

**ŽILINSKÁ UNIVERZITA V ŽILINE
FAKULTA RIADENIA A INFORMATIKY**

Semestrálna práca

Počítačová grafika – 2024

Bc. David Kučera

Akademický rok 2024/25, zimný semester



Obsah

ZADANIE A CIEĽ PRÁCE	3
POPIS A ANALÝZA PROBLÉMU	3
NÁVRH SAMOSTATNÝCH ČASŤÍ PROGRAMU	4
RIEŠENIE PROBLÉMU HĽADANIA BEZIÉROVEJ KRIVKY	4
ŠTRUKTÚRA PROGRAMU	5
MODUL PGVIEWER	6
MODUL PGRAPHICSLIB	7
<i>GrayscaleImage.cs</i>	7
<i>Bezier.cs</i>	7
<i>Gauss.cs</i>	8
<i>MiddlelineExtractor.cs</i>	8
<i>Sobel.cs</i>	8
<i>Threshold.cs</i>	8
MODUL PGTESTER	9
MOŽNOSTI TESTOV	9
EXTERNÉ KNIŽNICE	10
VYHODNOTENIE RÝCHLOSTÍ	11
POROVNANIE RÝCHLOSTÍ OPERÁCIÍ JEDNOTLIVÝCH OBRÁZKOV	11
POROVNANIE RÝCHLOSTÍ OPERÁCIÍ JEDNOTLIVÝCH OBRÁZKOV PRI SPRACOVANÍ KAŽDÉHO 10. RIADKU	11
POROVNANIE RÝCHLOSTÍ PRAHOVANÍ	12
POROVNANIE BUILD VERZIÍ x86	13
POROVNANIE BUILD VERZIÍ x64	13
CYKLICKÉ SPRACOVANIE OBRÁZKOV ROZMERU 640 x 480	14
ZÁVER	15



Zadanie a cieľ práce

Načítajte obrázok zo súboru, formát obrázka je YUV420. V prvej časti súboru sa nachádzajú dáta Luminancie. Veľkosť obrázka je 512 x 512 px, luminancia je zakódovaná ako hodnota 0..255 svietivosti pre 1 pixel. Farebné dáta v obrázku, ktoré nasledujú môžete ignorovať. V obrázku sa nachádza čierna čiara na bielom pozadí. Vašou úlohou je získať stred čiary a vektorizovať ho pomocou Beziérovej krivky.

Cieľom tejto práce je nájsť taký postup pre spracovanie obrázkov, aby bol z výpočtového hľadiska čo najefektívnejší a dokázal spracovávať dáta z kamery v čo najkratšom čase.

Taktiež je nutné riešenie implementovať v prostredí Microsoft Visual Studio 2022 na platforme .NET 8.0 v programovacom jazyku C#. Nie je dovolené používať OpenCV.

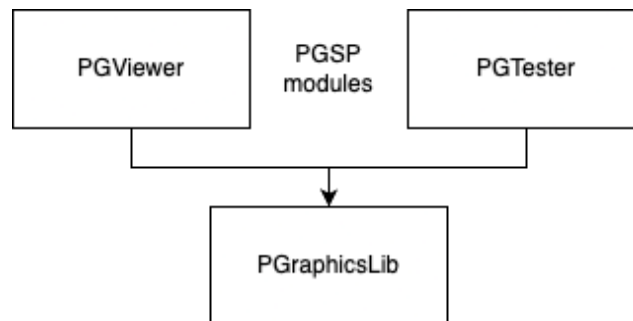
Popis a analýza problému

Počas riešenia úlohy budú potrebné rôzne metódy spracovania obrazu, napríklad vysokofrekvenčný Gaussov filter, alebo iný vhodný filter, histogram, algoritmus pre adaptívne prahovanie, napr. Otsu threshold, hľadanie hrán, napr. Sobel edge detector, práca s maticami a vektormi, knižnica na prekladanie krivky bodmi, napr. MathNet.Numerics.

Na otestovanie funkčnosti navrhnutého riešenia boli poskytnuté nasledovné 3 dátové súbory: *NewImage.txt*, *NewImage2.txt*, *NewImage3.txt*. Každý súbor obsahuje obrázok formátu YUV420 veľkosti 512 x 512 – na ňom sa nachádza čierna čiara na bielom pozadí. Na týchto obrázkoch bolo spravených niekoľko experimentov popísaných nižšie v dokumente. Taktiež bolo poskytnutých niekoľko podobných obrázkov, avšak veľkosti 640 x 480. Na týchto obrázkoch bolo testované cyklické spracovanie.

