Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra Informačních Technologií

Strojové vidění na iOS platformě

Diplomová práce

Autor práce: Bc. Tomáš Pařízek

Studijní obor: Aplikovaná Informatika

Vedoucí práce: Ing. Karel Mls, Ph.D.

Hradec Králové 2024

Prohlášení

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

.....

Tomáš Pařízek

17. dubna 2024

Poděkování

Děkuji vedoucímu diplomové práce Ing. Karel Mls, Ph.D. za metodické vedení práce.

Anotace

Diplomová práce "Strojové vidění na iOS platformě" analyzuje a porovnává technologie pro řešení úloh strojového vidění v kontextu vývoje nativních iOS aplikací. V rámci této analýzy se práce zaměřuje na tři vybrané technologie: Vision Framework, Core Image Framework a Google BlazeFace model. Hlavním cílem je vytvořit sadu doporučení pro řešení úloh strojového vidění zaměřenou na iOS vývojáře. Zjištění práce ukazují, že Vision Framework je pro většinu aplikací nejlepší volbou díky jeho integraci v iOS SDK a širokému spektru dostupných funkcí. Výsledky experimentů práce poskytují iOS vývojářům klíčové poznatky pro správný výběr technologie.

Anotation

Title: Computer Vision on iOS platform

The thesis "Machine Vision on iOS Platform" analyzes and compares available Computer vision solutions in the context of native iOS application development. Within this analysis, the thesis focuses on three selected technologies: the Vision Framework, the Core Image Framework and the Google BlazeFace model. The main goal is to create a set of recommendations for solving Computer vision tasks aimed at iOS developers. The findings of the thesis show that the Vision Framework is the best choice for most applications due to its integration in the iOS SDK and the wide range of available features. Results of the thesis experiments provide iOS developers with key insights for making the right technology choice.

Obsah

Seznam obrázků

Pokud u obrázku není uveden zdroj, tak zdrojem obrázku je autor práce.

Seznam tabulek

1 Úvod

V posledních letech se chytré mobilní telefony staly součástí každodenního života pro většinu lidí na naší planetě. [1] Telefony v základu poskytují užitečné funkce, ale jednou z jejich klíčových vlastností jsou aplikační obchody a obecně schopnost telefonu rozšiřovat své funkcionality v průběhu času. Každý člověk je odlišný a díky možnosti stahování aplikací od vývojářů třetích stran si může každý přizpůsobit mobilní telefon svým potřebám.

Problematika umělé inteligence je dnes velmi aktuální téma, mimo jiné kvůli populárnímu generativnímu modelu Chat-GPT, který představil možnosti umělé inteligence široké veřejnosti. [2] Mezi softwarovými inženýry byly nástroje a principy AI populární již v předchozích letech. Jasným důkazem je například Neural Engine - hardwarové čipy nacházející se v chytrých telefonech společnosti Apple od roku 2020, které umožňují hardwarovou akceleraci operací strojového učení a jsou dostupné vývojářům třetích stran prostřednictvím API. [3]

Jednou z nejčastějších aplikací AI ve světě mobilních aplikací je určitá forma strojového vidění neboli strojového porozumění obrazovým datům. Strojové vidění má mnoho využití, například v úpravě fotografie nebo práci s kamerou, a tato využití mohou zásadně zlepšit uživatelský zážitek z používání aplikace.

Tato diplomová práce se zaměřuje na oblast strojového vidění v nativních iOS aplikacích. Ačkoliv se téma může na první pohled zdát velmi specifické, opak je pravdou. Vývojáři iOS aplikací běžně čelí rozhodnutí, jaký nástroj a technologii zvolit pro úlohy v oblasti strojového vidění. Některé balíčky jsou tak rozsáhlé, že se mohou stát pro některé vývojáře nepřehlednými, což může vést k přehlížení užitečných funkcí. Tato práce má za cíl vývojářům pomoci se zorientovat a vybrat správné řešení pro aplikace, které vyvíjejí.

2 Cíl práce

Hlavním cílem diplomové práce "Strojové vidění na iOS platformě"je podrobná analýza problematiky strojového vidění při tvorbě nativních iOS aplikací. Práce se konkrétně zaměřuje na tři balíčky poskytující funkce strojového vidění v iOS ekosystému: Vision Framework, Core Image Framework a BlazeFace model od společnosti Google. Tato práce by měla sloužit jako užitečný přehled, který mohou iOS vývojáři využít při výběru správné technologie strojového vidění ve vývojovém procesu aplikace.

Práce začíná kapitolami, které popisují obecné principy strojového vidění a možnosti jeho implementace v iOS aplikacích. Tyto sekce čtenáře uvedou do správného kontextu a připraví ho na následující kapitoly.

Další sekcí práce je Detekce obličeje, což je ukázkový úkol, na jehož základě byl vytvořen experiment zkoumající vlastnostni testovaných řešení. Součástí této sekce je vývoj vlastní mobilní aplikace, která se následně použije k měření a porovnávání výsledků detekce obličejů v různých situacích. Sekce obsahuje podrobnou analýzu naměřených dat, která na konci vyvozuje doporučení pro implementaci detekce obličeje.

Poslední části práce čtenáře seznámí s tvorbou vlastního klasifikačního modelu v Create ML a detailně rozeberou Vision Framework, protože se jedná o aktuálně oficiálně doporučenou metodu řešení strojového vidění podle společnosti Apple, na rozdíl od Core Image Framework a BlazeFace modelu. [4]

3 Strojové vidění

3.1 Úvod do strojového vidění

Strojové vidění, známé také pod anglickým názvem Computer Vision, je proces umožňující strojům porozumět obrazovým datům, typicky fotografii nebo snímku z video sekvence. Velké množství smart systémů dnešní doby využívá určitou formu strojového vidění, například autonomní automobily, bezpečnostní systémy, fotoaparáty, automatizaci QA a mnohé další. Různá využití strojového vidění obvykle mají odlišné cíle při práci s daty, což vede k rozdělení oblasti na několik dílčích problémů, jako je detekce obličejů, klasifikace objektů, detekce emocí lidí atd.[5]

První náznaky strojového porozumění obrazovým datům pocházejí již z 50. let minulého století, kdy byly vyvinuty první algoritmy strojového vidění založené primárně na detekci geometrických vzorů. Jedním z prvních komplexnějších projektů byl Summer Vision, který vznikl v roce 1966 na Massachusettském technologickém institutu (MIT) s cílem vytvořit program pro rozpoznávání objektů na obrázcích. [6]

Dnes je většina implementací strojového vidění založena na principech konvulenčních neuronových sítí. Tyto sítě, díky procesu postupné konvulence skrz obraz, jsou schopné efektivně určit kontext v jednotlivých částech obrazu. Konvulenční neuronové sítě dosahují vysoké úrovně přesnosti pro různé úlohy strojového vidění jako je rozpoznávání objektů, segmentace obrazu, generování obrazu, stylizace obrazu nebo super rozlišení (umělé zvětšování rozlišení obrazu). Mezi některé z nejznámějších architektur konvulenčních neuronových sítí patří AlexNet, VGG, ResNet, Inception, U-Net, GAN nebo CycleGAN. Tyto sítě jsou obvykle trénovány na velkých datasetech, jako například ImageNet, COCO, CelebA nebo Cityscapes. [7]

Strojové vidění je stále aktivní a rychle se rozvíjející oblast. Současné výzvy v této oblasti zahrnují nejen zvýšení efektivity a přesnosti, ale v posledních letech se společnost začala více zabývat také otázkami etiky, rasové diskriminace a bezpečnosti osobních dat. Tyto problémy jsou závažné a stále nemají jasná řešení. Jako příklad lze uvést generativní umělou inteligenci ChatGPT, představenou v roce 2023 americkou společností OpenAI. ChatGPT rychle získal popularitu a miliony uživatelů, ale je otázkou, kolik z nich si je vědomo, že jakýkoliv vložený vstup do ChatGPT může být a pravděpodobně je využit pro další učení nástroje? Z toho vyplývá, že při vložení citlivých informací (například obsahu

smlouvy) do konverzace s ChatGPT, se tyto informace teoreticky mohou objevit v budoucí odpovědi na jiný dotaz jiného uživatele. Obdobné otázky mohou být položeny u nástrojů pracujících s technologiemi strojového vidění jako Google Photos

3.2 Mobilní aplikace

Procesy strojového vidění jsou dnes běžně aplikovány v oblasti mobilních aplikací. Typickým příkladem jsou aplikace pro ovládání kamery telefonu, které díky softwarovým vylepšením obrazu dokáží vytvořit vizuálně atraktivní fotografie a videa, i když senzor mobilního telefonu má obvykle malé rozměry. Klíčovou součástí těchto vylepšení je porozumění obrazu; například portrétní fotografie vyžadují odlišné úpravy než panoramatické snímky. [5]

Strojové vidění lze využít v mobilních aplikacích nejenom ke zlepšení kvality fotografií a videí. Aplikace mohou například detekovat nevhodný obsah pro děti, pokud uživatelé mají možnost nahrávat fotografie, které aplikace následně zobrazuje ostatním uživatelům (typicky na sociálních sítích). Další užitečnou aplikací strojového vidění ve světě mobilních aplikací je detekce obličejů. Populární součástí aplikací je forma nekonečně scrollovatelného obrazového obsahu, často označovaného jako "feed". Snímky zobrazované ve feedu obvykle mají různé poměry stran, což vyžaduje od aplikací oříznutí obrazu tak, aby vyplnily celý svůj prostor ve feedu. Standardní ořezem obrazu ze středu můžeme přijít o důležité části snímku, kde se třeba nachází obličej subjektu. Této situaci můžeme předejít pomocí detekce obličejů, jak je ilustrováno na Obrázku 1.

Kromě výše uvedených aplikací se strojové vidění v mobilních zařízeních využívá i v pokročilých technologiích, jako je rozšířená realita (AR) a virtuální realita (VR). Tyto technologie umožňují vytvářet interaktivní prostředí, ve kterém se virtuální objekty přirozeně integrují do reálného světa. Například, AR aplikace mohou využívat strojové vidění k rozpoznávání a sledování pohybu uživatele, umožňující tak interakci s virtuálními prvky ve fyzickém světě. Dalším rozšířeným využitím je rozpoznávání textu pro překlad nebo přístup k informacím, které se v reálném čase zobrazují na displeji uživatele. Tyto inovativní aplikace ukazují, jak se strojové vidění stává klíčovou technologií v mobilním ekosystému, zvyšující možnosti interakce a poskytující nové způsoby zážitku a učení. [5]

Obrázek 1: Ořez obrazu na feed obrazovce pomocí strojového vidění

3.3 Možnosti implementace strojového vidění na iOS platformě

Tato práce se primárně zaměřuje na nativní aplikace pro zařízení s operačním systémem iOS, ale obecné principy a využití se samozřejmě vztahují i ke konkurenční platformě Android.

Vývojáři iOS aplikací mají tři základní možnosti pro implementaci strojového vidění:

1. Core Image Framework
2. Vision Framework
3. Vlastní neuronová síť

Kromě výše uvedených možností existují také specializované služby a frameworky, které se obvykle soustřeďují na určité oblasti strojového vidění, jako je například OCR (optické rozpoznávání znaků). Tyto specializované služby často nabízejí omezenou funkcionalitu, jsou zpoplatněné a mohou mít složité licenční podmínky, což omezuje jejich široké využití ve srovnání s třemi výše zmíněnými možnostmi. Tato práce se detailněji zaměřuje na tyto tři základní možnosti, poskytující široké spektrum nástrojů a flexibilitu pro vývoj aplikací.

3.4 Core Image Framework

Core Image Framework je součástí iOS SDK již od verze iOS 5.0, která byla publikována v roce 2011. Tento framework se primárně využívá pro tvorbu a úpravu obrazových dat, ale obsahuje také funkce pro analýzu obrazu. [8]

Detekci objektů v obrazu framework umožňuje pomocí třídy CIDetector, která podporuje čtyři typy detekce:

1. Obličej
2. Obdélník
3. QR kód
4. Text

Core Image Framework je vhodný pro rychlé a efektivní zpracování obrazu, s důrazem na real-time aplikace, jako jsou filtry pro kamery a základní analýza obrazu. [8]

3.5 Vision Framework

Vision Framework, zavedený s iOS 11 v roce 2017, představuje nejnovější nástroj od Applu pro analýzu obrazových dat. Tento framework je založený primárně na konvulenčních neuronových sítích a využívá hardwarovou akceleraci výpočtů prostřednictvím neural engine čipu, který v iPhone 15 Pro nabízí 16 jader schopných provést 17 bilionů operací za sekundu. [4][9]

Vision Framework rozšiřuje možnosti Core Image Frameworku o komplexnější funkce, jako jsou klasifikace fotografií, srovnání podobností obrazů nebo trackování pozice objektů ve video sekvencích. Framework je ideální pro pokročilé úlohy strojového vidění, včetně rozpoznávání a interpretace obsahu v reálném čase. [4]

3.6 Vlastní neuronová síť

Kromě Core Image a Vision Frameworků, které jsou standardními součástmi iOS SDK a zjednodušují implementaci bez podrobných znalostí strojového učení, mohou vývojáři vytvářet i vlastní řešení. Tato možnost se hodí, pokud standardní frameworky nevyhovují specifickým požadavkům aplikace.

