Mobilní aplikace pro načítání a zpracování čísel

Bc. Oleksandr Dolomanov

Diplomová práce
2023

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2022/2023

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Oleksandr Dolomanov**
Osobní číslo:	**A21158**
Studijní program:	**N0613A140022 Informační technologie**
Specializace:	**Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Mobilní aplikace pro načítání a zpracování čísel**
Téma práce anglicky:	**Mobile Phone Application for Reading and Evaluation of Numbers**

Zásady pro vypracování

1. Vytvořte literární rešerši, která se bude zabývat metodami optického rozpoznávání znaků.
2. Stručně popište nástroje a vývojová prostředí používaná pro tvorbu aplikací pro chytré telefony. Zaměřte se na rozdíly oproti tvorbě desktopových aplikací.
3. Zvolte vhodnou mobilní platformu a vytvořte aplikaci, která bude schopna pomocí vestavěné kamery načítat sady čísel a tyto zasílat na externí úložiště pro další zpracování.
4. Doplňte aplikaci o možnost opravy chyb, základní statistické analýzy a grafické prezentace načtených dat ve formě grafů.
5. Otestujte vytvořenou aplikaci na rozsáhlejší množině čísel napsaných různými fonty a řezy písma. Vyzkoušejte také funkčnost pro ručně psaná čísla.
6. Formou případové studie ukažte rozšíření vytvořené aplikace pro použití na vybrané konkrétní praktické úloze.

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. CHAUDHURI, Arindam, Krupa MANDAVIYA, Pratixa BADELIA a Soumya K. GHOSH. *Optical character recognition systems for different languages with soft computing*. Cham, Switzerland: Springer, 2017, 1 online resource. Studies in fuzziness and soft computing. Dostupné z: doi:9783319502526.
2. PÁLKA, Jan. *Optimalizace systémů pro rozpoznávání ručně psaného textu pomocí metod umělé inteligence: disertační práce = Optimization of systems for handwritten text recognition using artificial intelligence methods*. 2013, 154 s. Disertační práce.
3. VÁVRŮ, Jiří a Miroslav UJBÁNYAI. *Programujeme pro Android*. 2., rozš. vyd. Praha: Grada, 2013, 250 s. Průvodce. ISBN 9788024748634.
4. UJBÁNYAI, Miroslav. *Programujeme pro Android*. Grada, 2012, 1 online zdroj (192 stran). ISBN 978-80-247-7983-6. Dostupné také z: https://www.bookport.cz/AccountSaml/SignIn/?idp=https://shibboleth.utb.cz/idp/shibboleth&returnUrl=/kniha/programujeme-pro-android-776/.
5. VÁVRŮ, Jiří. *IPhone: vývoj aplikací*. Grada, 2012, 1 online zdroj (192 stran). ISBN 978-80-247-8338-3. Dostupné také z: https://www.bookport.cz/AccountSaml/SignIn/?idp=https://shibboleth.utb.cz/idp/shibboleth&returnUrl=/kniha/iphone-957/.
6. MEDLÍK, Milan. *Rozpoznávání ručně psaných symbolů*. 2010, 70 s. Dostupné také z: http://hdl.handle.net/10563/11800.

Vedoucí diplomové práce: **Ing. Petr Chalupa, Ph.D.**
Ústav automatizace a řídicí techniky

Datum zadání diplomové práce: **2. prosince 2022**
Termín odevzdání diplomové práce: **26. května 2023**

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 7. prosince 2022

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomové práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové práce bude uložen v příruční knihovně Fakulty aplikované informatiky. Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 20.5.2023 Oleksandr Dolomanov v.r.

podpis studenta

ABSTRAKT

Tato práce se zaměřuje na návrh a vývoj mobilní aplikace pro načítání a zpracování čísel, s důkladným zkoumáním technologie OCR a vývoje mobilních aplikací. Práce je rozdělena na teoretickou část, zkoumající OCR a vývoj mobilních aplikací, a praktickou část, zaměřenou na návrh, implementaci a testování aplikace. V praktické části byla vyvinuta mobilní aplikace, která načítá a zpracovává čísla pomocí vestavěné kamery. Testování aplikace na různých testovacích sadách prokázalo její efektivitu. Práce nabízí detailní pohled na vývoj mobilní aplikace pro načítání a zpracování čísel, což je v praxi využitelné pro optimalizaci ustavování výkovků v průmyslovém prostředí. To dokazuje případová studie prezentovaná v závěru práce.

Klíčová slova: Optické rozpoznávání znaků (OCR), Mobilní aplikace, Načítání čísel, Zpracování čísel, Vývoj mobilních aplikací, Swift, SwiftUI, Vývojová prostředí, Testování aplikací, Architektura aplikace.

ABSTRACT

This work focuses on the design and development of a mobile application for reading and processing numbers, with thorough examination of OCR technology and mobile application development. The work is divided into a theoretical part, studying OCR and mobile application development, and a practical part, focused on the design, implementation, and testing of the application. A mobile application was developed that reads and processes numbers using a built-in camera. Testing the application on various test sets demonstrated its effectiveness. The work offers a detailed view of the development of a mobile application for reading and processing numbers, which is practically applicable for optimizing the establishment of forgings in an industrial environment. This is evidenced by a case study presented at the end of the work.

Keywords: Optical Character Recognition (OCR), Mobile Application, Number Reading, Number Processing, Mobile Application Development, Swift Programming Language, Development Environments, Application Testing, Application Architecture.

Tímto bych chtěl poděkovat Ing. Petru Chalupovi, Ph.D. za pomoc, cenné rady a čas, který mi věnoval při tvorbě teto diplomové práce.

Prohlašuji, že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD

Tato diplomová práce se zaměřuje na jednu z nejdůležitějších oblastí současné informatiky, a to vývoj mobilních aplikací. Zvláštní pozornost je věnována problematice optického rozpoznávání znaků (OCR) a jeho využití v kontextu mobilních aplikací zaměřených na načítání a zpracování čísel.

Rychlý rozvoj mobilních technologií v posledních letech umožnil nejen výrazné zlepšení kvality života mnoha lidí, ale také otevřel nové možnosti pro vývojáře softwaru. Mobilní aplikace se staly nedílnou součástí našeho každodenního života – pomáhají s komunikací, organizací času, prací, studiem a také zábavou. Z tohoto důvodu je důležité neustále zkoumat nové metody a technologie, které mohou přispět k dalšímu zlepšení těchto aplikací.

Tato práce je rozdělena do dvou hlavních částí. V teoretické části je proveden hloubkový průzkum technologie OCR, včetně její historie, principů fungování a budoucích trendů. Další kapitoly jsou věnovány nástrojům a vývojovým prostředím pro tvorbu aplikací pro chytré telefony, včetně srovnání s vývojem desktopových aplikací a detailnímu popisu jazyka Swift.

V praktické části je pak kladen důraz na návrh a implementaci mobilní aplikace pro načítání a zpracování čísel z fotografie. Práce popisuje architekturu aplikace, její funkční a nefunkční požadavky a podrobně se zabývá vývojem pomocných metod a jednotlivých obrazovek aplikace. Praktická část také zahrnuje komplexní testování aplikace na různých testovacích sadách.

Cílem práce je nejen navrhnout a vytvořit funkční aplikaci, ale také podrobně analyzovat všechny procesy, které vedou k jejímu vývoji, a poskytnout tak ucelený pohled na problematiku vývoje mobilních aplikací. Práce by měla sloužit jako užitečný průvodce pro ty, kteří se chtějí ponořit do světa vývoje mobilních aplikací, a poskytnout jim tak pevný základ pro další studium a praxi.

I. TEORETICKÁ ČÁST

1 OPTICKÉ ROZPOZNÁVÁNÍ ZNAKŮ

Optické rozpoznávání znaků (OCR, z anglického Optical Character Recognition) je technologie, která umožňuje převést různé typy dokumentů, jako jsou skenované papírové dokumenty, fotografie dokumentů, PDF soubory, nebo obrazy zachycené digitálním fotoaparátem, do editovatelných a prohledávatelných dat.

Tato technologie funguje tak, že rozpozná a analyzuje obrazy jednotlivých znaků v dokumentu, porovnává je s předem definovanou sadou možných znaků a převádí je do strojově čitelného textu. OCR je obzvláště užitečné pro převod a digitalizaci tisknutých dokumentů, což umožňuje strojové zpracování a analýzu textu, jeho vyhledávání, úpravy, ukládání a další manipulace.

OCR technologie má široké využití v různých oblastech, včetně vědy a výzkumu, vzdělání, obchodu, průmyslu a mnoha dalších. Může výrazně zvýšit efektivitu a produktivitu práce s textovými dokumenty, omezit chyby při přepisování textu a umožnit lepší přístup k informacím. [1]

1.1 Přehled technologie OCR

Technologie OCR zahrnuje několik fází: získávání obrazu, předběžného zpracování, extrakce rysů a klasifikace. Ve fázi získávání obrazu je obraz dokumentu zachycen pomocí skeneru nebo fotoaparátu. Ve fázi předběžného zpracování je obraz předzpracován, aby se odstranil šum, zkreslení. Předběžné zpracování je základním krokem v OCR, protože zvyšuje kvalitu obrazu a usnadňuje rozpoznání znaků. Ve fázi extrakce prvku jsou extrahovány prvky obrazu, jako jsou hrany, čáry a křivky. Extrakce rysů je jednou z nejkritičtějších fází OCR, protože zahrnuje identifikaci jednotlivých znaků v obrázku. Ve fázi klasifikace jsou extrahované prvky klasifikovány jako znaky nebo symboly pomocí algoritmů strojového učení. Fáze klasifikace určuje přesnost OCR, protože zahrnuje mapování extrahovaných prvků na odpovídající znaky nebo symboly. [2]

1.2 Historie a vývoj OCR

První OCR systémy se objevily již v polovině 20. století, ale jejich použití bylo omezené kvůli vysokým nákladům a nízké přesnosti. Od té doby technologie OCR prošla značným vývojem. Zásluhu na tom má především pokrok v oblasti umělé inteligence a strojového učení, který umožnil vývoj pokročilých algoritmů pro rozpoznávání textu.

1.2.1 Počátky OCR

První OCR systémy se objevily v polovině 20. století, konkrétně v roce 1950, kdy byla vyvinuta první komerční OCR technologie firmou RCA. Tato technologie, na-

zvaná "Gismo", byla schopna číst pouze jediný font a byla využívána především pro automatizaci procesu čtení telegrafických zpráv.

1.2.2 Vývoj během 60. a 70. let

V 60. a 70. letech 20. století došlo k významnému vývoji OCR technologie. V roce 1965 firma IBM představila první OCR zařízení pro široké použití, IBM 1287, které bylo schopné číst několik různých fontů. V této době byly také vyvinuty první systémy pro rozpoznávání ručně psaného textu.

1.2.3 Význam umělé inteligence a strojového učení

V 80. a 90. letech začaly mít vliv na OCR technologie pokroky v oblasti umělé inteligence a strojového učení. Byly vyvinuty nové metody pro rozpoznávání znaků, které se učily z tréninkových dat a byly schopné se přizpůsobit různým fontům a stylům psaní. Tyto metody vedly ke značnému zlepšení přesnosti OCR systémů.

1.2.4 Vliv digitalizace a internetu

S nástupem digitalizace a internetu na konci 20. století se zvýšila poptávka po OCR technologii, protože firmy a instituce potřebovaly způsob, jak převést velké množství tištěných dokumentů do digitální formy. To vedlo k dalšímu vývoji OCR technologie a k jejímu rozšíření do mnoha různých aplikací.

1.2.5 Současný vývoj a budoucí směřování

V posledních letech vedly pokroky v oblasti hlubokého učení a neuronových sítí k dalšímu zlepšení přesnosti a efektivity OCR systémů. Dnes jsou OCR systémy schopny rozpoznávat velmi širokou škálu fontů a stylů psaní, včetně některých typů ručně psaného textu. Budoucí vývoj OCR technologie pravděpodobně povede k dalšímu zlepšení přesnosti a rychlosti, a k jejímu využití v nových aplikacích, jako je rozpoznávání textu v reálném čase nebo zpracování dokumentů ve více jazycích.

Technologický pokrok také otevírá nové možnosti pro integraci OCR s dalšími technologiemi, jako je rozšířená realita, zpracování přirozeného jazyka, nebo strojový překlad. Tyto kombinace mohou vést k vytvoření nových nástrojů a služeb, které zlepší způsob, jakým lze interagovat s textem a informacemi v digitálním světě.

Vzhledem k rychlému vývoji v oblasti umělé inteligence a strojového učení je pravděpodobné, že OCR technologie bude i nadále pokračovat v inovacích a zlepšování, což povede k širšímu spektru použití a ještě vyšší přesnosti při rozpoznávání textu. [3] [4]

1.3 Principy fungování OCR

OCR technologie funguje na principu rozpoznávání vzorů nebo porovnávání obrazových dat. Základem je výpočetní algoritmus, který se snaží identifikovat jednotlivé znaky na skenované stránce nebo obrázku a převést je na text. K tomu může dojít buď porovnáním s předem definovanými vzory, nebo učením se na základě sady tréninkových dat.

1.3.1 Analýza obrazu

Prvním krokem v procesu OCR je analýza obrazu. Vstupní obrázek je analyzován a rozdělen na segmenty, které odpovídají jednotlivým znakům, slovům nebo jiným komponentům, jako jsou tabulky a obrázky. Tato fáze také obvykle zahrnuje předzpracování obrazu, například odstranění šumu, normalizaci osvětlení a kontrastu, a další techniky, které pomáhají zlepšit kvalitu obrázku pro následné zpracování.

1.3.2 Rozpoznávání znaků

V této fázi jsou jednotlivé segmenty analyzovány a porovnány s databází známých tvarů znaků. Tento proces může být založen na různých technikách, včetně metody porovnávání vzorů, výpočetní geometrie, statistických metod a strojového učení. Výsledkem je sada hypotéz o tom, jaký znak nebo znaky odpovídají každému segmentu.

1.3.3 Kontextová analýza

Po identifikaci jednotlivých znaků následuje kontextová analýza, která se snaží interpretovat znaky v kontextu sousedních znaků a slov. To zahrnuje například opravu chyb v rozpoznání znaků na základě pravděpodobnosti výskytu daného znaku v daném kontextu, rozpoznání slov na základě slovníku a gramatiky jazyka, nebo interpretaci formátování a rozložení dokumentu.

1.3.4 Použití strojového učení a hlubokého učení

Moderní OCR systémy často využívají techniky strojového učení a hlubokého učení pro zlepšení přesnosti rozpoznání znaků a interpretace dokumentů. Strojové učení umožňuje OCR systémům "učit se"z tréninkových dat a automaticky se přizpůsobit různým fontům, stylům a formátům dokumentů. Hluboké učení, konkrétně konvoluční neuronové sítě, jsou zvláště účinné pro zpracování obrazových dat a jsou často používány pro rozpoznávání znaků a slov v OCR systémech.

1.3.5 Post-processing

Poslední fází je post-processing, kde se dokončí formátování výstupního textu. To může zahrnovat opravu chyb, které nebyly detekovány v předchozích fázích, převod textu do požadovaného formátu, nebo začlenění výsledků do databází nebo jiných systémů pro další zpracování. Post-processing může také zahrnovat další analýzu textu, například extrakci informací, vyhledávání klíčových slov, nebo analýzu sentimentu. Výsledkem celého procesu OCR je digitální text, který odpovídá obsahu původního dokumentu a který lze snadno upravovat, prohledávat, a dále zpracovávat pomocí počítačových systémů. Přestože OCR technologie není dokonalá a může se stále objevit určitý počet chyb, její přesnost a efektivita se neustále zlepšuje díky pokroku v oblasti umělé inteligence a strojového učení. [5]

1.4 Formáty souborů a kompatibilita

Technologie OCR je kompatibilní s řadou různých formátů souborů, jak pro vstup, tak pro výstup.

1.4.1 Vstupní formáty souborů

OCR systémy jsou schopny zpracovat řadu různých formátů obrazových souborů, které obsahují text. Nejběžnějšími formáty jsou:

- JPEG: Jedná se o velmi běžný formát obrazových souborů, který je široce používán pro ukládání a přenos fotografií.
- PNG: Tento formát je oblíbený pro jeho schopnost ukládat obrazy s průhledností.
- TIFF: Tento formát se často používá pro skenování dokumentů a je oblíbený pro jeho schopnost ukládat obrazy ve vysoké kvalitě.
- PDF: Tento formát je široce používán pro distribuci digitálních dokumentů a může obsahovat jak text, tak obrazy.
- BMP: Toto je formát bitmapového obrazu, který je schopen ukládat detailní obrazy.

1.4.2 Výstupní formáty souborů

Po zpracování OCR mohou být výsledky uloženy v různých formátech textových souborů. Nejběžnějšími formáty jsou:

- TXT: Toto je jednoduchý formát textového souboru bez formátování.

- DOC a DOCX: Toto jsou formáty souborů používané programem Microsoft Word. Obsahují formátování a jsou široce používány pro vytváření a úpravu dokumentů.
- PDF: OCR systémy mohou také vytvářet prohledávatelné PDF soubory, kde je text převeden na strojově čitelný formát, zatímco původní vzhled dokumentu je zachován.
- RTF: Toto je univerzální formát souborů, který umožňuje formátování textu a je kompatibilní s většinou textových editorů.
- HTML: Tento formát je používán pro webové stránky a umožňuje vytvářet strukturované a formátované texty.

1.4.3 Kompatibilita s aplikacemi

Výsledky OCR mohou být také exportovány přímo do některých aplikací, jako jsou textové editory (například Microsoft Word, Google Docs), tabulkové procesory (například Microsoft Excel, Google Sheets) nebo databáze. Některé OCR systémy také umožňují přímý export do cloudových úložišť. [6]

1.5 Typy technik OCR

Technologie OCR se vyvíjela v průběhu let a existuje několik základních technik OCR, které jsou používány k rozpoznávání znaků v různých situacích a aplikacích.

1.5.1 Optické techniky založené na matici (Template matching)

Tyto techniky porovnávají obraz znaku s předem definovanými šablonami nebo maticemi, které reprezentují různé možné tvary a varianty znaků. Tento přístup je často omezen na specifické fonty a může mít potíže s rozpoznáváním znaků, které se liší od šablon.

1.5.2 Feature-based OCR

Tato technika analyzuje a extrahuje klíčové vlastnosti (features) znaků, jako jsou linie, křivky, úhly nebo přechody mezi černou a bílou barvou, a používá je k rozpoznání znaků. Feature-based OCR je obecně flexibilnější a schopnější rozpoznávat různé fonty a styly písma, ale může mít potíže s nízkou kvalitou obrazu nebo ručně psaným textem.

1.5.3 Neuronové sítě a strojové učení

Moderní OCR systémy často využívají neuronové sítě a algoritmy strojového učení, jako jsou konvoluční neuronové sítě (CNN) nebo rekurentní neuronové sítě (RNN), které se

učí rozpoznávat znaky na základě velkých datových sad obsahujících různé příklady textu. Tyto metody mají obecně vysokou úroveň přesnosti a mohou se přizpůsobit rozmanitosti fontů, stylů písma a jazyků.

1.5.4 Segmentace a kontext

Některé OCR techniky se zaměřují na segmentaci textu do jednotlivých znaků nebo slov a využívají kontextuální informace, jako je gramatika, syntax a sémantika, k vylepšení přesnosti rozpoznání znaků. Tento přístup může pomoci překonat některé z výzev spojených s nízkou kvalitou obrazu nebo ručně psaným textem.

1.5.5 Hybridní metody

Hybridní OCR systémy kombinují různé techniky, jako jsou optické techniky, feature-based metody, strojové učení a kontext, aby dosáhly co nejlepšího výkonu a přesnosti v různých situacích a aplikacích.

Vzhledem k různorodosti technik OCR je důležité vybrat správný přístup v závislosti na konkrétní aplikaci a požadavcích. Například pro archivaci historických dokumentů by mohlo být vhodné využití metody založené na strojovém učení, která je schopna se přizpůsobit rozmanitosti archaických fontů a stylů písma. Na druhou stranu, pro automatizované zpracování faktur by mohla být preferována technika založená na matici nebo feature-based metoda, která je rychlá a efektivní v rozpoznávání standardních fontů a formátů. Hybridní metody mohou nabídnout nejlepší z obou světů tím, že kombinují rychlost a efektivitu s flexibilitou a přizpůsobivostí. [7]

1.6 Metriky výkonu pro systémy OCR

Systémy OCR se používají k rozpoznávání textových znaků z obrázků nebo dokumentů a jejich výkon je obvykle hodnocen na základě určitých metrik.

1.6.1 Přesnost

Toto je nejzákladnější metrika výkonu pro systémy OCR. Je to procento správně rozpoznaných znaků z celkového počtu znaků v obrázku nebo dokumentu. Přesnost OCR je ovlivněna mnoha faktory, jako je kvalita obrazu, typ písma, velikost a styl.

1.6.2 Recall and Precision

Tyto metriky se obvykle používají k vyhodnocení výkonu systémů OCR při konkrétních úkolech, jako je rozpoznávání určitých znaků nebo slov. Recall je poměr správně rozpoznaných znaků k celkovému počtu znaků v obrázku, které měly být rozpoznány,

zatímco preciznost je poměr správně rozpoznaných znaků k celkovému počtu rozpoznaných znaků. Tyto metriky jsou důležité pro úkoly, jako je extrakce dat z dokumentů nebo hledání konkrétních slov ve velkém množství dokumentů.

1.6.3 Chybovost

Systémy OCR obvykle vytvářejí chyby při rozpoznávání znaků a chybovost je procento nesprávně rozpoznaných znaků z celkového počtu znaků v obrázku nebo dokumentu. Chybovost lze dále rozdělit na chyby nahrazování (kdy je znak rozpoznán jako jiný znak), chyby vkládání (kdy je přidán další znak) a chyby mazání (kdy znak chybí).

1.6.4 Rychlost zpracování

Systémy OCR musí zpracovávat obrázky nebo dokumenty rychle, zejména pro aplikace, jako je rozpoznávání textu z živých video streamů v reálném čase. Rychlost zpracování se obvykle měří ve stránkách za minutu nebo ve znacích za sekundu.

1.6.5 Robustnost

Systémy OCR by měly být schopny zpracovat různé typy vstupů, jako jsou obrázky s různým rozlišením nebo různými typy písem. Robustnost je schopnost systému OCR zvládnout takové variace bez výrazného poklesu přesnosti.

1.6.6 Škálovatelnost

Systémy OCR by měly být schopny efektivně zpracovávat velké objemy vstupních dat. Škálovatelnost je schopnost systému OCR zpracovávat velké objemy dat rychle a přesně. Celkově výkon systému OCR závisí na mnoha faktorech a metriky používané k hodnocení výkonu systému by měly být zvoleny na základě konkrétního úkolu nebo dané aplikace. [8]

1.7 Aplikace OCR v každodenním životě

Technologie OCR má široké využití v každodenním životě, které zahrnuje různé oblasti a úkoly. Některé z nich mohou byt:

1. **Digitalizace tištěných dokumentů**: OCR umožňuje převést tištěné dokumenty, jako jsou dopisy, smlouvy, články nebo knihy, na digitální text. To umožňuje snadnější práci s textem, jeho úpravu, vyhledávání, sdílení, třídění a ukládání.

2. **Čtení a převod tištěných knih a článků**: OCR lze použít k digitalizaci tištěných knih a článků, které jsou pak přístupné v elektronické podobě. To umožňuje

pohodlnější čtení na elektronických zařízeních, jako jsou čtečky e-knih, tablety nebo počítače.

3. **Překlad textu pomocí OCR**: Kombinace OCR s technologiemi strojového překladu umožňuje automatický překlad tištěných dokumentů do jiných jazyků. Stačí skenovat nebo vyfotit dokument a OCR software převede text na strojově čitelnou podobu, kterou lze poté snadno přeložit.