Návrh samostatných častí programu

Práca bola implementovaná v programovacom jazyku **C#** na platforme **.NET 8.0** a bola vyvíjaná vo vývojom prostredí Visual Studio 2022, podľa požiadaviek zadania. Obsahuje program s grafickým rozhraním pre načítanie a zobrazenie obrázkov typu YUV420, poskytuje aj rôzne možnosti ich úpravy pomocou implementovaných operácií. Ďalej obsahuje aj jednoduché konzolové rozhranie pre testovanie rýchlostí jednotlivých algoritmov, či už jednotlivo, alebo aj cyklicky. Tieto dva programy využívajú spoločnú knižnicu, v ktorej sú algoritmy implementované. Práca teda obsahuje dokopy 3 moduly – **PGViewer** (GUI), **PGraphicsLib** (DLL), **PGTester** (CLI). Prepojenie týchto modulov je možné vidieť v nasledovnom diagrame.



Obrázok 1 Diagram modulov

Riešenie problému hľadania beziérovej krivky

V tejto práci je cieľom zo vstupného súboru dostať ako výstup beziérovu krivku, ktorá bude čo najlepšie opisovať čiernu čiaru obsiahnutú v dátových súboroch.

Algoritmus je nasledovný:

1. **Načítanie** dátového súboru formátu YUV420. Tento obrázok obsahuje čiernu čiaru na pozadí, ktoré je odlišné od čiernej – biele, odtiene šedej. Je možné načítať len niektoré časti obrázka, pre zrýchlenie trvania algoritmu, napr. stačí prečítať každý 10. riadok.
2. Vykonanie **filtrácie** pomocou Gaussového algoritmu. Tento krok sa nemusí vykonať, avšak je dobré ho zahrnúť, pretože obrázky môžu obsahovať istý šum, ktorý vďaka tomuto kroku odfiltrujeme. Taktiež vďaka tomu zanedbáme nepotrebné detaily.
3. Binarizácia obrázka na čiernu a bielu farbu pomocou **prahovania**. Pre výpočet prahu je v práci implementovaný aj Otsu prah, ale v mojom testovaní nebol tak dobrý, ako jednoduché **priemerné** prahovanie.
4. Hľadanie **stred**u čiary. Tento stred čiary sa po predošlých operáciách hľadá pomerne ľahko, nakoľko v ideálnom prípade máme biele pozadie a čiernu čiaru – tak stačí len nájsť súradnice čiernych pixelov a vziať ich stred. Takto



to spravíme pre každý (alebo každý n-tý) riadok. Niektoré z týchto bodov nám budú popisovať výslednú beziérovu krivku.

5. Nakoniec sa z bodov vyberie začiatok a koniec – tie nám budú pevne definovať začiatok a koniec beziérovej krivky. K nim sa vyberú ešte dva kontrolné body, ktoré budú určovať tvar kubickej **beziérovej krivky**.

Štruktúra programu

Odovzdaný repozitár obsahuje nasledovné priečinky:

- data – obsahuje vstupné dáta (taktiež aj v prípade, že chceme nové dáta pridať, je nutné ich vkladať sem, inak sa v module PGViewer nezobrazia),
- doc – obsahuje dokumentáciu, UML diagramy, grafy,
- modules – obsahuje zdrojové kódy projektov (.csproj, .cs, ...),
- solution – obsahuje solution (.sln) pre otvorenie projektov v prostredí VS2022.

Po prípadnom builde vo VS2022 pribudnú nasledovné priečinky:

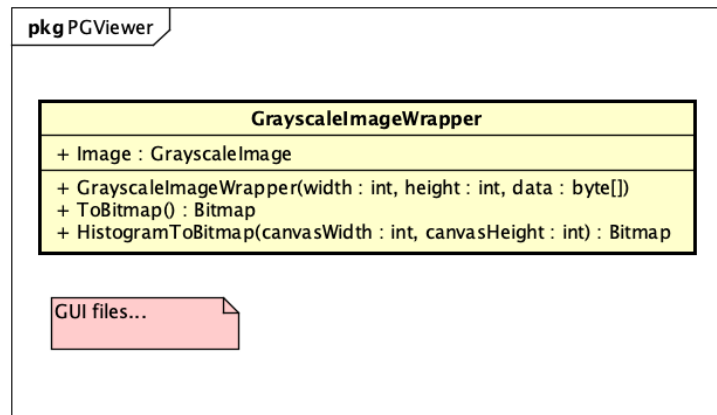
- run/run64/run_D/run64_D – obsahuje .exe binárky pre spustenie programu na základe konfigurácie (viď nižšie),
- tmp – dočasné súbory (priečinkov obj),
- output – obsahuje výstup z testovacej aplikácie, ak sa nastaví konštanta.