Apple nabízí integraci s Neural Engine čipem prostřednictvím Core ML frameworku. Modely pro Core ML mají příponu .mlmodel a jsou dostupné na webových stránkách Applu, včetně ukázkových modelů, jako je MNIST klasifikace nebo Resnet50 architektura. Core ML tools umožňují převod TensorFlow a PyTorch modelů do formátu Core ML, rozšiřují tak možnosti využití různých modelů. [10]

Funkce	Core Image Framework	Vision Framework	Vlastní neuronová síť
Detekce obličeje	Ano	Ano	Ano
Detekce obdélníku	Ano	Ano	Ano
Detekce QR kódu	Ano	Ano	Ano
Detekce textu	Ano	Ano	Ano
Klasifikace	Ne	Ano	Ano
Práce s video sekvencí	Ne	Ano	Ano
Podpora vlastního modelu	Ne	Ne	Ano

Tabulka 1: Porovnání poskytovaných funkcí řešení

TensorFlow Lite, specializovaná verze pro mobilní zařízení, nabízí optimalizaci pro běh na mobilních zařízeních. Pro implementaci TensorFlow Lite modelů do iOS aplikací je nutné přidat knihovnu TensorFlowLiteSwift. Limitací TensorFlow Lite modelů je absence podpory v Core ML Tools a nemožnost zpětného převodu na TensorFlow modely, což vyžaduje využití oficiální TensorFlowLiteSwift knihovny pro integraci modelů do aplikací. [11]

4 Detekce obličeje

4.1 Definice zadání

Předchozí kapitola představila tři způsoby implementace strojového vidění na iOS platformě: Core Image Framework, Vision Framework a vlastní neuronovou síť. Tabulka 1 poskytuje přehled o podporovaných úlohách strojového vidění pro každé z těchto řešení.

Cílem této sekce je srovnat přesnost a rychlost těchto tří řešení na rozličných zařízeních. Pro tento účel je třeba zvolit testovací úlohu, kterou všechna tři řešení podporují. Jako vhodnou testovací úlohu jsem vybral detekci obličeje ve statickém obrazu. Tato úloha je podporovaná všemi třemi řešeními a má široké uplatnění v reálném světě mobilních aplikací, jak ilustruje Obrázek 1.

Požadovaným výstupem pro úlohu detekce obličeje ve statickém obrazu je seznam oblastí, kde byl obličej detekován. Každá taková oblast má tvar obdélníku definovaného čtyřmi body v souřadnicovém systému rastrové mřížky vstupního obrazu. Některé metody detekce mohou navíc poskytovat procentuální pravděpodobnost existence obličeje v dané oblasti.

Úlohu detekce obličeje ve statickém obrazu lze dále rozšířit o dodatečná výstupní data, jako jsou souřadnice očí nebo úst. Jeden ze sofistikovanějších přístupů je, že detekovaná oblast není omezena na obdélníkový tvar, ale může být definována polem bodů, které přesně vymezují okraj obličeje. To je obzvláště potřebné pro aplikace jako jsou softwarové rozmazání pozadí portrétních fotografií.

Kromě základní detekce obličeje a souřadnic prvků obličeje je také možné provádět rozpoznávání emocí nebo identifikace jednotlivců. Tyto aplikace strojového vidění využívají pokročilé algoritmy a modely, které dokážou analyzovat nejen pozici obličeje, ale i jeho mikrovýrazy a další charakteristiky. [12] V rámci tohoto srovnání se však zaměříme pouze na základní výstupy detekce obličeje, jak bylo specifikováno výše.

4.2 Sada testovacích obrázků

Výběr obrázků pro porovnání řešení detekce obličeje je klíčový, protože cíl práce je objektivní srovnání řešení na základě jejich rychlosti a přesnosti na různých zařízeních za různých podmínek (míněno typu vlastností obrazu, na kterých je detekce prováděna). Pro objektivní srovnání jsem zvolil 3 obrázky (viz Obrázek 2) s rozdílným počtem obličejů (1, 2 a 7). Osoby na obrázcích mají různé oblečení, pokrývky hlavy, pohlaví a barvu pleti, tak aby

experiment otestoval vybrané řešení za různých podmínek. Všechny zvolené obrázky jsou veřejně zdarma dostupné na webovém portálu Unsplash.com

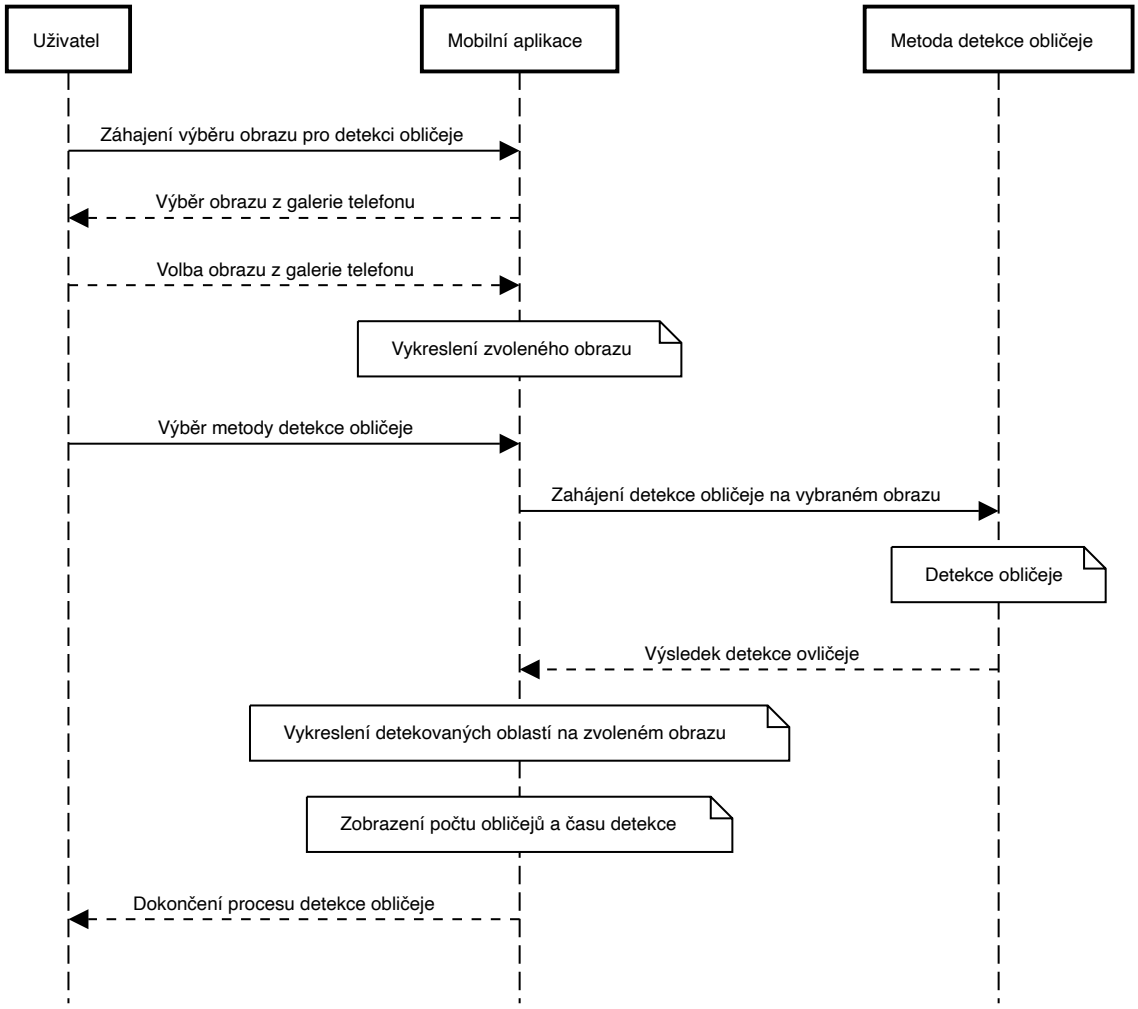
Obrázek 2: Sada testovacích obrázků pro detekci obličeje

4.3 Zadání pro testovací mobilní aplikaci

Cílem je objektivní porovnání různých řešení detekce obličeje a minimalizace vnějších faktorů, které by mohly detekci ovlivnit. Použití veřejně dostupných aplikací nebo ukázkových

projektů jednotlivých řešení by mohlo výsledky zkreslit. Proto jsem se rozhodl vytvořit vlastní nativní mobilní aplikaci, abych měl kontrolu nad celým procesem.

Aplikace umožní uživateli vybrat testovaný obrázek a provést detekci obličeje pomocí různých řešení. Po dokončení detekce uživatelské rozhraní zobrazí výsledky. Tento proces je zaznamenán v sekvenčním diagramu na Obrázku 3.

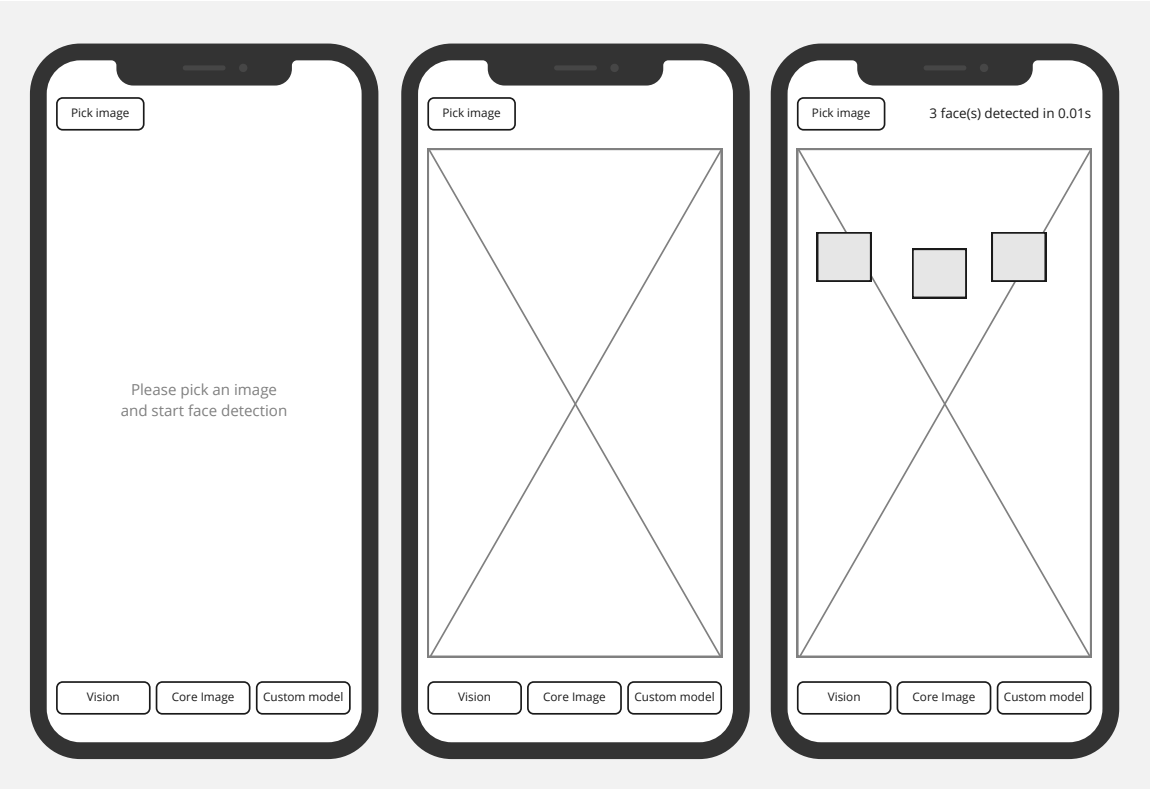


Obrázek 3: Sekvenční diagram detekce obličeje pomocí testovací mobilní aplikace

Uživatelské rozhraní testovací mobilní aplikace by mělo být jednoduché a intuitivní na ovládání, tak aby testovací měření mohla být provedena efektivně. Na základě těchto cílů a požadované funkcionality (viz sekvenční diagram na Obrázeku 3) byl vytvořen návrh uživatelského rozhraní, který se skládá z 1 obrazovky se 3 možnými stavy, viz Obrázek 4.

Horní čast aplikace bude obsahovat tlačítko pro výběr testovacího obrazu zarovnané k levé hraně displeje. Pravá horní část navržené obrazovky zobrazuje výsledek měření (počet

detekovaných obličejů a čas detekce) a je zobrazena až po dokončení procesu detekce obličeje. Tlačítka pro spuštění procesu detekce obličeje pomocí jednotlivých řešení bude aplikace zobrazovat v dolní části obrazovky. Střed aplikace bude dominován vybraným testovacím obrazem, na který aplikace vykreslí detekované oblasti po dokončení procesu detekce.



Obrázek 4: Drátový model uživatelského rozhraní testovací mobilní aplikace

4.4 Programování nativní iOS aplikace

Tvorba nativních mobilních aplikací pro iOS je dnes možná pouze pomocí integrovaného vývojářského prostředí Xcode od společnosti Apple. Prostředí Xcode je distribuováno pouze pro operační systém macOS; jeho stažení je možné bez poplatku skrze oficiální macOS aplikační obchod AppStore. Během vývoje aplikace je možné aplikaci testovat na simulátoru, jenž je součástí Xcode, ale pro reálné ověření funkčnosti je vždy lepší aplikaci spustit na reálném zažízení. Z toho vyplývá, že pro vývoj nativních iOS aplikací je dnes nutné mít k dispozici počítač s operačním systémem macOS a testovací iPhone/iPad zařízení. Nativní iOS aplikace se historicky psaly v programovacím jazyce Objective-C, ale dnes je již drtivá

většina nových aplikací a projektů psána pomocí jazyku Swift, jenž Apple představil v roce 2014 jako nástupce Objective-C.

