stíny na pozadí a v okraji je tmavá oblast.

Poslední vlastností videa je jeho snímkování, které dosahuje 60 snímků za sekundu. Pomocí této vlastnosti bude možné vyzkoušet zpracování a následné vykreslení na obrazovku, aniž by se výsledný obraz zdál zrychlený.

Sekundární video bude použito jako test stability. Obsahuje pouze modrou barvu pozadí a bílý text jako popředí. Toto video má menší snímkování nežli primární video, ale je výrazně delší. Díky této vlastnosti bude možné vyzkoušet schopnost implementace zpracovávat dlouhá videa a stabilní vypočítávání a určování pozadí.

Tabulka 4: Vlastnosti použitých videí

Typ videa	Délka	Snímků za sekundu	Celkový počet snímků
S postavou	6,5 s	60	395
S nápisem	28,9 s	30	869

Obě videa mají stejné rozlišení Full HD, které je relativně nízké (Tab. 2). Ačkoliv to v průběhu implementace nebude překážkou, pro potřeby testování a případné optimalizace a úpravy je nutné vytvořit daná videa ve větším rozlišení.

První možností je vytvořit stejná nebo alespoň obdobná videa pouze ve větším rozlišení. To s sebou nese několik problémů. První problém vyžaduje nové otestování výkonnosti na projektu k bakalářské práci. To musí být provedeno stejně kvůli změně hardwaru, tak aby bylo možné prezentovat výsledky. Hlavní problém je složitější a náročnější na pořízení dalších testovacích materiálů, konkrétně nahrání videa v 4K rozlišení s 60 snímkách za sekundu.

Druhá možnost je využití nástroje, který vytvoří ze zdrojového videa s nízkým rozlišením video v rozlišení 4K. Toto řešení bylo pro svou jednoduchost zvoleno.

Na zvětšení testovacích videí byl použit program VideoProc Converter, který umožňuje zpracování videa a jeho zvětšení. Tímto způsobem byla vytvořena pro každé z obou videí verze v rozlišení QuadHD a 4K.

5 Implementace algoritmů

Algoritmus pro implementaci byl zvolen na bázi shlukového algoritmu K-Means. Z výsledků bakalářské práce vyplývalo, že jeho potenciál nebyl plně využit a rozvinut. Dále byl implementován výsečový algoritmus pracující ve stejném barevném modelu jako shlukový algoritmus. Tento algoritmus je pouze derivátem shlukového algoritmu, který vyžaduje uživatelské nastavení, pro svůj správný chod. K-Means algoritmus byl také implementován do původního projektu použitého v bakalářské práci, aby bylo možné porovnat způsoby implementace.

5.1 Použité knihovny

Implementace probíhala ve dvou programovacích jazycích. Tudíž se v každé dané implementaci použité knihovny lišily. Obecně se využívá knihovna implementující OpenGL v dané jazykové iteraci a knihovna pro správu a načtení používaného videa.

Java implementace byla využita ve verzi, v jaké byla implementována v bakalářské práci, jedinou změnou bylo zlepšení *jdk* na verzi 20.0, která byla v době implementace nejnovější. Použité knihovny jsou OpenCV pro načítání videa a rozklad na jednotlivé snímky. Pro práci s transformacemi byl využit balíček tříd Transforms, využívaný pro předmět počítačová grafika $1-3$ na FIM UHK. Implementace OpenGL pro Javu byla zvolena knihovna LWJGL.

C# implementace využívá nejnovější verzi frameworku .NET ve verzi 7.0, což byla nejnovější verze v době implementace. Knihovna ImageSharp je využívána pro načítání a práci s obrázky. Nástavba AVCodecFormats byla upravena a využita pro načtení videa. Alternativní knihovna pro práci s videem byla využita implementace OpenCV v jazyce C# jménem EmguCV. Implementace OpenGL v dané jazykové mutaci byla zvolena knihovna Silk.NET. Poslední využitá knihovna s názvem BenchmarkDotNet byla použita pro otestování rychlosti dané implementace.

Inicializace algoritmů je ve svém jádru velmi obdobná bez ohledu na jazykovou verzi a typ implementace Kód 3. Oba implementované algoritmy potřebují pro svoji

činnost rastrový obraz, který budou klíčovat a obraz, který nahradí pozadí. Na shaderu nebo v programu lze nastavit barvu, kterou se pozadí bude nahrazovat. Na shaderu můžeme také nastavit viditelnost na nulovou hodnotu daným fragmentům patřícím do pozadí a tím je zprůhlednit.

```
Metoda INIT( )
    Načtení grafického API
    Vytvoření kreslících bufferů
    Načtení shaderu
    Načtení textury pozadí
    Načtení klíčovaného videa
```

Kód 3: Inicializační metoda

Přístup k vyklíčované oblasti byl zvolen stejný jako v bakalářské práci. Pozadí je vždy nahrazeno texturou pozadí. Textura pozadí poskytuje potřebný kontrast mezi vyklíčovaným pozadím a zachovaným popředím. Z optimalizačního hlediska je lepší pozadí určovat za pomoci statické barvy nebo za pomoci průhlednosti. V obou případech se řešení vyhýbá práci s texturou navíc.

Implementace OpenGL v C# s názvem Silk.NET využívá možnost delegátů funkce. Je tedy možné na místo implantace každé funkce, která se během životního cyklu aplikace na pozadí spouští, přiřadit pouze metody, které jsou potřeba pro požadovaný běh aplikace. Díky tomuto mechanismu není nutné vykreslovací část implementovat. Popřípadě vynechat zapnutí životního cyklu, který se stará běžně o vykreslování. Pouze inicializovat okno, do kterého se vykresluje a funkce starající se o funkcionalitu zapínat v aplikaci manuálně, respektive bez využití frameworku. Tato funkcionalita byla využita při testech rychlosti načítání textur.

Prvním krokem je vytvoření okna. Okno lze vytvořit pomocí výchozího nastavení nebo dané nastavení upravit. Dále se přiřadí implementované funkce, které jsou po zapnutí volány. Následně je funkcí *Run*() okno inicializováno a začne běžet jeho životní cyklus Kód 3. První funkce se spouští pouze při zapnutí a slouží pro načtení potřebných zdrojů. V této funkci se načítá video, které se bude klíčovat. Dále textura pozadí, což je v případě tohoto projektu pouze jednobarevná červená textura.

Poslední se načte využívaný shader a vytvoří se buffer určený pro vykreslování. Po inicializaci lze získat grafické API, které zajišťuje komunikaci s grafickou kartou a lze s jeho pomocí volat a provádět OpenGL funkce. Toto grafické API je potřebné pro vytvoření a úpravu jakéhokoliv vytvořeného zdroje.

```
Metoda RUN( )
   Init()
   while (!Closing)
      Render()
   Dispose()
```

Kód 4: Spustitelná metoda

Po inicializaci začíná nekonečný cyklus. V tu chvíli se volají dvě funkce určené pro aktualizaci zdrojů a vykreslení, popřípadě překreslení scény. Tento nekonečný cyklus se přeruší pouze při požadavku na zavření okna. Po ukončení cyklu se volá funkce pro uvolnění všech zdrojů. Zde se uvolní instance videa, shaderu, bufferu a shaderu z paměti.

5.2 Abstrakce použitých tříd

Knihovna Silk.NET implementuje pouze OpenGL volání, přesto je dobré vytvořené entity nějakým způsobem zapouzdřit pro snadnější znovupoužitelnost a obecnou práci s nimi. Některé využívané třídy byly získány z ukázkového projektu, který se vyskytuje v dokumentaci ke knihovně. Tyto třídy byly převzaty a vhodně upraveny, aby se hodily pro potřeby tohoto projektu.

Hlavní třída nese název BaseGLClass. Tato třída nese proměnnou třídy GL, což je grafické API a požaduje tuto proměnnou v konstruktoru. Takto lze snadno zabezpečit abstrakci nad všemi zděděnými třídami, které budou vyžadovat předání dané třídy GL v konstruktoru. Toto řešení není v rámci projektu potřeba, pouze usnadňovalo implementaci ostatních tříd a snížilo nutnost opakovat stejnou část kódu. Všechny třídy implementující abstrakci nad OpenGL implementují základní BaseGLClass třídu.