Programy je možné zbuildovať a následne spustiť v nasledovných konfiguráciách:

- Debug
 - x64 (priečinkov run64_D)
 - x86 (priečinkov run_D)
- Release
 - x64 (priečinkov run64)
 - x86 (priečinkov run)

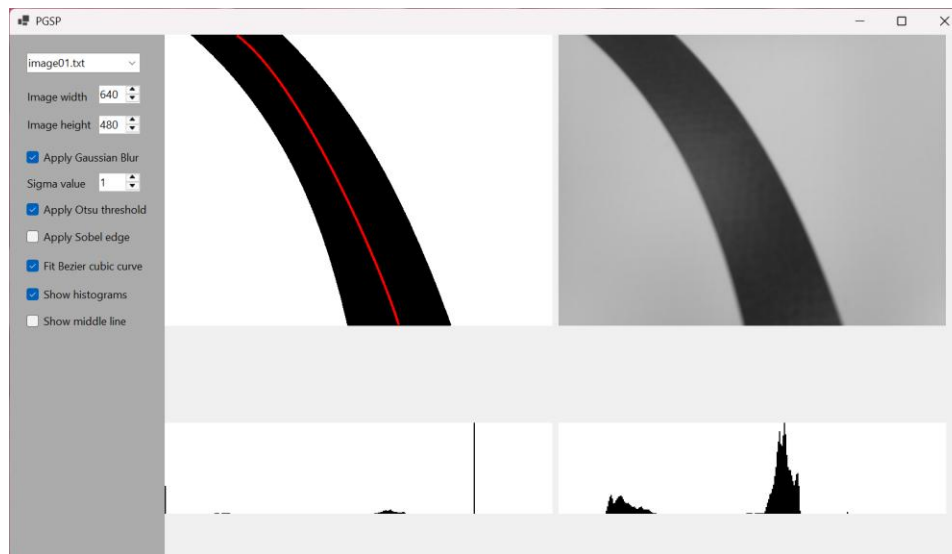


Modul PGViewer



Obrázok 2 Diagram tried modulu PGViewer

Modul obsahuje grafické rozhranie implementované pomocou **WinForms**, nakoľko sa s ním pracovalo na cvičeniach tohto predmetu. Ukážka vzhľadu je na nasledovnom obrázku.



Obrázok 3 Grafické rozhranie PGViewer

Okno obsahuje v ľavej časti niekoľko možností:

- voľba dátového súboru,
- nastavenie šírky a dĺžky obrázka pre prípad iných rozmerov,
- aplikovanie Gaussovho šumu s rôznym parametrom sigma (udáva mieru zašumenia),
- aplikovanie tresholdingu,
- aplikovanie Sobel Edge Detection,
- zobrazenie Beziérovej krivky,

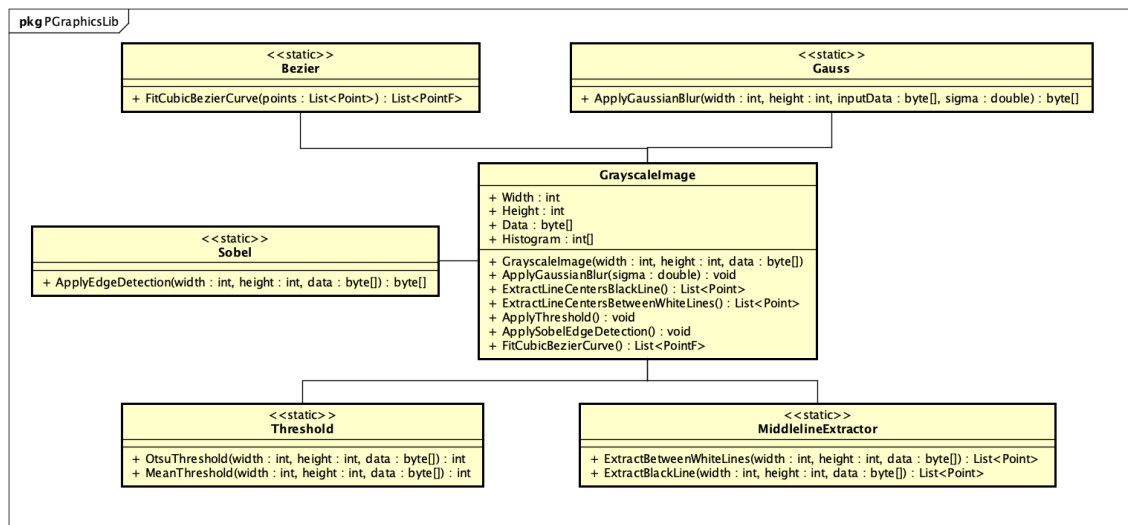


- zobrazenie histogramov obrázkov,
- zobrazenie stredovej čiary čiernej čiary.