Vývojáři nativních iOS aplikací mají dnes možnost volby mezi 2 způsoby tvorby uživatelského rozhraní aplikace - novější deklarativní SwiftUI a starší imperativní UIKit. Apple SwiftUI představil v roce 2019 jako budoucí standard tvorby UI pro všechna jejich zařízení. Velkou výhodou SwiftUI je fakt, že je vzájemně kompatibilní s UIKitem, takže je možné v rámci jednoho layoutu použít oba přístupy. Apple každý rok rozšiřuje SwiftUI o nové komponenty a modifiers (možnosti úprav komponent), ale už z deklarativního principu vyplývá, že UIKit poskytuje vývojářům lepší možnosti tvorby komplexních a specifických layoutů. Dnes již drtivou většinu layutů je možné postavit pouze se SwiftUI a pro drobné specifické situace, které SwiftUI nezvládne, není problém stále použít UIKit. [13]

4.5 Tvorba testovací mobilní aplikace

Prvním krokem v procesu tvorby testovací aplikace, po založení nového projektu v Xcode, je definice protokolu pro detekci obličeje a struktury pro výsledek detekce. Protokol v jazyce Swift funguje podobně jako rozhraní v jazyce Java, což znamená, že definuje soubor metod a vlastností, které musí splňovat třídy nebo struktury implementující tento protokol. Struktura výsledků detekce bude obsahovat informace o pozici a rozměrech detekovaných oblastí obličejů, případně pravděpodobnost detekce, pokud je dostupná.

Dalším krokem k dokončení aplikace je implementace jednotlivých detektorů.

```
import CoreGraphics

struct FaceDetection {
    let boundingBox: CGRect
    let confidence: Float?
}
```

Obrázek 5: FaceDetection struktura pro ukládání výsledků detekce obličeje

4.5.1 Core Image Framework

Implementace detektoru obličejů pomocí Core Image Framework je jednoduchá. Prvním krokem je konverze UIImage, což je standardní datová struktura pro rastrové obrazy v iOS

```swift
import UIKit

protocol FaceDetector {
    func detectFaces(in image: UIImage) async throws -> [FaceDetection]
}
```

Obrázek 6: FaceDetector protokol definující veřejné rozhraní detektorů obličeje

aplikacích, na CIImage (kde CI značí Core Image). Tato konverze je nezbytná, protože CIDetector pracuje s CIImage formátem.

Po konverzi je dalším krokem vytvoření instance třídy CIDetector. Tato instance se inicializuje s parametry, které určují požadovanou přesnost a typ detekce. V případě testovací aplikace této práce je typ detekce nastaven na detekci obličejů. Parametry konstruktoru CIDetectoru umožňují vývojářům nastavit různé možnosti, jako je například přesnost detekce, která může být nastavena na vysokou pro lepší výsledky nebo na nízkou pro rychlejší zpracování.

Následně se použije metoda features(in:) třídy CIDetector, která se zavolá s CIImage jako parametrem. Návratová hodnota této funkce představuje výsledky detekce obličejů, typicky ve formě kolekce objektů CIFeature. Každý objekt CIFeature reprezentuje jeden detekovaný obličej a obsahuje informace o jeho poloze a rozměrech.

Posledním krokem v procesu detekce je transformace souřadnic detekovaných regionů do souřadnicového systému původního obrázku. Tento krok je důležitý, protože souřadnicový systém CIImage může být odlišný od systému použitého v UIImage. Tato transformace zajišťuje, že detekované oblasti obličejů budou správně umístěny na původním obrázku. [8]

```
final class CoreImageDetector: FaceDetector {
    func detectFaces(in image: UIImage) async throws -> [FaceDetection] {
        let ciImage = CIImage(cgImage: image.cgImage!)

        let options = [
            CIDetectorAccuracy: CIDetectorAccuracyHigh
        ]

        let faceDetector = CIDetector(
            ofType: CIDetectorTypeFace,
            context: nil,
            options: options
        )!

        let faces = faceDetector.features(in: ciImage)

        let faceDetections = faces.map {
            FaceDetection(
                boundingBox: CGRect(
                    x: $0.bounds.origin.x,
                    y: image.size.height - $0.bounds.origin.y -
                        $0.bounds.size.height,
                    width: $0.bounds.size.width,
                    height: $0.bounds.size.height
                ),
                confidence: nil
            )
        }

        return faceDetections
    }
}
```

Obrázek 7: CoreImageDetector (Core Image Framework detektor)

4.5.2 Vision Framework

Implementace detekce obličeje pomocí Vision Framework vyžaduje více kódu ve srovnání s Core Image Frameworkem, což je dáno hlavně asynchronním způsobem zpracování a předávání výsledků. Proces detekce začíná vytvořením instance třídy VNDetectFaceRectanglesRequest. Tato třída je speciálně navržena pro detekci obličejů a vyžaduje closure funkci pro zpětné volání, které slouží k předání výsledků detekce.

Dále je třeba vytvořit instanci třídy VNImageRequestHandler s parametrem CGImage. CG prefix značí Core Graphics, a UIImage je možné převést na CGImage pomocí vlastnosti cgImage. Tato konverze je nutná, protože Vision Framework pracuje s CGImage formátem.

Posledním krokem v procesu detekce je spuštění metody perform třídy VNImageRequestHandler s předáním instance VNDetectFaceRectanglesRequest jako parametru. Tato metoda provede samotnou detekci obličeje. [4]

Zbytek kódu v třídě VisionDetector, jak je znázorněno na Obrázcích 8 a 9, se věnuje zpracování výsledků zpětného volání. To zahrnuje předání výsledku z closure funkce jako výstupní hodnoty asynchroní funkce detectFaces. Dalším krokem je transformace detekovaných regionů obličejů do souřadnicového systému původního obrazu.

```
final class VisionDetector: FaceDetector {
    private let results: CurrentValueSubject<Result<[FaceDetection], Error>?,
        Never> = .init(nil)
    private var cancellables: Set<AnyCancellable> = []

    func detectFaces(in image: UIImage) async throws -> [FaceDetection] {
        results.send(nil)

        let request = VNDetectFaceRectanglesRequest {
            self.handleDetectedFaces(
                image: image,
                request: $0,
                error: $1
            )
        }
        let imageRequestHandler = VNImageRequestHandler(cgImage:
            image.cgImage!)
        try imageRequestHandler.perform([request])

        return try await withCheckedThrowingContinuation { continuetion in
            results
                .compactMap { $0 }
                .first()
                .sink(receiveValue: { result in
                    switch result {
                    case .success(let faceDetections):
                        continuetion.resume(returning: faceDetections)
                    case .failure(let error):
                        continuetion.resume(throwing: error)
                    }
                })
                .store(in: &cancellables)
        }
    }
}
```

Obrázek 8: VisionDetector (Vision Framework detektor) - část 1/2

```
private extension VisionDetector {
    func handleDetectedFaces(image: UIImage, request: VNRequest?, error:
        Error?) {
        if let error {
            results.send(.failure(error))
            return
        }

        guard let faces = request?.results as? [VNFaceObservation]
        else {
            results.send(.success([]))
            return
        }

        let faceDetections = faces.map { detection in
            let detectionBounds = CGRect(
                x: detection.boundingBox.minX * image.size.width,
                y: (1 - detection.boundingBox.maxY) * image.size.height,
                width: detection.boundingBox.width * image.size.width,
                height: detection.boundingBox.height * image.size.height
            )
            return FaceDetection(
                boundingBox: detectionBounds,
                confidence: detection.confidence
            )
        }

        results.send(.success(faceDetections))
    }
}
```

Obrázek 9: VisionDetector (Vision Framework detektor) - část 2/2

4.5.3 Vlastní neuronová síť

Kromě standardních řešení poskytovaných iOS, jako jsou Vision Framework a Core Image Framework, existuje také možnost využít vlastní model neuronové sítě pro detekci obličeje. Apple sice poskytuje různé ukázkové modely v rámci Core ML Frameworku, ale žádný z nich se přímo nespecializuje na detekci obličeje. [10] Existuje velké množství open source modelů, které je možné zdarma využít. Některé modely jsou přímo v Core ML formátu, ale v iOS aplikaci je možné využít i TensorFlow nebo PyTorch modely po konverzi v CoreML Tools nástroji. [14] Další možností implementace vlastního modelu je TensorFlow Lite, což je zjednodušená verze TensorFlow určená a optimalizovaná primárně pro mobilní zařízení. [11]

V rámci výběru vhodného modelu pro tuto práci jsem po dlouhém srovnávání různých modelů zvolil BlazeFace model od společnosti Google. Jedná se o primární model, který společnost Google momentálně využívá pro detekci obličejů na všech mobilních platformách. BlazeFace model je veřejně dostupný v TensorFlow Lite formátu, takže není problém model integrovat do vlastní iOS aplikace. Primární důvod volby BlazeFace modelu byla důvěryhodnost tvůrce modelu - společnost Google - oproti ostatním opensource modelům. [15]

Pro implementaci BlazeFace modelu do iOS aplikace Google poskytuje knihovnu MediaPipeTasksVision, která je nadstavbou standardní TensorFlow Lite knihovny. Jak je ukázáno na Obrázku 10, prvním krokem je vytvoření konfigurace detekce pomocí FaceDetectorOptions. Následuje vytvoření instance třídy FaceDetector prostřednictvím funkce detectFaces, která se použije k detekci obličejů v testovaném obrazu. [16]

```
final class BlazeDetector: FaceDetector {
    private enum Constants {
        static let modelAssetPath = Bundle.main.path(forResource:
            "blaze_face_short_range", ofType: "tflite")!
    }

    func detectFaces(in image: UIImage) async throws -> [FaceDetection] {
        let faceDetectorOptions = FaceDetectorOptions()
        faceDetectorOptions.runningMode = .image
        faceDetectorOptions.baseOptions.modelAssetPath =
            Constants.modelAssetPath

        let faceDetector = try MediaPipeTasksVision.FaceDetector(options:
            faceDetectorOptions)

        let mpImage = try MPImage(uiImage: image)
        let result = try faceDetector.detect(image: mpImage)

        let faceDetections = result.detections.map { detection in
            FaceDetection(
                boundingBox: detection.boundingBox.rotated(by: image),
                confidence: nil
            )
        }

        return faceDetections
    }
}
```

Obrázek 10: BlazeDetector (detektor vlastní neuronové sítě)

4.5.4 Uživatelské rozhraní mobilní aplikace

Při tvorbě nativní iOS aplikace vývojáři volí mezi dvěma hlavními způsoby definice uživatelského rozhraní: UIKit a SwiftUI, jak je popsáno v sekci 4.4. Uživatelské rozhraní testovací mobilní aplikace se skládá pouze z jedné obrazovky, viz sekce 4.3.

Layout uživatelského rozhraní testovací aplikace byl vytvořen v .storyboard souboru pomocí UIKit ovládacích elementů. Zdrojový kód aplikace obsahuje třídu ViewController (potomek UIViewController třídy), jejímž úkolem je obsluha uživatelského rozhraní a propojení rozhraní s business logikou aplikace. Komunikace mezi ViewControllerem a layoutem v Storyboard souboru zajišťuje iOS SDK - @IBOutlet a @IBAction notace ve zdrojovém kódu (IB prefix značí interface builder).

Uživatel aplikace má možnost volby testovacícho obrázku z galerie mobilního telefonu. Pro tento účel aplikace využívá nativní třídu UIImagePickerController, která poskytuje rozhraní pro výběr obrázku, viz funkce pickImageButtonTapped na obrázku 11. UIImagePickerController zpět k aplikaci komunikuje pomocí protokolu UIImagePickerControllerDelegate, viz obrázek 13.

```
final class ViewController: UIViewController {
    @IBOutlet weak var imageView: UIImageView!
    @IBOutlet weak var numberOfFaces: UILabel!

    private let visionDetector = VisionDetector()
    private let coreImageDetector = CoreImageDetector()
    private let blazeDetector = BlazeDetector()
    private let imageSubject = CurrentValueSubject<UIImage?, Never>(nil)
    private let facesSubject = CurrentValueSubject<[FaceDetection], Never>([])
    private let detectionTimeSubject = CurrentValueSubject<CFAbsoluteTime?,
        Never>(nil)
    private var pathLayer: CALayer?
    private var cancellables = Set<AnyCancellable>()

    override func viewDidLoad() {
        super.viewDidLoad()
        setupBinding()
    }

    @IBAction func pickImageButtonTapped(_ sender: Any) {
        let picker = UIImagePickerController()
        picker.delegate = self
        present(picker, animated: true)
    }
    @IBAction func detectButtonTapped(_ sender: Any) {
        detectFaces(using: visionDetector)
    }
    @IBAction func ciDetectButtonTapped(_ sender: Any) {
        detectFaces(using: coreImageDetector)
    }
    @IBAction func blazeDetectButtonTapped(_ sender: Any) {
        detectFaces(using: blazeDetector)
    }
}
```

Obrázek 11: ViewController třída

```
private extension ViewController {
    func detectFaces(using detector: FaceDetector) {
        guard let image = imageSubject.value
        else { return }

        Task {
            do {
                let detectionStartTime = CFAbsoluteTimeGetCurrent()
                let detectionResult = try await detector.detectFaces(in: image)
                let executionTime = CFAbsoluteTimeGetCurrent() -
                    detectionStartTime
                detectionTimeSubject.send(executionTime)
                facesSubject.send(detectionResult)
            } catch {
                handleError(error)
            }
        }
    }
}
```

Obrázek 12: ViewController třída - detekce obličeje

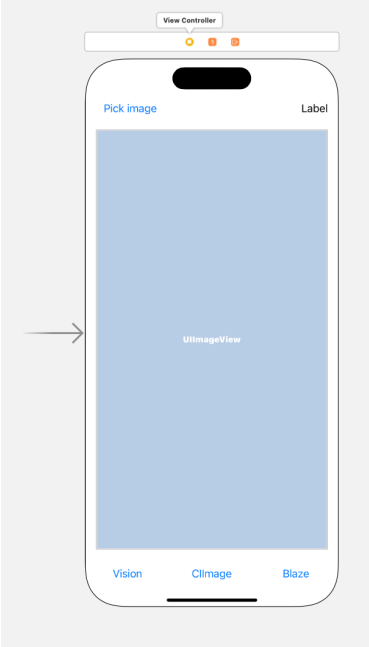
```
extension ViewController: UIImagePickerControllerDelegate,
    UINavigationControllerDelegate {
    func imagePickerController(_ picker: UIImagePickerController,
        didFinishPickingMediaWithInfo info: [UIImagePickerController.InfoKey :
        Any]) {
        guard let image = info[.originalImage] as? UIImage
        else { return }

        imageSubject.send(image)

        dismiss(animated: true)
    }

    func imagePickerControllerDidCancel(_ picker: UIImagePickerController) {
        picker.dismiss(animated: true)
    }
}
```

Obrázek 13: ViewController třída - výběr obrázku



Obrázek 14: Storyboard soubor

Obrázek 15: Snímek obrazovky výsledného uživatelského rozhraní

4.6 Metodika měření

Hlavním cílem experimentu je objektivně srovnat proces detekce obličejů za různých podmínek. Měření detekce bude prováděno na sadě 3 různorodých obrázků (viz kapitola 4.2) na 14 odlišných modelech iPhone. Před zahájením detekce bude baterie zařízení nabita na 100% a operační systém aktualizován na nejnovější verzi dostupnou pro dané zařízení.