4. **Přístupnost pro zrakově postižené uživatele**: OCR může být nástrojem, který zlepšuje přístup k informacím pro osoby se zrakovým postižením. Skenovaný text lze převést na digitální formát, který lze poté předčítat pomocí čtecích programů nebo zobrazit na zařízeních s hmatovým výstupem, jako jsou Braillské řádky.

5. **Skenování vizitek a ukládání kontaktních informací**: OCR technologie umožňuje snadno skenovat vizitky a extrahovat kontaktní informace, které lze poté uložit do adresáře telefonu nebo jiné aplikace pro správu kontaktů.

Takto OCR technologie zjednodušuje a zrychluje mnoho každodenních úkolů a pomáhá lidem efektivněji pracovat s informacemi a komunikovat s digitálními technologiemi.

1.8 Použití OCR v aplikacích pro mobilní telefony

Technologie optického rozpoznávání znaků je v aplikacích pro mobilní telefony stále oblíbenější. Technologie OCR umožňuje uživatelům skenovat a převádět text z papírových dokumentů do digitálního formátu přímo na jejich mobilních zařízeních. V této kapitole bude probráno použití OCR v aplikacích pro mobilní telefony a výhody, které nabízí.

1.8.1 Implementace OCR v aplikacích pro mobilní telefony

Implementace OCR v aplikacích pro mobilní telefony vyžaduje pečlivé zvážení různých faktorů, jako je kvalita obrazu, rozvržení dokumentu a jazyková podpora. Vývojáři si musí vybrat vhodnou knihovnu nebo službu OCR, optimalizovat algoritmy zpracování obrazu pro mobilní zařízení a zajistit, aby OCR engine zvládnul různé typy dokumentů a jazyků.

1.8.2 Výzvy a omezení OCR v aplikacích pro mobilní telefony

Navzdory výhodám OCR v aplikacích pro mobilní telefony jsou zde také některé výzvy a omezení, které je třeba vzít v úvahu. Technologie OCR se může potýkat s rozpoznáním ručně psaného textu, nekvalitních obrázků a složitých rozvržení dokumentů.

Přesnost OCR může být navíc ovlivněna světelnými podmínkami a kvalitou fotoaparátu v mobilním zařízení. Technologie OCR se stala důležitým nástrojem pro aplikace mobilních telefonů, poskytuje zvýšenou efektivitu, lepší přesnost a lepší uživatelskou zkušenost. I když existují určité výzvy a omezení, která je třeba vzít v úvahu, technologie OCR nabízí uživatelům mobilních telefonů mnoho výhod a její využití bude v budoucnu pravděpodobně dále růst.

1.8.3 Budoucnost OCR a další vývoj

Technologie OCR se neustále vyvíjí a zlepšuje, což otevírá nové možnosti pro její využití. Některé trendy a vývoj, které by mohly ovlivnit budoucnost OCR jsou zmíněny níže:

1. **Strojové učení a umělá inteligence**: Technologie OCR se stále více spoléhají na strojové učení a umělou inteligenci pro zlepšení přesnosti a účinnosti rozpoznávání znaků. To zahrnuje použití konvolučních neuronových sítí a jiných pokročilých algoritmů pro učení z velkých datových sad a lepší rozpoznávání různých stylů a formátů textu.

2. **Zlepšení rozpoznávání ručně psaného textu**: S využitím pokročilých technologií strojového učení a umělé inteligence se očekává, že se zlepší schopnost OCR systémů rozpoznávat ručně psaný text, což by mohlo rozšířit jejich využití v oblastech jako je vzdělání, práce s historickými dokumenty a další.

3. **Rozpoznávání hlasu a OCR**: Kombinace OCR a rozpoznávání hlasu by mohla umožnit nové způsoby interakce s textovými dokumenty. Například, uživatel by mohl požádat své zařízení o přečtení tištěného dokumentu nahlas nebo by mohl použít hlasové příkazy k navigaci v digitálně převedeném textu.

4. **Integrace s cloudovými službami**: Větší integrace OCR technologie s cloudovými službami by mohla umožnit lepší sdílení a přístup k digitálně převedeným dokumentům napříč různými zařízeními a platformami.

5. **Využití rozšířené reality (AR)**: V kombinaci s technologiemi rozšířené reality by OCR mohlo umožnit překlad textu v reálném čase přímo v uživatelově vizuálním poli, což by mohlo být užitečné například při cestování v zahraničí.

6. **Zlepšení přístupnosti**: Pokračující vývoj a vylepšení OCR technologie mohou přinést významný přínos pro osoby se zrakovým postižením nebo jinými omezeními přístupnosti, tím že zpřístupní více textového obsahu a umožní lepší interakci s digitálními zařízeními.

Vzhledem k těmto trendům je zřejmé, že OCR má před sebou světlou budoucnost, která bude nadále přinášet nové možnosti a zlepšovat efektivitu a přístupnost informací v digitálním věku.

1.9 Dopad OCR na společnost a technologický vývoj

Technologie OCR má hluboký dopad na společnost a technologický vývoj v několika klíčových oblastech:

1. **Digitalizace a zachování historických dokumentů**: OCR umožňuje digitalizaci a zachování historických a archivních dokumentů, které by mohly být jinak zničeny nebo ztraceny. Tímto způsobem OCR přispívá k ochraně kulturního dědictví a usnadňuje výzkum a studium historických textů.

2. **Zlepšení přístupnosti**: OCR zlepšuje přístupnost informací pro osoby se zrakovým postižením nebo dyslexií tím, že převádí tištěný text na formát, který lze předčítat pomocí čtecích programů nebo zobrazit na zařízeních s hmatovým výstupem, jako jsou Braillské řádky.

3. **Zvýšení efektivity a produktivity**: OCR usnadňuje správu a manipulaci s dokumenty v obchodním a akademickém prostředí tím, že umožňuje rychlé vyhledávání a editaci textu, automatické zpracování faktur a formulářů, a další úkoly, které by byly časově náročné nebo nemožné bez OCR.

4. **Podpora jazykových studií a překladů**: OCR umožňuje automatický překlad tištěných dokumentů do jiných jazyků, což může podpořit jazykové studium a multikulturní komunikaci.

5. **Inovace v technologickém vývoji**: OCR je klíčovou technologií, která podporuje vývoj a inovace v oblastech jako je umělá inteligence, strojové učení, rozšířená realita, robotika, a další. Například, roboti vybaveni OCR technologií mohou číst a rozumět textovým pokynům v reálném světě, což otevírá nové možnosti pro automatizaci a interakci mezi lidmi a stroji.

Celkově má technologie OCR široký dopad na společnost a technologický vývoj, který sahá od zlepšení přístupnosti a efektivity práce s dokumenty až po podporu inovací v široké škále technologických oblastí.

2 NÁSTROJE A VÝVOJOVÁ PROSTŘEDÍ PRO TVORBU APLIKACÍ PRO CHYTRÉ TELEFONY

V dnešní době mobilní aplikace zásadně mění způsob, jakým lidé pracují, komunikují a baví se. Počet chytrých telefonů a mobilních zařízení stále roste, což vede k větší poptávce po kvalitních aplikacích. Vývojáři se proto musí naučit pracovat s nástroji a technologiemi specifickými pro mobilní platformy, aby mohli vytvářet úspěšné a uživatelsky přívětivé aplikace.

Tato kapitola se zaměřuje na nástroje a vývojová prostředí používaná pro tvorbu aplikací pro chytré telefony. Budou v ní představeny hlavní jazyky, nástroje a frameworky používané v mobilním vývoji a budou porovnány s těmi, které se používají při vývoji desktopových aplikací. Také budou zmíněny klíčové rozdíly mezi vývojem pro mobilní a desktopové platformy, včetně hardwarových a operačních omezení, interakce uživatele, síťových omezení, bezpečnosti, soukromí a distribuce aplikací. Dále bude zkoumáno testování a ladění mobilních aplikací, vývoj pro více platforem a budoucnost vývoje mobilních aplikací.

2.1 Přehled hlavních rozdílů mezi vývojem mobilních a desktopových aplikací

Mobilní a desktopové aplikace jsou vytvářeny s odlišnými cíli a omezeními. Zatímco desktopové aplikace jsou navrženy pro výkonnější hardwarové prostředí s většími obrazovkami a mohou využívat komplexnější uživatelské rozhraní, mobilní aplikace musí být optimalizovány pro efektivní využití baterie, menší paměťovou kapacitu a omezený výkon.

1. **Hardwarové a operační systémové omezení**: Mobilní zařízení mají omezený výkon, paměť a úložiště. Velikost obrazovky, rozlišení a životnost baterie jsou také důležitými faktory. Na druhé straně desktopové počítače mají obecně vyšší výkon a kapacitu úložiště.

2. **Interakce uživatele a design**: Mobilní aplikace jsou obvykle ovládané pomocí dotykové obrazovky, zatímco desktopové aplikace využívají myš a klávesnici. To má zásadní dopad na design uživatelského rozhraní.

3. **Připojení a síťová omezení**: Mobilní aplikace musí být navrženy tak, aby mohly efektivně fungovat na mobilních datech, které mohou být pomalejší nebo méně stabilní než pevné internetové připojení, které se obvykle využívá u desktopových počítačů.

4. **Bezpečnost a soukromí**: Mobilní aplikace musí být navrženy s ohledem na

soukromí a bezpečnost, protože mobilní zařízení obecně obsahují více osobních informací než desktopové počítače.

5. **Distribuce a aktualizace**: Mobilní aplikace jsou obvykle distribuovány prostřednictvím obchodů s aplikacemi a musí dodržovat jejich politiky. Desktopové aplikace mohou být distribuovány různými způsoby, včetně přímého stažení z webových stránek.

Tyto rozdíly mají značný vliv na volbu nástrojů a postupů při vývoji mobilních a desktopových aplikací, což bude dále probíráno v následujících kapitolách.

2.2 Současný stav mobilního vývoje aplikací

Mobilní vývoj aplikací je dynamický a rychle se vyvíjející obor. Hlavními platformami pro mobilní aplikace jsou Android a iOS, každá s vlastní sadou nástrojů, jazyků a best practices pro vývoj.

- **Android**: Android je otevřený operační systém od společnosti Google, který je široce používán na mnoha typech zařízení od různých výrobců. Android aplikace jsou obvykle psané v Javě nebo Kotlinu a využívají Android Studio jako hlavní vývojové prostředí.
- **iOS**: iOS je operační systém od společnosti Apple a je používán na iPhonech a iPadech. Aplikace pro iOS jsou obvykle psané v Objective-C nebo Swiftu a využívají Xcode jako hlavní vývojové prostředí.

Kromě těchto nativních nástrojů existují také různé crossplatformní nástroje a frameworky, jako jsou React Native, Flutter a Xamarin, které umožňují vývojářům psát kód jednou a spustit ho na obou platformách.

V současné době je trendem vývoj více funkčních, uživatelsky přívětivých aplikací, které efektivně využívají omezené prostředky mobilních zařízení. K tomu se vývojáři stále častěji uchylují k pokročilým technologiím jako je strojové učení, rozšířená a virtuální realita a internet věcí (IoT). [9]

2.2.1 Statistiky a trendy

Podle statistik se počet uživatelů mobilních aplikací na celém světě neustále zvyšuje. V roce 2023 bylo například na celém světě více než 6,9 miliardy uživatelů chytrých telefonů a odhaduje se, že tento počet bude i nadále růst. [10]

V současné době existují dvě hlavní platformy pro mobilní aplikace: Android a iOS. Android má větší tržní podíl, což je dáno zejména větším počtem zařízení a nižšími

náklady na vstup pro vývojáře. Nicméně iOS uživatelé jsou často považováni za výdělečnější cílovou skupinu, jelikož průměrně utratí více za aplikace a nákupy v aplikacích. Co se týče trendů v mobilním vývoji, lze sledovat následující:

1. **Rozšířená a virtuální realita**: Technologie rozšířené a virtuální reality (AR a VR) jsou stále více využívány v mobilních aplikacích, zejména ve hrách a aplikacích pro vzdělávání.

2. **Strojové učení a umělá inteligence**: Strojové učení a umělá inteligence (AI) se stávají stále důležitějšími pro mobilní aplikace, umožňují personalizaci obsahu, rozpoznávání obrazu, předpovědi chování uživatele a mnoho dalšího.

3. **Internet věcí (IoT)**: S rostoucím počtem chytrých zařízení se stále více aplikací integruje s IoT zařízeními, což umožňuje uživatelům ovládat domácí zařízení, sledovat své zdraví a mnoho dalšího přímo z jejich mobilních telefonů.

4. **Multiplatformní vývoj**: S narůstající popularitou nástrojů jako React Native a Flutter, které umožňují vývojářům psát kód jednou a spustit ho na více platformách, se očekává, že multiplatformní vývoj bude i nadále v trendu.

5. **Bezpečnost**: S narůstajícím počtem kybernetických útoků a obavami o ochranu dat se bezpečnost stává klíčovou prioritou pro vývojáře mobilních aplikací. Aplikace, které dokáží chránit data uživatelů a zajistit jejich soukromí, budou mít výhodu.

Toto jsou jen některé z trendů, které ovlivňují současný stav a budoucí směr mobilního vývoje aplikací. Je důležité, aby vývojáři sledovali tyto trendy a přizpůsobovali své strategie a nástroje, aby mohli vytvářet aplikace, které splňují očekávání a potřeby uživatelů. [11]

2.3 Nástroje a jazyky pro vývoj mobilních aplikací

Existuje mnoho nástrojů a jazyků určených pro vývoj mobilních aplikací. Tyto nástroje a jazyky se liší v závislosti na platformě a na typu aplikace, kterou chce vývojář vytvořit.

- **Nativní vývoj**: Nativní vývoj znamená vytváření aplikací specificky pro jednu platformu pomocí nástrojů a jazyků podporovaných danou platformou. Pro Android jsou to obvykle Java a Kotlin s vývojovým prostředím Android Studio. Pro iOS jsou to jazyky Swift a Objective-C s vývojovým prostředím Xcode.

- **Multiplatformní vývoj**: Multiplatformní nástroje umožňují vývojářům psát kód jednou a spustit ho na více platformách. Toto může být časově efektivní

a může snížit náklady na vývoj. Některé z populárních multiplatformních nástrojů zahrnují React Native (využívající JavaScript), Flutter (využívající Dart) a Xamarin (využívající C#).

- **Hybridní vývoj**: Hybridní aplikace jsou vytvořeny pomocí webových technologií (jako jsou HTML, CSS a JavaScript) a jsou zabalené v nativním obalu, který umožňuje přístup k nativním funkcím zařízení. Cordova a Ionic jsou příklady nástrojů pro hybridní vývoj.

- **Vývojové prostředí (IDE)**: Vývojové prostředí zahrnují editor kódu, debugger a nástroje pro testování. Android Studio a Xcode jsou nativní IDE pro Android a iOS. Další IDE jako Visual Studio Code, IntelliJ IDEA nebo Eclipse mohou být použity s různými pluginy a rozšířeními pro vývoj mobilních aplikací.

- **Nástroje pro testování a ladění**: Tyto nástroje pomáhají vývojářům identifikovat a opravit chyby v jejich aplikacích. Příklady zahrnují nástroje pro jednotkové testování, jako je JUnit pro Java, XCTest pro Swift a Objective-C, a nástroje pro UI testování, jako je Espresso pro Android a XCUITest pro iOS.

Výběr správných nástrojů a jazyků pro vývoj mobilních aplikací závisí na řadě faktorů, včetně cílové platformy, požadavků na výkon aplikace, dovedností vývojářského týmu a časového rámce projektu. [12]

2.3.1 Jazyky pro vývoj mobilních aplikací

Různé jazyky se používají pro vývoj mobilních aplikací, přičemž každý jazyk má své vlastnosti a výhody. Některé z nejpoužívanějších jazyků pro vývoj mobilních aplikací zahrnují:

1. **Java**: Java je objektově orientovaný programovací jazyk, který je široce používán pro vývoj Android aplikací. Java je známá svou přenositelností, bezpečností a širokou podporou knihoven a frameworků.

2. **Kotlin**: Kotlin je moderní, staticky typovaný programovací jazyk, který je plně interoperabilní s Javou. Kotlin byl navržen s cílem zlepšit produktivitu vývojářů a eliminovat některé nedostatky Javy. Kotlin je nyní oficiálně podporován společností Google pro vývoj Android aplikací.

3. **Swift**: Swift je rychlý, moderní programovací jazyk vytvořený společností Apple pro vývoj aplikací pro iOS, macOS, watchOS a tvOS. Swift je navržen tak, aby byl snadno čitelný, bezpečný a vysoce výkonný.

4. **Objective-C**: Objective-C je starší programovací jazyk používaný pro vývoj aplikací pro Apple platformy. I když je Swift nyní preferovaným jazykem pro vývoj iOS aplikací, Objective-C je stále používán v mnoha starších aplikacích a knihovnách.

5. **JavaScript**: JavaScript je dynamický, interpretovaný programovací jazyk, který je nejčastěji používán pro webový vývoj. JavaScript se také používá pro vývoj mobilních aplikací pomocí multiplatformních nástrojů jako React Native a hybridních nástrojů jako Cordova a Ionic.

6. **Dart**: Dart je objektově orientovaný programovací jazyk navržený společností Google. Dart je primárně používán pro vývoj aplikací pomocí multiplatformního frameworku Flutter, který umožňuje vytvářet nativní aplikace pro Android a iOS s jedinou kódovou základnou.

Výběr jazyka pro vývoj mobilních aplikací závisí na platformě, na které chce vývojář aplikaci spustit, na nástrojích a frameworcích, které vývojář chce a nebo může použít, a na preferencích a zkušenostech vývojového týmu. [13] [14]

2.3.2 Vývojové prostředí: Android Studio, Xcode, Visual Studio Code atd.

Vývojové prostředí (IDE) hrají klíčovou roli v procesu vývoje aplikací. Poskytují nástroje a rozhraní potřebné pro psaní, testování a ladění kódu. Některá z nejběžnějších IDE pro vývoj mobilních aplikací zahrnují:

1. **Android Studio**: Android Studio je oficiální IDE pro vývoj Android aplikací. Poskytuje řadu nástrojů, včetně editoru kódu, emulátoru Android, profileru výkonu, systému pro správu verzí a mnoha dalších. Android Studio podporuje programovací jazyky Java a Kotlin. [15]

2. **Xcode**: Xcode je IDE vytvořené společností Apple pro vývoj aplikací pro iOS, macOS, watchOS a tvOS. Xcode poskytuje širokou škálu nástrojů pro vývoj, testování a ladění aplikací. Podporuje programovací jazyky Swift a Objective-C. [16]

3. **Visual Studio Code**: Visual Studio Code je univerzální textový editor vytvořený společností Microsoft. I když to není plně vybavené IDE jako Android Studio nebo Xcode, Visual Studio Code je velmi populární díky své flexibilitě a podpoře široké škály programovacích jazyků a nástrojů. Pro vývoj mobilních aplikací je často používán s multiplatformními nástroji, jako je React Native nebo Flutter. [17]

4. **IntelliJ IDEA**: IntelliJ IDEA je univerzální IDE vytvořené společností JetBrains, tvůrci Kotlinu. IntelliJ IDEA poskytuje pokročilé funkce pro psaní a ladění kódu, včetně inteligentního doplňování kódu, analýzy kódu, integrovaného správce verzí a mnoha dalších. Pro vývoj mobilních aplikací je často používán s Android Studio pluginem nebo pro vývoj aplikací ve frameworku Flutter. [18]

Každé IDE má své vlastní výhody a nevýhody, a výběr mezi nimi závisí na preferencích vývojáře, na požadavcích projektu a na platformě, pro kterou se vývoj provádí.

2.3.3 Frameworky pro vývoj mobilních aplikací: Flutter, React Native, Xamarin, Cordova

Frameworky pro vývoj mobilních aplikací umožňují vývojářům snadněji a rychleji vytvářet aplikace pro různé platformy. Tyto frameworky mohou být nativní, multiplatformní nebo hybridní, v závislosti na jejich přístupu k vývoji aplikací. Některé z nejběžnějších frameworků zahrnují:

1. **Flutter**: Flutter je open-source UI toolkit vytvořený společností Google, který umožňuje vývojářům vytvářet nativní aplikace pro Android a iOS z jedné základny kódu. Flutter používá programovací jazyk Dart a nabízí vysokou úroveň výkonu a plynulosti animací. Flutter také zahrnuje rozsáhlou sadu widgetů a nástrojů pro snadné vytváření uživatelských rozhraní. [19]

2. **React Native**: React Native je open-source framework vytvořený společností Meta, který umožňuje vývojářům vytvářet nativní aplikace pro Android a iOS pomocí JavaScriptu a Reactu. React Native využívá komponenty, které mapují na nativní UI prvky, což zajišťuje vysoký výkon a nativní vzhled a chování aplikací. [20]

3. **Xamarin**: Xamarin je open-source framework vytvořený společností Microsoft, který umožňuje vývojářům vytvářet nativní aplikace pro Android, iOS a Windows pomocí jednoho jazyka, C#. Xamarin využívá Mono runtime, který poskytuje společnou základnu kódu a přístup k nativním API pro všechny platformy. Xamarin.Forms je další součást Xamarinu, která umožňuje sdílet i kód pro uživatelské rozhraní mezi platformami. [21]

4. **Cordova / PhoneGap**: Cordova (dříve známá jako PhoneGap) je open-source framework pro vývoj hybridních mobilních aplikací. Cordova umožňuje vývojářům vytvářet aplikace pomocí webových technologií (HTML, CSS a JavaScript) a poskytuje nativní kontejnery pro jejich spuštění na různých platformách. Cordova také poskytuje rozhraní pro přístup k nativním funkcím zařízení, jako je kamera, GPS a kontakty. [22]

Volba frameworků pro vývoj mobilních aplikací závisí na požadavcích projektu, cílových platformách, zkušenostech a preferencích vývojářů a dalších faktorech. Například Flutter a React Native mohou být vhodné pro projekty, které vyžadují rychlý vývoj a vysokou úroveň sdílení kódu mezi platformami, zatímco Xamarin může být preferován pro projekty, které využívají .NET a C#, a Cordova pro projekty, které se zaměřují na maximalizaci opětovného použití webového kódu.

2.4 Porovnání s vývojem desktopových aplikací

I když existují určité podobnosti mezi vývojem mobilních a desktopových aplikací, existují také některé zásadní rozdíly, které je třeba vzít v úvahu.

1. **Platformy a jazyky**: Desktopové aplikace jsou obvykle vyvíjeny pro systémy Windows, MacOS nebo Linux, zatímco mobilní aplikace jsou primárně pro platformy Android a iOS. To ovlivňuje výběr programovacích jazyků a nástrojů. Například, zatímco C#, Java nebo C++ mohou být často použity pro vývoj desktopových aplikací, mobilní vývoj často využívá jazyky jako Kotlin, Swift nebo JavaScript.

2. **UI/UX Design**: Design uživatelského rozhraní a uživatelské zkušenosti je také velmi odlišný. Mobilní aplikace musí být navrženy s ohledem na menší obrazovky, dotykové ovládání a omezené systémové prostředky. Na druhé straně, desktopové aplikace mohou využívat větší obrazovky, myši a klávesnice a obecně disponují většími systémovými prostředky.

3. **Dostupnost a distribuce**: Zatímco desktopové aplikace mohou být často staženy přímo z webových stránek vývojářů, mobilní aplikace jsou obvykle distribuovány prostřednictvím obchodů s aplikacemi, jako je Google Play nebo Apple App Store, které mají své vlastní požadavky a omezení.

4. **Vývojové prostředí a nástroje**: Pro vývoj desktopových aplikací se často používají IDE jako Visual Studio, Eclipse nebo IntelliJ IDEA. Na druhé straně, pro vývoj mobilních aplikací se často používají specifické nástroje, jako je Android Studio pro Android nebo Xcode pro iOS.