V pravej časti sa nachádza po načítaní originálny obrázok spolu s manipulovaným obrázkom, aby bolo možné ihneď porovnať rozdiely. Taktiež pod obrázkami je možné zobrazovať ich histogramy, prípadne zobraziť aj nájdený stred čiernej čiary.

Modul PGraphicsLib

Táto knižnica obsahuje všetky implementované operácie a algoritmy pre manipuláciu s obrázkami. Všetky triedy obsahujú nejakú funkčnú časť použiteľnú pre rôzne obrázky. Preto sú všetky ich operácie statické a ako parameter majú rozmery obrázka a dáta obrázka.



Obrázok 4 Diagram tried modulu PGraphicsLib

GrayscaleImage.cs

Hlavná trieda celej knižnice. Obsahuje všetky potrebné dáta o obrázku, ako rozmery a jednotlivé bajty dát. Jediné obmedzenie je veľkosť obrázka – ten musí mať rozmer minimálne 16 x 16 a maximálne 1024 x 1024. Inak obsahuje metódy pre aplikáciu operácií na daný obrázok, ako napr. aplikovanie gaussovho šumu, extrakcia stredy čiary, aplikovanie prahovania, Sobel detekcie hrán a fitnutie beziérovej krivky.

Bezier.cs

V tejto triede je metóda *FitCubicBezierCurve*, ktorá má ako parameter body nájdené z triedy *MiddlelineExtractor*. Z týchto bodov zoberie prvý a posledný bod, ktoré definujú začiatok a koniec beziérovej krivky. Následne vezme podľa vzťahu ďalšie 2 kontrolné body, ktoré určujú tvar tejto beziérovej krivky. Na výstupe sú teda 4 body definujúce výslednú beziérovu krivku.

Gauss.cs

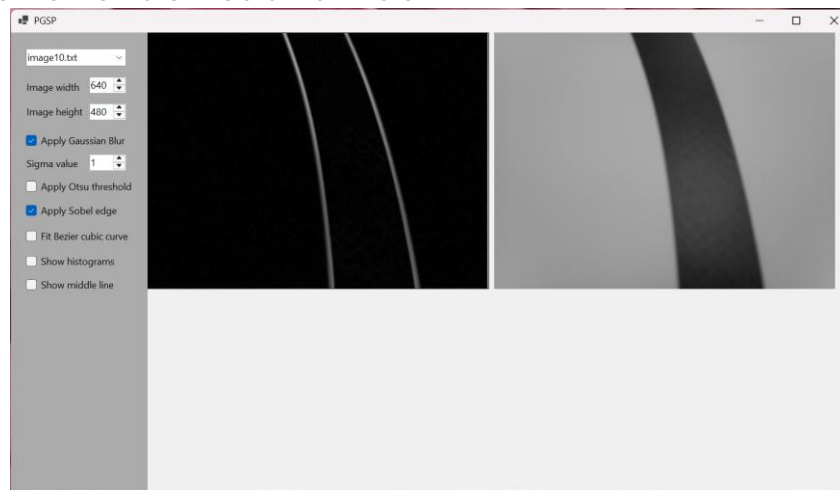
Metóda *ApplyGaussianBlur* zoberie dáta a aplikuje na ne gaussovský šum. Tým sa obrázok „vyhladí“ a zanedbajú sa nepotrebné detaily. Vďaka tomu sa potom lepšie aplikuje prahovanie – binarizácia obrázka.

MiddlelineExtractor.cs

Trieda obsahuje dve metódy – *ExtractBlackLine* – tá sa používa pri klasickom algoritme popísanom vyššie, teda hľadá stred čiernej čiary, ktorá je na bielom pozadí po aplikácii prahovania. Avšak, ak aplikujeme ešte predtým Sobel detekciu hrán, tak po prahovaní bude výsledný obrázok mať čierne pozadie a hrany čiary budú bielej farby. Teda metóda *ExtractBetweenWhiteLines* sa používa, ak sa aplikuje aj Sobel detekcia hrán. Pri mojom testovaní tento variant nebol efektívny a bol veľmi náchylný pre rôzne situácie, ako napr. že na hranách obrázku boli biele čiary – vtedy to zle vypočítalo stred a teda aj výslednú beziérovu krivku. Preto sa v hlavnom algoritme nepoužíva tento nespoľahlivý variant.