Detekce obličejů bude provedena 10x sekvenčně pro každou kombinaci zařízení, způsobu detekce a testovacího obrázku, aby experiment mohl určit rozdíly mezi první detekcí a následujícími. Před každou sérií deseti detekcí bude aplikace restartována, což zamezí ovlivnění měření předchozími detekcemi (například díky alokované paměti).

Celkový počet detekcí obličejů v rámci experimentu dosáhne 1260 (3 způsoby detekce x 3 obrázky x 10 detekcí x 14 zařízení). Výsledky detekce budou uloženy do CSV souboru, který umožní snadné následné zpracování dat pro analýzu výsledků. CSV soubor bude obsahovat dva sloupce: počet detekovaných obličejů a čas detekce, a to pro každé z 10 měření.

Záznamy z CSV souboru budou následně přesunuty do MySQL relační databáze pro snadnější analýzu dat. Transformaci záznamů z CSV souborů do databáze zajistí NodeJS script.

4.6.1 Seznam testovacích zařízení

- iPhone 6s
- iPhone SE (první generace)
- iPhone 7
- iPhone 8
- iPhone XR
- iPod touch (sedmá generace)
- iPhone 11
- iPhone 12
- iPhone 12 mini
- iPhone 13

- iPhone 13 Pro
- iPhone 14
- iPhone 14 Pro
- iPhone 15 Pro

4.7 Analýza výsledků

4.7.1 Počet detekovaných obličejů

Rychlost detekce a přesnost určení oblasti obličeje jsou důležité charakteristiky detekce obličejů, avšak nejdůležitější je schopnost správně určit počet obličejů na obrázku.

Detekce obličejů na testovacích obrázcích s 1 až 2 obličeji (viz obrázek 2) dosáhla 100% úspěšnosti ve správném určení počtu obličejů u všech metod detekce a na všech zařízeních.

Třetí testovací obrázek obsahující 7 obličejů představoval největší výzvu pro detekci, primárně kvůli menší velikosti obličejových oblastí a různorodosti osob na obrázku. Blaze Framework na všech zařízeních nedokázal detekovat žádný obličej. Vision Framework a Core Image Framework detekovaly 6 nebo 7 obličejů. Přesné počty detekcí jsou uvedeny v tabulce 2.

Způsob detekce	Počet detekcí	Počet záznamů
Blaze	0	140
Core Image Framework	6	50
Core Image Framework	7	90
Vision Framework	6	30
Vision Framework	7	110

Tabulka 2: Počet detekovaných obličejů

Způsob detekce	Úspěšnost
Blaze	0%
Core Image Framework	64%
Vision Framework	78%

Tabulka 3: Úspěšnost správné detekce počtu obličejů

Je zajímavé, že neúspěšná detekce se vyskytovala pouze na starších zařízeních, která nepodporují nejnovější verzi operačního systému iOS 17. Konkrétně se jednalo o zařízení:

- iPhone 6s
- iPhone 7
- iPhone 8
- iPhone SE (první generace)
- iPod touch (sedmá generace)

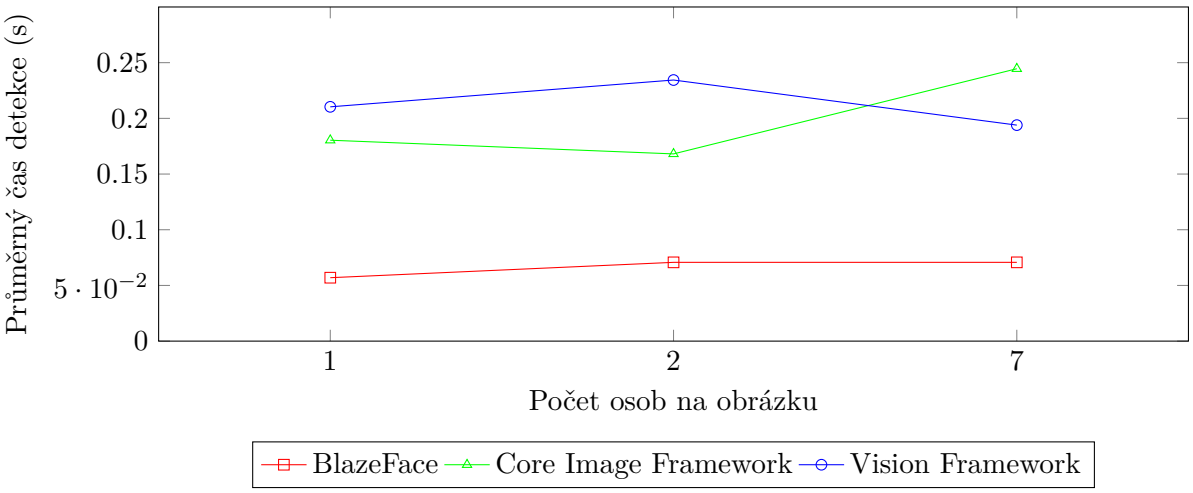
Počet detekovaných obličejů byl konzistentní při 10 sekvenčních detekcích pro každou kombinaci metody detekce a zařízení.

4.7.2 Doba detekce

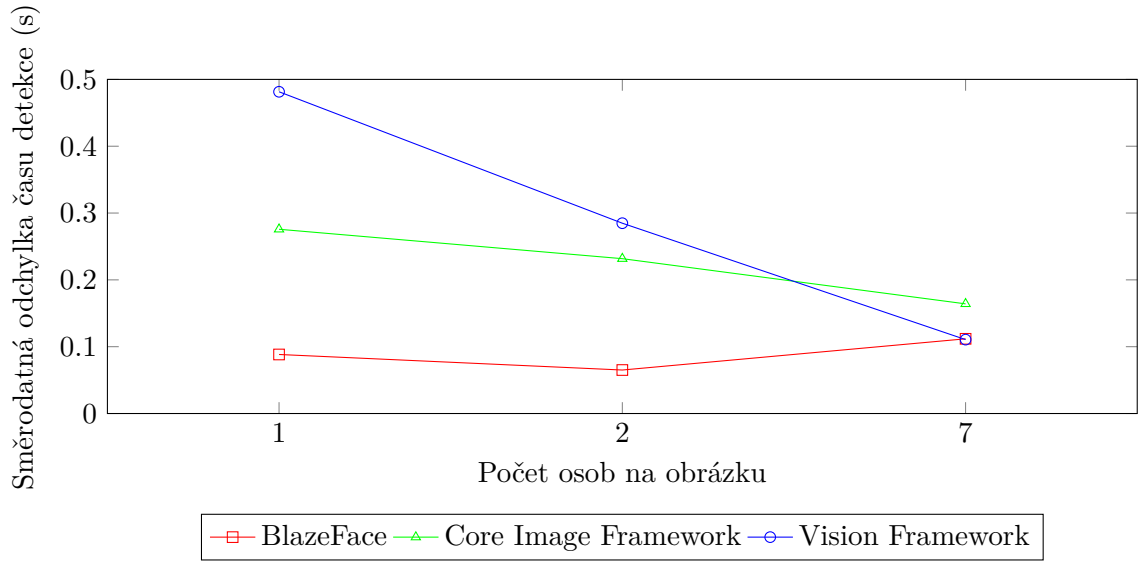
Rychlost odezvy mobilní aplikace představuje klíčový aspekt pro zajištění přirozeného a plynulého uživatelského prostředí. V kontextu vývoje aplikací je prioritou vývojářů optimalizace interních procesů s cílem dosáhnout co nejvyšší efektivity a minimalizovat dobu, po kterou je uživatelské rozhraní neaktuální. Z těchto důvodů je rychlost detekce obličeje kritická charakteristika procesu, kterou analyzuje následující sekce práce.

Testovací obrázek	Způsob detekce	Průměrný čas detekce	Sm. odchylka detekce
1 osoba	Blaze	0.05693797469	0.08835084971
1 osoba	Core Image Framework	0.1803853733	0.2757330666
1 osoba	Vision Framework	0.2103522003	0.4814866755
2 osoby	Blaze	0.07072555082	0.06509140228
2 osoby	Core Image Framework	0.1680895873	0.2317293679
2 osoby	Vision Framework	0.2344187949	0.2848017330
7 osob	Blaze	0.07073048609	0.1117928301
7 osob	Core Image Framework	0.2445551268	0.1641011237
7 osob	Vision Framework	0.1939320888	0.1106129904

Tabulka 4: Průměrný čas detekce modelů s odchylkou



Obrázek 16: Porovnání průměrného času detekce dle způsobu detekce



Obrázek 17: Porovnání směrodatné odchylky časů detekce dle způsobu detekce

Nejrychlejší proces detekce obličejů poskytl model BlazeFace ve všech třech variantách testovacích snímků. Model BlazeFace byl výrazně rychlejší než Core Image Framework a Vision Framework. Je však nutné zdůraznit, že u testovacího snímku se 7 osobami model nezaznamenal žádný obličej ve 100% případů detekcí, viz sekce 4.7.1.

Průměrná rychlost detekce obličejů na snímcích s jednou a dvěma osobami byla vyšší u Vision Framework; naopak, snímek se 7 osobami byl detekován nejpomaleji pomocí Core Image Framework. Rozdíl v rychlosti detekce mezi Vision Framework a Core Image Framework byl ovšem malý a na základě T-testu ho lze shledat statisticky nevýznamným.

4.7.3 Vliv modelu zařízení na dobu detekce

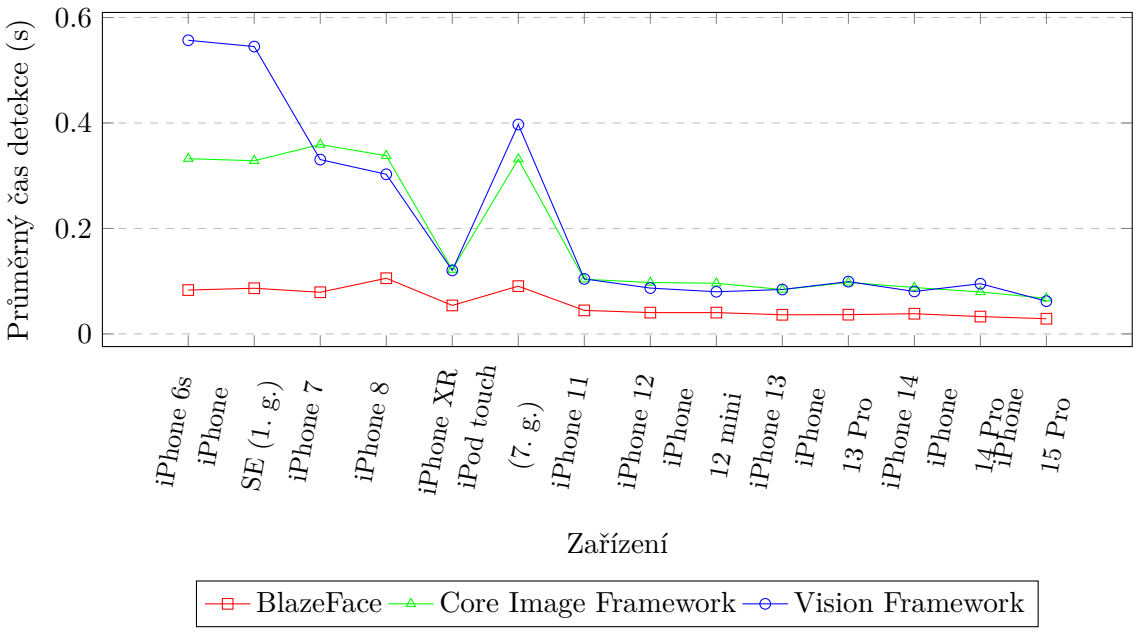
Tabulka 5 a grafy na obrázcích 18, 19 a 20 ukazují rozdíly v průměrném čase detekce a ve směrodatné odchylce pro všechna testovaná zařízení.