Třídy BufferObject, VertexArrayObject a BrawBuffer společně vytvářejí buffer pro vykreslování. Nastavení a inicializace byly ponechány z ukázkového projektu. Výchozí nastavení vykreslí za pomoci dvou trojúhelníků obrazovku. Toto chování pro potřeby projektu bylo dostačující a hodnoty daných tříd byly pouze zapouzdřeny pro snazší znovupoužitelnost a přehlednost. Využití je kvůli tomuto kroku velmi jednoduché. V konstruktoru je vytvořen kreslící buffer, který inicializuje vše potřebné pro svůj chod. Poté lze v případě potřeby překreslení obrazovky, nastavit příkazem *Bind*() a následně zavolat OpenGL příkaz pro vykreslení.

```
BaseGLClass
├── BufferObject
├── VertexArrayObject
├── BrawBuffer
├── Shader
├── BaseTexture
├── Texture
│   └── AvgTexture
├── BaseRenderTarget
│   └── RenderTarget
│       └── AvrRenderTarget
└── BaseVideo
    ├── SLVideo
    │   └── AvrSLVideo
    ├── EMGUVideo
    │   └── AvrEMGUVideo
    └── VLVideo
        └── AvrVLVideo
```

Třída Shader je určena pro načítání a práci se shadery. V konstruktoru je předáno jméno shaderu, které je načteno. V této implementaci se načítá pouze VS a FS. Pro momentální potřeby projektu je to dostačující. Třída dále nabízí funkce umožňující nastavit uniform proměnné do daného shaderu.

5.2.1 Textura

Zbylé třídy jsou implementovány pomocí abstraktní třídy, která poskytuje dané třídě základní implementaci a následně je daná abstraktní třída implementována jednou

nebo více možnými implementacemi. Tyto implementace pouze přebírají abstraktní třídu, popřípadě rozšiřují její funkce o další proměnné nebo metody.

Abstraktní třída BaseTexture a její implementační třída Texture zaštiťují operace vytváření a úpravu textury. Třída umožňuje nahrání obrázků pomocí souboru, popřípadě již načtených počítačových dat. Důležité je vždy zadat interní formát dat, které bude daná textura na grafické kartě mít. V případě vytváření za pomoci ukazatele na data v paměti je možné zadat formát daných dat. Pokud není formát dat zadán, je výchozí hodnota RGBA.

Parametry textury jsou výška a šířka. Dále interní formát a formát dat, které byly použity při inicializaci. Poslední parametry, které textura má, se vážou k výpočtu průměrné barvy dané textury. Proměnná TotalMipmapLevels udává, kolik mipmap úrovní má daná textura při své výšce a šířce. Poslední proměnná je AvrgColor, což je průměrná barva textury. Ta se vypočítá automaticky při deklaraci, ale při změně textury je ovšem nutné tento údaj přepočítat.

Textura implementuje pouze několik metod. Základní metoda *Bind*() umožňuje nastavit danou texturu pro další práci s ní. *Dispose*() umožní smazání dané textury. Dále Textura umožňuje přepočítání své průměrné barvy, tak aby byla aktuální k jejímu obsahu. Poslední metoda umožňuje za pomoci odkazu na data v paměti změnit obsah dané textury. Metoda je vhodná k optimalizaci, jelikož mění obsah již existující textury na grafické kartě. Tento přístup by měl být tudíž rychlejší než vytvářet vždy texturu novou a starou mazat. Tohoto se dá využít, pokud se s texturou ještě nějak pracuje mimo hlavní renderovací posloupnost, a je nutné mít vždy před renderem nově vytvořenou texturu.

V rámci optimalizace vznikla třída AvgTexture, která dědí z třídy Texture. Místo původní třídy implementuje metody týkající se výpočtu a práce s průměrnou barvou. Původní třída vyvolá výjimku, pokud jsou zavolány metody, které nikterak neimplementuje a proměnná reprezentující průměrnou barvu dané textury má výchozí nulové hodnoty.

5.2.2 Render target

Abstraktní třída BaseRenderTarget se svými dvěma implementačními třídami zajišťuje vytváření a práci s možným pomocným cílem renderování. Třída umožňuje vytvořit render target o určité velikosti, interního formátu a počtu textur, které bude takto vytvořený render target obsahovat.

BaseRenderTarget třída obsahuje parametry o výšce, šířce a počtu textur, které obsahuje. Zbylé dva parametry se stahují k texturám. Jeden je pole samotných textur, které v případě použití slouží jako plátno místo obrazovky. Druhý je pole s bufferem a informací o vykreslení. Za pomoci tohoto bufferu se ze shaderu zapíší výsledky dle pozice do příslušných textur render targetu.

Tato třída implementuje pouze základní metody. *Bind*() nastaví daný render target jako cíl následujícího renderování. *Dispose*() smaže všechny prostředky, nejdříve všechny textury, které obsahuje a následně celý render target. Poslední metoda je *UnBind*(), která umožňuje přepnutí renderování zpět do původního okna.

BaseRenderTarget má dvě implementační třídy. Třída RenderTarget pouze implementuje vše co jeho rodič, nic nového nepřidává. Druhá implementace dědí z té první a rozšiřuje implementaci o metodu, která přepočítá průměrnou barvu všem texturám v dané třídě a vrátí výsledek ve formě pole vektorů. Dané pole je stejně velké jako počet textur v dané RenderTarget.

5.2.3 Video

Implementace videa je nejkomplikovanější a její abstraktní třída BaseVideo nenabízí žádnou základní implementaci pouze abstraktní metodu a výchozí proměnné. Proměnné jsou Texture, což reprezentuje momentální načtený snímek ve videu. InternalFormat je údaj o interním formátu, ve kterém pracuje textura daného videa. Poslední proměnná FramePosition je údaj o aktuální pozici ve videu, přesněji kolikátý snímek z videa je načten. Metoda *NextFrame*() je jediná funkce, která posune video o snímek kupředu a načte daný snímek do textury.

První implementace videa je za využití knihovny Emgu.CV. Použitá knihovna se stará o veškerou práci s videem. Proměnná VideoData je třída představující zpracovávané video. Při inicializaci se zavolá načtení dalšího snímku. Při načtení snímku se snímek načte do operační paměti (RAM) a poté se vytvoří nová textura, kterou třída obsahuje, následně se tato textura pouze upravuje. Pokud se nepovede načíst následující snímek je vytvořena nová třída pro zpracování videa a následně se video začíná zpracovávat od začátku. Toto načítaní videa využívá finální shluková implementace EMGU. Název je odvozen od využité knihovny pro načítání videa Emgu.CV.

Druhá implementace videa využívá knihovnu ImageSharp. Použitá knihovna pouze umožňuje nahrát snímky daného videa do paměti. Proměnná VideoData je třída představující nahrané video v paměti. Při inicializaci se vytvoří nahraná textura. Následně se prochází snímky podle momentálního snímku, který je nahrán. Toto načítaní videa využívá finální shluková implementace SL. Název je odvozen od využité knihovny pro práci s obrázky SixLabors.ImageSharp.

Třetí implementace videa využívá stejný mechanismus jako knihovna ImageSharp, ale implementuje ho z pomocné třídy VideoLoader. Tato třída umožňuje načíst vždy pouze daný snímek a nemusí se celé video držet v paměti. Jinak se neliší od ostatních implementací videa. Toto načítaní videa využívá finální shluková implementace VL. Název je odvozen od využité třídy starající se o načtení snímků VideoLoader. Implementace WF využívající hodnoty nastavené uživatelem pro klíčování využívá také tento mechanizmus načítání videa. Název implementace je však odvozen od knihovny Windows Forms, která umožňuje uživateli měnit hodnoty při běhu aplikace.

5.3 Problém s rychlostí předchozí implementace

V průběhu implementace bylo nutné vyřešit problém s předchozí implementací, která vznikla k bakalářské práci. V té implementaci se pracovalo s videem s rozlišením Full HD. Toto rozlišení je z pohledu cíle této práce nízké, jelikož cílem je zpracovávat rozlišení 4K, které obsahuje čtyřikrát více barevných bodů nežli rozlišení zpracovávané (Tab. 2).