Sobel.cs

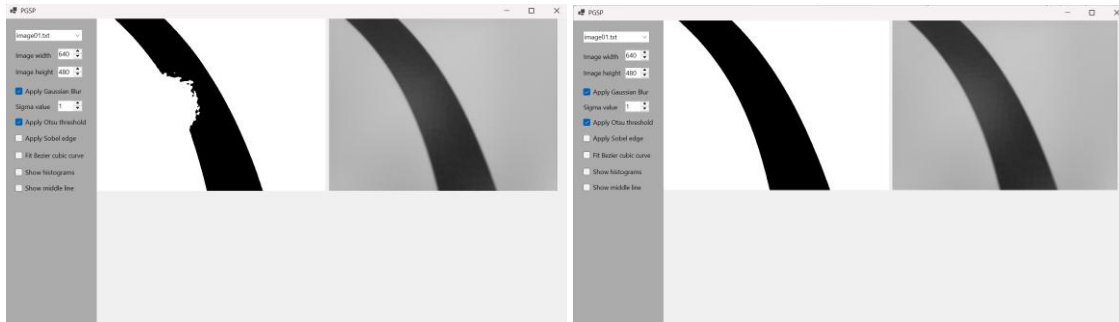
Obsahuje metódu *ApplyEdgeDetection*, ktorá aplikuje na dáta detekciu hrán. Výsledok je možné vidieť v obrázku nižšie.



Obrázok 5 Sobel detekcia hrán

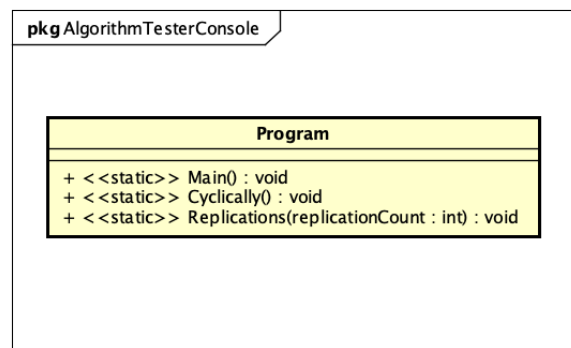
Threshold.cs

Trieda obsahuje dve rôzne metódy pre prahovanie. Jednou je *OtsuThreshold* a druhou je *MeanThreshold*. Obe hľadajú prah, podľa ktorého sa budú dáta-bajty obrázka binarizovať medzi čiernu a bielu farbu. Avšak, v mojom testovaní mi prišiel druhý variant spoľahlivejší, preto sa v hlavnom algoritme používa tento variant, vid' obrázok nižšie (vľavo Otsu, vpravo Mean, rovnaké parametre gauss. šumu).



Obrázok 6 Porovnanie Otsu vs Mean prahovania

Modul PGTester



Obrázok 7 Diagram tried modulu PGTester

Tento modul obsahuje len jeden súbor *Program.cs*, ktorý obsahuje jedinú triedu *Main*. V nej sú všetky potrebné náležitosti pre vykonanie testovania vybraných algoritmov tejto práce. Je možné si výsledky testov ukladať do externého csv súboru, napríklad pre ďalšiu analýzu, prípadne pre tvorbu prehľadných grafov.

Možnosti testov

Test je možné spustiť s uvedením parametrov

- n – počet behov,
- c – cyklické spracovanie všetkých poskytnutých obrázkov rozmeru 640 x 480 px.

Teda napr. zadaním 10 sa vykoná jedna replikácia s 10 behmi algoritmov. Ak zadáme c, tak sa vykoná jedna replikácia v ktorej sa spracujú po sebe všetky obrázky daného rozmeru.

Ak chceme zmeniť počet replikácií, je nutné to zmeniť v zdrojovom kóde v sekcii *Constants* – *NUMBER_OF_REPLICATIONS*. Ak chceme zmeniť obrázok pre testovanie behov – *IMAGE_NAME*, prípadne *DIRECTORY_PATH*. Ak chceme ukladať výsledky replikácií do CSV súboru, je nutné nastaviť konštantu *SAVE_RESULTS* na



true. Ak to nastavíme, výstup sa uloží do priečinku output v hlavnom repozitári. Ak chceme nastaviť, aby sa spracoval len každý n-tý riadok, je nutné zmeniť konštantu PROCESS_EVERY_NTH_LINE.