5. **Multiplatformní vývoj**: Pro multiplatformní vývoj mobilních aplikací existuje mnoho nástrojů a frameworků, jako je Flutter, React Native nebo Xamarin. Na straně desktopu, existují také některé multiplatformní nástroje, jako je Electron nebo Qt, ale jejich použití není tak rozšířené jako v mobilním prostředí.

6. **Hardware a přístup k funkcím zařízení**: Mobilní zařízení mají řadu hardwarových prvků, jako jsou GPS, akcelerometr, gyroskop, kamera atd., které nejsou

obvykle k dispozici na desktopových počítačích. To otevírá nové možnosti pro mobilní aplikace, ale také představuje další výzvy pro vývojáře, kteří musí tyto funkce správně integrovat a ošetřit různé stavové scénáře (například, co se stane, když je GPS vypnuta).

7. **Životní cyklus aplikace a omezení na pozadí**: Na rozdíl od desktopových aplikací, které mohou běžet na pozadí po dlouhou dobu, mobilní operační systémy mají striktní omezení pro aplikace běžící na pozadí, aby ušetřily baterii a zdroje. To znamená, že vývojáři musí pečlivě spravovat životní cyklus aplikace a ukládat a obnovovat stav aplikace, když je to nutné.

8. **Aktualizace a kompatibilita**: Mobilní aplikace musí být kompatibilní s mnoha různými verzemi operačního systému a různými typy zařízení s různými velikostmi obrazovky, rozlišením a hardwarovými schopnostmi. Na druhé straně, ačkoli desktopové aplikace také musí zvládnout různé konfigurace systému, obecně jsou tyto problémy méně komplexní než v mobilním prostředí.

Celkově platí, že ačkoli existují určité podobnosti mezi vývojem mobilních a desktopových aplikací, existují také značné rozdíly, které vývojáři musí vzít v úvahu. Výběr správných nástrojů, jazyků a postupů závisí na konkrétních požadavcích a omezeních každého projektu. [23] [24]

2.4.1 Jazyky a nástroje pro vývoj desktopových aplikací: C++, C#, Java, Python atd.

Při vývoji desktopových aplikací mají vývojáři k dispozici širokou škálu jazyků a nástrojů. Následující jazyky a nástroje jsou mezi nejpopulárnější:

1. **C++**: Tento jazyk je často používán pro vývoj vysokovýkonných a náročných aplikací, jako jsou hry, grafické editory nebo systémový software. C++ poskytuje velkou kontrolu nad systémovými prostředky, ale je také složitější a náročnější na vývoj a údržbu. Nejběžněji používané vývojové prostředí pro C++ zahrnují Visual Studio, Code::Blocks a CLion.

2. **C#**: C# je jazyk vytvořený společností Microsoft, který je široce používán pro vývoj Windows aplikací. C# je součástí platformy .NET, která poskytuje rozsáhlou knihovnu tříd a nástrojů pro vývoj aplikací. Visual Studio je nejběžněji používané vývojové prostředí pro C#.

3. **Java**: Java je také široce používaný jazyk pro vývoj desktopových aplikací, zejména pro podnikové a síťové aplikace. Java poskytuje platformu nezávislou na architektuře a má rozsáhlou standardní knihovnu. Nejběžněji používané vývojové prostředí pro Java zahrnují IntelliJ IDEA a Eclipse.

4. **Python**: Python je relativně jednoduchý a čitelný jazyk, který je často používán pro vývoj skriptů, datové analýzy a webových aplikací. Python má také řadu knihoven pro vývoj desktopových aplikací, jako jsou PyQt nebo Tkinter. PyCharm a Jupyter jsou populární vývojová prostředí pro Python.

Každý z těchto jazyků má své výhody a nevýhody a je vhodný pro různé druhy projektů. Volba jazyka a nástrojů závisí na požadavcích projektu, cílové platformě, zkušenostech a preferencích vývojářů a dalších faktorech.

2.4.2 Vývojové prostředí: Visual Studio, IntelliJ IDEA, Eclipse, PyCharm atd.

Vývojové prostředí (IDE) jsou klíčovou součástí softwarového vývoje. Poskytují vývojářům nástroje pro psaní, ladění a testování kódu. Následující jsou mezi nejpopulárnějšími IDE pro vývoj desktopových aplikací:

1. **Visual Studio**: Je to výkonné a komplexní vývojové prostředí od společnosti Microsoft, které je široce používáno pro vývoj aplikací v C++, C# a Visual Basic. Visual Studio poskytuje mnoho nástrojů a funkcí, včetně editoru kódu, debuggeru, návrháře formulářů, profileru a mnoha dalších. [25]

2. **IntelliJ IDEA**: Toto IDE od společnosti JetBrains je velmi populární pro vývoj aplikací v Javě. Poskytuje mnoho pokročilých funkcí, jako je inteligentní doplňování kódu, refaktorizace, statická analýza kódu a integrace s různými nástroji pro správu verzí, jako je Git. [18]

3. **Eclipse**: Eclipse je další populární IDE pro vývoj v Javě, ačkoli podporuje i mnoho dalších jazyků pomocí pluginů. Eclipse je open-source a poskytuje mnoho nástrojů a funkcí, včetně editoru kódu, debuggeru, návrháře GUI a podpory pro vývoj pluginů. [26]

4. **PyCharm**: PyCharm je vývojové prostředí od JetBrains pro Python. Poskytuje mnoho funkcí pro efektivní vývoj v Pythonu, včetně inteligentního doplňování kódu, podpory pro webový vývoj a datovou analýzu, integrace s nástroji pro správu verzí a mnoha dalších. [27]

Každé IDE má své silné a slabé stránky, a volba mezi nimi často závisí na konkrétním jazyku, který se používá, na požadavcích projektu a na osobních preferencích vývojáře.

2.4.3 Frameworky: .NET, JavaFX, Qt, GTK+ atd.

Frameworky jsou nástroje, které poskytují strukturovanou základnu pro vývoj aplikací. Pomáhají vývojářům efektivněji vytvářet aplikace tím, že poskytují předdefinované

třídy a funkce pro běžné úkoly. Následující frameworky jsou často používány při vývoji desktopových aplikací:

1. **.NET**: .NET je platforma od Microsoftu, která podporuje několik jazyků, včetně C#, Visual Basic a F#. Poskytuje rozsáhlou knihovnu tříd pro různé úkoly, včetně vytváření uživatelských rozhraní, práce se sítí, přístupu k databázím a mnoho dalšího. .NET je také základem pro několik populárních frameworků pro vývoj desktopových aplikací, jako je Windows Forms a WPF (Windows Presentation Foundation). [28]

2. **JavaFX**: JavaFX je framework pro vývoj bohatých internetových aplikací (RIA) v Javě. Umožňuje vytvářet aplikace s moderními grafickými prvky a animacemi. JavaFX podporuje také CSS a jiné webové technologie, což usnadňuje vytváření atraktivních uživatelských rozhraní. [29]

3. **Qt**: Qt je multiplatformní framework pro vývoj aplikací v C++. Je široce používán pro vývoj GUI aplikací, ale také podporuje vývoj konzolových aplikací a serverů. Qt poskytuje širokou škálu nástrojů a funkcí, včetně návrháře formulářů, nástrojů pro internacionalizaci, přístupu k databázím a podpory pro síťové programování. [30]

4. **GTK+**: GTK+ je multiplatformní framework pro vývoj GUI aplikací v C. Je základem pro několik desktopových prostředí v Linuxu, včetně GNOME a Xfce. GTK+ poskytuje řadu nástrojů pro vytváření uživatelských rozhraní, včetně návrháře formulářů a knihovny widgetů. [31]

Tyto frameworky mají různé výhody a nevýhody a jsou vhodné pro různé druhy projektů. Volba mezi nimi závisí na požadavcích projektu, cílové platformě, zkušenostech a preferencích vývojářů a dalších faktorech.

3 SWIFT

Z důvodu výběru mobilní platformy iOS pro vývoje mobilní aplikace, která bude schopna pomocí vestavěné kamery načítat sady čísel je potřeba se seznámit s klíčovými vlastnosti programovacího jazyka, který se používá pro vývoj iOS aplikací.

Swift je moderní, bezpečný a výkonný programovací jazyk navržený společností Apple. Byl představen v roce 2014 jako nástupce jazyka Objective-C a stal se hlavním jazykem pro vývoj aplikací pro iOS, macOS, watchOS a tvOS. Swift je také open-source, což umožňuje jeho využití na různých platformách a vývojářských ekosystémech. [32]

3.1 Klíčové vlastnosti a výhody Swiftu

Swift je moderní, bezpečný a výkonný programovací jazyk navržený společností Apple, který přináší několik klíčových vlastností a výhod pro vývojáře:

1. **Čitelnost**: Swift má čistou a snadno čitelnou syntaxi, což usnadňuje psaní a údržbu kódu. To také zjednodušuje spolupráci ve vývojářských týmech, protože kód je snadnější pochopit a přečíst.

2. **Bezpečnost**: Jazyk byl navržen s důrazem na bezpečnost a odolnost proti chybám. Swift zavádí koncept nepovinných (optionals) pro řešení problému s nulovými hodnotami a má silnou typovou kontrolu, což pomáhá zabránit běžným chybám při programování.

3. **Výkon**: Swift je optimalizován pro výkon a rychlost, což z něj dělá ideální jazyk pro vývoj mobilních aplikací, které vyžadují rychlost a efektivitu. Swift je navržen s využitím moderních kompilátorů a technologií, což zajišťuje vysoký výkon aplikací.

4. **Interoperabilita**: Swift je zpětně kompatibilní s Objective-C, což umožňuje vývojářům snadno přecházet mezi oběma jazyky a přebírat stávající kódy. To umožňuje hladký přechod pro týmy, které přecházejí z Objective-C na Swift.

5. **Funkcionální a objektově orientovaný jazyk**: Swift podporuje jak funkcionální, tak i objektově orientované programovací koncepty. To umožňuje vývojářům kombinovat nejlepší z obou světů a vytvářet flexibilní a udržitelný kód.

6. **Open-source**: Swift je open-source jazyk, což znamená, že je volně dostupný pro vývojáře a komunitu. To umožňuje jeho využití na různých platformách a vývojářských ekosystémech a zároveň podporuje inovace a rychlý vývoj jazyka.

7. **Bohatý ekosystém a podpora společnosti Apple**: Swift je hlavním jazykem pro vývoj aplikací pro Apple platformy a má silnou podporu od společnosti Apple. To zahrnuje rozsáhlou dokumentaci, nástroje jako Xcode a Interface Builder, a integraci s Apple technologiemi a službami, jako je iCloud, Core Data, Core Graphics a další. Toto je klíčové pro vývojáře, kteří se zaměřují na vývoj aplikací pro iOS, macOS, watchOS a tvOS.

8. **Moderní prvky jazyka**: Swift zahrnuje některé moderní prvky jazyka, jako je správa paměti pomocí automatického počítání referencí (ARC), podpora pro closure, generika, rychlé a snadné práce s kolekcemi jako jsou pole a slovníky, a podpora pro funkcionální programování.

9. **Playgrounds**: Swift zahrnuje funkci zvanou Playgrounds v Xcode, což je interaktivní prostředí, které umožňuje vývojářům testovat kód Swiftu v reálném čase. To je vhodné pro experimentování, učení jazyka nebo rychlé prototypování.

Tyto klíčové vlastnosti a výhody činí Swift populárním a oblíbeným výběrem pro vývoj aplikací na platformách Apple, ať už se jedná o mobilní aplikace pro iOS, desktopové aplikace pro macOS, nebo aplikace pro watchOS a tvOS. Navíc, díky jeho open-source povaze a snadné čitelnosti, je Swift také dobrým jazykem pro začínající vývojáře, kteří se chtějí naučit programování.

3.2 Syntaxe Swiftu a základní koncepty

Swift je jazyk s jednoduchou a přehlednou syntaxí, která je snadno čitelná a umožňuje vývojářům psát čistý a kompaktní kód. Níže je představena základní syntaxe a klíčové koncepty jazyka Swift. Swift je case sensitive jazyk, a to znamená že proměnné isValid a isvalid jsou dvě různé proměnné.

3.2.1 Proměnné a konstanty

V Swiftu se proměnné definují klíčovým slovem var a konstanty pomocí let.

```
1 var promenna = "Hello, variable"
2 let konstanta = "Hello, constant"
```

3.2.2 Typy dat

Swift je silně typovaný jazyk a podporuje následující datové typy: Int, Double, Float, Bool, String, Array, Dictionary, Set, Optional.

```
1 var promenna_integer: Int = 10
2 var promenna_double: Double = 10.0
3 var promenna_boolean: Bool = true
```

```
var promenna_string: String = "Hello"
var promenna_array: [Int] = [1, 2, 3]
var promenna_dictionary: [String: Int] = ["one": 1, "two": 2]
var promenna_optionalString: String? = nil
```

3.2.3 Funkce

Funkce v Swiftu se definují pomocí klíčového slova func.

```
func greet(name: String) -> String {
  return "Hello, \(name)!"
}
```

3.2.4 Podmínky a cykly

Swift podporuje standardní podmínky a cykly, jako jsou if, switch, for-in, while, a repeat-while.

```
if score > 10 {
  print("You scored high!")
} else {
  print("Keep trying!")
}

for i in 1...5 {
  print(i)
}

while count < 10 {
  count += 1
}

repeat {
  print("This will be done at least once")
} while false
```

3.2.5 Třídy a struktury

V jazyce Swift jsou struktury (Structs) a třídy (Classes) podobné ve smyslu, že mohou mít vlastnosti a metody. Obě mohou mít konstruktory (tzv. init metody), které definují, jak se instance vytvoří. Ale mají také některé klíčové rozdíly:

1. **Reference vs. hodnotové typy**: Třídy jsou reference typy, zatímco struktury jsou hodnotové typy. To znamená, že při přiřazení instanci třídy do nové proměnné nebo jejímu předání funkci, obě reference ukazují na stejnou instanci - změna jedné ovlivní i druhou. Na druhou stranu, je-li přiřazena instanci struktury do nové proměnné nebo je předána funkci, vytvoří se kopie instance - změna jedné neovlivní druhou.

2. **Dědičnost**: Třídy v jazyce Swift podporují dědičnost, což znamená, že jedna třída může dědit vlastnosti a metody od jiné třídy. Struktury však dědičnost nepodporují.

3. **Deinicializace**: Třídy podporují deinicializátory, které umožňují uvolnit zdroje, než se instance třídy odstraní z paměti. Struktury však deinicializátory nepodporují.

4. **Mutability**: V případě struktur, pokud je instance vytvořena jako konstanta pomocí klíčového slova let, nelze měnit její vlastnosti. U tříd toto omezení neplatí.

```
1 class MyClass {
2   var property: Int = 0
3 }
4
5 struct MyStruct {
6   var property: Int = 0
7 }
```

3.2.6 Optionals

Swift má unikátní koncept nazývaný "optional", který umožňuje proměnným mít hodnotu nil, což znamená, že nemají žádnou hodnotu. To pomáhá předejít chybám při práci s hodnotami, které mohou být prázdné.

```
1 var optionalInt: Int? = nil
2 optionalInt = 10
```

Tyto jsou jen některé z klíčových konceptů a syntaxe Swiftu. Swift také podporuje pokročilé koncepty jako:

3.2.7 Generics

Generics jsou nástroje pro psaní flexibilního, opakovatelného kódu. Lze s nimi definovat funkce nebo typy, které pracují s jakýmkoli typem, přičemž skutečný typ použitý je určený, když je funkce nebo typ instancován.

```
1 func swapValues<T>(a: inout T, b: inout T) {
2   let temp = a
3   a = b
4   b = temp
5 }
```

3.2.8 Closures

Closures jsou samostatné bloky funkcionality, které lze předávat a používat v kódu. Closures v Swiftu jsou podobné lambda funkcím v jiných jazycích.

```
let numbers = [1, 2, 3, 4, 5]
let doubledNumbers = numbers.map { $0 * 2 }
```

3.2.9 Protokoly

Protokoly jsou způsob, jak definovat sadu požadavků, jako jsou metody nebo vlastnosti, které musí splňovat třída, struktura nebo výčtový typ.

```
protocol Identifiable {
  var id: String { get set }
}
```

3.2.10 Enums

Enum, nebo celým názvem "enumeration", je speciální datový typ, který umožňuje skupině souvisejících hodnot sdílet společný typ. Enumy jsou velmi výkonné a nabízejí více funkcí než většina jiných jazyků.

```
enum Direction {
  case north
  case south
  case east
  case west
}
```

Swift také podporuje "raw values"pro enumy, což jsou předdefinované hodnoty přiřazené jednotlivým hodnotám enumu. Tyto hodnoty mohou být například čísla nebo řetězce:

```
enum ASCIIControlCharacter: Character {
  case tab = "\t"
  case lineFeed = "\n"
  case carriageReturn = "\r"
}
```

Enumy v Swiftu také podporují metody, které mohou pracovat s hodnotami enumu:

```
enum Planet {
  case mercury, venus, earth

  func simpleDescription() -> String {
   switch self {
   case .earth:
     return "Mostly harmless"
   default:
    return "Not yet visited by humans"
   }
  }
}
```

Tyto pokročilé koncepty a funkce jazyka Swift umožňují vývojářům psát robustní, bezpečný a efektivní kód. Swift je navržen tak, aby byl snadno čitelný a pochopitelný,

zatímco poskytuje výkonné nástroje a konstrukce, které vývojářům umožňují psát kód, který je snadno udržovatelný a škálovatelný. [33]

3.3 UIKit a SwiftUI

UIKit a SwiftUI jsou dvě klíčové technologie pro vývoj uživatelských rozhraní v aplikacích pro iOS a další platformy od společnosti Apple. Zatímco UIKit byl dlouhou dobu standardem pro vývoj uživatelských rozhraní, SwiftUI, které bylo představeno v roce 2019, představuje nový, deklarativní přístup k vytváření uživatelských rozhraní.

3.3.1 Co je UIKit a jak funguje

UIKit je framework pro vývoj grafických, orientovaných na události rozhraní pro iOS a tvOS aplikace. UIKit poskytuje širokou škálu nástrojů a tříd pro vytváření a správu uživatelských rozhraní pro tyto platformy. Je napsán v Objective-C, ale je plně kompatibilní se Swift. Níže je struktura toho, jak UIKit funguje:

1. **Uživatelské rozhraní (UI) prvky**: UIKit poskytuje sadu předdefinovaných tříd pro vytváření prvků uživatelského rozhraní, jako jsou tlačítka, popisky, textová pole, posuvníky, přepínače a další. Tyto prvky jsou reprezentovány jako instance tříd, které dědí od třídy UIView.

2. **Správa rozvržení**: UIKit poskytuje nástroje pro uspořádání a uskutečnění prvků uživatelského rozhraní na obrazovce. To zahrnuje systém pro relativní pozicování prvků pomocí konstruktérů (Auto Layout) a podporu pro různé velikosti obrazovky a orientace.

3. **Interakce uživatele**: UIKit poskytuje rozhraní pro zacházení s interakcemi uživatele, jako jsou dotyky, tahy, švihy, stisky, otáčení a další. Třída UIResponder a její podtřídy poskytují metody pro reakci na tyto události.

4. **Navigace a řízení toku aplikace**: UIKit poskytuje třídy a struktury pro řízení toku mezi různými obrazovkami v aplikaci. Toto zahrnuje UINavigationController, který zajišťuje navigaci pomocí zásobníku obrazovek, a UITabBarController, který umožňuje přepínání mezi různými sekcemi aplikace.

5. **Animace a přechody**: UIKit obsahuje nástroje pro vytváření animovaných přechodů a interakcí. Tyto nástroje umožňují animovat vlastnosti prvků uživatelského rozhraní, vytvářet složité animace a vytvářet přizpůsobené přechody mezi obrazovkami.

6. **Podpora pro přizpůsobení**: UIKit umožňuje vývojářům přizpůsobit vzhled a chování standardních prvků uživatelského rozhraní, a to buď globálně pro celou aplikaci, nebo individuálně pro jednotlivé prvky.

7. **Podpora pro přístupnost**: UIKit poskytuje nástroje pro vytváření aplikací, které jsou přístupné uživatelům se zdravotním postižením. To zahrnuje podporu pro VoiceOver, zvětšení textu a další funkce.

Při práci s UIKit vývojáři obvykle začnou vytvořením instance třídy, jako je UIViewController, který spravuje jednu obrazovku aplikace. Poté mohou přidat prvky uživatelského rozhraní k tomuto kontroléru a nastavit jejich vlastnosti a chování. Nakonec vývojáři nastaví, jak aplikace reaguje na interakce uživatele s těmito prvky. [34]

3.3.2 Co je SwiftUI a jak funguje

SwiftUI je moderní, deklarativní framework pro vývoj uživatelských rozhraní od společnosti Apple. Byl představen v roce 2019 a je napsán v jazyce Swift. SwiftUI umožňuje vývojářům snadno vytvářet uživatelská rozhraní pro platformy iOS, macOS, watchOS a tvOS pomocí jednotného, konzistentního API. Níže je struktura toho, jak SwiftUI funguje:

1. **Deklarativní syntaxe**: Na rozdíl od imperativního přístupu používaného v UIKit, SwiftUI využívá deklarativní syntaxi. To znamená, že vývojáři definují, jak by mělo uživatelské rozhraní vypadat a jak se má chovat, a SwiftUI se postará o zbytek. Například, místo toho, aby bylo nutné explicitně nastavit stavy a přechody mezi nimi, vývojáři jednoduše definují, jak by měl vypadat každý stav, a SwiftUI automaticky řídí přechody.

2. **Komponentový přístup**: SwiftUI je založen na komponentovém přístupu k vývoji uživatelských rozhraní. Uživatelské rozhraní je sestaveno z menších, znovupoužitelných komponent (nazývaných "Views"), které lze kombinovat a skládat do složitějších rozhraní. Každý View je funkce svého vstupního stavu a nemění se po svém vytvoření.

3. **Data Flow**: SwiftUI používá koncepty jako @State, @Binding, @ObservedObject, @EnvironmentObject a další pro správu toku dat mezi různými částmi aplikace. To umožňuje vývojářům snadno synchronizovat stav mezi různými částmi uživatelského rozhraní a reagovat na změny stavu.

4. **Podpora pro všechny Apple platformy**: SwiftUI umožňuje vývojářům vytvářet aplikace pro všechny Apple platformy pomocí jednotného API. To znamená,

že stejný kód může být použit pro vytvoření uživatelských rozhraní pro iOS, macOS, watchOS a tvOS s minimálními úpravami.

5. **Live Previews**: S pomocí nástroje Xcode, SwiftUI poskytuje funkci "Live Previews", která umožňuje vývojářům okamžitě vidět, jak jejich kód ovlivní vzhled a chování uživatelského rozhraní. To výrazně zrychluje proces vývoje a zlepšuje zpětnou vazbu mezi vývojářem a jeho kódem.

6. **Animace a přechody**: SwiftUI poskytuje jednoduchý a srozumitelný způsob, jak přidávat animace a přechody do uživatelského rozhraní. Vývojáři mohou snadno animovat změny stavu nebo vytvářet složitější animace s minimálním kódem.

7. **Přístupnost a lokalizace**: SwiftUI automaticky poskytuje podporu pro přístupnost a lokalizaci. Umožňuje vývojářům snadno vytvářet aplikace, které jsou přístupné uživatelům se zdravotním postižením, a podporuje lokalizaci textu a dalších prvků uživatelského rozhraní pro různé jazyky a regiony.

8. **Dark Mode a další vlastnosti systému**: SwiftUI umožňuje snadnou integraci s funkcemi systému, jako je Dark Mode nebo Dynamic Type. Vývojáři mohou snadno přizpůsobit vzhled a chování svých aplikací na základě uživatelských preferencí a nastavení systému.