Výsledky bakalářské práce byly takové, že pokud by se zahazovali některé snímky, bylo možné zpracovat použité video v reálném čase. Navíc se ještě nepočítá s velmi dlouhou inicializací, která byla nutná pro správné fungování implementace (Tab. 5).

Tabulka 5: Výsledky implementace bakalářské práce [1]

Typ operace	Čas zpracování
K-Means inicializace	13 398 ms
K-Means zpracování	11 294 ms
Výsečový algoritmus	11 452 ms

Bylo nutné prozkoumat tuto implementaci a zjistit, které kroky vedly k těmto výsledkům. Dále bylo potřebné tato zjištění aplikovat v nové implementaci v jazyce C#.

5.3.1 Výsledky analýzy implementace

Inicializační část se využívá pro načtení všech potřebných zdrojů. Následně se provádí vzorkování a výpočet K-Means algoritmu. Tato pasáž byla velmi časově náročná, ačkoliv se dle předpokladů mělo vše odehrávat velmi rychle.

Problém v dlouhé době zpracování je v samotném vzorkování. Na začátku se získají data reprezentující první snímek videa. Následně jsou získaná data odeslána na GPU, čímž se vytvoří v paměti GPU objekt dané textury. Jádro problému je, že pro získávání vzorků se využívá tento objekt textury. To způsobuje v dané implementaci, že pokaždé kdy je vzat vzorek barvy, musí se stáhnou textura z GPU a přemístit data do paměti, kde s nimi již dokáže pracovat CPU.

Řešení v tomto případě je přiřadit data snímku do proměnné a pracovat posléze s nimi. V rámci optimalizace jsou data načtená z videa před odesláním na GPU. Stahování dat z GPU způsobuje zpomalení, jelikož se čeká na přesun dat mezi paměťmi GPU a CPU, přesněji RAM.

Toto řešení způsobí velký rychlostní růst inicializační části, už není nutné čekat několik sekund před úpravou, nyní tato část trvá v řádech stovek milisekund. Tato

jednoduchá optimalizace, dokázala celý výpočet K-Means prováděný na CPU, výrazně urychlit.

Další část je část zobrazovací, zajišťující dva kroky. První je posunutí snímku ve videu a druhá operace je samotné vyklíčování a vykreslení na obrazovku. Klíčování a vykreslení není časově náročné. Během testování bylo možné nastavit shader a provést vykreslení v řádu milisekund, tudíž hlavní časová ztráta je při načítání a práci s texturou.

Získání snímku a následné vytvoření, popřípadě úprava textury, zabírali v podstatě celý čas zpracování. Textura je definována s vnitřním formátem $RGBA32F$. Tento formát používá 32 bitů ve formě floatu s desetinnou čárkou pro každý člen barevného modelu. Toto zjištění vedlo k nutnosti vyzkoušet jiné formáty pro ukládání textury.

Hlavním problémem bylo získávání snímku. Tento úkon byl velmi náročný na čas, ačkoliv byla využita specializovaná knihovna, která by toho měla být bez problému schopna. Toto zjištění vedlo k nutnosti implementace možné alternativy.

5.4 Nutné vylepšení

Bylo nutné otestovat možné alternativy a následně výsledky použít pro snadnější načítání snímku videa, překlopení snímku na GPU a hlavně rychlejší práci s texturou. Toto se týkalo převážně vnitřního formátu textury. Vznikl předpoklad o možném zrychlení procesu. Toto se týkalo hlavně změny formátu, kde se nebude alokovat tolik bytů dat na jeden kanál barevného modelu.

Hlavní úskalí v bakalářské práci přinášelo načítání videa. Tudíž vznikly v této práci další dva alternativní přístupy, jak ho načítat.

První přístup (SL) je držet celý obsah v RAM. Toto řešení není vhodné pro velká, popřípadě dlouhá videa, ale reprezentuje možnost, že potřebný snímek se bude nacházet v RAM. V reálném řešení, lze vytvořit mechanismus, který dané video bude přednačítat a následně se bude s daty pouze pracovat. Tento princip se dá skloubit s ostatními řešeními, které načítají video postupně.

Druhý přístup (VL) využívá stejný princip jako stávající řešení v bakalářské práci. Rozdíl je pouze v použité knihovně a řešení, jak se jednotlivé snímky načítají. Toto je hlavně z důvodu možného špatného využití původní knihovny, která se stará o načítání snímků.

5.5 Algoritmus klíčování

Algoritmus klíčování byl přesunut na shadery. Z této změny bylo nutno provést několik úprav. Hlavní spočívá v načítání snímku videa a následného vypočítání hodnot využívaných centroid. Na základě těchto hodnot se snímek klíčuje.

5.5.1 Výpočet centroid

Hodnota využívaných centroid se při prvotní inicializaci nastaví na výchozí hodnotu a následně se při načtení snímku přepočítávají. Tudíž se nečeká na ustálení shlukového algoritmu při první iteraci, ale vypočítává se postupně. Díky tomuto jevu je algoritmus schopen reagovat na změnu scény.

Shader vyžaduje vstupní texturu, která je aktuální snímek ve videu. Další vstupní parametry jsou momentální hodnoty centroid. Výstup shaderu je zakreslen do RenderTargetu, přesněji proběhne vykreslení do textur, které obsahuje. RenderTarget musí obsahovat stejný počet textur jako je počet centroid. V případě tohoto projektu, stejně jako u bakalářské práce, byl zachován počet centorid u shlukového algoritmu na 3. Tento počet je ideální, kvůli výslednému uspořádaní bodů ve skupinách, kde jedna skupina obsahuje body patřící do pozadí a zbylé dvě body popředí.

```
Metoda KmeamsShader(Texture text, Vector3[] cent)
   Získání barevné hodnoty a její přepočet do HSB modelu
   Přepočet hodnot centroid do HSB modelu
   Výpočet vzdálenosti od jednotlivých centroid
   Zapsání výsledku do RenderTargetu
```

Kód 5: Shader pro výpočet centroid

Shader vždy přepočítá aktuální hodnotu získané barvy z textury, se kterou pracuje, do modelu HSB. Toto platí také pro hodnoty centroid. Následně se vypočítá vzdálenost přepočítané hodnoty textury od předpočítaných centroid. Na toto se využije jeden ze tři vzorců v závislosti na vzájemné pozici obou barev v barevném modelu. Kód 5

$$V_{+} = (H_1 - H_2)^2 + (S_1 - S_2)^2 + (B_1 - B_2)^2 \tag{1}$$

$$V_{-} = (H_1 - (H_2 - 360))^2 + (S_1 - S_2)^2 + (B_1 - B_2)^2 \tag{2}$$

$$V_{-} = ((H_1 - 360) - H_2)^2 + (S_1 - S_2)^2 + (B_1 - B_2)^2 \tag{3}$$

Rovnice 1 se využívá pro výpočet vzdálenosti, když oba body leží v barevném modelu blízko sebe. Rovnice 2 a Rovnice 3 se využívají, pokud body leží na opačných stranách barevného spektra barevného modelu. Rovnice je využita v závislosti na pozici bodů. Vzdálenost bodů je nejmenší vypočítaná hodnota daných rovnic.

Po výpočtu vzdálenosti je vybrána nejmenší vzdálenost. Následně se postupně zapisuje hodnota do textur v render targetu. Pokud nejmenší vzdálenost odpovídá hodnotě vzdálenosti n-té centroidy, které odpovídá n-té textuře, je zakreslena do této textury hodnota barvy výchozí textury snímku. V opačném případě je zapsána nulová hodnota ve všech barevných kanálech RGBA.

$$RGB_{Cent} = \frac{RGB}{A} \tag{4}$$

Tímto vznikají 3 textury. Tyto textury představují body spadající do centroidy, která tuto texturu reprezentuje. V dalším kroku se vygenerují mipmapy a vyčte se hodnota poslední úrovně mipmap textury o rozměru 1 x 1. Tato hodnota reprezentuje průměrnou barvu dané textury. Nová hodnota centroidy se získá aplikací Rovnice 4, která vypočítá barevnou hodnotu pixelů s průhledností nastavenou na hodnotu 1.