Externé knižnice

V práci sa nepoužívajú žiadne externé knižnice, len tie, čo sú súčasťou .NET.



Vyhodnotenie rýchlostí

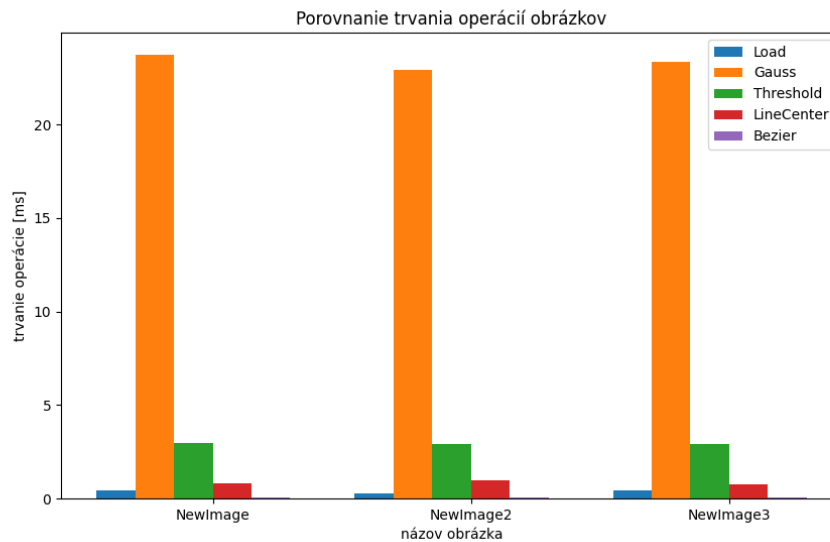
V tejto sekcii sa zhodnotia a porovnajú jednotlivé rýchlosti a experimenty, ktoré som vykonal.

Pokiaľ nie je uvedené inak, tak výsledky sú z build verzie Debug x86, pri počte behov 10 a 10 replikáciách a spracovaní všetkých dát/riadkov obrázka.

Grafy boli vytvorené v programovacom jazyku Python pomocou knižnice matplotlib.

Porovnanie rýchlostí operácií jednotlivých obrázkov

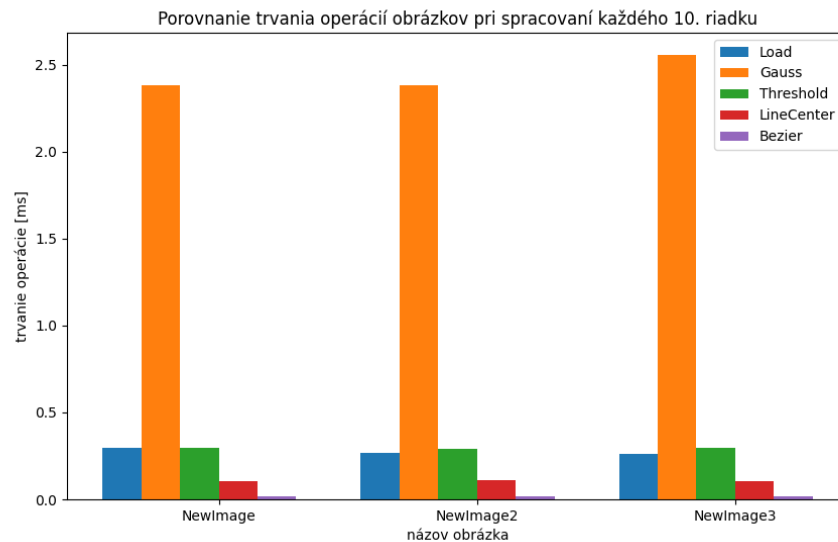
V prvom vyhodnotení si porovnáme, či rozličné obrázky majú nejaký vplyv na trvanie jednotlivých operácií. Ako je možné vidieť v grafe, každá operácia trvá približne rovnako bez ohľadu na obrázok. Najdlhšie trvá operácia aplikácie gaussovho šumu, potom aplikácia prahovania, hľadanie stredu hrany a nakoniec samotné načítanie obrázkov. Hľadanie beziérovej krivky je oproti ostatným operáciám časovo takmer zanedbateľné. Výsledky sú uvedené v nasledovnom grafe.