Celkově nabízí SwiftUI vývojářům jednoduchý, deklarativní způsob pro vytváření uživatelských rozhraní pro všechny Apple platformy. Je navržen tak, aby byl snadno použitelný, efektivní a umožňoval rychlý vývoj a ladění. [35]

3.3.3 Klíčové rozdíly mezi UIKit a SwiftUI

UIKit a SwiftUI jsou oba frameworky pro vývoj uživatelských rozhraní na Apple zařízeních, ale mají několik klíčových rozdílů, které ovlivňují způsob, jakým s nimi vývojáři pracují:

1. **Deklarativní vs. imperativní přístup**: SwiftUI využívá deklarativní přístup k vývoji uživatelských rozhraní, což znamená, že vývojáři definují, jak by mělo uživatelské rozhraní vypadat a jak se má chovat, a SwiftUI se postará o zbytek. Naproti tomu UIKit využívá imperativní přístup, kde vývojáři musí explicitně řídit stav a chování uživatelského rozhraní.

2. **Syntaxe a jazyk**: SwiftUI je napsán v jazyce Swift a využívá jeho moderní syntaxi a funkce. UIKit je napsán v Objective-C, ale je plně kompatibilní se

Swift. Avšak některé starší koncepty z Objective-C mohou být přítomné při práci s UIKit.

3. **Platforma a kompatibilita**: SwiftUI je dostupný pro iOS 13 a novější verze, zatímco UIKit je dostupný i pro starší verze iOS. SwiftUI také umožňuje jednodušší multiplatformní vývoj pro iOS, macOS, watchOS a tvOS s jednotným API, zatímco pro UIKit by bylo nutné použít AppKit pro macOS a další frameworky pro ostatní platformy.

4. **Komponentový přístup a modularity**: SwiftUI podporuje komponentový přístup k vývoji uživatelských rozhraní, což usnadňuje znovupoužití a skládání kódu. S UIKit je znovupoužití kódu také možné, ale může být složitější a méně přirozené.

5. **Data Flow**: SwiftUI zavádí nové koncepty pro správu toku dat mezi různými částmi aplikace, jako jsou @State, @Binding, @ObservedObject a @EnvironmentObject. UIKit má svůj vlastní způsob správy toku dat pomocí delegate pattern, KVO (Key-Value Observing) a NotificationCenter.

6. **Animace a přechody**: SwiftUI poskytuje jednoduchý a srozumitelný způsob, jak přidávat animace a přechody do uživatelského rozhraní. S UIKit je také možné vytvářet animace a přechody, ale může být složitější a zahrnovat více kódu.

7. **Live Previews**: SwiftUI nabízí funkci "Live Previews"ve spojení s Xcode, která umožňuje vývojářům okamžitě vidět, jak jejich kód ovlivňuje vzhled a chování uživatelského rozhraní. To zrychluje proces vývoje a zlepšuje zpětnou vazbu mezi vývojářem a jeho kódem. UIKit tuto funkci nepodporuje přímo, ale je možné použít nástroje jako Interface Builder pro vizuální návrh uživatelského rozhraní.

8. **Přechod mezi frameworky**: SwiftUI poskytuje možnost integrace s existujícím UIKit kódem, což umožňuje plynulý přechod pro aplikace, které již používají UIKit. To znamená, že vývojáři mohou začít používat SwiftUI v jejich existujících projektech bez nutnosti kompletního přepsání kódu.

9. **Přístupnost a lokalizace**: Oba frameworky poskytují podporu pro přístupnost a lokalizaci, ale SwiftUI to často automatizuje a zjednodušuje.

10. **Dark Mode a další vlastnosti systému**: SwiftUI umožňuje snadnou integraci s funkcemi systému, jako je Dark Mode nebo Dynamic Type. S UIKit je možné tyto funkce také podporovat, ale může to vyžadovat více kódu a práce.

Celkově platí, že UIKit je osvědčený a robustní framework, který je dostatečně flexibilní na to, aby vývojáři mohli vytvořit téměř jakékoliv uživatelské rozhraní, které

si představí. Na druhou stranu, SwiftUI nabízí modernější a čistější přístup k vývoji uživatelských rozhraní, který může usnadnit a zrychlit proces vývoje, ale může být omezenější v některých pokročilých případech užití.

3.3.4 Přechod z UIKit na SwiftUI: Kdy a proč to dělat

Přechod z UIKit na SwiftUI je velké rozhodnutí, které závisí na několika faktorech. Zde je několik důvodů, proč by vývojáři mohli zvážit tento přechod, a některé aspekty, které je třeba vzít v úvahu:

1. **Aktualizace technologického stacku**: SwiftUI je moderní a inovativní framework, který přináší řadu výhod, jako je deklarativní syntaxe, live previews, jednoduché animace a přechody, a snadná podpora pro dark mode a další systémové funkce. Pokud chce vývojář využít tyto výhody a udržet svůj kód aktuální s nejnovějšími technologiemi Apple, může být přechod na SwiftUI dobrým krokem.
2. **Multiplatformní vývoj**: SwiftUI umožňuje jednodušší multiplatformní vývoj pro iOS, macOS, watchOS a tvOS s jednotným API. Pokud je v plánu vytvářet aplikace pro více než jednu z těchto platforem, může být SwiftUI efektivnější volbou.
3. **Zjednodušení kódu a udržitelnost**: Díky deklarativní syntaxi a komponentovému přístupu může SwiftUI zjednodušit strukturu kódu a usnadnit jeho čtení a údržbu. To může zlepšit efektivitu vývoje a usnadnit práci novým členům týmu.
4. **Rychlejší a jednodušší vývoj**: S funkcemi, jako jsou live previews a automatické rozložení, může SwiftUI zrychlit proces vývoje a snížit počet chyb tím, že umožní vývojářům okamžitě vidět, jak se jejich kód projeví na uživatelském rozhraní.

Nicméně, přechod na SwiftUI také přináší některé výzvy a omezení:

1. **Kompatibilita**: SwiftUI vyžaduje iOS 13 nebo novější, takže pokud aplikace potřebuje podporovat starší verze iOS, vývoj může být omezen na použití UIKit.
2. **Pokročilé přizpůsobení a složité uživatelské rozhraní**: UIKit je velmi flexibilní a umožňuje vývojářům vytvořit téměř jakékoliv uživatelské rozhraní, které si představí. S SwiftUI může být některé pokročilé přizpůsobení nebo složité uživatelské rozhraní obtížnější nebo méně přirozené.
3. **Dostupnost dokumentace a zdrojů**: Jelikož je UIKit starší a více zavedený, existuje pro něj více zdrojů, tutoriálů a odpovědí na otázky. Se SwiftUI může být některé informace těžší najít, i když se situace rychle zlepšuje.

4. **Přechod a doba učení**: Přechod na nový framework vyžaduje čas na učení a experimentování. To může být zvláště významné pro velké týmy nebo pro projekty s náročnými časovými rámci.

Pokud se vývojář nebo tým rozhodne přejít na SwiftUI, doporučuje se plánovat přechod postupně. SwiftUI umožňuje integraci s existujícím UIKit kódem, takže lze začít přidávat SwiftUI do svého projektu postupně a učit se, jak s ním pracovat, aniž by bylo potřeba přepsat celou aplikaci najednou.

3.4 Práce s frameworky v Swiftu

Práce s frameworky v Swiftu umožňuje vývojářům přistupovat k funkcím a třídám poskytovaným Apple nebo třetími stranami. Frameworky zjednodušují vývoj aplikací tím, že poskytují předem vytvořené a optimalizované součásti kódu, které řeší běžné úkoly. Následující jsou některé z běžných frameworků, které lze použít v projektech:

1. **UIKit / SwiftUI**: Tyto frameworky se používají pro vytváření uživatelského rozhraní a interakce s uživatelem. UIKit je tradiční způsob vytváření uživatelských rozhraní pro iOS, zatímco SwiftUI je novější deklarativní způsob vytváření uživatelských rozhraní.

2. **Foundation**: Foundation poskytuje základní datové typy, kolekce a operace pro práci s textem, daty a časem. Tento framework také obsahuje funkce pro práci se soubory a adresáři, komunikaci se sítí a správu vláken.

3. **AVFoundation**: AVFoundation je framework pro práci s audiovizuálním obsahem. Umožňuje vývojářům nahrávat, přehrávat, analyzovat a upravovat zvukové a video soubory.

4. **Core Data**: Core Data je framework pro správu a ukládání datových modelů v iOS aplikacích. Umožňuje vývojářům vytvářet, číst, aktualizovat a mazat objekty a jejich vztahy s minimálním kódem a snadno integrovat tyto funkce do uživatelského rozhraní.

5. **Core Graphics**: Core Graphics je framework pro kreslení 2D grafiky a manipulaci s obrázky. Umožňuje vývojářům vytvářet a upravovat obrázky, kreslit tvary a text, pracovat s vrstvami a transformacemi.

6. **Core Location**: Core Location je framework pro získávání a sledování informací o poloze a orientaci zařízení. Umožňuje vývojářům získat aktuální polohu uživatele, sledovat změny polohy a provádět geofencing.

7. **Vision**: Vision je framework pro analýzu obrázků a videí. Umožňuje vývojářům detekovat a rozpoznávat tváře, text, bar kódy a další objekty na obrázcích. Tento framework také podporuje sledování objektů ve videu a může být použit s Core ML pro přizpůsobené modely strojového učení.

8. **Charts**: Charts je populární open-source knihovna pro vytváření interaktivních grafů a diagramů. Je kompatibilní s iOS, macOS, tvOS a je napsána ve Swiftu.

Každý framework má své vlastní API a způsob práce. Proto je důležité pochopit, jak fungují a jak je v kódu používat efektivně. Studium dokumentace a návodů pro každý framework, který bude použit, je dobrým způsobem, jak se seznámit s těmito funkcemi frameworku.

3.4.1 Popis a přehled frameworku Vision: jeho použití a výhody

Vision je framework, který poskytuje vysokou úroveň rozhraní pro provádění rozpoznávání obrazů a detekce na obrázcích a videích. Byl vytvořen společností Apple a je součástí iOS, macOS, tvOS a watchOS. Vision poskytuje různé funkce pro analýzu obrázků a videí, včetně detekce tváří, rozpoznávání textu, detekce objektů, sledování objektů a dalších.

Struktura Vision:

1. **Detekce objektů a tváří**: Vision může detekovat tváře, oči, ústa, nosy a další rysy na fotografiích a ve videích. Také může rozpoznat a sledovat objekty.

2. **Rozpoznávání textu**: Vision může rozpoznávat a číst text na obrázcích. To je užitečné pro aplikace, které chtějí číst a interpretovat text z fotografie, například skenování dokumentů nebo rozpoznávání registračních značek automobilů.

3. **Detekce obrazců a barev**: Vision může detekovat a rozpoznávat obrazce a barvy. To je užitečné pro aplikace, které chtějí rozpoznávat určité symboly nebo barvy na obrázcích.

4. **Integrace s Core ML**: Vision může být použit společně s Core ML (framework od Apple, který umožňuje vývojářům integrovat předem natrénované modely strojového učení do aplikací) pro přizpůsobené modely strojového učení. To znamená, že lze vytvořit vlastní model pro rozpoznávání specifických objektů nebo obrazců a použít jej v kombinaci s Vision.

Výhody použití Vision:

1. **Vysoká přesnost a rychlost**: Vision poskytuje vysokou přesnost a rychlost při rozpoznávání a detekci obrázků a videí.

2. **Snadná integrace**: Vision je snadno integrovatelný do aplikací a poskytuje jednoduché API pro provádění složitých operací rozpoznávání obrazů.

3. **Široká škála funkcí**: Vision nabízí širokou škálu funkcí pro rozpoznávání obrazů a videí, což z něj dělá silný nástroj pro vývojáře.

4. **Bezpečnost a soukromí**: Protože veškeré zpracování probíhá na zařízení, data uživatelů zůstávají soukromá a bezpečná.

5. **Kompatibilita**: Vision je kompatibilní s většinou zařízení Apple, včetně iPhone, iPad, Mac, Apple Watch a Apple TV. To znamená, že lze vytvářet aplikace, které využívají schopnosti rozpoznávání obrazů a videí napříč různými platformami.

6. **Podpora pro reálný čas**: Vision podporuje zpracování obrazu a videa v reálném čase, což je důležité pro aplikace, jako jsou hry nebo rozšířená realita (AR).

Při práci s frameworkem Vision je důležité pochopit, jak efektivně využít jeho funkcí a jak ho integrovat do vyvíjené aplikace. Seznámené se s dokumentací a návody, lze začít využívat jeho plný potenciál v konkrétním projektu. [36]

3.4.2 Popis a přehled frameworku AVFoundation: jeho použití a výhody

AVFoundation je jedním z klíčových frameworků pro práci s audiovizuálním obsahem na platformách Apple, jako je iOS, macOS, watchOS a tvOS. Tento framework poskytuje sadu nástrojů pro manipulaci s audiem a videem, včetně nahrávání, přehrávání, editace a konverze mezi různými formáty.

Struktura AVFoundation:

1. **Práce s audiem**: AVFoundation umožňuje nahrávání a přehrávání zvuku, stejně jako přímé zpracování audio dat. Toto je užitečné pro aplikace, které potřebují nahrávat hlas, přehrávat hudbu nebo manipulovat se zvukem.

2. **Práce s videem**: AVFoundation poskytuje nástroje pro nahrávání videa, přehrávání videa a zpracování video dat. To může zahrnovat změnu rychlosti přehrávání, přidání efektů nebo dokonce vytváření vlastního video přehrávače.

3. **Editace a konverze**: AVFoundation obsahuje nástroje pro editaci audio a video dat, včetně stříhání, sloučení a přidání efektů. Framework také umožňuje konvertovat data mezi různými audio a video formáty.

4. **Metadata a analýza**: AVFoundation umožňuje číst a psát metadata do audio a video souborů. Tento framework také umožňuje analyzovat obsah videa, včetně detekce obličeje a sledování pohybu.

Výhody použití AVFoundation:

1. **Flexibilita a kontrola**: AVFoundation poskytuje vysokou úroveň kontroly nad audio a video daty. Lze přizpůsobit, jak se data nahrávají, přehrávají a zpracovávají, což je ideální pro aplikace, které potřebují specifické nebo složité audiovizuální funkce.

2. **Vysoká kvalita a výkon**: AVFoundation je navržen tak, aby poskytoval vysokou kvalitu zvuku a videa, zatímco udržuje vysokou účinnost a výkon.

3. **Kompatibilita**: AVFoundation podporuje širokou škálu audio a video formátů, což znamená, že je kompatibilní s téměř jakýmkoli typem audio nebo video dat.

4. **Integrace s ostatními frameworky Apple**: AVFoundation se snadno integruje s ostatními frameworky Apple, jako je Core Audio, Core Video a Core Media.

   Při práci s frameworkem AVFoundation je důležité pochopit, jak efektivně využít jeho funkcí a jak ho integrovat do aplikace.

5. **Multimediální možnosti**: AVFoundation nabízí komplexní řešení pro práci s audiem a videem. Lze s ním vytvářet aplikace s pokročilými multimediálními funkcemi, jako je nahrávání, editace, přehrávání a analýza audiovizuálního obsahu.

6. **Podpora pro reálný čas**: AVFoundation podporuje zpracování audia a videa v reálném čase, což je důležité pro aplikace, jako jsou hry nebo rozšířená realita (AR).

7. **Bezpečnost a soukromí**: Jakožto framework od Apple, AVFoundation poskytuje silnou podporu pro bezpečnost a soukromí uživatelů. Veškeré zpracování probíhá na zařízení, což znamená, že data uživatelů zůstávají soukromá a bezpečná.

8. **Podpora pro hardware**: AVFoundation dokáže efektivně využívat hardware zařízení pro zpracování audia a videa. To znamená, že může přizpůsobit své chování v závislosti na dostupném hardware, což může vést k lepšímu výkonu a efektivitě.

Vývojáři by měli vědět, jak efektivně využít AVFoundation pro vytváření bohatých a interaktivních multimediálních aplikací. S tímto frameworkem mohou vytvářet aplikace,

které mohou nahrávat, editovat, přehrávat a analyzovat audiovizuální obsah, což přináší nové možnosti pro vytváření inovativních a uživatelsky přívětivých aplikací. [37]

3.4.3 Popis a přehled frameworku Charts: jeho použití a výhody

Charts je populární knihovna pro vytváření grafů a diagramů v aplikacích pro iOS a macOS. Poskytuje širokou škálu možností pro vizualizaci dat a je kompatibilní jak s UIKit, tak s SwiftUI.

Struktura frameworku Charts:

1. **Typy grafů**: Framework Charts podporuje mnoho různých typů grafů, včetně čárových, sloupcových, koláčových, bublinových a dalších. Každý typ grafu má vlastní sadu vlastností a lze je přizpůsobit podle potřeb aplikace.

2. **Vizualizace dat**: Charts poskytuje nástroje pro vizualizaci a zobrazování dat. To zahrnuje možnost zobrazování popisků, os, matic, legend a dalších prvků grafu.

3. **Interaktivita**: S frameworkem Charts lze vytvářet interaktivní grafy, které uživatelé mohou prozkoumávat a s kterými mohou interagovat. To zahrnuje možnosti jako zoom, posun, výběr bodů a další.

Výhody použití frameworku Charts:

1. **Flexibilita a přizpůsobitelnost**: Framework Charts je velmi flexibilní a lze ho přizpůsobit podle potřeb aplikace. Lze upravit vzhled a chování grafů, aby odpovídaly konkrétnímu designu a požadavkům na funkčnost.

2. **Podpora pro různé typy dat**: Charts může pracovat s různými typy dat, včetně číselných, časových a kategoriálních dat. To znamená, že může vytvářet vytvářet širokou škálu grafů pro různé účely.

3. **Integrace s UIKit a SwiftUI**: Charts je kompatibilní jak s UIKit, tak s SwiftUI, což znamená, že ho lze používat v jakémkoli typu aplikace pro iOS nebo macOS.

4. **Podpora pro přístupnost**: Charts obsahuje funkce pro přístupnost, což znamená, že grafy a diagramy mohou být přístupné i pro uživatele s omezeným viděním nebo jinými omezeními.

Při práci s frameworkem Charts je důležité pochopit, jak efektivně využít jeho funkcí a jak ho integrovat do vyvíjené aplikace.

5. **Snadná implementace**: Framework Charts je navržen tak, aby byl snadno implementovatelný do konkrétního projektu. To znamená, že s ním lze rychle a efektivně vytvářet grafy a vizualizace dat pro aplikaci.

6. **Široká komunita a podpora**: Charts má velkou a aktivní komunitu vývojářů, což znamená, že začínající vývojář má přístup k širokému spektru zdrojů, návodů a podpory.

7. **Výkon a optimalizace**: Charts je optimalizován pro vysoký výkon a nízkou spotřebu zdrojů. To znamená, že grafy budou rychlé a plynulé, i když zobrazují velké množství dat.

Použitím frameworku Charts mohou vývojáři snadno vytvářet sofistikované a interaktivní vizualizace dat pro své aplikace. Díky jeho flexibilitě, přizpůsobitelnosti a podpoře pro různé typy dat je tento framework ideální volbou pro prezentaci informací uživatelům v atraktivní a uživatelsky přívětivé formě. [38]

II. PRAKTICKÁ ČÁST

4 NÁVRH APLIKACE

V této kapitole je prezentován konkrétní návrh aplikace využívající technologii optického rozpoznávání znaků (OCR). Aplikace byla vyvinuta ve frameworku SwiftUI společnosti Apple. Vycházející z teoretických poznatků a technologických dovedností získaných v předchozích kapitolách, je cílem navrhnout funkční a efektivní řešení, které splňuje praktické požadavky a zároveň odpovídá současným trendům v oblasti mobilních aplikací.

Nejprve jsou definovány funkční a nefunkční požadavky, které představují klíčová očekávání a cíle návrhu. Jsou zde rozebírány specifické funkce, které by měla aplikace poskytnout.

Dalším bodem návrhu je popis jednotlivých obrazovek aplikace, které by měly uživatelům poskytnout intuitivní a uživatelsky přívětivé rozhraní pro interakci s aplikací.

Poslední, ale rozhodně ne nevýznamnou částí návrhu je architektura aplikace. Tato část se zabývá strukturou a uspořádáním aplikace na úrovni kódu, definováním hlavních komponent, jejich funkcí a vztahů.

Celkově je tento návrh aplikace považován za základ pro další fázi vývoje, která je detailněji popsána v následujících kapitolách.

4.1 Funkční požadavky

Funkční požadavky definují specifické funkce, které musí systém nebo aplikace splnit, aby byl splněn její účel nebo cíl. Jsou to konkrétní činnosti, které může systém provést, nebo operace, které musí být aplikace schopna vykonat.

Například, v případě vývoje mobilní aplikace, funkční požadavky by mohly zahrnovat schopnost přihlášení a ověření uživatele, možnost sdílet obsah, schopnost zobrazit určité informace atd.

Funkční požadavky jsou často specifikovány v detailu a je důležité je dobře definovat, aby bylo možné úspěšně dokončit vývoj systému nebo aplikace. Tato specifikace také pomáhá při testování, protože může být použita jako základ pro vytvoření testovacích scénářů. Základní funkční požadavky jsou znázorněny na obrázku 4.1.

Tabulka 4.1 Základní funkční požadavky

ID	Požadavek
FP1	**OCR načítání čísel**: Aplikace musí být schopna načítat sady čísel pomocí vestavěné kamery zařízení.
FP2	**Oprava chyb**: Aplikace by měla umožnit uživatelům korigovat načtená data v případě, že OCR neposkytne přesné výsledky.
FP3	**Sdílení dat**: Aplikace musí umožnit sdílení vytvořených CSV souborů pomocí standardních sdílení v iPhone.
FP4	**Statistická analýza a vizualizace dat**: Aplikace musí být schopna provést základní statistickou analýzu načtených dat a prezentovat výsledky ve formě grafů.

4.2 Nefunkční požadavky

Nefunkční požadavky, někdy také označované jako kvalitativní požadavky, se týkají vlastností systému nebo aplikace, které nejsou přímo spojeny s konkrétními funkcemi, které systém či aplikace vykonává.

Nefunkční požadavky určují, jak by systém měl fungovat, a to v různých aspektech jako jsou výkon, bezpečnost, spolehlivost, použitelnost, kompatibilita a další. Mohou například specifikovat, jak rychlá by měla aplikace být, jak bezpečně by měla zacházet s uživatelskými daty, jak stabilní by měla být, jaké platformy by měla podporovat, jak by měla vypadat a jak snadno by měla být ovladatelná.

Zatímco funkční požadavky jsou často vymezeny jako “co“ systém dělá, nefunkční požadavky definují “jak“ by systém měl tyto funkce vykonávat. Jedná se o klíčový aspekt vývoje, který může mít velký vliv na celkovou kvalitu a úspěšnost systému nebo aplikace. Základní nefunkční požadavky jsou znázorněny na obrázku 4.2.

Tabulka 4.2 Základní nefunkční požadavky

ID	Požadavek
NP1	**Výkon**: Aplikace by měla být schopna rychle a efektivně zpracovat velké množství dat bez významného ovlivnění výkonu zařízení.
NP2	**Kompatibilita**: Aplikace by měla být kompatibilní s nejnovějšími verzemi iOS.
NP3	**Uživatelské rozhraní**: Uživatelské rozhraní by mělo být intuitivní a snadno ovladatelné.
NP4	**Spolehlivost**: Aplikace by měla být stabilní a spolehlivá při každém použití, a to i při zpracování velkých objemů dat.

4.3 Popis obrazovek

Aplikace se bude skládat ze tří obrazovek. Drátěné modely obrazovek byly vytvořené pomoci nástroje Balsamiq Wireframes.