5.5.2 Rozdělení popředí a pozadí

V předchozím kroku byly přepočítány centroidy pro aktuální snímek videa, tudíž je možné snímek vyklíčovat a následně vykreslit na obrazovku. Toto se děje v druhém využitém shaderu Kód 6.

```
Metoda Split(Texture video, Texture back, Texture mask)
   Získání barevné hodnoty v textuře
   Pokud se hodnota masky rovná 1
      Zapsání hodnoty textury pozadí
   Pokud není hodnota masky rovná 1
      Zapsání hodnoty textury snímku
```

Kód 6: Shader pro odstranění barvy pozadí

Shader vyžaduje tři vstupní textury. První je textura snímku z videa, druhá je textura pozadí a poslední je textura z RenderTargetu. Poslední textura slouží jako maska, pro určení oblasti k nahrazení.

Funkce shaderu je, kvůli využití textury pro výpočet hodnot centroid, velmi jednoduchá. Pokud má maska nastavenou hodnotu průhlednosti na hodnotu 1, je tato oblast nahrazena barvou pozadí. V opačném případě je ponechána barva snímku z videa.

Oblast pro odstranění lze vypočítávat za pomoci hodnoty centroidy, stejně jako v bakalářském projektu. Tudíž se znovu určuje k jaké centroidě patří daný bod. Metoda využívající masku byla zvolena, protože oblast centroidy je již vypočítána v předchozím kroku a tudíž není třeba opakovat stejný úkon znovu.

5.6 Úpravy algoritmu

V modelu RGB se vypočítával průměr pomocné textury, respektive hodnota nové centroidy. Tento způsob byl původně myšlen jako ulehčení vypočítávání barvy v periodickém HSB modelu, kde barevné složce H je umožněno jít mimo původní hranice, který byl použit v bakalářském projektu. Tento způsob však nepracoval, tak jak bylo zamýšleno a výsledky se lišily od předpokladů.

K vyřešení problému bylo nutno upravit algoritmus, tak aby centroidy a pomocné textury vyžívaly barevný model HSB. Navíc textury bylo nutno vytvářet s vnitřním formátem, který podporuje záporné hodnoty a umožňuje vypočítávat průměr barvy v nekonečném HSB modelu.

Shader byl pozměněn, tak aby podporoval tuto úpravu. Výpočet vzdálenosti vrací hodnotu vzdálenosti dvou barevných bodů v barevném modelu, a také informaci, jestli byla využita hodnota vzdálenosti přes hranice barevného modelu. Tato informace je ve formě kladné nebo záporné jedničky. Výsledek je nutný pro přepočet barvy.

Tato úprava byla učiněna pro lepší vypočítávání průměrů textur, ale umožnila odstranit neustálé přepočítávání barevných modelů v shaderu. Bohužel shader pracoval v rozsahu pro HSB, kde H nabývá hodnot od 0 do 360 a prvky mají hodnoty od 0 do 1. Tato vlastnost byla zachována, ale zbylé úpravy pracují v rozsahu od -1 do 1 pro všechny prvky.

Rozdílné rozsahy textury na GPU a shaderu je nutno vždy přepočítat do daného barevného rozsahu. Toto je poměrně nešťastné řešení, ale je nutné zabezpečit, že hodnoty modelu v shaderu budou v kladných hodnotách a hodnoty mimo shader budou dosahovat i záporných hodnot.

Přepočet do rozsahu shaderu za pomoci Rovnice 5 pouze zabezpečí, aby byly hodnoty v kladných číslech. Přepočet do rozsahu textury za pomoci Rovnice 6 bere příznak V, jako informaci o překročení hranice barevného modelu při výpočtu vzdálenosti ve formě kladné nebo záporné jedničky. H_T je barevná hodnota H textury a H_S shaderu.

$$H_S = ((H_T + 1) \mod 1) * 360 \tag{5}$$

$$H_T = (V + \frac{H_S}{360}) \mod 1 \tag{6}$$

5.7 Rozšíření funkcionalit

V upravené verzi implementace byla k třídě reprezentující video přidána metoda umožňující automatické vybrání textury masky, kterou se následně klíčuje. Tato metoda vezme cílovou barvu, která se má klíčovat a vybere dle hodnot centroid pozici textury pozadí. Vzdálenost prvku se vypočítává za pomoci Rovnice 7 a následně se vybere ta nejmenší. H_x je barevná hodnota H centroidy a x je pořadí textury reprezentující využívanou centroidu. H_{Back} je hodnota určující, který barevný tón je klíčován. V_x vypočítaná vzdálenost od klíčovaného barevného tónu pro jednotlivé textury render targetu.

$$V_x = |H_x - H_{Back}| \tag{7}$$

Tato úprava umožňuje automaticky vybírat, dle které centroidy se bude klíčovat bez prvotního vybrání, která centroida reprezentuje pozadí. Je nutné, aby bylo toto automatické, jelikož je možné, že v některých případech, pokud se změní scéna, změní se i centroida reprezentující pozadí.

Třída videa také implementuje metodu umožňující nastavit centroidy na výchozí nulovou hodnotu, pokud je jejich hodnota NaN. Tato metoda se volá při vypočítávání průměrné barvy centroid. Přesun výpočtu shlukové analýzy na GPU požaduje kontrolu a nastavení centroidy, pokud nabudou nedefinovaných hodnot.

V původním algoritmu se při zahájení výpočtu vždy přiřadí ke každé z centroid výchozí hodnota. Tento úkon zabezpečí, že každá centroida bude obsahovat alespoň jeden bod, tudíž bude mít definovanou hodnotu.

Při výpočtu na GPU nelze zabezpečit, že daná textura masky nebude prázdná. V případě prázdné textury bude její průměrná hodnota nulová a při následné normalizaci hodnot dojde k dělení nulou. Toto nakonec vyústí k nastavení vypočítaných hodnot centroidy na hodnoty NaN pro její barevné prvky. Metoda pouze nastaví všechny tyto hodnoty na nulovou hodnotu v případě zavolání.

5.8 Optimalizace

Bylo nutné provést několik optimalizací a úprav. Úpravy a hlavně optimalizace se týkaly převážně práce s texturami a videem. Což je způsobeno pomalou rychlostí při špatné manipulaci s daty mezi CPU a GPU.

Jediná úprava shaderu týkající se jeho fungování, byla nutnost otáčet texturu masky při klíčování. I přes značné snahy se nepodařilo naleznout důvod, proč je maska v RenderTargetu špatně otočená. Nejjednodušší řešení je danou texturu v shaderu otočit.

5.8.1 Vnitřní formát

Hlavní optimalizace se týkala vnitřního formátu textury a práce s danou texturou. Implementace využívá textury v každém kroku a velmi záleží na jejím vysokém rychlostním výkonu. Vnitřní formát textury se jevil během optimalizací jako hlavní zpomalující prvek.

Vnitřní formát textury udává jak budou reprezentovány data na GPU. Hlavně udávají, kolik bitů bude reprezentovat každý barevný prvek. Pokud bude každý prvek reprezentován velkým počtem bitů, bude přesnost dané barvy textury velmi velká, ale bude se s ní pracovat pomaleji kvůli většímu počtu dat, které se musí při změně přepsat. Vnitřních formátů textury je velké množství. Běžné typy jsou sepsány v Tab. 3.

Implementace shaderu využívá rozsah normalizovaného kladného integeru, proto bylo nutno zachovat požadovaný rozsah. Pro práci s texturou samotnou stačilo využít 8bitovou verzi daného formátu. Toto platí hlavně pro texturu reprezentující aktuální snímek ve videu, kde při použití větších formátů bylo prvotní vytvoření objektu a jeho překreslování velmi pomalé. Lze využít i s menším počtem bitů, ale získaná rychlost nevyvažuje pokles kvality, kterou ukládání barvy na méně bitů přináší.

Možné je využít 16bitovou verzi daného formátu. Toto přináší možnost lepší barevné reprezentace a zachování barevné kvality, ale výrazně tím poklesne rychlost celého procesu.

Textura pozadí kopíruje vnitřní formát textury videa, ačkoliv její formát může být kvalitnější. Toto je umožněno tím, že se s danou texturou po načtení pracuje pouze v shaderu. Proto nemá vnitřní formát na danou texturu moc velký výkonnostní vliv.