Obrázok 8 Graf porovnania trvania operácií obrázkov

Porovnanie rýchlostí operácií jednotlivých obrázkov pri spracovaní každého 10. riadku

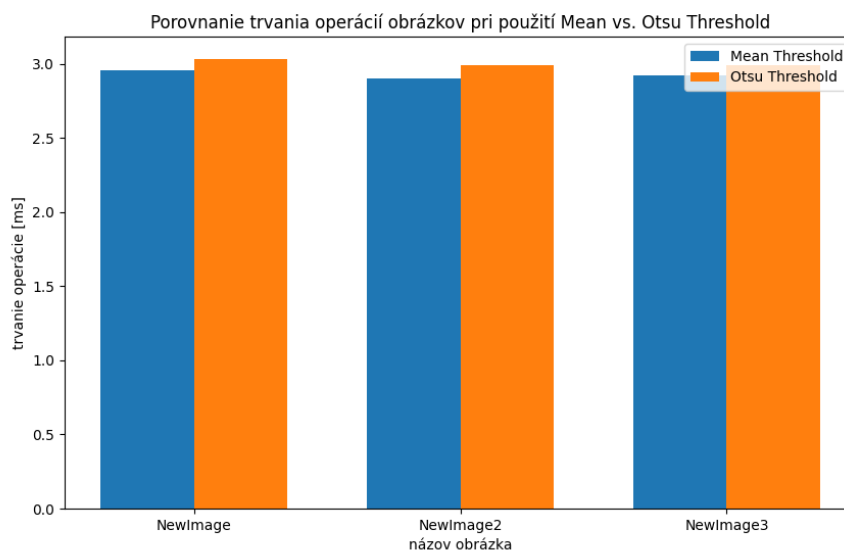
Toto porovnanie porovnáva, aký veľký rozdiel bude oproti predošlému experimentu, ak budeme spracovávať len každý 10. riadok obrázku. Je jasné, že časy budú výrazne lepšie, takmer 10x lepšie. Avšak, čo je zaujímavé, že trvanie načítania obrázku a prahovania trvá takmer rovnako. Zvyšné operácie trvajú proporcionálne rovnako ako v predošlom prípade. Výsledky je možné vidieť v grafe nižšie.



Obrázok 9 Graf porovnania trvania operácií obrázkov pri spracovaní každého 10. riadku

Porovnanie rýchlostí prahovania

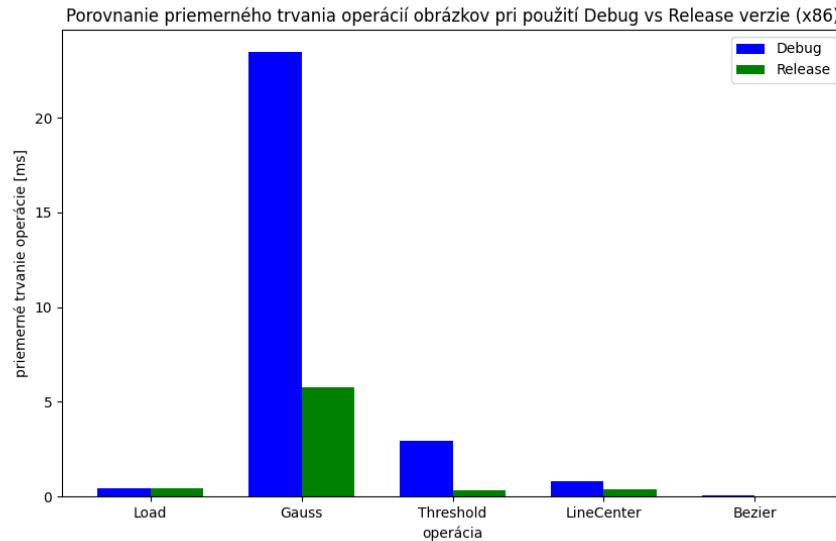
V tomto experimente porovnáme časové trvanie dvoch druhov prahovania, ktoré som implementoval a testoval v práci. Vyššie bolo uvedené, že spoľahlivejšie je Mean prahovanie, ktoré je odôvodnené obrázkom 6. V tomto experimente som chcel porovnať, či je rozdielny aj čas týchto verzií. Avšak, vyzerá to tak, že ich trvanie je takmer totožné. Preto v algoritme bol použitý Mean variant, nakoľko bol naoko spoľahlivejší.