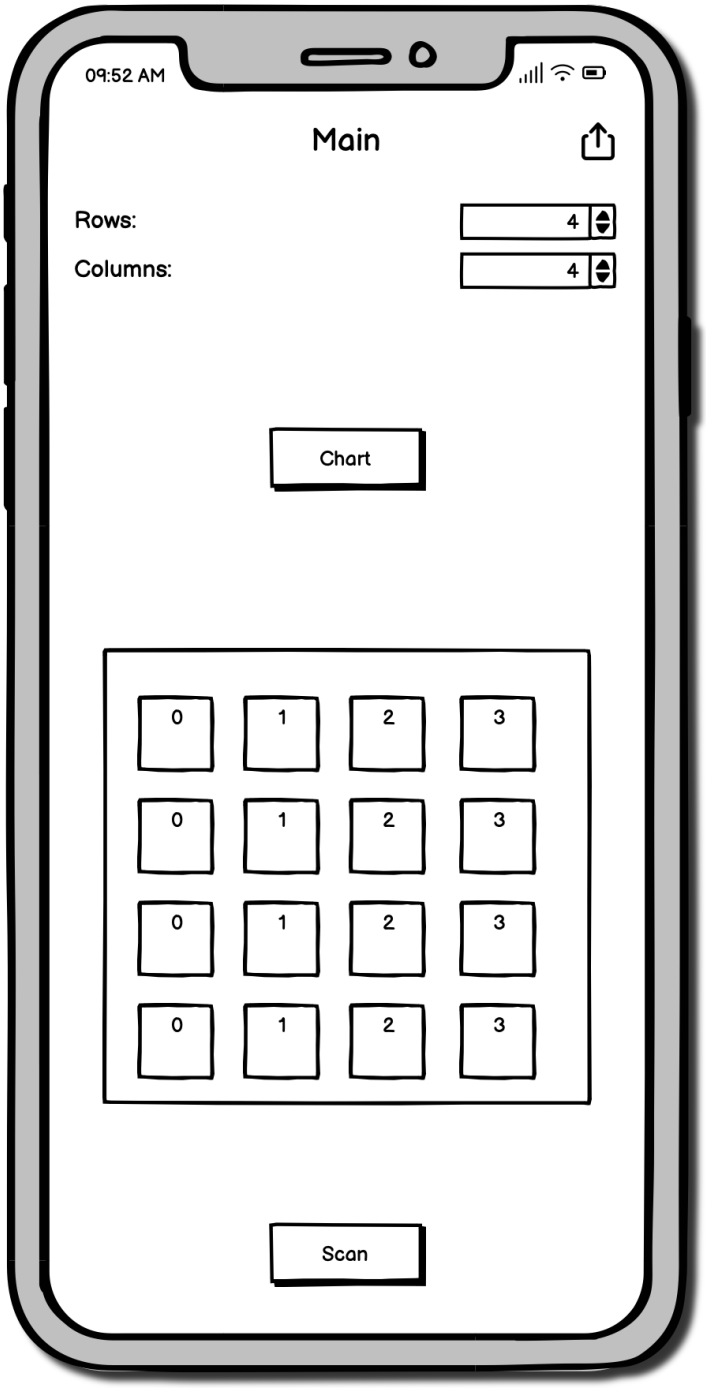
4.3.1 Hlavní obrazovka

Hlavní obrazovka má název „Main“ a je rozdělena na několik částí. Na vrchu je komponenta, která je určena k nastavení počtu sloupců a řádku, které si uživatel bude chtít naskenovat. Pod touto komponentou je navigační odkaz na obrazovku „Chart“, která obsahuje zobrazení dat ve formě grafu. Toto tlačítko je označeno ikonou grafu a slovem „Chart“.

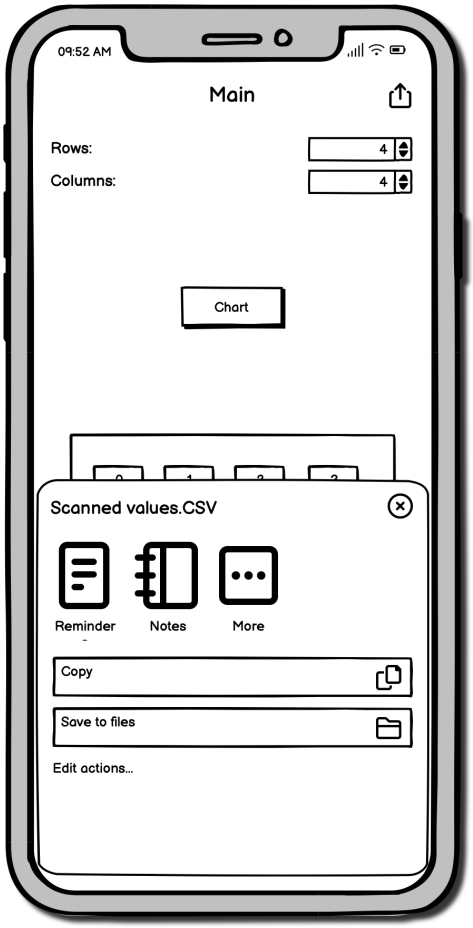
Další část obrazovky zobrazuje pole s naskenovanými hodnoty s nazvaném „Scanned values“. Tyto hodnoty jsou zobrazeny v tabulce, kde každá buňka obsahuje textové pole, které zobrazuje jednu hodnotu. Uživatel může tyto hodnoty upravovat. Pokud je buňka vybraná, její okraj se zbarví do defaultní zvýrazňující barvy na iOS (modré).

Na spodní části obrazovky je nástrojová lišta s tlačítkem „Scan“, která slouží k otevřeni modální obrazovky s názvem „Scanner“, která umožňuje uživateli skenovat čísla.

Na pravé straně navigační lišty je tlačítko pro sdílení, které je aktivováno, pokud jsou k dispozici data k exportu do CSV (tabulka obsahuje naskenovaná data). Po stisknutí tohoto tlačítka se vytvoří CSV, do něj se vloží načtená data a otevře se menu pro sdílení s možnostmi sdílení CSV souboru. Na obrázku 4.1 je znázorněn drátěný model hlavni obrazovky. Obrázek 4.2 znázorňuje drátěný model hlavní obrazovky (s otevřeným menu pro sdílení CSV).



Obrázek 4.1 Drátěný model hlavní obrazovky



Obrázek 4.2 Drátěný model hlavní obrazovky (s otevřeným menu pro sdílení CSV)

4.3.2 Obrazovka pro skenovaní dat

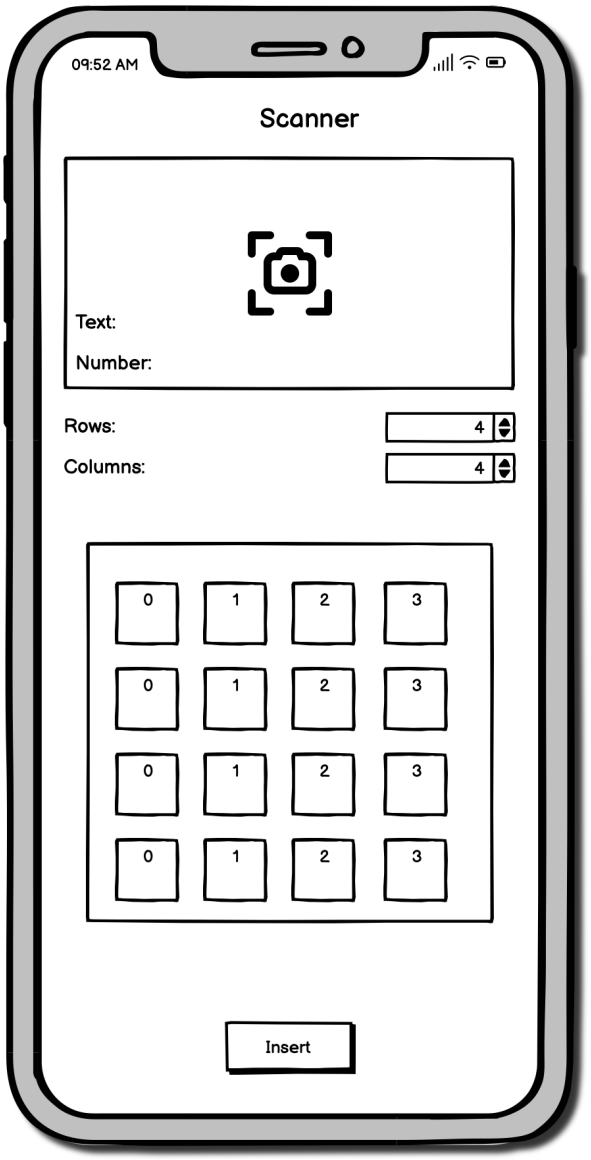
Hlavní část obrazovky je rozdělena na tři sekcí. Nahoře je pole, které obsahuje komponentu, umožňující skenování pomoci vestavěné kamery. Tato komponenta zobrazuje obraz, který je právě načítaný z kamery. Výsledky skenovaní jsou zobrazeny v rohu této komponenty, ve dvou textových polích. První dvojice zobrazuje skenovaný text a druhá dvojice zobrazuje skenované číslo, pokud naskenovaný text lze transformovat na číslo.

Druhá sekce obrazovky obsahuje komponentu, která je určena k manipulaci množství sloupců a řádku, které uživatel bude chtít naskenovat, stejně jak na hlavni obrazovce.

Třetí sekce obrazovky zobrazuje naskenované hodnoty v tabulce, kde každá buňka obsahuje tlačítko, které zobrazuje jednu hodnotu. Uživatel může vybrat buňku kliknutím na ni. Vybraná buňka je označena okrajem v modré barvě. Vizuálně tato sekce vypadá stejně jak tabulka na hlavni obrazovce.

Na spodní části obrazovky je nástrojová lišta s tlačítkem „Insert“, která slouží pro vkládání skenované hodnoty do vybrané buňky. Po vložení se výběr automaticky posune

na další buňku. Na obrázku 4.3 lze vidět drátěný model obrazovky pro skenování dat.



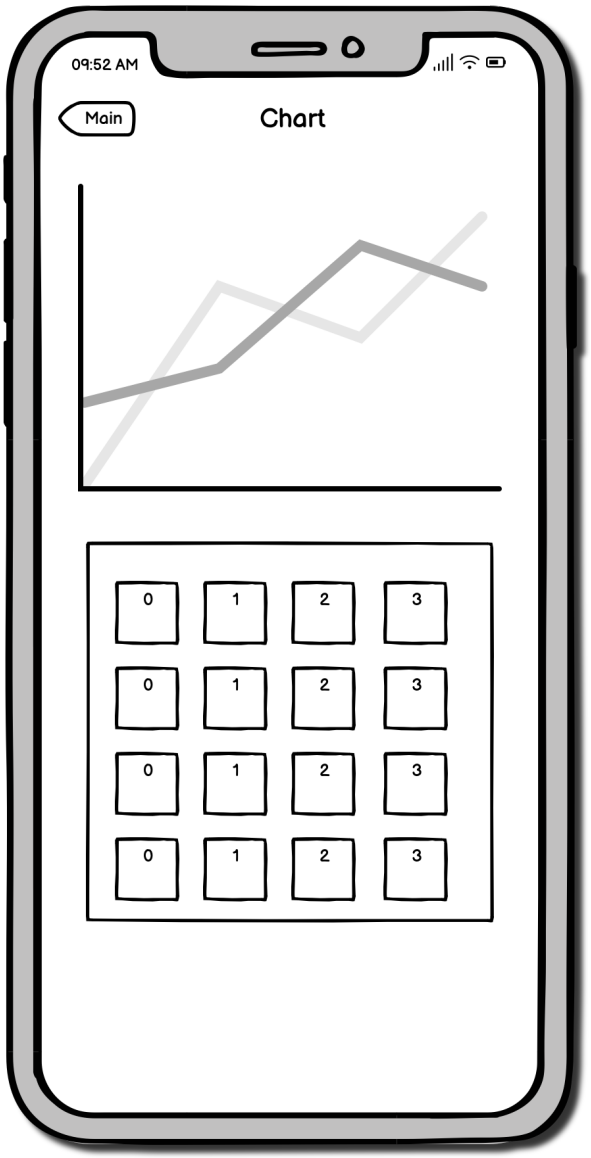
Obrázek 4.3 Drátěný model obrazovky pro skenovaní data

4.3.3 Obrazovka se zobrazením naskenovaných dat ve formě grafu

Hlavní oblast obrazovky je rozdělena do dvou částí. V horní části je graf, který je vytvořen pomocí nativní nově prezentované komponenty pro zobrazeni grafu ve SwiftUI. Vybraný styl grafu je spojnicový. Každý sloupec dat je reprezentován jako samostatná linie v grafu. Každý bod na linii odpovídá buňce dat v daném sloupci. Graf obsahuje legendu, která je umístěna na spodní straně grafu. Graf je ohraničen zaobleným obdélníkem s tenkým okrajem.

Druhá část obrazovky obsahuje tabulku z hlavní obrazovky, takže uživatel má v případě potřeby možnost ručně editovat data a ihned vidět změnu grafu. Na obrázku

4.4 je znázorněn drátěný model obrazovky s grafem.



Obrázek 4.4 Drátěný model obrazovky s grafem

4.4 Architektura aplikace

Aplikace má tři hlavní obrazovky, které tvoří její uživatelské rozhraní:

1. **ContentView**: Toto je hlavní obrazovka aplikace. Zobrazuje tabulku s daty, která lze upravit. Má také tlačítka pro přechod na obrazovku **ChartView** a pro otevření skenovací funkce.

2. **ScannerSheetView**: Tato obrazovka je otevřena z **ContentView**. Umožňuje uživatelům skenovat text a čísla pomocí OCR a vložit skenované hodnoty do tabulky na **ContentView**.

3. **ChartView**: Tato obrazovka zobrazuje graf na základě dat z **ContentView**. Může také zobrazovat další obsah, který je definován při inicializaci **ChartView**.

Tyto tři obrazovky jsou navzájem propojeny následovně:

- Z **ContentView** může uživatel přejít na **ChartView** pomocí tlačítka "Chart". Uživatel také může otevřít **ScannerSheetView** kliknutím na tlačítko "Scan"na nástrojové liště.
- Ze **ScannerSheetView** může uživatel vložit skenované hodnoty do tabulky na **ContentView**. Po dokončení skenování může uživatel zavřít **ScannerSheetView** a vrátit se zpět na **ContentView**.
- **ChartView** je obrazovka pro prohlížení, takže z ní není žádný přímý přechod na jiné obrazovky. Uživatel se může vrátit na **ContentView** pomocí navigační lišty.

Všechny tři obrazovky sdílejí stejný model dat **ContentViewModel**. Tento model dat je použit pro ukládání a manipulaci s daty v tabulce na **ContentView**. Změny v tomto modelu dat jsou reflektovány na všech třech obrazovkách.

Návrhový model aplikace je MVVM (Model-View-ViewModel), který je často používán ve SwiftUI aplikacích. **ContentViewModel** je ViewModel, **ContentView**, **ScannerSheetView** a **ChartView** jsou Views a data v ContentViewModel jsou Model.

5 VÝVOJ APLIKACE

5.1 Pomocné metody

Pro zjednodušení práce na vývoji aplikace byli přidané některé rozšířeni existujících tříd. Rozšíření (Extensions) v jazyce Swift jsou nástroje, které umožňují přidávat nové funkce k existujícím třídám, strukturám, výčtovým typům nebo protokolům. Toto je velmi mocná vlastnost Swiftu, která zvyšuje jeho flexibilitu a zjednodušuje jeho použití. Rozšíření mohou přidávat:

1. Vypočítané vlastnosti (Computed properties)
2. Metody instancí
3. Metody typů
4. Pravidla pro indexování (Subscripts)
5. Vnořené typy
6. Konformitu k protokolu

Přestože rozšíření mohou přidávat nové funkce, nemohou přepisovat existující funkce.

5.1.1 CGImageExtension

K třídě CGImage prostřednictvím rozšíření byla přidána metoda cropTo(rect: CGRect). Tato metoda umožňuje oříznout obrázek (CGImage) na určitý obdélníkový výřez (CGRect).

```
1  extension CGImage {
2    func cropTo(rect: CGRect) -> CGImage? {
3     let imageWidth = CGFloat(self.width)
4     let imageHeight = CGFloat(self.height)
5
6     let croppedImage = self.cropping(to: CGRect(
7       x: rect.origin.x * imageWidth,
8       y: rect.origin.y * imageHeight,
9       width: rect.width * imageWidth,
10      height: rect.height * imageHeight
11    ))
12
13    return croppedImage
14   }
15 }
```

Funkce cropTo(rect: CGRect) provádí následující kroky:

1. Získá šířku a výšku obrázku pomocí vlastností **self.width** a **self.height** a převede je na hodnoty typu **CGFloat**.

2. Vytvoří nový obdélník pro ořez. Tento obdélník je založen na vstupním obdélníku **rect**, ale jeho souřadnice a rozměry jsou upraveny tak, aby odpovídaly skutečné šířce a výšce obrázku. To je důležité, protože souřadnice a rozměry vstupního obdélníka **rect** jsou obvykle v rozsahu od 0 do 1, kde 1 odpovídá plné šířce nebo výšce obrázku.

3. Provede ořez obrázku pomocí metody **self.cropping(to:)**, která přijímá obdélník pro ořez a vrátí nový obrázek oříznutý podle tohoto obdélníku.

4. Vrátí oříznutý obrázek. Pokud se ořez nepodaří (například pokud je vstupní obdélník mimo rozměry obrázku), vrátí **nil**.

Tato funkce je užitečná například v situacích, kdy je potřeba oříznout obrázek na určitou část. Tato metoda je poté použita pro zpracování pouze té části obrazu načteného kamerou, která se zobrazuje v okně skeneru.

5.1.2 CGPointExtension

Pro nalezení vzdálenosti mezi body v 2D prostoru byla přidana metoda **distanceTo(point: CGPoint)** do třídy **CGPoint**. **CGPoint** je struktura, která reprezentuje bod v 2D prostoru a je běžně používána v rámci Apple's UIKit a Core Graphics frameworků.

```
1 extension CGPoint {
2   func distanceTo(point: CGPoint) -> CGFloat {
3    sqrt(pow((self.x - point.x), 2) + pow((self.y - point.y), 2))
4   }
5 }
```

Metoda **distanceTo(point: CGPoint)** vrací vzdálenost mezi aktuálním bodem (**self**) a bodem předaným jako parametr (**point**). Vzdálenost mezi dvěma body je vypočítána pomocí Pythagorovy věty, která je základní vlastností Eukleidovské geometrie.

5.1.3 ViewExtension

Třida **View** ve Swift UI byla rozšířená metodou **hideKeyboard()**. Tato metoda umožňuje skrýt klávesnici, což je často potřebné v aplikacích, kde uživatel zadává text do textových polí. Tímto způsobem může být tato funkce použita v různých částech UI pro skrytí klávesnice, například po stisknutí tlačítka nebo při klepnutí mimo textové pole.

```
1 extension View {
2 func hideKeyboard() {
3  let resign = #selector(UIResponder.resignFirstResponder)
4  UIApplication.shared.sendAction(resign, to: nil, from: nil, for: nil)
5  }
6 }
```

5.2 OCR

5.2.1 Implementace OCR jádra pro rozpoznávání textu v obrázku

K rozpoznávání textu z obrázků pomocí technologie OCR (Optical Character Recognition) slouží třída **OCREngine**. Třída **OCREngine** je navržena tak, aby efektivně zpracovávala požadavky na OCR a vracela výsledky prostřednictvím funkcí zpětného volání (callback). Je navržena tak, aby byla snadno použitelná pro ostatní části kódu, které potřebují provádět OCR. V třídě jsou definovány následující vlastnosti:

1. **pendingOCRRequests**: Toto je soukromé pole typu **OCREngineRequest**, které udržuje seznam čekajících požadavků na OCR. Požadavek na OCR je reprezentován strukturou **OCREngineRequest**, obsahující obrázek, který se má zpracovat, a funkci zpětného volání, která se má zavolat po dokončení OCR.

2. **imageSize**: Toto je soukromá vlastnost, která udržuje velikost obrázku, který se má zpracovat. Je typu **CGSize**, což je struktura definovaná v UIKit pro uchování rozměrů ve 2D prostoru.

Metody třídy OCREngine jsou:

1. **getTextFromImage**: Tato funkce přijímá obrázek typu **CGImage** a funkci zpětného volání typu **OCREngineCallback** jako parametry. Funkce zpětného volání je volána po dokončení procesu OCR s výsledkem operace.

```
public func getTextFromImage(
  _ image: CGImage,
  callback: @escaping OCREngineCallback
  ) {
   if imageSize == .zero {
     imageSize = .init(
      width: image.width,
      height: image.height
     )
   }

   addRequest(withImage: image, callback: callback)
}
```

2. **addRequest**: Tato funkce přijímá obrázek typu **CGImage** a funkci zpětného volání typu **OCREngineCallback** jako parametry. Vytvoří nový požadavek na OCR a přidá ho do seznamu čekajících požadavků. Pokud je to první požadavek v seznamu, ihned začne proces OCR.

```
private func addRequest(
  withImage image: CGImage,
  callback: @escaping OCREngineCallback
) {
```

```
5     let request = OCREngineRequest(image: image, callback:
       callback)
6     pendingOCRRequests.append(request)
7
8     if pendingOCRRequests.count == 1 {
9      processOCRRequest(request)
10    }
11  }
```

3. **processOCRRequest**: Tato funkce zpracovává požadavek na OCR. Přijímá požadavek typu **OCREngineRequest** jako parametr. Používá Apple Vision knihovnu k analýze obrázku a extrakci textu.

```
1  private func processOCRRequest(_ request: OCREngineRequest) {
2    let requestHandler = VNImageRequestHandler(
3     cgImage: request.image,
4     orientation: CGImagePropertyOrientation.right,
5     options: [:]
6    )
7
8    let visionRequest = VNRecognizeTextRequest(completionHandler:
      recognizeTextHandler)
9
10   visionRequest.recognitionLevel = .accurate
11   visionRequest.usesLanguageCorrection = false
12
13   do {
14    try requestHandler.perform([visionRequest])
15   } catch {
16    print("Error while performing vision request: \(error).")
17    currentRequestProcessed(string: nil)
18   }
19 }
```

4. **recognizeTextHandler**: Tato funkce je volána po dokončení Vision knihovny zpracování obrázku. Přijímá parametry **VNRequest** a volitelnou **Error**. Tato funkce zpracovává výsledky a seřadí rozpoznané texty podle jejich vzdálenosti od středu obrázku.

```
1  private func recognizeTextHandler(
2    request: VNRequest,
3    error: Error?
4  ) {
5    guard let observations = request.results
6       as? [VNRecognizedTextObservation] else {
7        currentRequestProcessed(string: nil)
8        return
9       }
10
11   // Sort the recognized strings based on their distance from
      the center of the image
12   let recognizedStrings = observations
13    .compactMap { observation -> (String, CGFloat)? in
14      guard let candidate = observation.topCandidates(1)
```

```
15          .first else { return nil }
16
17        let boundingBox = observation.boundingBox
18        let boundingBoxCenter = CGPoint(
19          x: boundingBox.origin.x + (boundingBox.width / 2),
20          y: boundingBox.origin.y + (boundingBox.height / 2)
21        )
22
23        let distance = boundingBoxCenter.distanceTo(
24          point: CGPoint(x: 0.5, y: 0.5)
25        )
26
27        return (candidate.string, CGFloat(distance))
28      }
29      .sorted { $0.1 < $1.1 } // sort by distance
30
31    currentRequestProcessed(string: recognizedStrings.first?.0)
32  }
```

5. **currentRequestProcessed**: Tato funkce se volá, když je aktuální požadavek zpracován. Pokud byl text úspěšně rozpoznán, vrátí ho pomocí funkce zpětného volání. Pokud ne, vrátí chybu.

```
1  private func currentRequestProcessed(string: String?) {
2    guard let request = pendingOCRRequests.first else { return }
3
4    pendingOCRRequests.removeFirst()
5    let callback = request.callback
6
7    if let string {
8      callback(.success(string))
9    } else {
10     callback(.failure(NSError(
11       domain: "GFLiveScanner",
12       code: 0,
13       userInfo: ["Message" : "cannot perform OCR on image"]
14     )))
15   }
16 }
```

5.2.2 OCRScannerViewController

Pro skenovaní textu pomoci vestavěné kamery byla vytvořena třida **OCRScanner-ViewController**. Tato třída slouží k digitalizaci textu z obrazovky v reálném čase pomocí kamerového snímku.