RenderTarget využívaný ve videu pro výpočet nových centroid požaduje rozsah hodnot v textuře i v záporných normalizovaných hodnotách. Tyto rozsahy umožňuje řada typů vnitřního formátu. V této implementaci se využívá 16bitová normalizovaná floatová verze, ačkoliv pro potřebu lepšího výkonu lze využít 8bitový normalizovaný integer se znaménkem.

Možností je využití 32-bitového normalizovaného floatu, kdy vzroste teoretická kvalita výpočtu a výkon poklesne pouze mírně. Ačkoliv určitá ztráta výkonu výměnou za malý vzrůst kvality se nemusí vyplatit.

5.8.2 Problematika videa

Načítání videa a práce s ním byla nejproblematičtější část celé implementace. Hlavní důvod obtíží je, že tento úkon v celé posloupnosti zabírá největší část vyhrazeného času a možnost optimalizace je velmi obtížná při využití již hotových řešení ve formě knihoven.

Knihovna EMGU poskytuje C# zabalení knihovny původně psané v programovacím jazyce C++. Knihovna v tomto provedení je pomalejší, nežli při použití v originálním jazyce, ve kterém původně vznikla. Hlavní problém však je práce s daty, které knihovna vrací. Data jsou v barevném formátu BGR.

Většina GPU dokáže pracovat pouze s RGBA nebo BGRA. Tento formát zabírá v paměti 32 bitů. Pokud GPU pracuje s formátem zabírajícím 24 bitů, což jsou formáty RGB a BGR, je nutné je transformovat do 4 bytového formátu. Tyto formáty jsou automaticky transformovány při odesílání dat na GPU. [24]

Transformace vyžaduje čas a zpomaluje celý proces. Menší čas vyžaduje tento proces, pokud je vnitřní formát RGB. Řešením, ačkoliv není ideální, je odesílat data ve formátu RGB, ačkoliv jsou ve formátu BGR a následně na shaderu prohodit barvy podle potřeby.

Při načítání videa do paměti bylo nutné upravit knihovnu. Knihovna původně podporovala pouze načtení prvního snímku z videa, což nebylo vyhovující. Při práci s videem o rozlišení 4K se ukázalo, že je nutno přidat i limitaci počtu možných načtených snímků. Momentální verze umožní načíst až 500 snímků videa a následně je načítání zastaveno. Při načtení více snímků zároveň byly nároky na místo v RAM příliš vysoké a nebylo by při dostatečné kapacitě RAM možné celou implementaci spustit. I přes toto opatření je nárok na místo v RAM kolem 16 GB, pokud je načteno všech 500 snímků.

V RAM jsou snímky uchovány ve formátu, který byl použit již na snímky v tutoriálu. Tento formát však neuchovává data v chronologickém sledu a je nutno při odesílání dat na GPU procházet data snímku po jednotlivých řádcích a odesílat data daného řádku. Optimální by byl převod dat na pole bytů a práce s ním, ale byl ponechán kvůli ukázce práce s jiným formátem dat.

Poslední implementace načítání videa je v pouhém zjednodušení předchozí knihovny. Tato implementace vrací načtený snímek jako pole bytů, kde každý barevný kanál je reprezentován jedním bytem. Tato implementace nabízí nejlepší formát dat, které vrací a tato data lze rovnou odesílat na GPU.

5.9 Uživatelsky nastavitelná oblast

V projektu vznikla implementace (WF) využívající výpočtu vzdálenosti od cílové barvy. Tato implementace vyžaduje pro správnou funkčnost nastavení od uživatele. Uživatel může určit typ barvy a vzdálenost od vybrané barvy, která se bude odstraňovat. Dále je na výběr neodstraňovat příliš tmavé, popřípadě světlé části barvy.

Implementace využívá stejné Rovnice 1 pro výpočet vzdálenosti, jako implementace využívající K-Means. Hlavní výhodou je jednoduchost celé implementace Kód 7, kde se pouze aplikuje shader na texturu snímku videa, vykreslí se a načítá se další snímek.

Uživatelsky nastavitelná implementace využívá typ textur a videa, který neumožňuje výpočet průměrné barvy textury. Po optimalizaci je celý algoritmus výkonnější

```
Metoda RENDER( )
    Nastavení hodnot pro shader
    Vykreslení výsledku shaderu na obrazovku
    Načtení dalšího snímku ve videu
```

Kód 7: Render funkce pro implementaci (WF) s uživatelským nastavením

nežli implementace využívající shlukovou analýzu. Nevýhodou je nutnost vykreslovat druhé okno ve formě formuláře, kde uživatel může nastavit hodnoty pro klíčování. Další nevýhoda je plná závislost na nastavení od uživatele, tudíž při zapnutí dané implementace se nic neklíčuje a zobrazuje se na obrazovku momentální snímek videa v původní neupravené formě.

5.10 Java implementace

Java implementace využívá shadery z upravené verze algoritmu a implementuje je. Proces této implementace je obdobný jako v jazyce C#. Nebylo zasahováno do již existující abstrakce a funkcionality, tudíž všechny textury mají interní formát 32bitového normalizovaného floatu. [25]

Implementace načte snímek do textury. Vypočtou se nové centroidy a následně je provedeno klíčování a vykreslen výsledek na obrazovku. V principu stejná sekvence jako je u C# implementace, pouze byla uzpůsobena prostředí jiného jazyka s již vytvořenou strukturou abstrakcí tříd.

5.11 Popis struktury implementačních projektů

C# implementace obsahuje sdílený projekt, který obsahuje všechny abstraktní třídy a obecné potřebné zdroje. Na tento projekt odkazují všechny ostatní spustitelné projekty. Tyto ostatní projekty obsahují spustitelnou třídu. Pokud to implementace projektu vyžaduje, tak obsahuje potřebné rozšíření základních abstrakčních tříd, a popřípadě také složku obsahující soubory shaderů.

Celé řešení mimo sdíleného projektu, obsahuje několik implementačních na sobě nezávislých projektů. [26]

- 2 mezistupně implementačních projektů - vytvořeny během vývoje a výsledkem jsou finální implementace
- 3 finální implementační projekty (VL, SL, EMGU) - funkčnost je identická a liší se pouze typem načítání videa
- 1 uživatelsky nastavitelná implementace (WF) - umožňuje uživateli nastavit, jaká oblast se bude klíčovat
- 1 projekt pro měření výkonu - pouze pro měření rychlosti a vytvoření výsledků

6 Souhrn výsledků

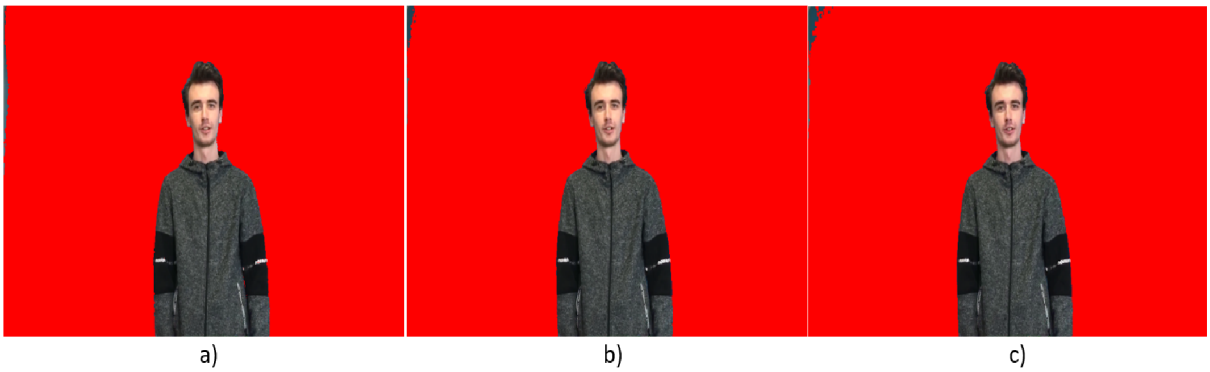
Cílem bylo zachovat kvalitu, kterou nastavil algoritmus představený v bakalářské práci a celý proces klíčování urychlit. Z důvodu provádění výpočtů na GPU se předpokládalo, že výsledná kvalita bude zachována nebo se mírně zhorší.