Obrázok 10 Graf porovnania Mean vs Otsu prahovania

Porovnanie build verzií x86

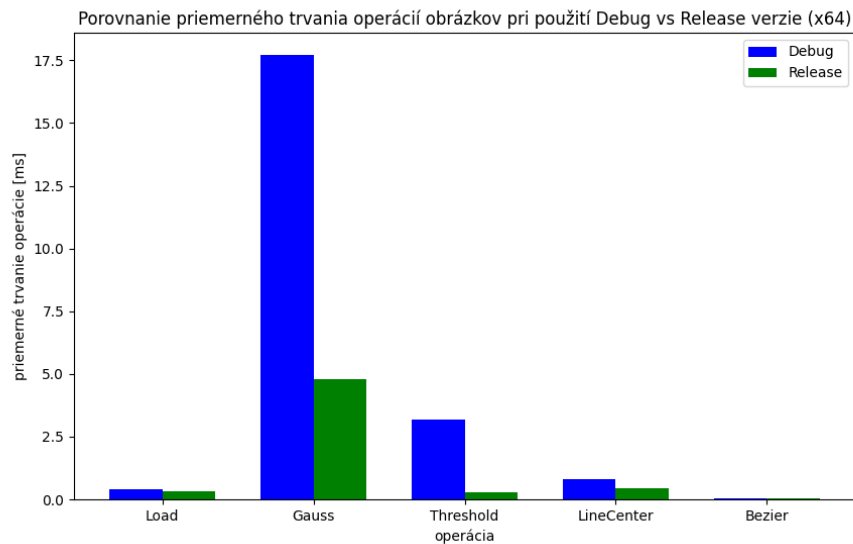
V tomto experimente som chcel otestovať, aký vplyv má verzia Debug a Release programu pre architektúru x86. Ako je možné vidieť v grafe nižšie, načítanie trvá rovnako, avšak pri ostatných operáciách je signifikantný rozdiel trvania operácií v prospech Release verzie.



Obrázok 11 Graf porovnania build verzií x86

Porovnanie build verzií x64

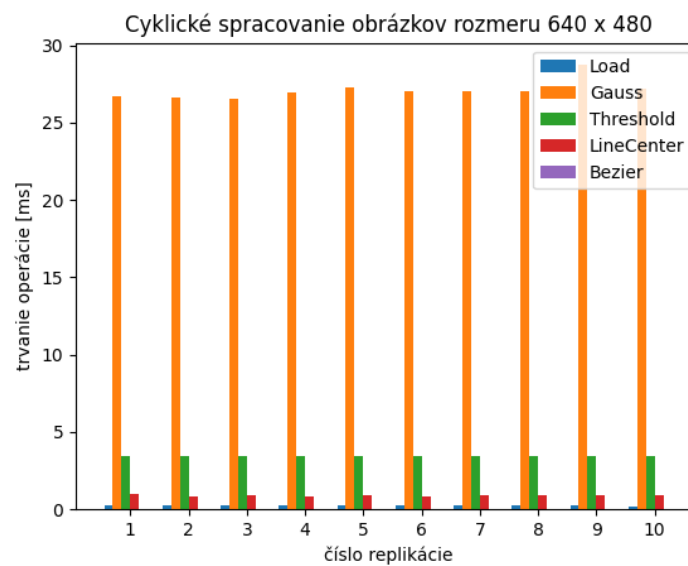
Tento experiment je rovnaký, ako predošlý, avšak pre architektúru x64. Výsledky sú porovnateľné s predošlým prípadom v prospech Release verzie. Rozdiel oproti x86 je v hodnotách osi y, teda časoch operácií – x64 je o čosi rýchlejšie ako x86.



Obrázok 12 Graf porovnania build verzií x64

Cyklické spracovanie obrázkov rozmeru 640 x 480

V poslednom experimente je možné vidieť trvanie 10 replikácií cyklických spracovaní všetkých obrázkov formátu YUV420 a rozmeru 640 x 480 px, ktoré boli poskytnuté. Všetky operácie trvajú v každej replikácii takmer rovnako. Najdlhšie trvá gaussovo zašumenie, prahovanie, hľadanie stredu čiary, načítanie a zanedbateľne fit beziérovej krivky.



Obrázok 13 Graf cyklického spracovania



Záver

Cieľ a zadanie práce bolo splnené. Súčasťou práce sú 3 moduly, ktoré efektívne riešia danú problematiku. Jeden modul obsahuje grafické rozhranie pre manipuláciu s obrázkami. Druhý modul obsahuje konzolové rozhranie pre testovanie rýchlostí jednotlivých algoritmov. Posledný modul je knižnica, ktorá sa využíva v oboch predošlých moduloch. V práci sme popísali jednotlivé moduly, algoritmus riešenia a dôvody použitia daných krokov. Nakoniec sme spravili a vyhodnotili niekoľko experimentov s obrázkami a ich trvaniami. Moduly je možné využiť pre ďalšiu budúcu analýzu a prípadné testovanie algoritmov a operácií.