Z údajů v tabulce a grafech vyplývá, že model zařízení má vliv na rychlost procesu detekce obličeje, avšak velikost tohoto rozdílu závisí na zvolené metodě detekce. Obecně platí, že s vyšším výkonem zařízení (čím je model zařízení novější), rychleji probíhá detekce. Detekce obličeje pomocí Vision Framework na snímku s jednou osobou byla na nejnovějším zařízení iPhone 15 Pro přibližně 9x rychlejší než na nejpomalejším zařízení iPhone 6s. Rozdíl v rychlosti detekce pomocí modelu BlazeFace na těchto zařízeních a testovacím snímku byl však pouze trojnásobný, což je stále signifikantní rozdíl, ale oproti Vision Framework je tento rozdíl třikrát menší.

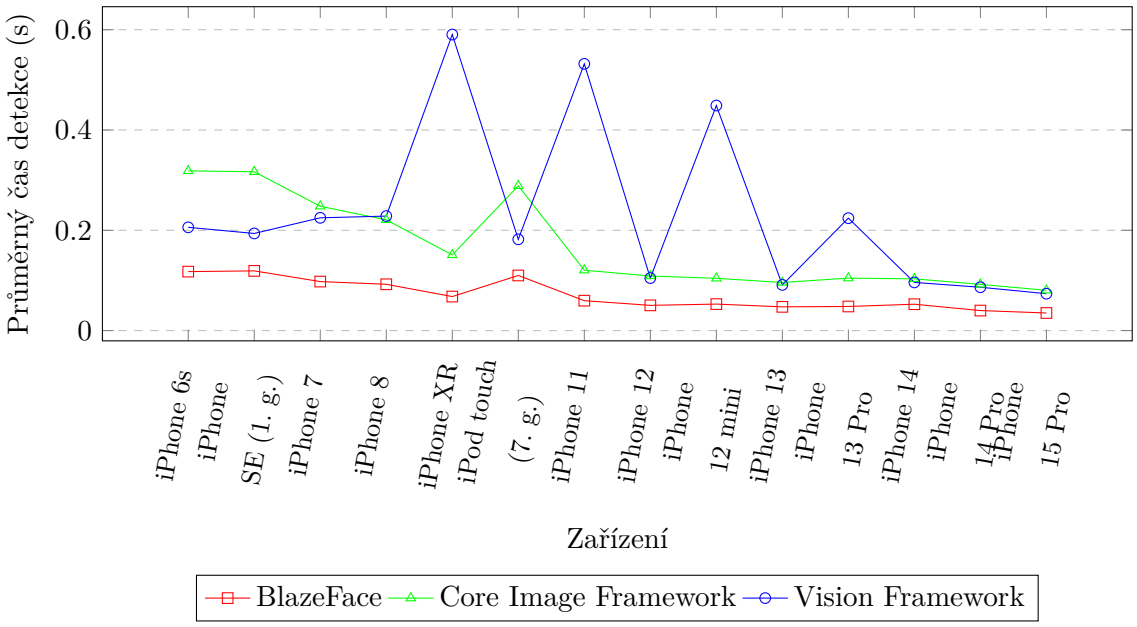
Graf na obrázku 19 prezentuje porovnání průměrné rychlosti detekce na jednotlivých zařízeních pro snímek s dvěma osobami. Tento graf upozorňuje na anomálii v naměřených datech – detekce pomocí Vision Framework byla výrazně pomalejší na zařízeních iPhone XR, iPhone 11 a iPhone 12 mini. Měření detekce na těchto zařízeních bylo opakováno a výsledky druhého měření již anomálie nevykazovaly. Jelikož je Vision Framework uzavřený systém, nelze bohužel určit přesnou příčinu zpomalení při původním měření. Z dat je zřejmé, že všech 10 sekvenčních detekcí bylo na těchto zařízeních při prvním měření výrazně pomalejších. Tento jev mohl být způsoben například vytížením zařízení procesy na pozadí, interním výkonostním problémem Vision Frameworku nebo chybou v měření. Kvůli uzavřenosti Vision Frameworku nelze přesnou příčinu definitivně určit.

Zařízení	Testovací obrázek	Vision Framework: Průměrný čas detekce	Vision Framework: Sm. odchylka	BlazeFace: Průměrný čas detekce	BlazeFace: Sm. odchylka	Core Image Framework: Průměrný čas detekce	Core Image Framework: Sm. odchylka
iPhone 6s	1 osoba	0.5568268061	1.044885384	0.08318969011	0.08051590911	0.3322737217	0.3062798723
iPhone 6s	2 osoby	0.2059051037	0.3365494765	0.1176104903	0.07313171122	0.318533802	0.2984988154
iPhone 6s	7 osob	0.3238307834	0.05990518899	0.1181850076	0.1609209537	0.4854139209	0.0688038048
iPhone SE (1. generace)	1 osoba	0.5448712945	1.021692211	0.08661819696	0.08844505226	0.3284344077	0.3058045958
iPhone SE (1. generace)	2 osoby	0.1936993241	0.3347496095	0.1190424681	0.07264611419	0.3167050242	0.2988360305
iPhone SE (1. generace)	7 osob	0.32559551	0.07068929695	0.1130021811	0.1400299059	0.4821960926	0.07143343437
iPhone 7	1 osoba	0.3305559039	0.3770452824	0.07900848389	0.1002896111	0.3591236949	0.5039672983
iPhone 7	2 osoby	0.224857688	0.3084062075	0.09774688482	0.07752478086	0.2479263067	0.246679237
iPhone 7	7 osob	0.3086610079	0.07525338073	0.09492598772	0.1357537277	0.3734032035	0.06190718438
iPhone 8	1 osoba	0.3027038097	0.4107276	0.1055545926	0.1926395196	0.3378622055	0.5471453426
iPhone 8	2 osoby	0.2283230662	0.3718024585	0.09231798649	0.07270371367	0.2208069921	0.2305671661
iPhone 8	7 osob	0.2585857868	0.09048159846	0.08540458679	0.11804708	0.327008605	0.1510600992
iPhone XR	1 osoba	0.1205217957	0.07119183464	0.05398905277	0.06564361271	0.1219403863	0.08082844282
iPhone XR	2 osoby	0.5903530836	0.05598679843	0.06782237291	0.0500773047	0.150880909	0.2104987318
iPhone XR	7 osob	0.1806663275	0.05767265666	0.06891599894	0.09831014488	0.1899116039	0.06678319431
iPod touch (7. generace)	1 osoba	0.3971416235	0.5649965237	0.09056961536	0.1206824726	0.3315376401	0.2700969526
iPod touch (7. generace)	2 osoby	0.1820068121	0.3636563076	0.1098954916	0.08926587769	0.2886810064	0.3065854153
iPod touch (7. generace)	7 osob	0.3336041093	0.07066471623	0.109409976	0.1609905629	0.4664717913	0.06906596765
iPhone 11	1 osoba	0.104310739	0.05311948692	0.04456578493	0.0548191598	0.1037033081	0.06319295089
iPhone 11	2 osoby	0.5319245934	0.03922084984	0.05963670015	0.0498325366	0.1204638839	0.1815095423
iPhone 11	7 osob	0.1523722172	0.03473038697	0.06146408319	0.09347103562	0.1673358321	0.04914686979
iPhone 12	1 osoba	0.086852777	0.06526373812	0.04044890404	0.05391518755	0.09749991894	0.08401635504
iPhone 12	2 osoby	0.1046928048	0.1805431426	0.05030602217	0.04404134954	0.1088175893	0.1712277782
iPhone 12	7 osob	0.1311071157	0.05978351931	0.0534968853	0.08367184595	0.1445377946	0.06466375019
iPhone 12 mini	1 osoba	0.07992187738	0.02541943495	0.04037220478	0.05074026342	0.09610599279	0.07855541085
iPhone 12 mini	2 osoby	0.4487815142	0.04983042654	0.05282050371	0.04290383712	0.1042278886	0.1612906198
iPhone 12 mini	7 osob	0.1351801038	0.05228894977	0.05308198929	0.08219037199	0.140872705	0.05146602579
iPhone 13	1 osoba	0.08415929079	0.07766740465	0.03632469177	0.05148409514	0.08405438662	0.06786007953
iPhone 13	2 osoby	0.09110326767	0.1605517764	0.04733290672	0.04256530456	0.09592969418	0.1591167882
iPhone 13	7 osob	0.1220529795	0.07469658686	0.04788178205	0.07738221544	0.1268657804	0.0607108715
iPhone 13 Pro	1 osoba	0.09924279451	0.125502534	0.03651168346	0.05068852173	0.09772679806	0.1141417429
iPhone 13 Pro	2 osoby	0.2242092967	0.1958524238	0.04806740284	0.04648921996	0.1046296954	0.1773551537
iPhone 13 Pro	7 osob	0.1215239048	0.07675308926	0.05009310246	0.08325386766	0.1524099112	0.1410086271
iPhone 14	1 osoba	0.08049241304	0.07300202156	0.03832052946	0.04846957436	0.08799468279	0.08103393343
iPhone 14	2 osoby	0.09603410959	0.17623508	0.05256420374	0.04477034125	0.103157115	0.1809524443
iPhone 14	7 osob	0.1168845057	0.05641537721	0.05135622025	0.07935647451	0.1335719466	0.0802545289
iPhone 14 Pro	1 osoba	0.09518760443	0.128049502	0.03285131454	0.04872155695	0.07962139845	0.07145166405
iPhone 14 Pro	2 osoby	0.08642169237	0.1553252741	0.03993268013	0.04104056762	0.09216210842	0.1510213838
iPhone 14 Pro	7 osob	0.1107708931	0.0630432275	0.04332630634	0.07438586917	0.1249075055	0.0780962045
iPhone 15 Pro	1 osoba	0.06214207411	0.0427630126	0.02880690098	0.04105800797	0.06751668453	0.0481838471
iPhone 15 Pro	2 osoby	0.07355077267	0.1211416495	0.03506159782	0.03536087988	0.08033220768	0.1223387723
iPhone 15 Pro	7 osob	0.09421399832	0.02878375812	0.03968269825	0.06481801665	0.1088650823	0.04139395111

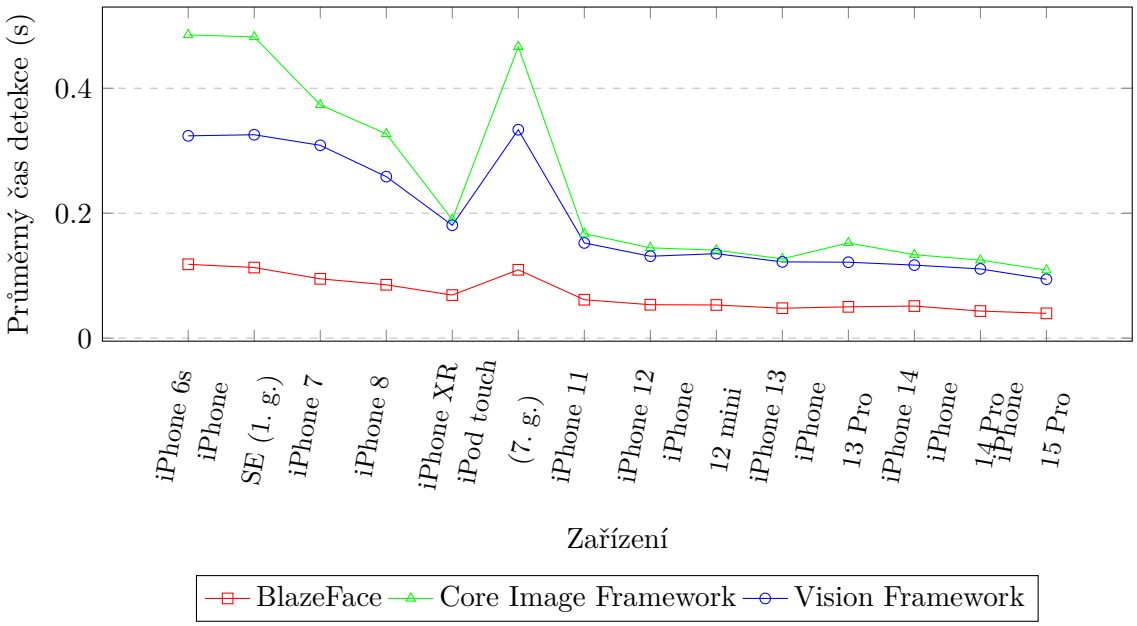
Tabulka 5: Porovnání směrodatné odchylky časů detekce pro různá zařízení



Obrázek 18: Porovnání průměrné rychlosti detekce na jednotlivých zařízeních (1 osoba)



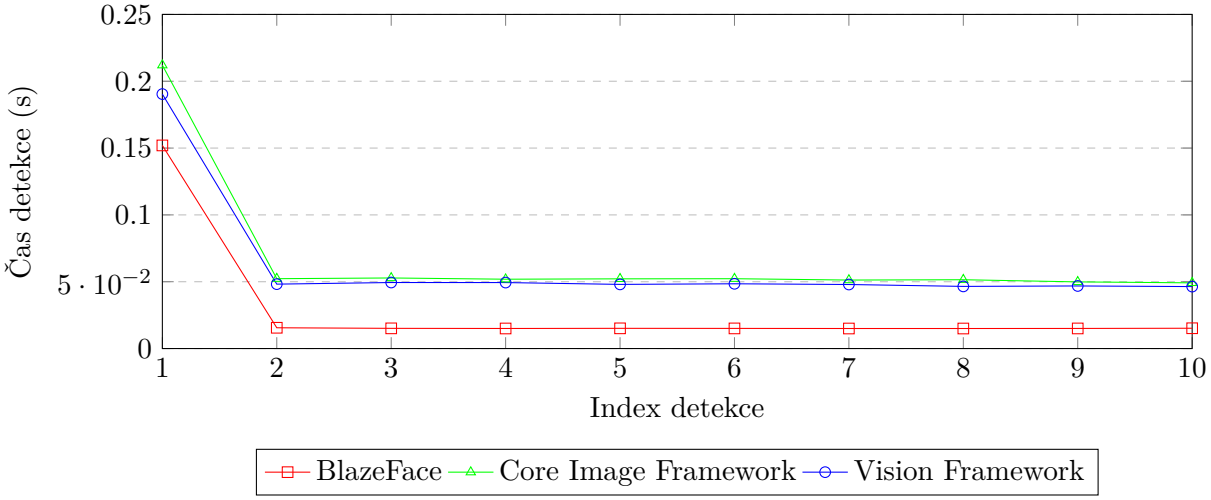
Obrázek 19: Porovnání průměrné rychlosti detekce na jednotlivých zařízeních (2 osoby)



Obrázek 20: Porovnání průměrné rychlosti detekce na jednotlivých zařízeních (7 osob)

4.7.4 Sekvenční detekce

Detekce byla vždy prováděna sekvenčně desetkrát pro každou testovanou kombinaci zařízení a obrázku. Důvodem pro sekvenční testování bylo, že první detekce může být teoreticky zpomalena kvůli přípravným činnostem algoritmu, jako například nahrávání zdrojů do paměti zařízení.