Hodnocení kvality je prováděno vizuálním srovnáním výsledků oproti upravené verzi algoritmu implementovaného v bakalářské práci. Je kladen důraz, aby algoritmus neklíčoval části popředí, které původní implementace neklíčovala. Obecně se očekává zdědění stejných problémů při klíčování jako měla původní upravená implementace.

Vzorek, který je následně porovnáván, je odebrán ze stého snímku. Což je z důvodu dostatečného času na výpočet správného nastavení oblasti pro klíčování v novém algoritmu. Navíc by se za pomoci této strategie mohlo promítnout u nové implementace dynamické vypočítávání, které se děje při každém snímku.

6.1 Přesun implementace na GPU

Výsledky ukazují, že přesun algoritmu na GPU a jeho uzpůsobení na tento typ zpracování, neměl velký dopad na kvalitu zpracování (Obr. 6). Obě nové implementace bez rozdílu v jazyce zvládají klíčovat problematické části oblečení a přechodu popředí a pozadí.



Obrázek 6: Ukázka výsledků klíčování a) původní (Java) b) nový (Java) c) nová implementace

Jediné místo, kde obě implementace selhávají a původní implementace dosahovala kvalitnějších výsledků je oblast na okraji scény. S touto oblastí měla problém již původní implementace, přesto ji dokázala odstranit ve velmi dobré kvalitě.

Nová implementace má velké problémy s tmavými, popřípadě velmi světlými odstíny klíčované barvy. To plyne z omezení v shaderu, aby se při klíčování vynechávaly příliš světlé a příliš tmavé odstíny barvy, které jsou určené maskou centroid.

Rozdíl mezi výsledky v implementacích Javy a C# je způsoben možností úpravy kvality a nastavení využívaných textur. V jazyce Java je textura nastavena na největší kvalitu a barevnou přesnost, což se příznivě promítá na mírně lepší kvalitě klíčování.

6.2 Neupravené implementace v C#

V jazyce C# vznikly mimo finální implementace shlukového algoritmu také dvě implementace, které využívaly stejný princip klíčování jako verze finální, ale nebyly dostatečně funkční. Probíhal na nich vývoj a následná implementace využívaného algoritmu a princip načítání snímků, aby byl splněn cíl zachování kvality a také velká rychlost celého algoritmu.

Obě verze využívají stejný mechanismus práce s texturou a mají stejnou posloupnost procesu. První verze využívá mimo shader barevný model RGB pro úschovu dat. Toto způsobuje velké nepřesnosti ve výpočtu průměrné barvy.

Výpočetní nepřesnosti kvůli špatné konverzi do jiného barevného modelu jsou však zahlazeny, pokud se světlé a tmavé části masky začnou filtrovat při samotném klíčování. Kvůli využívání RBG barevného modelu v texturách je však nutné texturu před filtrací nejdříve přepočítat do modelu HSB. Původní nefiltrovaný vyklíčovaný snímek ukazuje stejný problém, který měla implementace bakalářské práce. Špatné zařazení části popředí, nacházející ho se v barevném modelu blízko barvy pozadí, do pozadí. Tuto vlastnost se však ani úpravou práce s barvami nepovedlo eliminovat a řeší se limitací světlých a tmavých částí snímku při samotném klíčování.

Druhá verze vykazuje lepší schopnost zařazení části oblečení do popředí, přesto se větší část oblečení zařazuje do oblasti pozadí. Změna je viditelná, ačkoliv se jednalo

Obrázek 7: Ukázka první implementace v jazyce C# pracující interně v modelu RGB a) původní b) upravený

pouze o opravu výpočtu průměrné barvy textury masky za pomoci barevného modelu HSB.

Obrázek 8: Ukázka upravené implementace v jazyce C# pracující interně v modelu HSB a) původní b) upravený

6.3 Nedokonalosti implementace

Ačkoliv algoritmus vykazuje dobré výsledky, má také své nedostatky. Tyto nedostatky jsou dobře viditelné při klíčování druhého videa. Toto video obsahuje pouze bílý nápis a modrou barvu pozadí. Dle očekávání by klíčování mělo být stabilní, bohužel se tomu tak neděje.

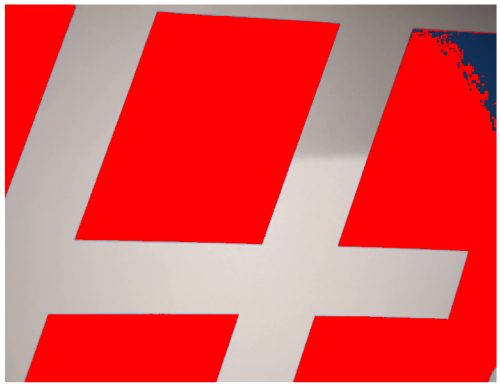
Při klíčování videa jsou místa, kde je špatně určeno pozadí a není v podstatě nic klíčováno. Tato situace vždy vydrží několik snímků a následně se změní osvětlení a barva scény dostatečně na to, aby algoritmus byl opět schopen scénu klíčovat správně (Obr. 9).

Obrázek 9: Přechod od špatně klíčované scény po dobře klíčovanou, během třech navazujících snímků

Nestabilita algoritmu u videa s bílým textem je způsobena mechanismem, který vypočítává vzdálenost bodů od sebe přes hranice barevného modelu. Toto způsobuje nechtěné výsledky při práci s bílým textem, který je v použitém barevném modelu na dolní nebo horní hranici barvy. Pokud se u algoritmu odebere schopnost výpočtu vzdálenosti přes barevné hranice, je video vyklíčováno stabilně s očekávaným výsledkem klíčování.

Algoritmus také nezvládá odstranit tmavé části pozadí, což je způsobeno limitací odstiňování tmavých a světlých částí scény. S tímto se bohužel nedá nic dělat, jelikož tato vlastnost vychází z chování implementace (Obr. 10).

I přes prezentované nedostatky je algoritmus schopen klíčovat video s textem z velké části velmi kvalitně. Nasvícení, poměr popředí a pozadí se během videa mění, a tudíž předchozí implementace, která vypočítala své nastavení pouze při inicializaci není schopna toto video klíčovat s dobrými výsledky. Výhodou schopnosti algoritmu je přepočítávat klíčované hodnoty při zpracování každého snímku.



Obrázek 10: Ukázka neklíčování tmavé části scény

6.4 Uživatelsky nastavitelná oblast

Nejlepší výsledky při klíčování videa s bílím textem měl algoritmus, kde je možné nastavit oblast, která bude odstraněna. Tento algoritmus vykazoval po nastavení rozsahu schopnost klíčovat celé video, aniž by se objevovaly během klíčování nedokonalosti nebo nevyklíčované oblasti. U primárního videa dosahuje tento algoritmus také velmi dobrých výsledků, jenom se objevuje nedokonalost klíčování u okraje scény a klíčuje se malá část oblečení.

Obrázek 11: Ukázka klíčování uživatelsky nastaveného algoritmu

6.5 Rychlost implementace

Pro měření rychlosti zpracování jednotlivých implementací bylo použito stejné video s postavou jako pro porovnávání účinnosti klíčování. Byla testována rychlost zpracování videa různého rozlišení, aby bylo možné určit náročnost rozlišení na čas zpracování a určit jakou velikost rozlišení lze zpracovávat v reálném čase a kterou již nikoliv.

Celý proces testování výkonosti byl prováděn na počítači obsahující procesor Intel Core i7-12700KF, grafiku GeForce GTX 1070 a 32 GB operační paměti. Počítač využívá SSD disky, což umožňuje rychleji číst zpracovávané video z disku.

Tabulka 6: Specifika využitého počítače

Procesor	Grafika	RAM
Intel Core i7-12700KF	GeForce GTX 1070	32 GB

6.5.1 Implementace v jazyce Java

Implementace v jazyce Java využívala nejlepší nastavení textury. Textura je uložena v 32-bitovém floatu a podporuje vytváření mipmap textur. Tudíž nebyl předpoklad schopnosti zpracovávat vyšší rozlišení. Očekávání bylo, že daná implementace budu umožňovat klíčování nejnižšího testovacího rozlišení na více nežli 60 snímků za sekundu. Původní implementace byla upravena v části, kde se využívá optimalizace vzorkování textury, tudíž inicializační část se značně urychlila.