Životní cyklus třídy **OCRScannerViewController** lze popsat následovně:

1. **Inicializace** – Instance třídy **OCRScannerViewController** se vytvoří. V tomto okamžiku se vytvoří i všechny její instanční proměnné, včetně **captureSession**, **previewLayer** a **videoOutput**.

2. **Načtení pohledu** - Metoda **viewDidLoad()** se volá poté, co je pohled načten do paměti, ale předtím, než se zobrazí na obrazovce. Tato metoda kontroluje oprávnění ke kameře a poté zahajuje proces skenování zavoláním metody **startScanning()**.

3. **Začátek skenování** - Metoda **startScanning()** zahajuje proces skenování tím, že zahájí běh **captureSession** na **sessionQueue**.

4. **Získání oprávnění** - Pokud nebylo ještě uděleno oprávnění ke kameře, volá se metoda **requestPermission()**, která požádá uživatele o udělení oprávnění. Pokud je oprávnění uděleno, **sessionQueue** se obnoví a pokračuje proces skenování.

5. **Nastavení zachytávacího sezení** - Metoda **setupCaptureSession()** nastaví vstup, náhled a výstup pro **captureSession**. Také nastaví oblast zájmu pro rozpoznání textu.

6. **Zachytávání snímků** - Jakmile je **captureSession** spuštěno, začne získávat video snímky z kamery. Tyto snímky jsou zpracovány v metodě **captureOutput(_,didOutput,from)**, která je definována v rozšíření **AVCaptureVideoDataOutputSampleBufferDelegate**. Snímek se převede na obrázek, ořízne se na oblast zájmu a poté se z něj získá text pomocí **ocrEngine**.

7. **Zastavení skenování** - Proces skenování lze zastavit voláním metody **stopScanning()**, která zastaví běh **captureSession**.

8. **Ukončení** - V momentě kdy již **OCRScannerViewController** není potřeba (například když uživatel odejde z pohledu), je instance **OCRScannerViewController** uvolněna stejně jako všechny její zdroje.

Podrobnější popis třídy **OCRScannerViewController**:

1. **delegate**: Instanční proměnná, která je typu **GFLiveScannerDelegate?**. Toto je delegát, který bude informován, když je detekován text.

2. **screenRect**: Instanční proměnná pro uložení rozměrů obrazovky.

3. **permissionGranted**: Boolovská hodnota, která označuje, zda bylo uděleno oprávnění k přístupu ke kameře.

4. **captureSession, previewLayer, videoOutput**: Tyto tři proměnné řídí proces získávání snímků z kamery, zobrazování náhledu videa a výstupu videa.

5. **sessionQueue**: **sessionQueue** je fronta, ve které běží zpracování snímků z kamery.

6. **ocrEngine** je instance třídy **OCREngine**, která je použita pro rozpoznání textu z obrázků.

7. **viewDidLoad()**: Tato metoda se volá, když je načten pohled. Zde se ověřuje přístup ke kameře a začíná proces skenování.

8. **startScanning(), stopScanning()**: Tyto metody slouží k zahájení a zastavení skenování.

9. **checkPermission(), requestPermission()**: Tyto metody ověřují a vyžadují povolení uživatele k přístupu ke kameře.

```
func checkPermission() {
  switch AVCaptureDevice.authorizationStatus(for: .video) {
  // Permission has been granted before
  case .authorized:
   permissionGranted = true
  // Permission has not been requested yet
  case .notDetermined:
   requestPermission()
  default:
   permissionGranted = false
  }
}

func requestPermission() {
  sessionQueue.suspend()

  AVCaptureDevice
   .requestAccess(for: .video) { [unowned self] granted in
     self.permissionGranted = granted
     self.sessionQueue.resume()
  }
}
```

10. **setupCaptureSession(), setupCaptureSessionInput(), setupCaptureSessionPreview(), setupCaptureSessionOutput()**: Tyto metody konfigurují zachytávání snímků z kamery.

```
func setupCaptureSession() {
  guard setupCaptureSessionInput() else { return }
  setupCaptureSessionPreview()
  setupCaptureSessionOutput()
}

func setupCaptureSessionInput() -> Bool {
  guard let videoDevice = AVCaptureDevice.default(
   .builtInDualWideCamera,
   for: .video,
   position: .back
  ),

  let videoDeviceInput = try? AVCaptureDeviceInput(
```

```
15      device: videoDevice
16    ),
17      captureSession.canAddInput(videoDeviceInput) else {
18        return false
19    }
20
21    captureSession.addInput(videoDeviceInput)
22
23    // make a little zoom in the camera preview to be able to scan
       from a monitor
24    do {
25      try videoDevice.lockForConfiguration()
26      if videoDevice.activeFormat.videoMaxZoomFactor >= 8.0 {
27         videoDevice.videoZoomFactor = 8.0
28      }
29      videoDevice.unlockForConfiguration()
30    } catch {
31       print("Could not configure video device: \(error)")
32    }
33
34    return true
35  }
36
37  func setupCaptureSessionPreview() {
38    previewLayer.session = captureSession
39    previewLayer.videoGravity =
       AVLayerVideoGravity.resizeAspectFill
40    previewLayer.connection?.videoOrientation = .portrait
41
42    let maskLayer = CAShapeLayer()
43    maskLayer.fillRule = .evenOdd
44    maskLayer.fillColor = UIColor.black.cgColor
45    maskLayer.opacity = 0.6
46    previewLayer.addSublayer(maskLayer)
47
48    let regionOfInterestOutline = CAShapeLayer()
49    regionOfInterestOutline.path = UIBezierPath(rect:
       rectOfInterest).cgPath
50    regionOfInterestOutline.fillColor = UIColor.clear.cgColor
51    regionOfInterestOutline.strokeColor = UIColor.yellow.cgColor
52    previewLayer.addSublayer(regionOfInterestOutline)
53
54    // Updates to UI must be on main queue
55    DispatchQueue.main.async { [weak self] in
56      self?.view.layer.addSublayer(self!.previewLayer)
57      self?.previewLayer.frame = self?.view.layer.bounds ?? CGRect()
58    }
59  }
60
61  func setupCaptureSessionOutput() {
62   // Detector
63   videoOutput.videoSettings = [
       String(kCVPixelBufferPixelFormatTypeKey):Int(kCVPixelFormatType_32BGRA)
64   ]
65
66   videoOutput.setSampleBufferDelegate(
67     self,
```

```
68      queue: DispatchQueue(label: "sampleBufferQueue")
69    )
70
71    videoOutput.connection(with: .video)?.videoOrientation =
        .portrait
72    videoOutput.alwaysDiscardsLateVideoFrames = true
73
74    captureSession.addOutput(videoOutput)
75  }
```

11. **rectOfInterest**: Tato metoda vrací obdélník zájmu, který je oblastí, v níž se má provést rozpoznání textu.

```
1  var rectOfInterest: CGRect {
2    previewLayer.metadataOutputRectConverted(
3      fromLayerRect: previewLayer.bounds
4    )
5  }
```

Rozšíření **AVCaptureVideoDataOutputSampleBufferDelegate**:

1. **captureOutput(_, didOutput, from)**: Tato metoda se volá, když je k dispozici nový video snímek. Zde se snímek převede na obrázek, ořízne se na oblast zájmu a poté se z něj získá text pomocí **ocrEngine**.

```
1  func captureOutput(
2    _ output: AVCaptureOutput,
3    didOutput sampleBuffer: CMSampleBuffer,
4    from connection: AVCaptureConnection
5  ) {
6
7    guard let image = getCGImageFromSampleBuffer(sampleBuffer),
8       let croppedImage = image.cropTo(rect: rectOfInterest)
9       else { return }
10
11   ocrEngine.getTextFromImage(croppedImage) { result in
12     switch result {
13     case .success(let string):
14       self.delegate?.capturedString(string)
15     default:
16       break
17     }
18   }
19 }
```

2. **getCGImageFromSampleBuffer(_)**: Tato metoda převádí video snímek (reprezentovaný jako **CMSampleBuffer**) na obrázek (reprezentovaný jako **CGImage**).

```
1  func getCGImageFromSampleBuffer(
2    _ sampleBuffer: CMSampleBuffer
3  ) -> CGImage? {
4    guard let pixelBuffer =
```

```
5       CMSampleBufferGetImageBuffer(sampleBuffer)
6         else { return nil }
7
8    CVPixelBufferLockBaseAddress(pixelBuffer, .readOnly)
9
10   let baseAddress = CVPixelBufferGetBaseAddress(pixelBuffer)
11   let width = CVPixelBufferGetWidth(pixelBuffer)
12   let height = CVPixelBufferGetHeight(pixelBuffer)
13   let bytesPerRow = CVPixelBufferGetBytesPerRow(pixelBuffer)
14   let colorSpace = CGColorSpaceCreateDeviceRGB()
15   let bitmapInfo = CGBitmapInfo(
16    rawValue: CGImageAlphaInfo.premultipliedFirst.rawValue |
17         CGBitmapInfo.byteOrder32Little.rawValue
18   )
19
20   guard let context = CGContext(
21    data: baseAddress,
22    width: width,
23    height: height,
24    bitsPerComponent: 8,
25    bytesPerRow: bytesPerRow,
26    space: colorSpace,
27    bitmapInfo: bitmapInfo.rawValue
28   ) else { return nil }
29
30   let cgImage = context.makeImage()
31
32   return cgImage
33 }
```

Protokol **GFLiveScannerDelegate**:

1. **capturedString(_:)**: Tato metoda je volána, když je detekován text. Je určena k implementaci delegátem.

```
1 public protocol GFLiveScannerDelegate {
2   func capturedString(_ string: String)
3 }
```

5.2.3 OCRScannerView

Pro účely použiti **OCRScannerViewController** v SwiftUI aplikaci je potřeba vytvořit určitou komponentu, která by pomohla propojit dva UI frameworky Apple. K tomu byl použít protokol **UIViewControllerRepresentable**, navržený pro tyto účely. **OCRScannerView** slouží jako obal pro **OCRScannerViewController**, díky tomu, že implementuje protokol **UIViewControllerRepresentable**.

Struktura **OCRScannerView** obsahuje dvě proměnné **@Binding - scannedText a scannedNumber**. Tyto proměnné se aktualizují, když **OCRScannerViewController** zachytí a zpracuje text pomocí OCR.

Metoda **makeUIViewController(context:)** vytváří instanci **OCRScannerViewController** a nastavuje její delegáta na koordinátora, který je vytvořen metodou **makeCoordinator()**.

```
func makeUIViewController(context: Context) ->
    OCRScannerViewController {

  let viewController = OCRScannerViewController()
  viewController.delegate = context.coordinator
  return viewController
}
```

Metoda **updateUIViewController(_:context:)** je prázdná, protože v tomto případě není třeba aktualizovat **OCRScannerViewController** s novými daty z SwiftUI.

Koordinátor je pomocná třída, která koordinuje interakci mezi **OCRScannerViewController** a **OCRScannerView**.

```
class Coordinator: NSObject, GFLiveScannerDelegate {
  var parent: OCRScannerView

  init(_ parent: OCRScannerView) {
   self.parent = parent
   super.init()
  }
}
```

Koordinátor implementuje protokol **GFLiveScannerDelegate**, který obsahuje metodu **capturedString(_:)**, kde se zpracuje zachycený text. Tato metoda aktualizuje proměnné **scannedText** a **scannedNumber**.

```
func capturedString(_ string: String) {
  parent.scannedText = "\(string)"

  if let number = parseNumber(text: string) {
   parent.scannedNumber = number
  }
}
```

Funkce **parseNumber(text:)** je pomocná funkce, která se pokouší převést zpracovaný text na číslo typu **Double**. Pro tento proces je třeba odstranit všechny neplatné znaky a nahradit některé často zaměňovaných znaků za číslice.

```
private func parseNumber(text: String) -> Double? {
  let acceptedLetters = Array(0...9)
    .map(String.init) + ["-"] + [","] + ["."]

  let characters = text
   .replacingOccurrences(of: ",", with: ".")
   .replacingOccurrences(of: "\n", with: "")
   .replacingOccurrences(of: "o", with: "0")
   .replacingOccurrences(of: "|", with: "1")
   .replacingOccurrences(of: "l", with: "1")
   .replacingOccurrences(of: "L", with: "1")
   .replacingOccurrences(of: "z", with: "2")
```

```
13      .replacingOccurrences(of: "Z", with: "2")
14      .replacingOccurrences(of: "s", with: "2")
15      .replacingOccurrences(of: "S", with: "2")
16      .filter({ acceptedLetters.contains(String($0)) })
17
18      return Double(characters)
19  }
```

Hlavní role koordinátoru je zajištění komunikace mezi SwiftUI a UIKit komponentami. SwiftUI a UIKit mají různé životní cykly a způsoby, jakými řeší změny stavu.

Koordinátor je objekt, který je vytvořen a spravován SwiftUI a poskytuje stabilní a trvalý kontext pro UIKit třídy, které mohou být vytvářeny a zničeny během životního cyklu SwiftUI zobrazení.

5.3 Datové sktruktury

5.3.1 ContentViewModel

Třída **ContentViewModel** je třída "observabilního"objektu, která slouží jako ViewModel pro SwiftUI **View**.

- **@Published var rowsCount: Int: rowsCount** je proměnná, která sleduje počet řádků v tabulce. **@Published** znamená, že když se tato hodnota změní, oznámí se to všem, kteří tuto proměnnou sledují. **willSet** je tzv. property observer, který je volán předtím, než je nastavena nová hodnota proměnné. V tomto případě se přidává nebo odebírá řádek ve **values** podle toho, jestli se počet řádků zvětšil nebo zmenšil.

```
1  @Published var rowsCount: Int
2  {
3    willSet {
4     if newValue > rowsCount {
5       values.append(.init(repeating: nil, count: columnsCount))
6     } else {
7       values.removeLast()
8     }
9    }
10 }
```

- **@Published var columnsCount: Int**: Podobně jako **rowsCount**, **columnsCount** sleduje počet sloupců v tabulce a přidává nebo odebírá sloupec ve **values** podle změny počtu sloupců.

```
1  @Published var columnsCount: Int
2  {
3    willSet {
4     if newValue > columnsCount {
5       values.indices.forEach { rowIndex in
6        values[rowIndex].append(nil)
7       }
```

```
    } else {
      values.indices.forEach { rowIndex in
       values[rowIndex].removeLast()
      }
    }
  }
}
```

- **@Published var values: [[Double?]]**: **values** je dvourozměrné pole, které reprezentuje hodnoty v tabulce. Každý prvek ve **values** je buď **Double** hodnota nebo **nil** (prázdna hodnota ve Swift).
- **let tempFileURL: URL**: **tempFileURL** je dočasný soubor, do kterého se ukládá CSV soubor.
- **init(rowsCount: Int = 4, columnsCount: Int = 4)**: Konstruktor pro **ContentViewModel**. Vytváří instanci **ContentViewModel** s daným počtem řádků a sloupců.

```
init(rowsCount: Int = 4, columnsCount: Int = 4) {
  self.rowsCount = rowsCount
  self.columnsCount = columnsCount
  self.values = [
   [0, 1, 2, 3],
   [0, 1, 2, 3],
   [0, 1, 2, 3],
   [0, 1, 2, 3]
  ]

  let tempDirectoryURL = URL(
   fileURLWithPath: NSTemporaryDirectory(),
   isDirectory: true
  )

  tempFileURL = tempDirectoryURL
   .appendingPathComponent(UUID().uuidString)
   .appendingPathExtension("csv")
}
```

- **func isCSVAvailable() -> Bool**: Tato funkce generuje CSV soubor z hodnot ve **values** a ukládá ho do **tempFileURL**. Pokud se CSV soubor podaří vytvořit a uložit, funkce vrátí **true**. Pokud dojde k chybě, funkce vrátí **false**.

```
func isCSVAvailable() -> Bool {
  do {
   let tmpValues = [Array(
     repeating: 0.0,
     count: values.first?.count ?? 1
     )] + values

   let csvBodyString = tmpValues.map({ row in
     row.enumerated().map({ index, value in
```

```
10          if row == tmpValues.first {
11            return index == 0 ? "x" : "y\(index)"
12          } else {
13            if let value {
14             return "\(value)"
15            } else {
16             return ""
17            }
18          }
19        }).joined(separator: ",")
20      }).joined(separator: "\n")
21
22      try String(csvBodyString).write(
23        to: tempFileURL,
24        atomically: true,
25        encoding: .utf8
26      )
27
28      return true
29    } catch {
30      print(error)
31      return false
32    }
33  }
```

Celkově **ContentViewModel** spravuje data tabulky a poskytuje metodu pro generování CSV souboru.

5.3.2 FocusedCell

FocusedCell implementuje protokol **Hashable**. **Hashable** je protokol, který vyžaduje, aby objekty mohly být jedinečné a použitelné jako klíče v hashovacích strukturách, jako jsou například **Dictionary** a **Set**.

Struktura **FocusedCell** obsahuje dvě konstanty:

- **row: Int**: Tato konstanta označuje řádek buňky, která je aktuálně zaměřena.
- **column: Int**: Tato konstanta označuje sloupec buňky, která je aktuálně zaměřena.

```
1  struct FocusedCell: Hashable {
2    let row: Int
3    let column: Int
4  }
```

Cílem této struktury je uchovat informace o poloze buňky, která je aktuálně vybrána v nějakém kontextu, například v tabulce.

Implementace protokolu **Hashable** umožňuje porovnávat instance **FocusedCell** a používat je jako klíče v datových strukturách. To je užitečné například v případě, kdy

je potřeba sledovat, které buňky jsou vybrány. Díky tomu lze jednoduše zjistit, jestli je konkrétní buňka vybrána, porovnáním její pozice (řádek a sloupec) s pozicemi všech vybraných buněk.

5.4 Obrazovky

5.4.1 Pomocné komponenty

TableManipulatorView TableManipulatorView je struktura, která představuje SwiftUI **View**. Toto **View** zobrazuje dva ovládací prvky **Stepper**, které umožňují uživateli upravit počet řádků a sloupců v tabulce.

- **@ObservedObject var viewModel: ContentViewModel: TableManipulatorView** používá **ContentViewModel** jako svůj **ObservedObject**. Tímto způsobem **TableManipulatorView** může sledovat změny ve **viewModel** a automaticky aktualizovat svůj obsah, když se **viewModel** změní.
- **VStack**: Oba **Steppers** jsou umístěny uvnitř **VStack**, což je SwiftUI prvek pro vertikální zarovnání podřízených prvků.
- První **Stepper**: zobrazuje a upravuje počet řádků (**viewModel.rowsCount**). Text "Rows:" je zobrazen vedle **Stepper**, a je doplněn o aktuální počet řádků. Hodnoty, které může **Stepper** nabývat, jsou od 1 do maximálního celého čísla (**Int.max**).
- Druhý **Stepper**: zobrazuje a upravuje počet sloupců (**viewModel.columnsCount**). Stejně jako u prvního **Stepper**, text "Columns:" je zobrazen vedle **Stepper**, a je doplněn o aktuální počet sloupců.

Steppery, stejně jak ostatní ovládací prvky ve SwiftUI funguji na zaklade obousměrného provázaní (**Binidng**), který se stará o to, aby se při změně hodnoty v ovládacím prvku se automaticky měnila jemu předaná jemu hodnota a naopak. Tuto vazbu lze vidět při inicializaci **Stepperu** a předáni jako **value** $**viewModel.rowsCount** a $**viewModel.columnsCount**, takže když uživatel upravuje **Stepper**, přímo se upravuje hodnota ve **viewModel**. To poté vede k automatické aktualizaci **View**.

```
1 struct TableManipulatorView: View {
2   @ObservedObject var viewModel: ContentViewModel
3
4   var body: some View {
5    VStack {
6      Stepper(
7       "Rows: \(viewModel.rowsCount)",
8       value: $viewModel.rowsCount,
9       in: 1...Int.max
```

```
10        )
11
12        Stepper(
13         "Columns: \(viewModel.columnsCount)",
14         value: $viewModel.columnsCount,
15         in: 1...Int.max
16        )
17      }
18    }
19  }
```

5.5 Hlavni obrazovka

ContentView je hlavní obrazovka aplikace, která slouží jako kontejner pro ostatní podřízené obrazovky. ContentView se skládá z následujících části:

- **Proměnné stavu**:
  - **scannedText**: drží text, který byl naskenován pomocí OCR.
  - **scannedNumber**: drží číslo, které bylo naskenováno pomocí OCR a převedeno na **Double**.
  - **isScannerSheetPresented**: řídí, zda se má zobrazit modální okno pro skenovaní textu – **ScannerSheetView**.
  - **isShareSheetPresented**: řídí, zda se má zobrazit modální okno pro sdílení.
  - **viewModel**: je objekt **ContentViewModel**, který spravuje data aplikace.
  - **focusedCell**: řídí, která buňka v tabulce aktuálně je vybraná.
- Tělo **View**: Zobrazuje hlavní rozložení **View**, které obsahuje **TableManipulatorView**, tlačítko pro navigaci na **ChartView** a skupinu s naskenovanými hodnotami.

```
1  var body: some View {
2    VStack {
3     TableManipulatorView(viewModel: viewModel)
4     Spacer()
5
6     NavigationLink(
7       destination: ChartView(data: viewModel.values) {
8        scannerViewScannedValuesSectionView
9       },
10      label: {
11       Label("Chart", systemImage: "chart.xyaxis.line")
12         .padding(.horizontal, 10)
13      }
14    )
15    .buttonStyle(.borderedProminent)
16
17    Spacer()
```

```
18
19      scannerViewScannedValuesSectionView
20      Spacer()
21    }
22    .sheet(
23      isPresented: $isScannerSheetPresented,
24      content: scannerSheetView
25    )
26    .padding()
27    .toolbar(content: toolbarView)
28    .navigationTitle("Main")
29    .navigationBarTitleDisplayMode(.inline)
30    .sheet(isPresented: $isShareSheetPresented) {
31      ShareSheetView(activityItems: [viewModel.tempFileURL])
32        .presentationDetents([.medium])
33    }
34  }
```

- Funkce **toolbarView**: Zobrazuje nástrojovou lištu s tlačítkem „Scan“ na spodní liště a tlačítkem pro sdílení na horní liště.

```
1   @ToolbarContentBuilder
2   func toolbarView() -> some ToolbarContent {
3     ToolbarItem(placement: .bottomBar, content: {
4       Button(action: {
5         focusedCell = nil
6         isScannerSheetPresented = true
7       }, label: {
8         Label("Scan", systemImage: "qrcode.viewfinder")
9           .labelStyle(.titleAndIcon)
10          .padding(.horizontal, 10)
11          .padding(.vertical, 5)
12      }
13    )
14    .buttonStyle(.borderedProminent)
15    })
16
17    ToolbarItem(placement: .navigationBarTrailing, content: {
18      Button(action: {
19        guard viewModel.isCSVAvailable() else { return }
20        self.isShareSheetPresented.toggle()
21      }, label: {
22        Image(systemName: "square.and.arrow.up")
23      })
24    })
25  }
```

- Funkce **scannerViewScannedValuesSectionView**: Tato funkce vytváří **View**, který zobrazuje naskenované hodnoty ve formě tabulky.

```
1   var scannerViewScannedValuesSectionView: some View {
2     GroupBox("Scanned values") {
3       ScrollViewReader { scrollProxy in
4         ScrollView([.vertical, .horizontal]) {
5           Grid {
6             ForEach(
```

```
7           viewModel.values.indices,
8           id: \.self
9         ) { row in
10          GridRow {
11            ForEach(
12              viewModel.values[row].indices,
13              id: \.self
14            ) { column in
15              valueCellView(row, column)
16            }
17          }
18        }
19      }
20    }
21    .onAppear {
22      scrollProxy.scrollTo("0 0", anchor: .center)
23    }
24  }
25 }
26 .frame(maxHeight: 350, alignment: .topLeading)
27 .fixedSize(horizontal: false, vertical: true)
28 .clipShape(RoundedRectangle(cornerRadius: 15))
29 }
```