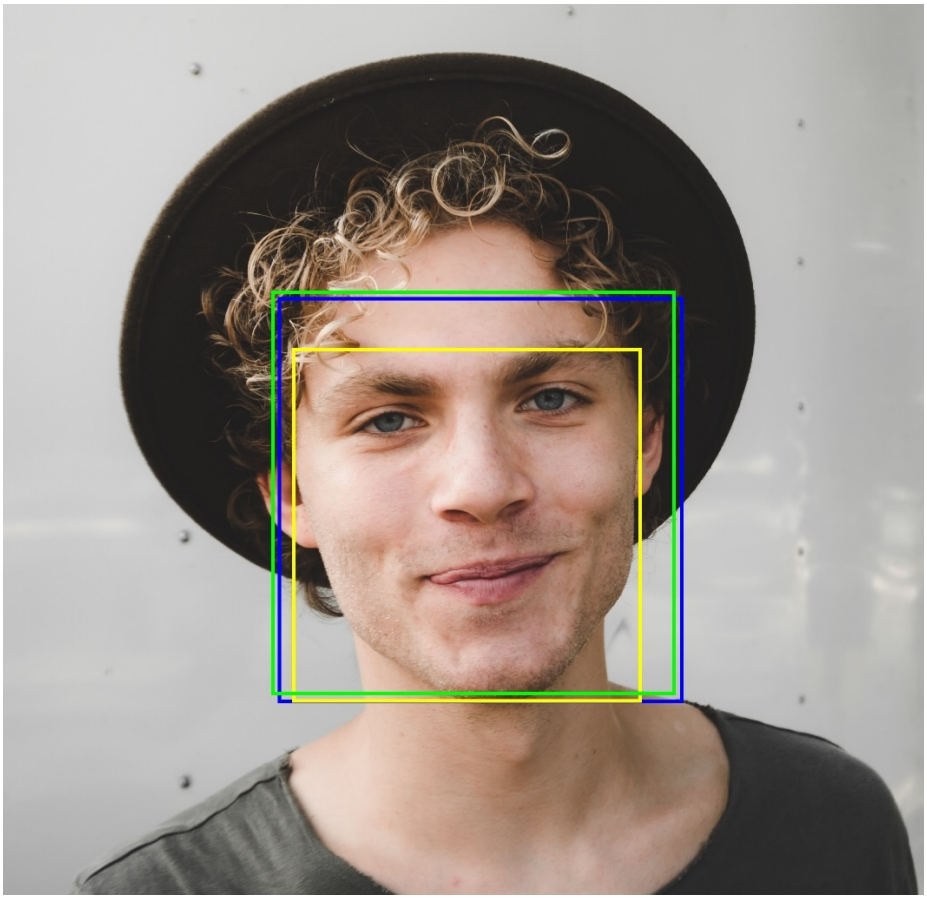
Obrázek 21: Sekvenční detekce na iPhone 15 Pro se snímkem jedné osoby

Model	**Prům. čas 1. detekce**	**Prům. čas 2. detekce**	**Prům. čas 10. detekce**
BlazeFace	0.3006	0.0405	0.0399
Core Image Framework	0.6669	0.1457	0.1445
Vision Framework	0.7686	0.1549	0.1518

Tabulka 6: Průměrný čas 1.,2. a 10. detekce

Graf na obrázku 21 a tabulka 6 potvrzují domněnku, která byla uvedena na úvodu sekce, že první detekce trvá déle než následné detekce. Rozdíl mezi první a následujícími detekcemi závisí na konkrétním zařízení a modelu detekce. Tabulka 6 ukazuje celkový průměr ze všech testovaných zařízení a snímků. Z dat této tabulky vyplývá, že nejvýraznější rozdíl je u modelu BlazeFace, kde je druhá detekce více než 7krát rychlejší než první. Frameworky Core Image a Vision vykazují rozdíl mezi první a druhou detekcí v rozmezí 4 až 5 násobku. Po druhé detekci se rychlost detekce již signifikantně nezlepšuje, což dokládá graf na obrázku 21 a porovnání mezi druhým a třetím sloupcem v tabulce 6.

4.7.5 Přesnost detekované oblasti



Obrázek 22: Srovnání detekovaných oblastí dle modelů

Výstupní data detekce obličeje jsou ve formátu obdélníkových oblastí definovaných čtyřmi body, které vymezují oblasti, ve kterých model detekoval obličeje. Obrázek 22 zobrazuje oblasti detekované všemi testovanými modely. Zelená oblast byla detekovaná BlazeFace modelem, modrá Vision Frameworkem a žlutá Core Image Frameworkem.

BlazeFace model a Vision Framework detekovaly větší oblasti zahrnující i čelo obličeje. Naopak Core Image Framework detekoval menší oblast, jejíž horní hranice byla vždy lehce nad obočím osoby. Toto chování bylo konzistentní na všech testovaných zařízeních a nebylo ovlivněno počtem osob na testovaném snímku.

4.8 Módy přesnosti Core Image Frameworku

Detekce obličejů pomocí Core Image Frameworku podporuje dva módy přesnosti: vysoký a nízký. Dokumentace Apple však neuvádí detailní informace o rozdílech mezi těmito módy, kromě zmínky, že režim s nízkou přesností by měl být rychlejší. [17]

Pro objasnění specifických rozdílů mezi módy byla provedena detekce v obou režimech na testovacích snímcích obsahujících 1 a 7 osob z předchozích kapitol. Měření proběhlo na zařízení iPhone 14 a následující analýza vychází z druhé hodnoty sekvenčního měření, aby byl eliminován vliv zpomalování při první detekci, viz sekce 4.7.4.

Výsledky z tabulky 7 a srovnání detekovaných oblastí na obrázku 23 dokládají, že mód vysoké přesnosti je přibližně o 20% pomalejší než mód nízké přesnosti, ale zároveň poskytuje signifikantně lepší výsledky. Detekované oblasti v módu vysoké přesnosti jsou menší a přesněji odpovídají skutečné oblasti obličeje. U snímku se 7 osobami mód nízké přesnosti detekoval pouze 6 osob.

Získané poznatky vedou k doporučení využívat mód vysoké přesnosti. Všechny experimenty popsané v předchozích sekcích, kde byla aplikována detekce obličejů skrze Core Image Framework, využívaly mód vysoké přesnosti.

Mód přesnosti	1 osoba	7 osob
High	0.0693	0.1078
Low	0.0527	0.0931

Tabulka 7: Porovnání módů přesnosti Core Image frameworku

Obrázek 23: Porovnání módů přesnosti Core Image frameworku

5 Create ML a tvorba vlastního modelu klasifikace obrázků

Create ML je aplikace vyvinutá společností Apple, která umožňuje vývojářům trénovat vlastní modely strojového učení pomocí jednoduchého a intuitivního uživatelského rozhraní. Tradičně vytváření vlastního modelu strojového učení zahrnuje složité procesy jako programování v Pythonu a používání nástrojů TensorFlow nebo PyTorch. Create ML tuto komplexitu zjednodušuje, umožňuje prakticky každému vytvořit vlastní modely. [18]

Aplikace Create ML je integrována do vývojového prostředí Xcode. Pro přístup ke CreateML je nutné v menu Xcode vybrat *Otevřít nástroje vývojáře* a poté zvolit *Create ML*.

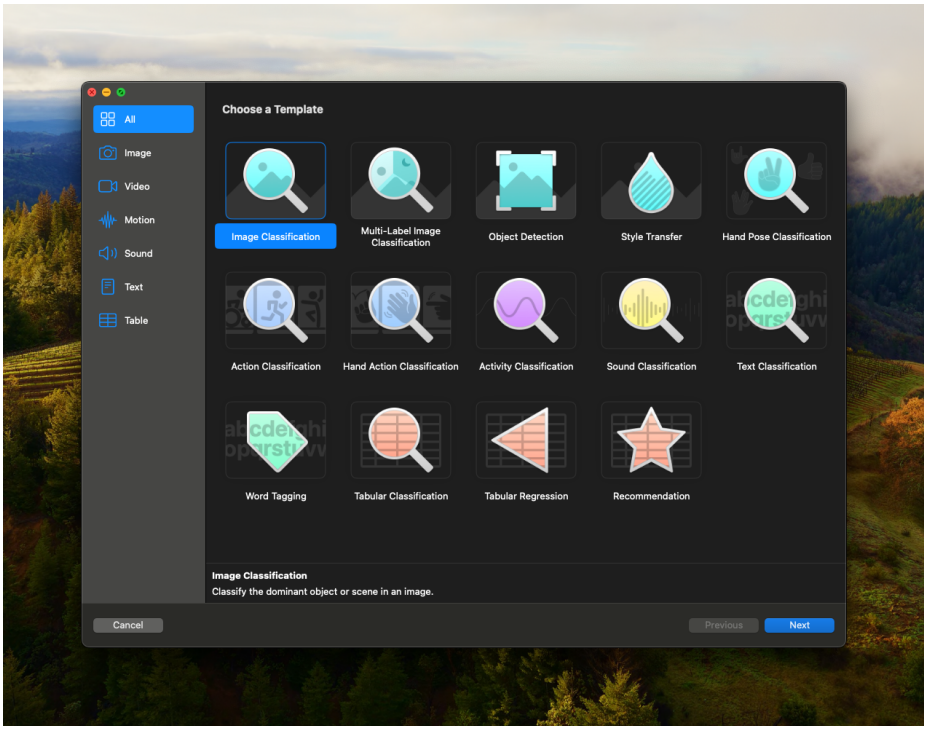
Po spuštění Create ML se zobrazí výběr typu projektu, jako je klasifikace obrázků nebo detekce objektů. Pro příklad zkoumaný touto prací vybereme *Klasifikaci obrázků*.

Následuje stránka s detaily modelu, kde je možné nastavit parametry tréninku. Nejdůležitějším parametrem je *Tréninková data*. Nastavení tréninkových dat vyžaduje pouze jednoduchou strukturu složek s datasetem:

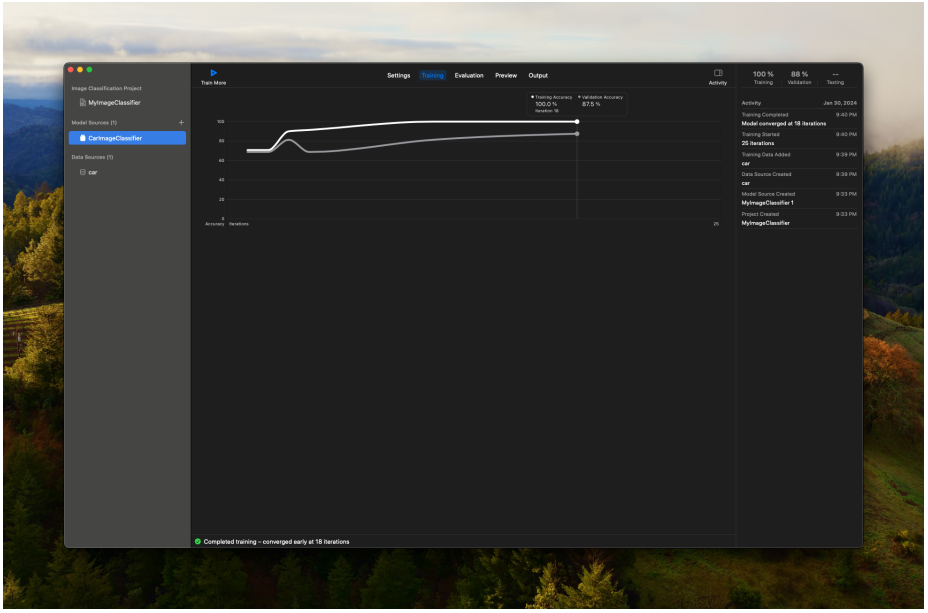
- Složka Třída A:
  - Obrázky JPG nebo PNG
- Složka Třída B:
  - Obrázky JPG nebo PNG

Po vybrání datové sady a dalších tréninkových parametrů stačí kliknout na tlačítko *Trénovat*.