Tabulka 7: Rychlost zpracování Java implementace

Typ	Rozlišení	Inicializace	Čas zpracování	Snímek	FPS
Původní	1920 x 1080	368 ms	6577 ms	16,6 ms	60,2
Původní	2560 x 1440	453 ms	6986 ms	17,6 ms	56,7
Původní	3840 x 2160	781 ms	15332 ms	38,7 ms	25,8
Nový	1920 x 1080	240 ms	6591 ms	16,6 ms	60,1
Nový	2560 x 1440	250 ms	6978 ms	17,6 ms	56,8
Nový	3840 x 2160	300 ms	15065 ms	38,0 ms	26,3

Jediné rozlišení, které lze zpracovat v reálném čase v 60 snímcích nebo více, je to nejmenší testované Full HD. Ostatní rozlišení se již v reálném čase zpracovávat nepodařilo (Tab. 7).

6.5.2 Implementace v jazyce C#

Testování rychlosti překreslení bylo provedeno na textuře o velkém objemu dat, kde obsah není důležitý. Tato textura byla během inicializační části nahrána do RAM ve formě pole bytů, což je nejefektivnější možný formát pro odesílání dat. Následně je v každé z iterací testovaná textura stokrát překreslena. Rychlost daného překreslování se odvíjí od velikosti textury, tudíž u menšího rozlišení je čas výrazně menší.

Z testování během vývoje vyplynulo, že počet ani náročnost shaderů nemá velký dopad na výkon dané implementace. Nejdelší čas vyhrazený pro zpracování daného snímku zabere nahrání snímku do RAM a následně do GPU za pomoci překreslení použité textury.

Po tomto zjištění má implementace v C# tři přístupy, jak načítat klíčované video a byla využita textura umožňující nejrychlejší překreslení obsahu, pokud to načítání videa dovoluje.

Testování rychlosti překreslení přineslo překvapivé výsledky, které se následně promítly do finální implementace. Nejrychlejší možnost překreslit svůj obsah umožňuje vnitřní formát textury obsahující 4 a 8 bitů (Tab. 8).

Nejpomalejší bylo zpracování textury ve formátu 16-bitového floatu, a to i ve srovnání s 32-bitovou verzí, což bylo překvapivé. Ta si v testu nevedla moc špatně, přesto není vhodná pro operace, kde je potřebné dosáhnout operací v co nejlepším čase, aniž by byl kladen důraz na kvalitu textury.

Velký propad rychlosti způsobuje netradiční formát dat, který grafický driver při odeslání na grafickou kartu přeformátuje na požadovaný tvar [24]. Toto má bohužel velký dopad na nahrávání. Tento jev se týká jedné implementace videa (EMGU), kde jsou data ve formátu BGR, a ačkoliv se posílají na GPU jako RGB, jsou přeformátována při odeslání na formát RGBA a je tu tudíž časové zpoždění.

Tabulka 8: Rychlost sto-násobného překreslení textury o velikosti 3840 x 3840

Typ	Počet meření	Rychlost	Rychlost jedné operace	Op/s
Rgba8	125	823 ms	8,2 ms	121,5
Rgba4	125	834 ms	8,3 ms	119,9
Rgba8 z Rgb	125	1115 ms	11,2 ms	89,7
Rgba8 z Bgra	125	1368 ms	13,7 ms	73,1
Rgba8 z Bgr	125	1446 ms	14,5 ms	69,2
Rgba16	125	2292 ms	22,9 ms	43,6
Rgba32f	125	3596 ms	36,0 ms	27,8
Rgba16f z Bgr	125	7780 ms	77,8 ms	12,9
Rgba16f z Rgb	125	7800 ms	78,0 ms	12,8
Rgba16f z Bgra	125	8776 ms	87,8 ms	11,4
Rgba16f	125	8881 ms	88,8 ms	11,3

Testování rychlosti zpracování jednotlivých implementací C# se provádělo ve formě měření rychlosti renderovací funkce. V měření byl obsažen celý proces, který byl potřebný pro vykreslení jednoho snímku. Výsledek samotný se skládal z 500 zprůměrovaných vzorků.

Tímto způsobem byla testována každá z implementací a následně proběhl také test, kde byly využívané textury nastaveny na stejnou kvalitu, kterou využívala implementace v jazyce Java (VL Max). Za pomoci tohoto kroku bylo následně možné porovnat obě jazykové implementace a odhadnout jaký výkon by měla implementace v Javě, pokud by se upravila kvalita textury i v této implementaci.

Předpokládané výsledky byly, že implementace (SL) využívající data v RAM bude mít ze všech nejrychlejší zpracování. U implementace (EMGU) využívající stejnou knihovnu jako implementace v Javě se předpokládalo malé rychlostní zpomalení kvůli nutnosti úpravy dat pro texturu nežli v implementaci (VL), která vrací data textury v ideálním formátu. Přesto se předpokládalo, že obě zbývající implementace budou mít rychlost zpracování porovnatelnou nebo stejnou.

U poslední implementace (WF) využívající možnost nastavení klíčované oblasti uživatelem, se předpokládala nejlepší rychlost zpracování vzhledem k použitému načítání videa. Pro testování byla využita verze VL pro načítání videa.

Tabulka 9: Rychlost zpracování jednotlivých implementací

Typ	Op. (HD)	Op/s (HD)	Op. (QHD)	Op/s (QHD)	Op. (4K)	Op/s (4K)
SL	2,1 ms	482,6	2,8 ms	353,9	5,0 ms	198,6
EMGU	11,3 ms	88,8	11,4 ms	88,0	23,3 ms	42,8
VL	3,8 ms	263,0	6,9 ms	145,3	8,9 ms	112,0
VL Max	6,5 ms	152,7	11,4 ms	87,6	22,0 ms	45,5
WF	3,8 ms	261,4	6,6 ms	150,5	8,7 ms	115,0

Výsledky ukazují možnost klíčování více nežli sto snímků za sekundu. Hlavní problém je schopnost dostatečně rychle získat snímek a následně ho nahrát na GPU. Rychlost samotného zpracování na GPU je pouze minoritní a lze provádět tento úkon v řádech nízkých jednotek milisekund. Zbytek času zpracování je přesun a jiné operace s texturou.

Načítání videa za pomoci SL dosahovalo bezkonkurenčních výsledků, bohužel reálná implementace musí nějakým způsobem zabezpečit přítomnost textury v RAM. Kvůli těmto omezením je nejlepší implementace VL, která je také z pohledu formátu dat a práce s nimi nejlepší.

Implementace využívající knihovnu EMGU dosahovala velmi nepříznivých výsledků, zpracovat testované video v reálném čase se povedlo pouze při menším rozlišení, nežli byl daný cíl. Tato knihovna nedosahuje dobrých výsledků při načítání textury a následně načtená data poskytuje ve formátu, který způsobí další zdržení načítání textury. Ačkoliv je možné, že špatné výsledky produkuje pouze špatná implementace dané knihovny do projektu.

Implementace WF umožňující nastavit oblast klíčování uživatelem z celého testu vykazuje nejlepší výsledky. To se týká nejen testování rychlosti, ale také v testování kvality a stability klíčování. Jediná nevýhoda je nastavení uživatelem během klíčování.

7 Závěry a doporučení

Diplomová práce se zabývá odstraňováním barevného pozadí a jeho nahrazením. Tato technologie je hojně využívána v zábavním a filmovém průmyslu pro úpravu natáčené scény. Práce navazuje na bakalářskou práci, přebírá použitý algoritmus a implementuje ho novým způsobem, zatímco ho optimalizuje pro nejrychlejší možný chod.

Implementace byla primárně prováděna v programovacím jazyce C# s využitím knihovny Silk.Net. V této jazykové implementaci vznikly dvě ukázky možného řešení dané problematiky.

První využívala plně autonomní vypočítávání oblasti a následného odstraňování. Přepočítávání oblasti bylo prováděno při zpracování každého snímku. Díky tomuto typu práce se scénou je tato implementace schopna reagovat na změnu nasvícení scény, aniž by utrpěla kvalita zpracování.