- Funkce **valueCellView**: Tato funkce vytváří jednotlivé buňky pro tabulku s naskenovanými hodnotami.

```
1  func valueCellView(_ row: Int, _ column: Int) -> some View {
2    TextField(
3      value: $viewModel.values[row][column],
4      format: .number,
5      label: {}
6    )
7    .multilineTextAlignment(.center)
8    .keyboardType(.numbersAndPunctuation)
9    .autocorrectionDisabled()
10   .frame(minWidth: 30, minHeight: 50)
11   .scenePadding(.minimum, edges: .horizontal)
12   .focused(
13     $focusedCell,
14     equals: FocusedCell(row: row, column: column)
15   )
16   .border(
17     isCellSelected(row, column)
18       ? Color.accentColor
19       : .secondary,
20     width: isCellSelected(row, column) ? 2 : 1
21   )
22   .onTapGesture {}
23   .onSubmit(hideKeyboard)
24   .id("\(row) \(column)")
25 }
```

- Funkce **isCellSelected**: Tato funkce kontroluje, zda je buňka vybrána.

```
1  func isCellSelected(_ row: Int, _ column: Int) -> Bool {
```

```
2     focusedCell == FocusedCell(row: row, column: column)
3   }
```

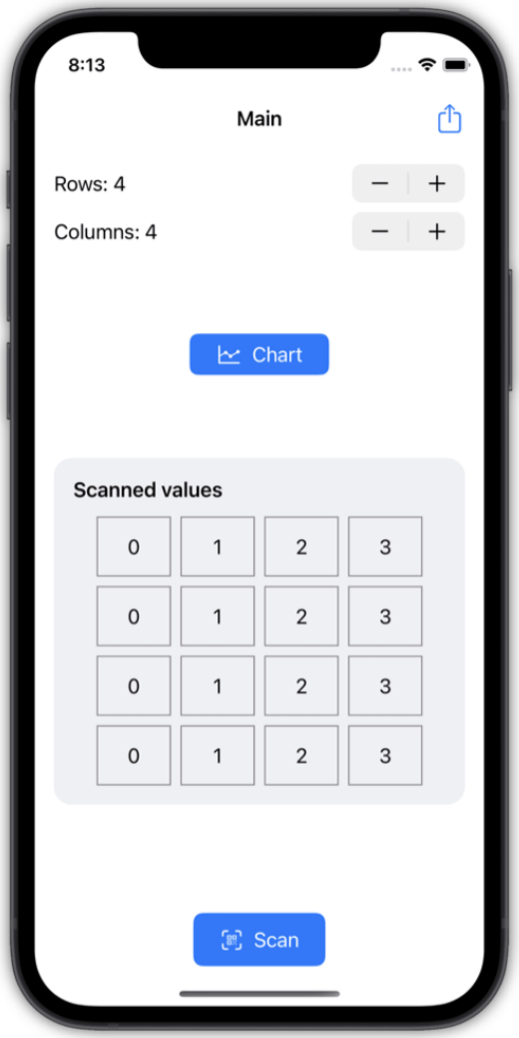
- Funkce **scannerSheetView**: Tato funkce vytváří **ScannerSheetView**, který je zobrazen v **NavigationView**.

```
1   func scannerSheetView() -> some View {
2     NavigationView {
3       ScannerSheetView(
4         scannedText: $scannedText,
5         scannedNumber: $scannedNumber,
6         viewModel: viewModel
7       )
8       .edgesIgnoringSafeArea(.vertical)
9       .navigationTitle("Scanner")
10      .navigationBarTitleDisplayMode(.inline)
11    }
12  }
```

Kód obsahuje několik **@State** a **@StateObject** proměnných, které jsou v SwiftUI používány k sledování stavu a změn v aplikaci. Také zde je použit **@FocusState** pro řízení fokusu buněk v tabulce.

NavigationLink je použitý k navigaci na **ChartView**, zatímco **sheet** metoda je použita pro zobrazení **ScannerSheetView** a **ShareSheetView** v závislosti na stavu odpovídajících **@State** proměnných.

ContentView poskytuje náhled na modifikátory **ViewBuilder** a **ToolbarContentBuilder**, které umožňují snadné sestavování složitějších **View** a **Toolbar** komponent. Na obrázku 5.1 lze vidět výslednou hlavni obrazovku.



Obrázek 5.1 Hlavni obrazovka

5.5.1 Obrazovka skeneru

ScannerSheetView reprezentuje obrazovku skeneru a zobrazuje různé informace a ovládací prvky související s procesem skenování.

- **@Binding var scannedText: String** a **@Binding var scannedNumber: Double?**: **Binding** označuje odkaz na data, která mohou být sdílena mezi více **View**. Tato data jsou **scannedText**, což je text získaný ze skenování, a **scannedNumber**, což je číslo získané ze skenování.
- **@ObservedObject var viewModel: ContentViewModel**: **ScannerSheetView** používá **ContentViewModel** jako svůj **ObservedObject**, což znamená,

že jakékoli změny ve **viewModel** vyvolají aktualizaci **ScannerSheetView**.

- **@State private var selectedCell: FocusedCell = .init(row: 0, column: 0): selectedCell** je stavová proměnná, která udržuje informace o aktuálně vybrané buňce. **@State** značí, že **selectedCell** je zdrojem pravdy a jakékoli změny v ní vyvolají aktualizaci **View**.
- **UITableView.appearance().isScrollEnabled = false**: Tento řádek deaktivuje rolování pro všechny instance **UITableView** v rámci této **View**. Je to globální nastavení, které ovlivní všechny **UITableView** prezentované v rámci tohoto **View**.
- Tělo **View**: Zobrazuje hlavní rozložení **View**, které obsahuje **OCRScannerView**, skenerem načtenou textovou a číslelnou hodnotu, **TableManipulatorView** a skupinu s naskenovanými hodnotami.

```
var body: some View {
  NavigationStack {
   GroupBox {
      ZStack {
        OCRScannerView(
         scannedText: $scannedText,
         scannedNumber: $scannedNumber
        )

        VStack(alignment: .leading) {
            Spacer()
            HStack {
              Text("Text: ")
              Text(scannedText)
          }

         HStack {
            Text("Number: ")

            if let scannedNumber {
              Text(scannedNumber, format: .number)
            } else {
              Text("")
            }
         }
        }
        .frame(maxWidth: .infinity, alignment: .leading)
      }
      .frame(height: 170)
   }
   .clipShape(RoundedRectangle(cornerRadius: 15))

   TableManipulatorView(viewModel: viewModel)
      .padding(.horizontal)

   scannerViewScannedValuesSectionView
      .toolbar(content: scannerViewToolbar)
```

```
38
39      Spacer()
40    }
41    .padding()
42  }
```

- **valueCellView(_ row: Int, _ column: Int)**: Tato funkce vytváří buňky tabulky, které reprezentují hodnoty v jednotlivých buňkách tabulky.

```
1  func valueCellView(_ row: Int, _ column: Int) -> some View {
2      Button(action: {
3        selectedCell = .init(row: row, column: column)
4      }, label: {
5        Text(viewModel.values[row][column]?.formatted() ?? "")
6          .frame(minWidth: 30, maxWidth: .infinity, minHeight: 50)
7          .scenePadding(.minimum, edges: .horizontal)
8          .border(
9            isCellSelected(row, column)
10             ? Color.accentColor
11             : .secondary,
12           width: isCellSelected(row, column) ? 2 : 1
13         )
14         .contentShape(Rectangle())
15     })
16     .buttonStyle(.plain)
17     .id(FocusedCell(row: row, column: column))
18   }
```

- **isCellSelected(_ row: Int, _ column: Int) -> Bool**: Tato funkce kontroluje, zda je buňka definovaná daným řádkem a sloupcem vybrána.

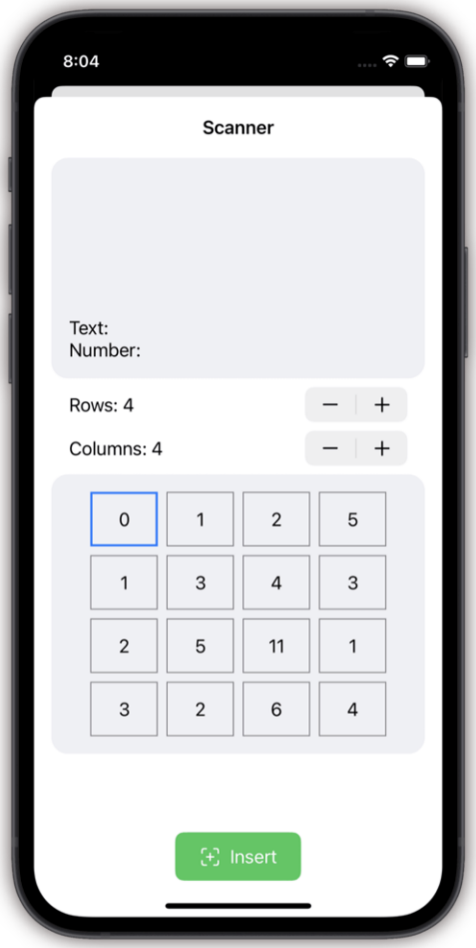
```
1  func isCellSelected(_ row: Int, _ column: Int) -> Bool {
2      selectedCell == .init(row: row, column: column)
3  }
```

- **scannerViewToolbar() -> some ToolbarContent**: Tato funkce vytváří nástrojovou lištu pro **View**. Nástrojová lišta obsahuje tlačítko, které umožňuje vložení skenovaného čísla do vybrané buňky.

```
1  func scannerViewToolbar() -> some ToolbarContent {
2      ToolbarItem(placement: .bottomBar, content: {
3        Button(action: {
4          viewModel.values
5            [selectedCell.row]
6            [selectedCell.column] = scannedNumber
7
8          let newColumn = (selectedCell.column + 1)
9              % (viewModel.values.first?.count ?? 1)
10         let newRow = (selectedCell.row
11             + (newColumn == 0 ? 1 : 0))
12                 % viewModel.values.count
13
14         selectedCell = .init(row: newRow, column: newColumn)
15       }, label: {
```

```
16            Label("Insert", systemImage: "plus.viewfinder")
17              .labelStyle(.titleAndIcon)
18              .padding(.horizontal, 10)
19              .padding(.vertical, 5)
20          })
21          .buttonStyle(.borderedProminent)
22          .tint(.green)
23      })
24  }
```

Celkově je cílem tohoto **View** poskytnout uživateli interaktivní nástroj pro skenování textu a čísel a jejich následné umístění do tabulky. Obrazovka skeneru je reprezentovaná na obrázku 5.2.



Obrázek 5.2 Obrazovka skeneru

5.5.2 Menu pro sdílení načtených dat ve formátu CSV

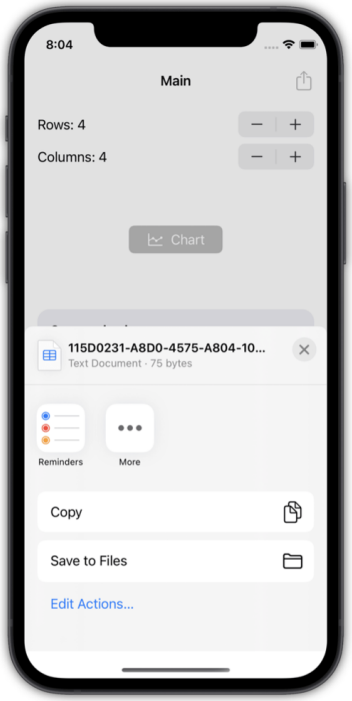
Tato obrazovka umožňuje sdílet různý obsah, v tomto případě CSV soubor, s různými službami a aplikacemi v systému iOS. **ShareSheetView**, implementuje protokol **UIViewControllerRepresentable**. **UIViewControllerRepresentable** je protokol ve SwiftUI, který umožňuje vytvářet a spravovat **UIViewController** instance ve SwiftUI.

Struktura **ShareSheetView** obsahuje následující vlastnosti:

- **activityItems**: pole položek, které budou sdíleny.
- **applicationActivities**: pole vlastních **UIActivity** instancí, které budou k dispozici ve zobrazení sdílení. Defaultně je to **nil** (prázdna hodnota).
- **excludedActivityTypes**: pole typů aktivit, které budou vynechány ze zobrazení sdílení. Defaultně je to **nil**.
- **callback**: volitelná zpětná vazba, která bude volána po dokončení aktivit.

Metoda **makeUIViewController(context: Context) -> UIActivityViewController** vytváří instanci **UIActivityViewController**, která je náhledem sdílení v iOS. Tento náhled umožňuje uživatelům sdílet obsah s různými službami a aplikacemi.

Metoda **updateUIViewController(_ uiViewController: UIActivityViewController, context: Context)** je vyžadována protokolem **UIViewControllerRepresentable**, ale v tomto případě není potřeba nic aktualizovat, takže je prázdná. Tato metoda by se použila, pokud by bylo potřeba aktualizovat stav **UIViewController** v reakci na změny v SwiftUI. Menu pro sdíleni naskenovaných hodnot ve formátu CSV souboru lze vidět na níže uvedeném obrázku 5.3.



Obrázek 5.3 Menu pro sdílení
načtených dat ve formátu CSV

5.5.3 Obrazovka s grafy

ChartView je generickým SwiftUI zobrazením, které vytváří sloupcový graf na základě poskytnutých dat a obsahuje další libovolné zobrazení, v tomto případě je to tabulka načtených dat, stejně jak na hlavní obrazovce.

Struktura **ChartView** má tři hlavní vlastnosti:

1. **data**: Dvourozměrné pole **Double?**, které reprezentuje data k zobrazení v grafu. Každé vnitřní pole reprezentuje řadu dat, přičemž první prvek v řadě reprezentuje hodnotu x a další prvky reprezentují hodnoty y pro různé řady dat.

2. **content**: Typu **Content**, který je generickým parametrem omezeným na **View**. Tento prvek zobrazuje další obsah, který má být zobrazen pod grafem.

3. Inicializátor, který přijímá **data** a **content** jako parametry. Parametr content je funkce označená **@ViewBuilder**, která umožňuje předat více SwiftUI zobrazení, které se pak skombinují do jednoho.

V těle **ChartView** je vytvořena instance **Chart**, která se používá k vytvoření grafu. Je-li v datech více než jeden sloupec, jsou vytvořeny série **LineMark** a **PointMark** pro každý sloupec dat.

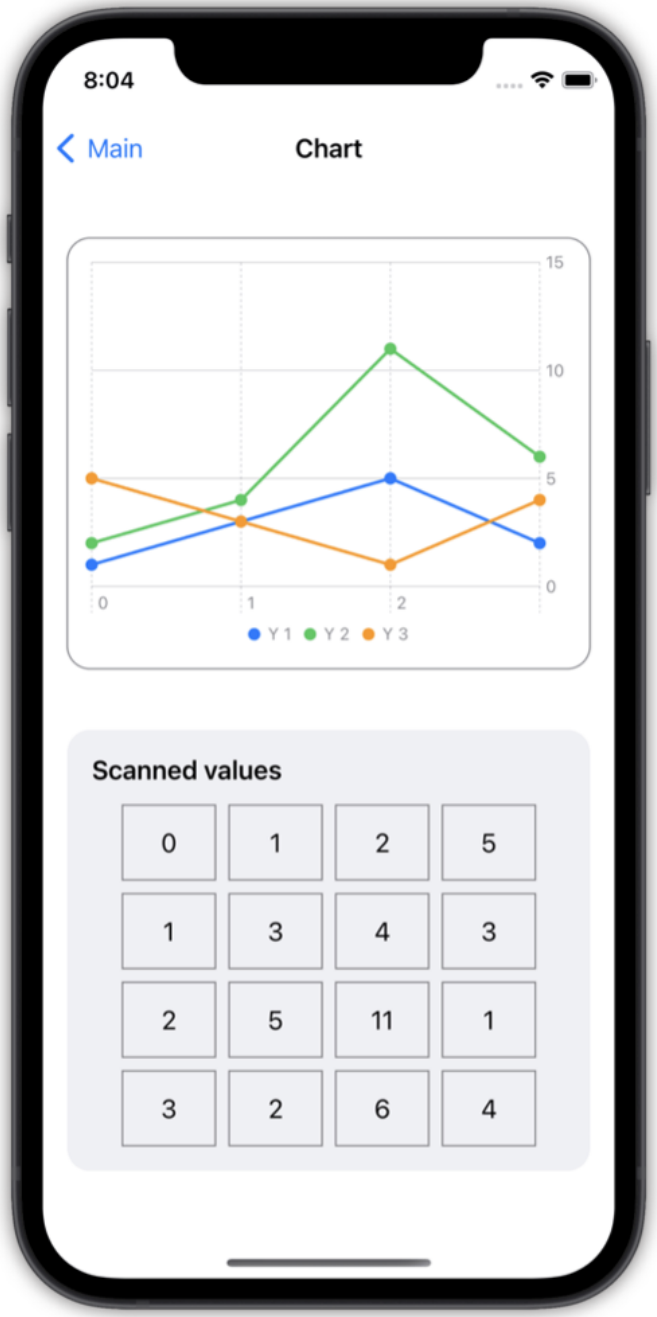
```
var body: some View {
  VStack {
   Chart {
      if let columnsCount = data.first?.count, columnsCount > 1 {
         let rowLength = data.count

        ForEach(1..<columnsCount, id: \.self) { column in
         ForEach(0..<rowLength, id: \.self) { row in
            if let y = data[row][column],
               let x = data[row].first as? Double {
                  LineMark(
                    x: .value("X", x),
                    y: .value("Y", y)
                  )
                  .foregroundStyle(by: .value("Type", "Y \(column)"))

                  PointMark(
                    x: .value("X", x),
                    y: .value("Y", y)
                  )
                  .foregroundStyle(by: .value("Type", "Y \(column)"))
            }
          }
        }
      }
   }
   .chartLegend(position: .bottom, alignment: .center)
   .frame(height: 250)
   .padding()
   .background(RoundedRectangle(cornerRadius: 15).stroke(.secondary))
   .padding()

   content.padding()
  }
  .navigationTitle("Chart")
  .navigationBarTitleDisplayMode(.inline)
}
```

LineMark a **PointMark** jsou body na čáře a na grafech, které zobrazují hodnoty y na základě hodnoty **x**.

Následně je vytvořen grafický popis (**chartLegend**) a nastavena velikost a styl pro graf.

Pod grafem je zobrazen **content**, který je další SwiftUI zobrazení, které je k dispozici pro zobrazení pod grafem. **content** být cokoli, co splňuje požadavky na **View**, a je dodáno jako parametr při vytváření **ChartView**.

Nakonec je pro celý **ChartView** nastaven název navigace na "Chart"a je zvolen režim zobrazení titulu navigace na **.inline**. Obrázek 5.4 znázorňuje obrazovku s grafy.



Obrázek 5.4 Obrazovka s grafy

6 TESTOVANÍ

Jednou z klíčových částí procesu vývoje a testování rozpoznávací aplikace čísel je dataset. Dataset zvolený pro tento úkol obsahuje mnoho obrázků čísel napsaných v různých fontech, stylech a řezech písma. Dataset zahrnuje také několik příkladů ručně psaných čísel.

6.1 Výběr datasetu

Hlavním kritériem pro výběr datasetu byla jeho rozmanitost. Rozpoznávání čísel v různých stylech a formátech je obtížné, a proto je důležité, aby zvolený dataset zahrnoval co nejširší možné spektrum příkladů. To zahrnuje nejen různé fonty a styly písma, ale také rozdílné formáty, jako jsou ručně psané číslice, a různé úrovně kvality obrázků.

6.2 Proces testovaní

Manuální testování rozpoznávání čísel je klíčovou částí procesu vývoje aplikace. Tento typ testování umožňuje ověřit funkčnost a přesnost aplikace v praxi a identifikovat potenciální problémy.

Před zahájením manuálního testování bylo připraveno několik různých datasetů, které reprezentují různé typy čísel, fontů a řezů písma. Tyto datasety také zahrnují čísla zobrazená v různých kvalitách obrazu, aby bylo možné ověřit, jak dobře aplikace zvládne čtení čísel z obrazů s nižší kvalitou. Všechny testy se provádí za normálních světelných podmínek během dne, aby byly podmínky co nejblíže běžnému užívání.

Pro všechny manuální testy byl použít iPhone 12 Mini.

Během testování aplikaci byla čísla načítána z obrazovky monitoru. Každý dataset byl zobrazen na monitoru a poté načten pomocí aplikace na iPhone 12 Mini.

6.3 Výsledky testovaní

6.3.1 První testovací sada

První testovací sada (Obr. 6.1) se skládá z čísel dobře čitelných, přehledných a zobrazených v dobré kvalitě. Standartní font a řez. Žádné potíže při skenování daného datasetu nevznikly. Díky menším úpravám v načteném textu před transformaci textu na čísla není problémem ani desetinná téčka ani čárka. Daný dataset byl zpracován rychle a bez problémů.

1	**12,5**	**18**	**156**	**245,00**
15	**258**	**0.256**	**18.3**	**865**
10	**13**	**5 845**	**45 258,3**	**1789**

Obrázek 6.1 První testovací sada

6.3.2 Druhá testovací sada

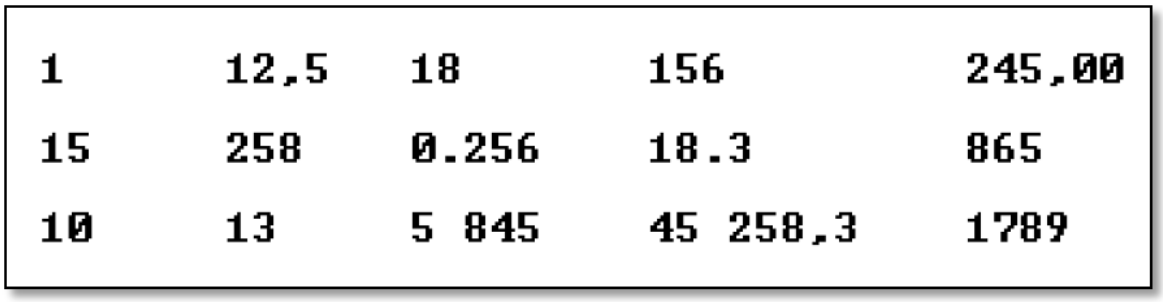
Při testovaní druhé testovací sady (Obr. 6.2) se vyskytly některé problémy s rozpoznáním čísel: 5845 a 45258,3. Skener je sice zachytával, ale výsledek nebyl stabilní a správná hodnota se střídala s jinou podobnou. 5845 se často střídalo s 845 a 45258.3 s 258,3 nebo 258. Načítaní dalších čísel z tohoto datasetu probíhalo bez problémů.

1	12,5	18	156	245,00
15	258	0.256	18.3	865
10	13	5 845	45 258,3	1789

Obrázek 6.2 Druhá testovací sada

6.3.3 Třetí testovací sada

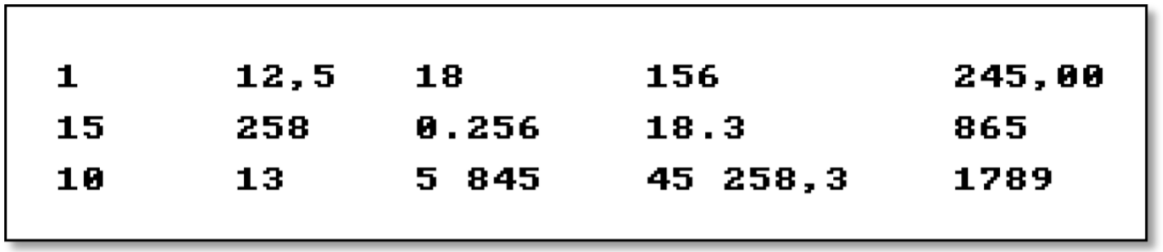
V třetí testovací sadě (Obr. 6.3) se zopakovaly problémy s načítáním, stejně jak ve druhé testovací sade a rozšířily se na další čísla v datasetu. Největší problém byl s načtením čísla 5845, skenovací algoritmus často vynechával číslici 5 z výsledku.