Pro demonstraci možností Create ML byl vytvořen základní klasifikátor obrázků schopný rozlišovat mezi jablky a banány, využívající datovou sadu složenou pouze ze 78 fotografií.



Obrázek 24: Create ML podporované typy modelů



Obrázek 25: Create ML proces trénování modelu

Pro integraci souboru `.mlmodel` do vašeho projektu Xcode jej jednoduše zahrňte do souborů projektu a proveďte build projektu. Xcode automaticky generuje třídy související s modelem, které můžete použít pro klasifikaci obrázků. Generované třídy nejsou viditelné ve struktuře projektu podobně jako Core Data.

```
let classifier = try FruitImageClassifier(configuration: .init())
let input = try FruitImageClassifierInput(imageWith: image.cgImage!)
let output = try classifier.prediction(input: input)

guard let probability = output.classProbability[output.label]
else { return }

let formattedProbability = String(format: "%.2f", probability * 100)
resultLabel.text = "\(output.label) (\(formattedProbability)%)"
```

Obrázek 26: Příklad použití modelu FruitImageClassifier

Model je efektivní i při tréninku na malém množství obrázků, což demonstruje schopnost Create ML rychle vytvářet užitečné modely strojového učení. Velikost výsledného modelu je pouze 17kB. Model měl 100% úspěšnost v rozpoznání jablek a banánů. Dokonce klasifikace byla správná i v případě obrázku žlutého jablka, i přesto že model byl trénován pouze na jablkách červených a zelených. Důležité je ale uvědomit si, že výstupem modelu je pravděpodobnost každé známé třídy a součet všech pravděpodobností tříd vždy rovná 100%. To znamená, že pokud je do modelu vložen obrázek nesouvisející s tréninkovou sadou, výstupem budou neplatné hodnoty, nikoli 0% pro jablka a 0% pro banány, jak by mohl někdo očekávat.

6 Vision Framework

Vision Framework, doporučený způsob implementace strojového vidění na iOS platformě dle společnosti Apple, je součástí iOS SDK. Jeho pravidelné aktualizace jsou prezentovány na každoroční vývojářské konferenci WWDC (World Wide Developer Conference), takže se jedná o aktivně vyvíjenou a udržovanou součást iOS SDK. [4]

V této kapitole se práce věnuje obecnému popisu Vision Frameworku a detailně rozebírá jeho klíčové části, s výjimkou dříve zmíněné detekce obličejů ve statických obrazech.

6.1 Historie

Vision Framework byl představen společností Apple v roce 2017 jako součást aktualizace operačního systému iOS 11 a příslušného iOS SDK. Představení tohoto frameworku představovalo zásadní krok pro strojové vidění na platformě iOS, neboť Vision Framework nabízel rozsáhlejší sadu funkcí ve srovnání s předchozími řešeními. Byl založen na moderních principech neuronových sítí a poskytl pevný základ pro budoucí rozvoj strojového vidění na iOS platformě.

Apple neustále vylepšuje Vision Framework a pravidelně přidává nové funkce. V iOS 13 SDK byla například přidána podpora pro detekci textu v obraze, což předtím vyžadovalo použití vlastních řešení nebo externích knihoven. Aktualizace iOS 14 rozšířila Vision Framework o schopnost analýzy optického toku mezi snímky, což významně usnadnilo operace strojového vidění při práci s video sekvencemi.

Jednou z nesporných výhod Vision Frameworku je, že výpočty probíhají lokálně na zařízení. To znamená, že pro analýzu dat není potřebné internetové připojení a data neopouštějí uživatelské zařízení, což je významné z hlediska ochrany soukromí uživatele. [4]

6.2 Základní principy a elementy

Vision Framework poskytuje mnoho funkcí a většina z nich je založena na třech základních komponentách: Request, RequestHandler a Observation.

Request je definice specifické úlohy, kterou chceme provést. Každý Request specifikuje typ operace (například detekci obličeje) a zahrnuje zpětné volání pro zpracování výsledků. Důležité je, že Request sám o sobě neobsahuje data k analýze. Instance Requestu může být použita opakovaně pro stejný typ operací.

Existují dva typy Requestů: *stateless* (bezstavové) a *stateful* (stavové). Bezstavový Request, jako je například detekce textu, neuchovává informace mezi jednotlivými spuštěními a jedná se o nezávislé operace. Stavové Requesty, na druhé straně, si uchovávají stav mezi spuštěními, což je výhodné při analýze sekvencí videí, jako je například GeneratePersonSegmentationRequest.

RequestHandler je objekt zodpovědný za zpracování Requestu na konkrétních datech. Jako vstupní parametr konstruktoru RequestHandleru se používají data určená k analýze. Instance RequestHandleru může být použita opakovaně pro různé Requesty. Vision Framework dnes obsahuje 2 RequestHandlers - VNImageRequestHandler a VNSequenceRequestHandler.

Observation je datová struktura, kterou Vision Framework používá k předávání výsledků zpět aplikaci. Každá operace má typicky svůj specifický typ Observation, který obsahuje relevantní výstupní hodnoty pro danou operaci. Například VNGenerateObjectnessBasedSaliencyImageRequest používá jako Observation instanci třídy VNSaliencyImageObservation, která obsahuje relevantní informace o výsledku operace. [4]

```
let request = VNGenerateObjectnessBasedSaliencyImageRequest { request, error
    in
    // Zpracovani vysledku
}

let requestHandler = VNImageRequestHandler(cgImage: cgImage, options: [:])
try requestHandler.perform([request])
```

Obrázek 27: Ukázka základních elementů Vision Framework

6.3 Detekce význačnosti

Obrázek 1 v kapitole 3.2 představuje ukázku chytrého ořezu obrazu na základě detekce obličeje. Vision Framework podporuje detekci význačnosti, která je sofistikovanější řešením pro ukázanou situaci než detekce obličeje. Hlavní výhodou detekce význačnosti je, že se nevztahuje pouze na obličeje osob. Z toho vyplývá, že detekce bude fungovat i na obrázcích, kde se nenacházejí osoby (panoráma, zvířata, atd.).

Výstupní data detekce význačnosti jsou až tři oblasti obdélníkového tvaru, každá s vlastní hodnotou konfidence.

Vision Framework podporuje dva módy detekce význačnosti: pozornostní a objektový. Pozornostní mód byl natrénován tak, aby detekoval oblasti, na které se lidské oko zaměří jako první po zobrazení obrazu. Naopak objektový mód pracuje čistě s význačností objektů obrazu bez zohlednění priority pro lidské oko. [19]

Request třídy detekce význačnosti:

- VNGenerateAttentionBasedSaliencyImageRequest: pozornostní mód
- VNGenerateObjectnessBasedSaliencyImageRequest: objektový mód

```
let request = VNGenerateObjectnessBasedSaliencyImageRequest { request, error
    in
    // Zpracovani vysledku
}

let requestHandler = VNImageRequestHandler(cgImage: cgImage, options: [:])
try requestHandler.perform([request])
```

Obrázek 28: Ukázka detekce význačnosti pomocí Vision Framework

Výsledky detekce jsou získávány v datové struktuře VNSaliencyImageObservation, která obsahuje vlastnost salientObjects. Tato vlastnost definuje oblasti, jež Vision Framework identifikuje jako významné.

Výstup detekce může obsahovat až tři oblasti. Nicméně, mobilní aplikace obvykle vyžaduje pouze jednu oblast, například pro účely ořezu obrazu. Vývojář aplikace má k dispozici dvě možnosti:

1. Vybrat oblast s nejvyšší hodnotou konfidence.
2. Sloučit všechny detekované oblasti (pomocí operace sjednocení) do jedné větší oblasti.

Volba mezi těmito přístupy závisí na specifickém použití v mobilní aplikaci. Z testování při tvorbě této práce jsem došel k závěru, že ve většině případů vypadá vizuálně lépe sloučení oblastí.

```
guard
    let saliencyObservation = results.first(ofType:
        VNSaliencyImageObservation.self),
    let salientObjects = saliencyObservation.salientObjects,
    !salientObjects.isEmpty
else {
    return // Detekce byla neuspesna
}

let salientBox: CGRect
switch mode {
case .union:
    var unionOfSalientRegions = salientObjects.first!.boundingBox
    for salientObject in salientObjects.dropFirst() {
        unionOfSalientRegions =
            unionOfSalientRegions.union(salientObject.boundingBox)
    }
    salientBox = VNImageRectForNormalizedRect(
        unionOfSalientRegions,
        Int(originalImage.size.width),
        Int(originalImage.size.height)
    )
case .confidency:
    let salientObject = salientObjects.sorted(by: \.confidence, isAscending:
        false).first!
    salientBox = VNImageRectForNormalizedRect(
        salientObject.boundingBox,
        Int(originalImage.size.width),
        Int(originalImage.size.height)
    )
}

let croppedImage = originalImage.cgImage!.cropping(to: salientBox)!
```

Obrázek 29: Ukázka zpracování výsledku detekce význačnosti pomocí Vision Framework

Detekce význačnosti je užitečná funkce Vision Frameworku, která by mohla pomoci vývojářům v reálném světě při tvorbě mobilních aplikací. Proto jsem se rozhodl tuto detekci v rámci této práce vyzkoušet a otestovat. Testovací mobilní aplikaci ze sekce 4 jsem rozšířil o obrazovku menu poskytující uživatelům možnost výběru požadované detekce. Detekci význačnosti jsem implementoval na samostatné obrazovce, jejíž součástí je výběr obrázku z galerie telefonu. Následně aplikace obrázek ořízne podle výsledků detekce význačnosti. Uživatel má možnost vybrat, zda proces využije sjednocení detekovaných oblastí nebo se zaměří pouze na oblast s nejvyšší konfidencí. Aplikace na obrazovce zobrazuje jak původní obrázek, tak jeho ořezanou verzi.

Obrázek 30: Snímek obrazovky experimentu s detekcí význačnosti

6.4 Detekce hlavního předmětu

Apple v roce 2023 představil verzi mobilního operačního systému iOS 17, která přinesla řadu nových funkcí. Mezi hlavní novinky patří schopnost detekce hlavního předmětu na fotografii a následné vytvoření masky ořezávající hlavní předmět. [20] Uživatelé mohou tuto funkci aktivovat dlouhým stiskem prstu na hlavním předmětu fotografie, čímž systém následně umožní přesunutí oříznutého předmětu do jiné aplikace, odeslání zprávou nebo třeba vytvoření štítku pro iMessage.

S uvedením iOS 17 Apple představil také nové SDK pro iOS 17, jehož součástí je rozšíření Vision Frameworku o funkci detekce hlavního předmětu na fotografii. Považuji za důležité a osobně mě těší, že Apple poskytuje vývojářům přístup k nejmodernějším technologiím systému prostřednictvím API, což umožňuje aplikacím třetích stran dosahovat funkcionalit srovnatelných s vestavěnými aplikacemi v zařízení.

Hlavní rozdíl mezi detekcí hlavního předmětu a detekcí význačnosti spočívá ve formě výstupních dat. Zatímco detekce význačnosti identifikuje až tři obdélníkové oblasti, detekce hlavního předmětu poskytuje aplikaci přesně oříznutý obraz hlavního předmětu, buď v rastrové podobě nebo jako masku.

Vision Framework spolu s VisionKit Framework, který představuje oficiální nadstavbu nad Vision Frameworkem, zahrnují dvě hlavní třídy pro práci s detekcí hlavního předmětu:

- ImageAnalyzer
- ImageAnalysisInteraction

Instance třídy ImageAnalyzer, jak již název napovídá, slouží k analýze obrazových dat. Výstupy analýzy jsou aplikaci poskytnuty prostřednictvím instance třídy ImageAnalysis, která však neobsahuje přímý přístup k výsledkům detekce. Pro získání výsledků detekce je nutné využít třídu ImageAnalysisInteraction.