Druhé řešení je plně odkázáno na nastavení uživatelem, která oblast se bude klíčovat. Ten může určit jaká barva se budu cíleně odstraňovat. Ukázkový příklad umožňuje vybrat pouze předvolenou barvu, ale v reálné implementaci by se spíš nabízela možnost výběru konkrétní barevné oblasti ze zpracovávané scény. Dále uživatel volí vzdálenost v použitém barevném modelu od zvolené barvy, která se bude odstraňovat. Také lze výběrem omezit odstraňování světlých a tmavých barev.

Obě řešení poskytovala dobré výsledky, přesto implementace využívající uživatelské nastavení nabízela stabilnější řešení. Ačkoliv je nutností, aby uživatel vše nastavil. Toto řešení navíc nabízí možnost rychlejšího zpracovávání, ale rozdíl není markantní.

Rychlost zpracování se ukázal jako největší problém celé problematiky. Konkrétně načtení snímku do GPU, kde lze následně s daty snímku pracovat. Tento úkon trval většinu času celého zpracování jednotlivého snímku a v tomto směru je možné celý proces dále optimalizovat. Přesto se podařilo zpracovávat největší rozlišení ve více nežli 60 snímcích za sekundu, tudíž dané rozlišení lze zpracovávat v reálném čase, pokud by byla implementace uzpůsobena, aby zpracovala za sekundu pouze daný počet snímků, nebo snímky načítala z jiného zdroje nežli videa. Zlepšení procesu

klíčování, by mohlo přinést využití umělé inteligence, která by určovala oblast pozadí nebo popředí.

Popřípadě určovat popředí, v takovém případě by nezáleželo na barvě pozadí. Pro zlepšení rychlosti zpracování se nabízí přednačítání snímků nezávisle na zpracování, popřípadě načítat snímek přímo na GPU mezi uložení v RAM. Tyto kroky přinesou zlepšení kvality zpracování a možnost zpracovávat více snímků, popřípadě ve větší rozlišení.

Literatura

[1] Hvězda, M. Segmentace objektů ve videu s využitím klíčování barevného pozadí. Available from: https://theses.cz/id/3jqfqs

[2] android-mipmap.png (256×384). Available from: https://www.learnopengles.com/wordpress/wp-content/uploads/2012/02/android-mipmap.png

[3] Rendering Pipeline Overview - OpenGL Wiki. Available from: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

[4] Image Format - OpenGL Wiki. Available from: https://www.khronos.org/opengl/wiki/Image_Format

[5] Yang, H.-s.; Kupferschmidt, B. Time Stamp Synchronization in Video Systems.

[6] Griesser, A.; Van Gool, L. Rtsyncnet-a flexible real-time synchronisation network for cluster based vision-and graphics-architectures.

[7] Jack, K. *Video demystified: a handbook for the digital engineer*. Demystifying technology series, Elsevier, fourth edition, ISBN 978-0-7506-7822-3.

[8] Tai, N. N. Chroma-Key Algorithm Based on Combination of K-Means and Confident Coefficients. volume 4, no. 3, ISSN 20103719, doi:10.7763/IJIEE.2014.V4.432. Available from: http://www.ijiee.org/index.php?m=content&c=index&a=show&catid=45&id=464

[9] Truong Van, C.; Truong Quang, V. *FPGA implementation of real-time growcut based object segmentation for chroma-key effect*. doi:10.1109/ComManTel.2015.7394259, pages: 56.

[10] Agata, H.; Yamashita, A.; Kaneko, T. Chroma Key Using a Checker Pattern Background. volume E90-D, no. 1: pp. 242–249, ISSN 0916-8532, 1745-1361,

doi:10.1093/ietisy/e90-1.1.242. Available from: http://ietisy.oxfordjournals.org/cgi/doi/10.1093/ietisy/e90-1.1.242

[11] NVIDIA Developer. Available from: https://developer.nvidia.com/

[12] Stevewhims. Texture Filtering with Mipmaps (Direct3D 9) - Win32 apps. Available from: https://learn.microsoft.com/en-us/windows/win32/direct3d9/texture-filtering-with-mipmaps

[13] Reed, N. Answer to "Calculating maximum number of mipmap levels for OpenGL automatic mipmap storage generation". Available from: https://computergraphics.stackexchange.com/a/12735

[14] Framebuffer Object - OpenGL Wiki. Available from: https://www.khronos.org/opengl/wiki/Framebuffer_Object

[15] Framebuffer - OpenGL Wiki. Available from: https://www.khronos.org/opengl/wiki/Framebuffer

[16] Texture - OpenGL Wiki. Available from: https://www.khronos.org/opengl/wiki/Texture

[17] Renderbuffer Object - OpenGL Wiki. Available from: https://www.khronos.org/opengl/wiki/Renderbuffer_Object

[18] Shader - OpenGL Wiki. Available from: https://www.khronos.org/opengl/wiki/Shader

[19] Vertex Shader - OpenGL Wiki. Available from: https://www.khronos.org/opengl/wiki/Vertex_Shader

[20] Tessellation Control Shader - OpenGL Wiki. Available from: https://www.khronos.org/opengl/wiki/Tessellation_Control_Shader

[21] Geometry Shader - OpenGL Wiki. Available from: https://www.khronos.org/opengl/wiki/Geometry_Shader

[22] Fragment Shader - OpenGL Wiki. Available from:
https://www.khronos.org/opengl/wiki/Fragment_Shader

[23] Compute Shader - OpenGL Wiki. Available from:
https://www.khronos.org/opengl/wiki/Compute_Shader

[24] Common Mistakes - OpenGL Wiki. Available from:
https://www.khronos.org/opengl/wiki/Common_Mistakes

[25] MichaelHvezda. MichaelHvezda/KZP. Original-date: 2019-10-26T12:03:56Z.
Available from: https://github.com/MichaelHvezda/KZP

[26] MichaelHvezda. MichaelHvezda/KBPvCS. Original-date:
2023-02-03T13:46:53Z. Available from:
https://github.com/MichaelHvezda/KBPvCS

Seznam zkratek

API aplikační programovatelný interface

CPU procesoru

CS Compute Shader

FB Framebuffer

FBO Framebuffer Objekt

FS Fragment Shader

GLSL OpenGL Shading Language

GPU grafické karty

GS Geometry Shader

RAM operační paměti

RBO RenderBuffer Objekt

TCS Tessellation Control Shader

TES Tessellation Evaluation Shader

VS Vertex Shader

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání diplomové práce

Autor:	**Bc. Michal Hvězda**
Studium:	I2000042
Studijní program:	N1802 Aplikovaná informatika
Studijní obor:	Aplikovaná informatika
Název diplomové práce:	**Klíčování videa na GPU**
Název diplomové práce AJ:	Chroma key video segmentation on GPU

Cíl, metody, literatura, předpoklady:

Cíl práce:

Cílem práce je prozkoumat možnosti klíčování video záznamu prováděné v reálném čase. Pro zpracování použijte možnosti hardwarové akcelerace na grafické kartě.

Postup prací:

1. Prozkoumat principy klíčování videa v reálném čase.
2. Vytvořit přehled algoritmů pro výběr klíčované oblasti.
3. Navrhnout implementaci výběru klíčované oblasti s využitím možností programovatelných grafických karet.
4. Navržené řešení implementovat a otestovat pro různá osvětlení snímané scény.
5. Zhodnotit dosažené výsledky.

HAO, Chengcheng, Wenyi WANG a Jiying ZHAO, 2016. Video chroma keying via global sampling and trimap propagation. *Multimedia Systems* [online]. **22**(6), 693–707. ISSN 1432-1882. Dostupné z: doi:10.1007/s00530-015-0493-2 SHALOM, S. A. Arul, Manoranjan DASH a Minh TUE, 2008. Efficient K-Means Clustering Using Accelerated Graphics Processors. In: Il-Yeol SONG, Johann EDER a Tho Manh NGUYEN, ed. *Data Warehousing and Knowledge Discovery* [online]. Berlin, Heidelberg: Springer, s. 166–175. Lecture Notes in Computer Science. ISBN 978-3-540-85836-2. Dostupné z: doi:10.1007/978-3-540-85836-2_16

Zadávající pracoviště:	Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu
Vedoucí práce:	Ing. Bruno Ježek, Ph.D.
Oponent:	prof. RNDr. PhDr. Antonín Slabý, CSc.
Datum zadání závěrečné práce:	26.1.2021