Třída ImageAnalysisInteraction umožňuje integraci dat z ImageAnalysis s uživatelským rozhraním mobilní aplikace. Tato interakce může být navázána s libovolným potomkem třídy UIView, což zahrnuje většinu komponent uživatelského rozhraní v rámci UIKit frameworku.

```
final class SubjectDetectionViewController: UIViewController,
    ImagePickingController, StoryboardedController {
    @IBOutlet weak var imageView: UIImageView!
    private let analyzer = ImageAnalyzer()
    private let interaction = ImageAnalysisInteraction()
    ...

    private func setupInteractions() {
        interaction.preferredInteractionTypes = .automatic
        imageView.isUserInteractionEnabled = true
        imageView.addInteraction(interaction)
    }

    @IBAction func analyzeTapped(_ sender: Any) {
        guard let image = ....
        else { return }

        Task {
            do {
                let configuration =
                    ImageAnalyzer.Configuration([.machineReadableCode, .text,
                    .visualLookUp])
                let analysis = try await analyzer.analyze(image, configuration:
                    configuration)
                interaction.analysis = analysis
                interaction.setContentsRectNeedsUpdate()
            } catch {
                handleError(error)
            }
        }
    }
}
```

Obrázek 31: Ukázka konfigurace ImageAnalyzer a ImageAnalysisInteraction

Zdrojový kód na obrázku 31 představuje ukázku využití ImageAnalyzer a ImageAnalysisInteraction tříd pro přidání gesta extrakce hlavního předmětu na imageView komponentu. Rozhraní VisionKit Frameworku umožňuje aplikacím také získat obraz (UIImage) s maskou na hlavní předměty, viz ukázka na obrázku 32.

```
let subjects = await interaction.subjects
let subjectImages = try await withThrowingTaskGroup(
    of: UIImage.self,
    returning: [UIImage].self
) { group in
    for subject in subjects {
        group.addTask {
            return try await subject.image
        }
    }

    var result: [UIImage] = []
    for try await image in group {
        result.append(image)
    }
    return result
}
```

Obrázek 32: Ukázka UIImage s maskou na hlavní předměty obrazu

6.5 Detekce obličeje ve videu v reálném čase

Vision Framework poskytuje řadu pokročilých funkcí včetně možnosti real-time sledování obličeje ve videu. Framework umožňuje s vysokou přesností lokalizovat specifické části obličeje jako jsou ústa, nos a oči. Jedním z využití této funkce je například umísťování grafických efektů a elementů na obličej, což uživatelé mohou znát z populární aplikace Snapchat. [21]

Přestože Vision Framework výrazně zjednodušuje zpracování této náročné úlohy, tak je implementace výrazně komplexnější než v úlohách předchozích. Postup je následující:

1. Získání souhlasu uživatele s přístupem ke kameře.

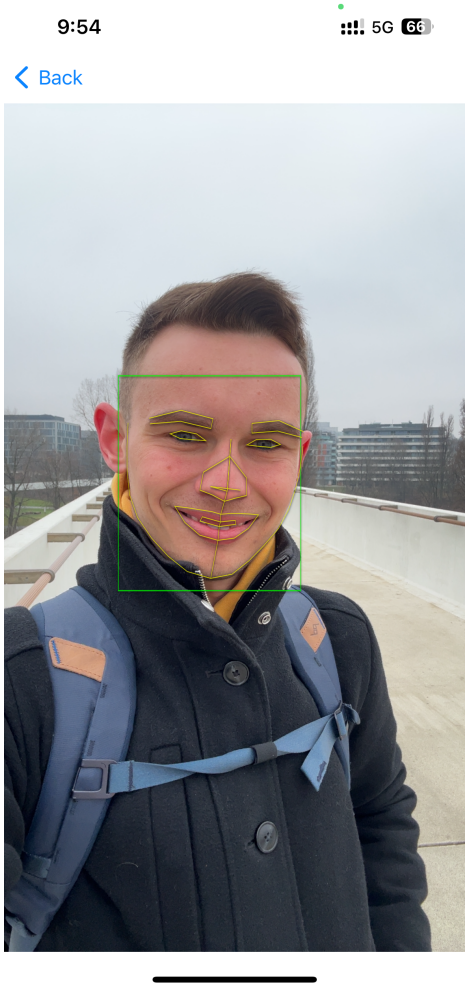
2. Zobrazení živého přenosu z kamery na obrazovku.

3. Pořízení jednoho statického snímku z kamery a následná detekce obličejů s využitím VNDetectFaceRectanglesRequest a VNImageRequestHandler, jak je popsáno v sekci 4.5.2.

4. Sledování rozpoznaných obličejů pomocí VNTrackObjectRequest a VNSequenceRequestHandler, kombinovaně s daty SampleBuffer z kamery.

5. Pravidelné opakování předchozích dvou kroků pro adaptaci na dynamiku scény, jako je například změna počtu osob ve scéně.

Apple poskytuje ukázkový projekt „Tracking the User's Face in Real Time“, který demonstruje výše uvedený proces. [22] Sekce implementující tento postup je také součástí testovací mobilní aplikace, která je součástí této práce.



Obrázek 33: Detekce obličeje ve videu v reálném čase

7 Závěry a doporučení

Diplomová práce se zaměřuje na podrobné zkoumání a analýzu strojového vidění v rámci vývoje nativních aplikací pro iOS platformu. Hlavním cílem práce je poskytnout sadu doporučení, která budou sloužit jako průvodce strojovým viděním pro vývojáře iOS aplikací.

Úvodní kapitoly práce poskytují čtenářům důležité informace potřebné pro porozumění sekcím následujícím a seznámí čtenáře s klíčovými aspekty strojového vidění.

Sekce zaměřená na detekci obličejů porovnává tři odlišná řešení: Vision Framework, Core Image Framework a model BlazeFace od společnosti Google. Z experimentů vyplynulo, že všechny zkoumané metody jsou schopny úspěšně detekovat obličeje ve statickém obrazu, typicky do několika set milisekund. Mezi zkoumanými modely byly zaznamenány rozdíly, které mohou být rozhodující při výběru nejvhodnějšího řešení pro konkrétní aplikaci.

Model BlazeFace byl obecně nejrychlejší, avšak spolehlivě fungoval pouze v případě detekce větších obličejů. To znamená, že model je vhodný pro detekci obličejů na obrázcích typu selfie pořízených přední kamerou telefonu, ale není vhodný pro detekci obličejů na fotografiích s velkým počtem osob (například skupinové fotografie z události). Další nevýhodou modelu BlazeFace je, že model není součástí iOS SDK, což znamená, že se jedná o externí knihovnu, přičemž její začlenění do projektu přináší rizika spojená s používáním externího softwaru (závislost na třetí straně).

Výsledky detekce pomocí Core Image Frameworku a Vision Frameworku byly velmi podobné. Při rozhodování mezi těmito frameworky bych doporučil dát přednost Vision Frameworku, jelikož se jedná o novější framework, který Apple nadále aktivně udržuje a který nabízí širší spektrum funkcionalit ve srovnání s Core Image Frameworkem.

Vliv hardwarového vybavení zařízení (tj. výkonu zařízení) má signifikantní dopad na rychlost detekce. V nejextrémnějším případě byl zaznamenán devítinásobný rozdíl v rychlosti detekce mezi nejpomalejším a nejrychlejším zařízením. Ve většině případů však rozdíl není tak extrémní, a obecně platí, že na pomalejších zařízeních trvají všechny operace déle, takže rozdíly v rychlosti detekce nejsou kritické.

Všechny tři testované metody detekce obličejů vykazovaly stejný jev: první detekce byla výrazně pomalejší než detekce následné. Tento jev je pravděpodobně způsoben potřebou načíst nezbytné zdroje do paměti zařízení a inicializovat proces detekce. Na základě analýzy

výsledků lze doporučit provedení první detekce již během spouštění aplikace na pozadí, tak aby byla detekční pipeline připravená a rychlá pro reálnou potřebu detekce za běhu aplikace.

Potenciální nedostatek práce je omezený vzorek testovaných zařízení. Je možné, že výsledky experimentů obsahují anomálie, jež ovlivňují zjištěné informace. Nicméně to nic nemění na faktu, že práce objektivně analyzuje získané měření v daném kontextu.

Na základě zjištěných informací je možné prohlásit, že pro aplikace bez specifických požadavků je implementace strojového vidění prostřednictvím Vision Frameworku nejlepší volbou. Balíček poskytuje jednoduché a funkční řešení pro mnoho úloh strojového vidění a zároveň se jedná o oficiální, aktivně udržovanou součást iOS SDK, což eliminuje závislost na externích knihovnách. Poslední kapitoly práce se proto podrobně věnují Vision Frameworku, jeho principům a vybraným funkcím.

Práce potvrzuje, že dnes dostupné balíčky umožňují efektivní řešení úloh strojového vidění s dobou zpracování v řádu stovek milisekund, tudíž přijatelnou pro produkční nasazení v mobilních aplikacích. Výhodou je, že vývojáři nepotřebují pokročilé znalosti umělé inteligence a neuronových sítí, což umožňuje snadnější integraci strojového vidění do iOS aplikací.

Literatura

[1] Office, G. P.: Smartphone owners are now the global majority, New GSMA report reveals.
URL https://www.gsma.com/newsroom/press-release/smartphone-owners-are-now-the-global-majority-new-gsma-report-reveals/

[2] OpenAI: Introducing ChatGPT.
URL https://openai.com/blog/chatgpt

[3] Apple, C. V. M. L. T.: Deploying Transformers on the Apple Neural Engine.
URL
https://machinelearning.apple.com/research/neural-engine-transformers

[4] Apple: Vision Framework Documentation.
URL https://developer.apple.com/documentation/vision/

[5] Marques, O.: *Image Processing and Computer Vision in iOS*. SpringerBriefs in Computer Science, Springer International Publishing, ISBN 978-3-030-54030-2 978-3-030-54032-6, doi:10.1007/978-3-030-54032-6.
URL http://link.springer.com/10.1007/978-3-030-54032-6

[6] Savin, A.: Computer Vision History: Milestones and Breakthroughs.
URL https://scilifestyle.com/computer-vision-history/

[7] Voulodimos, A.; Doulamis, N.; Doulamis, A.; aj.: Deep Learning for Computer Vision: A Brief Review. ročník 2018: s. 1–13, ISSN 1687-5265, 1687-5273, doi:10.1155/2018/7068349.
URL https://www.hindawi.com/journals/cin/2018/7068349/

[8] Apple: Core Image documentation.
URL https://developer.apple.com/documentation/coreimage

[9] Apple: iPhone 15 Pro Technická specifikace.
URL https://www.apple.com/iphone-15-pro/specs/

[10] Apple: Core ML Models.
URL https://developer.apple.com/machine-learning/models/

[11] David, R.; Duke, J.; Jain, A.; aj.: TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems. In *Proceedings of Machine Learning and Systems*, ročník 3, editace A. Smola; A. Dimakis; I. Stoica, s. 800–811.
URL https://proceedings.mlsys.org/paper_files/paper/2021/file/6c44dc73014d66ba49b28d483a8f8b0d-Paper.pdf

[12] Huang, Z.-Y.; Chiang, C.-C.; Chen, J.-H.; aj.: A study on computer vision for facial emotion recognition. ročník 13, è. 1: str. 8425, ISSN 2045-2322, doi:10.1038/s41598-023-35446-4.
URL https://www.nature.com/articles/s41598-023-35446-4

[13] Apple: Apple Developer Documentation.
URL https://developer.apple.com/documentation/

[14] Thakkar, M.: *Introduction to Core ML Framework*. Apress, ISBN 978-1-4842-4296-4 978-1-4842-4297-1, s. 15–49, doi:10.1007/978-1-4842-4297-1_2.
URL http://link.springer.com/10.1007/978-1-4842-4297-1_2

[15] Bazarevsky, V.; Kartynnik, Y.; Vakunov, A.; aj.: BlazeFace: Sub-millisecond Neural Face Detection on Mobile GPUs. doi:10.48550/ARXIV.1907.05047, publisher: arXiv Version Number: 2.
URL https://arxiv.org/abs/1907.05047

[16] Google: MediaPipe Solutions guide.
URL https://developers.google.com/mediapipe/solutions/guide

[17] Apple: Detector Accuracy Options.
URL https://developer.apple.com/documentation/coreimage/cidetector/detector_accuracy_options

[18] Apple: Create ML.
URL https://developer.apple.com/machine-learning/create-ml/

[19] Apple: Cropping Images Using Saliency.
URL https://developer.apple.com/documentation/vision/cropping_images_using_saliency

[20] Apple: What's new in iOS 17.
URL https:
//support.apple.com/guide/iphone/whats-new-in-ios-17-iphfed2c4091/ios

[21] Apple: Snapchat | AppStore.
URL https://apps.apple.com/us/app/snapchat/id447188370

[22] Apple: Tracking the User's Face in Real Time.
URL https://developer.apple.com/documentation/vision/
tracking_the_user_s_face_in_real_time

Seznam zkratek

iOS SDK iOS software development kit (softwarová vývojová sada)

AI artificial intelligence (umělá inteligence)

API application programming interface (aplikační programové rozhraní)

QA quality assurance (zajištění kvality)

UNIVERZITA HRADEC KRÁLOVÉ
Fakulta informatiky a managementu
Akademický rok: 2023/2024

Studijní program: Aplikovaná informatika
Forma studia: Kombinovaná
Obor/kombinace: Aplikovaná informatika (ai2-k)

Podklad pro zadání DIPLOMOVÉ práce studenta

Jméno a příjmení: **Bc. Tomáš Pařízek**
Osobní číslo: **I2100438**
Adresa: Hlavní 38/53, Hradec Králové – Nový Hradec Králové, 50008 Hradec Králové 8, Česká republika

Téma práce: Strojové vidění na iOS platformě
Téma práce anglicky: Computer Vision on iOS Platform
Jazyk práce: Čeština

Vedoucí práce: Ing. Karel Mls, Ph.D.
Katedra informačních technologií

Zásady pro vypracování:

Cíl: Provést analýzu, porovnání a doporučení možností implementace strojového vidění na iOS platformě.
Osnova:
- Úvod
- Cíl a metodika práce
- Analýza dostupných řešení strojového vidění
- Návrh a Implementace
- Závěr, zhodnocení

Seznam doporučené literatury:

- MARQUES, Oge. Image Processing and Computer Vision in iOS. SpringerBriefs in Computer Science. Cham: Springer International Publishing, 2020. ISBN 978-3--030-54030-2.
- HU, Han; HUANG, Yujin; CHEN, Qiuyuan; ZHUO, Terry Yue a CHEN, Chunyang. A First Look at On-device Models in iOS Apps. ACM Transactions on Software Engineering and Methodology. 2023, roč. 33, č. 1, s. 1-30. ISSN 1049-331X.
- Karthikeyan, N. G. Machine Learning Projects for Mobile Applications: Build Android and IOS Applications Using TensorFlow Lite and Core ML. Packt Publishing Ltd, 2018.

Podpis studenta: Datum:

Podpis vedoucího práce: Datum:

© IS/STAG, Portál – Podklad kvalifikační práce , parizto1, 28. ledna 2024 16:15