Obrázek 6.3 Třetí testovací sada

6.3.4 Čtvrtá testovací sada

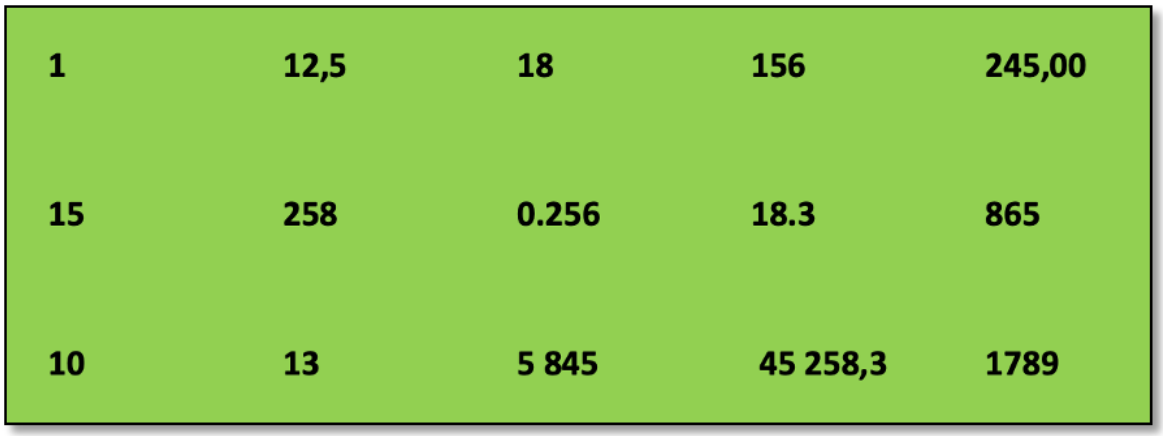
Čtvrtá testovací sada se skládá z čísel výrazně kostičkovaných (Obr. 6.4), čím schopnost aplikace správně detekovat symboly byla trochu snížena. Aplikace byla schopna načíst většinu čísel bez problémů. Potíže nastaly s rozpoznáním 0 a občas s desetinnou čárkou.



Obrázek 6.4 Čtvrtá testovací sada

6.3.5 Pátá testovací sada

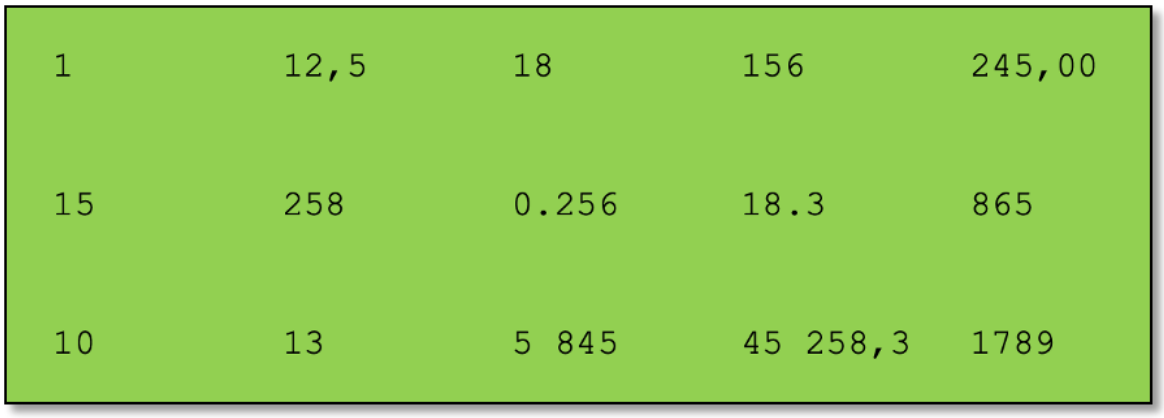
Pátá testovací sada (Obr. 6.5) stejně jak první nevyvolala žádné potíže při skenovaní.



Obrázek 6.5 Pátá testovací sada

6.3.6 Šestá testovací sada

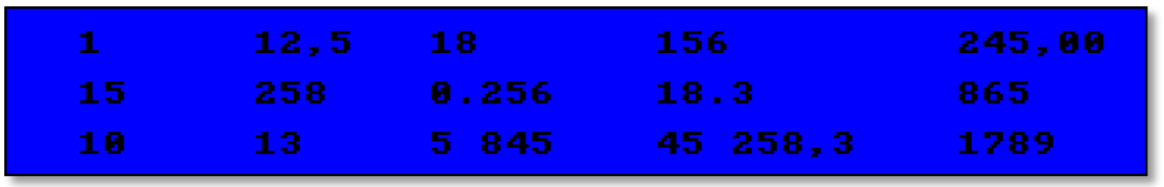
V šesté testovací sadě (Obr. 6.6) se vyskytly podobné problémy, jak při skenování čísel z druhé testovací sady, tj. nestabilní výsledek skenování čísel 5845 a 45258,3.



Obrázek 6.6 Šestá testovací sada

6.3.7 Sedmá testovací sada

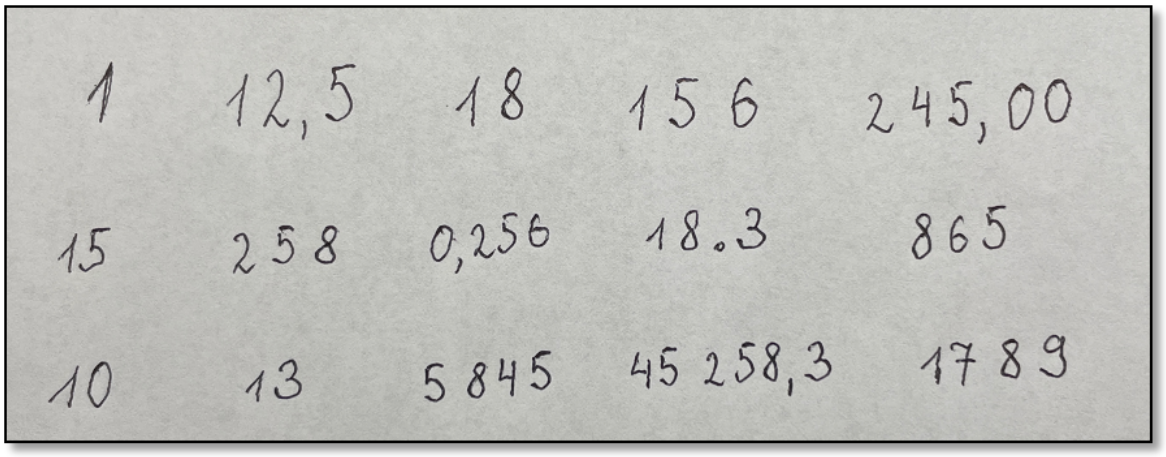
Při skenování této testovací sady (Obr. 6.7) nastával problém skoro u každého čísla, buď to se stabilitou výsledné hodnoty nebo s tím, že OCR vůbec nechtěl rozpoznat některé symboly. Nepříjemným překvapením bylo, že OCR nezvládl přečíst ani 156.



Obrázek 6.7 Sedmá testovací sada

6.3.8 Osmá testovací sada

Osmá testovací sada se skládá s ručně psaných čísel (Obr. 6.8). Při skenovaní této sady se dalo očekávat velmi slabou účinnost OCR, ale výsledky skenovaní jsou přijatelné. Většina čísel byla detekována správně. Problémem byly čísla 12.5, 18.3 a 45 258.3.



Obrázek 6.8 Osmá testovací sada

6.3.9 Zhodnoceni testovaní

Po testování aplikace na sadách čísel napsaných různými řezy písma a různými úrovněmi kvality obrazu byly shromážděny cenné informace, které budou užitečné při dalším vývoji aplikace a vylepšení jejich funkcionality.

V případech, kdy byla kvalita obrazu normální, aplikace předvedla dostačující výsledky. Testování načítání ručně psaných čísel dopadlo uspokojivě. Aplikace byla schopna úspěšně rozpoznat většinu čísel z vybraných testovacích sad, a tím potvrdila schopnost fungovat efektivně v typických podmínkách.

Avšak když se kvalita obrazu snížila a obraz byl výrazněji kostičkovaný, kvalita rozpoznávání rapidně klesla. To naznačuje, že aktuální algoritmus pro rozpoznání čísel má určité omezení, pokud jde o práci s nižší kvalitou obrazu.

Dalším zjištěným problémem byla nestabilita výsledků při horší kvalitě obrazu. Bylo zjištěno, že výsledek rozpoznání se často střídal mezi správným a nesprávným. To bylo zvláště problematické, protože algoritmus v současné době neobsahuje žádnou metodu pro preferování výsledků, které se častěji vyskytují po provedení OCR. Tento problém může být řešen implementací technik strojového učení, které mohou rozpoznat a upřednostnit konzistentnější výsledky.

Informace zjištěné při testovaní mohou být užitečné při dalším vývoji aplikace a mohou pomoci vytvořit robustnější a přesnější aplikaci pro rozpoznávání čísel. Je důležité

zdůraznit, že ačkoliv se při testovaní narazilo na určité výzvy, tato zkušenost poskytla cenný náhled na to, jak by se dalo aplikaci vylepšit a na co je potřeba dát vetší pozor.

7 PŘÍPADOVÁ STUDIE – OPTIMALIZACE USTAVOVÁNÍ VÝKOVKŮ

V této kapitole je demonstrován potenciál aplikace pro rozpoznání číselných dat pro konkrétní praktické úlohy. Případová studie je zaměřena na proces ustavování výkovků, konkrétně na vyrovnání obrobku tak, aby jeho horní stěna byla vodorovná.

7.1 Specifika úlohy vyrovnání obrobku v průmyslovém prostředí

Tato podkapitola se věnuje podrobnému rozebrání vybrané průmyslové úlohy, konkrétně vyrovnání obrobku a vysvětluje, proč je tento proces tak zásadní v mnoha průmyslových aplikacích.

Vyrovnání obrobku, neboli ustavování výkovků, je proces, kde se velký ocelový kvádr má upnout tak, aby jeho horní stěna byla vodorovná. Z hlediska průmyslové výroby má tento proces značný význam. Přesné vyrovnání obrobku je nezbytné pro další zpracování, jako je například frézování, obrábění, vrtání či svařování. Pokud není obrobek správně vyrovnaný, mohou být následné operace narušeny a výsledný produkt může být vadný, což může způsobit vysoké náklady na opravy nebo výrobu nového kusu.

Pro správné vyrovnání obrobku je nezbytné provést přesné měření x, y, z souřadnic bodů na horní stěně obrobku. Tyto hodnoty se zobrazují na displeji měřicího přístroje. Změřené hodnoty poté poskytují informace o poloze a orientaci obrobku v prostoru.

Cílem je dosáhnout toho, aby všechny hodnoty na vertikální ose z byly co nejvíce stejné - to by znamenalo, že horní stěna obrobku je vodorovná. Pro dosažení tohoto cíle je nutné určit, jaký úhel rotace kolem os x a y je potřebný. Tento úkol je komplikovaný a časově náročný, zejména pokud se provádí manuálně, a vyžaduje vysokou úroveň odborných znalostí a zručností.

Na tomto místě se stává evidentním, jak může digitální transformace a použití moderních technologií, jako je aplikace pro rozpoznání čísel, přinést významný přínos pro průmyslovou výrobu. Aplikace by mohla zjednodušit a zefektivnit proces vyrovnání obrobku tím, že by rychle a přesně načetla číselné hodnoty souřadnic z displeje měřicího přístroje a následně by na základě těchto hodnot vypočetla potřebný úhel rotace.

7.2 Postup využití aplikace pro řešení úlohy vyrovnání obrobku

Aplikace, jejíž přizpůsobení a využití je cílem této případové studie, je strukturována do několika fází, které postupně vedou k řešení konkrétní úlohy optimalizace ustavování výkovků. Tyto fáze jsou:

1. **Načítání dat**: Načítání čísel přes kameru je využíváno k zaznamenání souřadnic x, y, z, zobrazených na displeji měřicího přístroje. Aplikace je navržena tak, aby rozpoznala jednotlivé souřadnice a oddělila je pro další zpracování.

2. **Oprava chyb a analýza dat**: Po načtení dat jsou uživatelům poskytnuty možnosti pro úpravy a opravu případných chyb v datech. Dále je umožněno provést základní statistickou analýzu dat, která poskytuje uživatelům hlubší porozumění o orientaci obrobku.

3. **Optimalizace**: Na základě načtených dat je aplikací řešena optimalizační úloha, která spočívá v nalezení úhlu otáčení obrobku kolem osy x a y tak, aby byl rozptyl z-souřadnic co nejmenší po transformaci. K řešení této úlohy je využívána numerická metoda nejmenších čtverců.

4. **Grafická prezentace výsledků**: Po nalezení optimálního úhlu otáčení je aplikací poskytnuta vizualizace výsledků ve formě grafů. Tato vizualizace umožňuje uživatelům lépe pochopit, jak by se obrobek měl otáčet.

5. **Uložení a sdílení výsledků**: Na závěr jsou uživatelům poskytnuty možnosti pro uložení výsledků do CSV souboru a pro sdílení těchto výsledků pro další použití.

Celý tento proces je přizpůsoben tak, aby vyhovoval konkrétním potřebám průmyslového procesu ustavování výkovků, a přináší uživatelům přesné a efektivní řešení pro vyrovnání obrobku.

7.3 Modifikace a rozšíření aplikace pro optimalizaci ustavování výkovků

Ačkoli aplikace již disponuje základními funkcemi pro načítání a zpracování číselných dat, pro úspěšné řešení této konkrétní úlohy je nutné provést některá rozšíření a adaptace.

1. **Přizpůsobení načítání dat**: Načítání dat je klíčovou funkcí aplikace. Pro účely optimalizace ustavování výkovků je třeba tuto funkcionalitu rozšířit tak, aby bylo možné načítat trojice čísel, které reprezentují x, y, z souřadnice bodů na horní stěně obrobku. To vyžaduje značnou změnu v algoritmu zpracování vstupu. Uživatelské rozhraní také může vyžadovat aktualizaci, aby mohlo efektivně zobrazovat a správně interpretovat tato data.

2. **Implementace specifického algoritmu optimalizace**: Pro úlohu vyrovnání obrobku je třeba implementovat specifický algoritmus optimalizace. Tento algoritmus by měl být schopen provést matematickou operaci rotace bodů v 3D prostoru a najít takový úhel rotace, který minimalizuje rozptyl z-souřadnic. Metoda nejmenších čtverců může být využita jako numerická metoda pro řešení této optimalizační úlohy.

3. **Zlepšení přesnosti načítání dat**: Přesnost načítání čísel z displeje měřicího přístroje je kritická pro úspěšné vyrovnání obrobku. Jakákoli nepřesnost v načítání číselných dat může výrazně ovlivnit přesnost výsledné rotace obrobku. Aplikace by tedy měla být dále testována a optimalizována pro zlepšení přesnosti načítání číselných dat.

4. **Stanovení optimálního počtu měřených bodů**: Další důležitou otázkou je určení optimálního počtu bodů, které je třeba načíst pro dosažení nejlepších výsledků. Je třeba provést analýzu, která určí, kolik bodů je nezbytných pro dosažení požadované přesnosti bez zbytečného prodlužování času potřebného pro načítání a zpracování dat.

Tyto úpravy a rozšíření aplikace by měly poskytnout uživatelům efektivní nástroj pro optimalizaci procesu ustavování výkovků, čímž by mohly být značně sníženy náklady na výrobu a zvýšena kvalita výrobků.

ZÁVĚR

V rámci této diplomové práce byla provedena podrobná analýza oblasti optického rozpoznávání znaků, známého jako OCR (Optical Character Recognition), s hlavním zaměřením na využití této technologie v kontextu mobilních aplikací pro načítání a zpracování číselných dat. Detailní prozkoumání OCR, jeho historie, principů fungování, technik a aplikací, bylo doplněno studiem současných trendů a nástrojů v oblasti vývoje mobilních aplikací. Tento teoretický základ poskytl hluboké a komplexní pochopení dané problematiky, což bylo klíčové pro následující praktickou část práce.

Praktická část byla zaměřena na vytvoření mobilní aplikace, jejímž úkolem bylo efektivní využití technologie OCR pro načítání a zpracování číselných dat z fotografie. Tato aplikace byla navržena a implementována s ohledem na optimální využití možností OCR technologie, a tím dosažení robustnosti a efektivity. Výsledky testování potvrdily dostatečnou úroveň přesnosti a spolehlivosti aplikace při zpracování různých testovacích sad dat. Stále ale tato technologie má určíte nedostatky a člověk stále rozpoznává znaky o něco lépe než aplikace.

Přestože bylo dosaženo významných výsledků a cíl práce byl naplněn, otevírají se stále nové možnosti pro další vývoj a vylepšení. To může zahrnovat integraci funkcí pro skenovaní vice čísel najednou nebo rozšíření analytických nástrojů a funkcí aplikace.

V závěru práce je představena případová studie, kde je aplikace využita pro optimalizaci ustavování výkovků v průmyslovém prostředí. Studie ukazuje, jak může být aplikace upravena a rozšířena pro specifické průmyslové použití.

Výsledkem této práce je funkční mobilní aplikace, která demonstruje uplatnění technologie OCR v praxi a potenciál pro další rozšíření a optimalizaci. Výsledky testování ukazují, že aplikace je schopna efektivně načítat a zpracovávat čísla z obrázků s vysokou úrovní přesnosti. Případová studie ukazuje, jak může být aplikace upravena a využita v průmyslovém prostředí pro řešení specifických úloh.

SEZNAM POUŽITÉ LITERATURY

[1] CHRISTENSSON, Per: OCR Definition[online]. https://techterms.com/definition/ocr, 16.04.2018, [cit. 2023-04-15].

[2] Mittal, R.; Garg, A.: Text extraction using OCR: A Systematic Review. In *2020 Second International Conference on Inventive Research in Computing Applications (ICIRCA)*, 2020, s. 357–362, doi:10.1109/ICIRCA48905.2020.9183326.

[3] Schantz, H. F.: *The history of OCR, optical character recognition.* [Manchester Center, Vt.] : Recognition Technologies Users Association, 1982, ISBN 9780943072012, [cit. 2023-04-15].

[4] SpeechOcean: The Future of OCR Technology: Advancements and Applications [online]. https://en.speechocean.com/Cy/921.html, 2023-04-19, [cit. 2023-04-15].

[5] Eikvil, L.: Optical Character Recognition. 1993, [cit. 2023-04-15].

[6] Data@Urban: Choosing the Right OCR Service for Extracting Text Data [online]. https://urban-institute.medium.com/choosing-the-right-ocr-service-for-extracting-text-data-d7830399ec5, 2022-03-25, [cit. 2023-04-15].

[7] Rao, N.; Sastry, A.; Chakravarthy, D.; aj.: Optical character recognition technique algorithms. ročník 83, 01 2016: s. 275–282, [cit. 2023-04-15].

[8] Alotaibi, F.; Abdullah, M.; Abdullah, R.; aj.: Optical Character Recognition for Quranic Image Similarity Matching. *IEEE Access*, ročník PP, 11 2017: s. 1–1, doi:10.1109/ACCESS.2017.2771621, [cit. 2023-04-15].

[9] Michael Allius: Android Vs iOS: Which Is An Ideal Platform For Mobile App Development [online]. https://www.linkedin.com/pulse/android-vs-ios-which-ideal-platform-mobile-app-2021-michael-allius/, 2021-01-05, [cit. 2023-04-15].

[10] Statista: How many smartphones are in the world? [online]. https://www.statista.com/topics/840/smartphones/topicOverview, 2023, [cit. 2023-04-15].

[11] Charles McLellan: Smartphone trends in 2023: Here's what's coming next [online]. https://www.zdnet.com/article/smartphone-trends-in-2023-heres-whats-coming-next/, 2023-02-17, [cit. 2023-04-15].

[12] Erin Gilliam Haije: Top 20 Mobile App Development Tools: An Overview [online]. https://mopinion.com/mobile-app-development-tools-an-overview/, 2019-07-08, [cit. 2023-04-15].

[13] javinpaul: Top 5 Programming languages for Mobile App Development in 2023 [online]. https://medium.com/javarevisited/top-5-programming-languages-for-mobile-app-development-in-2021-19a1778195b8, 2021-03-06, [cit. 2023-04-15].

[14] Apoorva Bellapu: Top 10 Most In-Demand Programming Languages for Mobile App Development [online]. https://www.analyticsinsight.net/top-10-most-in-demand-programming-languages-for-mobile-app-development/, 2023-03-19, [cit. 2023-04-15].

[15] CST Studio Suite 3D EM simulation and analysis software [online]. https://developer.android.com/studio, [cit. 2023-04-15].

[16] Xcode 14 [online]. https://developer.apple.com/xcode/, [cit. 2023-04-15].

[17] Documentation for Visual Studio Code [online]. https://code.visualstudio.com/docs, [cit. 2023-04-15].

[18] IntelliJ IDEA – the Leading Java and Kotlin IDE [online]. https://www.jetbrains.com/idea/, [cit. 2023-04-15].

[19] Flutter [online]. https://flutter.dev, [cit. 2023-04-15].

[20] React Native · Learn once, write anywhere [online]. https://reactnative.dev, [cit. 2023-04-15].

[21] Xamarin: Open-source mobile app platform for .NET [online]. https://dotnet.microsoft.com/en-us/apps/xamarin, [cit. 2023-04-15].

[22] Apache Cordova [online]. https://cordova.apache.org, [cit. 2023-04-15].

[23] Joe Stangarone: Mobile vs. Traditional Development: 7 Key Differences [online]. https://www.mrc-productivity.com/blog/2015/06/mobile-vs-traditional-development-7-key-differences/, 2015, [cit. 2023-04-15].

[24] ARROWCORE GROUP: Web, Mobile or Desktop App – Which is Right for Your Project? [online]. https://arrowcore.com/blogs/web-mobile-or-desktop-app-which-is-right-for-your-project/, 2021-10-14, [cit. 2023-04-15].

[25] Visual Studio IDE and Code Editor for Software Developers and Teams [online]. https://visualstudio.microsoft.com, [cit. 2023-04-15].

[26] Eclipse Desktop & Web IDEs: The Eclipse Foundation [online]. https://www.eclipse.org/ide/, [cit. 2023-04-15].

[27] PyCharm: the Python IDE for Professional Developers by JetBrains [online]. https://www.jetbrains.com/pycharm/, [cit. 2023-04-15].

[28] .NET: Build. Test. Deploy. [online]. https://dotnet.microsoft.com/en-us/, [cit. 2023-04-15].

[29] JavaFX [online]. https://openjfx.io, [cit. 2023-04-15].

[30] QT: Tools for Each Stage of Software Development Lifecycle [online]. https://www.qt.io, [cit. 2023-04-15].

[31] The GTK Project - A free and open-source cross-platform widget toolkit [online]. https://www.gtk.org, [cit. 2023-04-15].

[32] Swift.org [online]. https://www.swift.org, [cit. 2023-04-15].

[33] The Swift Programming Language [online]. https://docs.swift.org/swift-book/documentation/the-swift-programming-language/, [cit. 2023-04-15].

[34] UIKit [online]. https://developer.apple.com/documentation/uikit, [cit. 2023-04-15].

[35] SwiftUI [online]. https://developer.apple.com/documentation/swiftui, [cit. 2023-04-15].

[36] Vision [online]. https://developer.apple.com/documentation/vision, [cit. 2023-04-15].

[37] AVFoundation [online]. https://developer.apple.com/documentation/avfoundation/, [cit. 2023-04-15].

[38] Swift Charts [online]. https://developer.apple.com/documentation/charts, [cit. 2023-04-15].

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

OCR	Optical Character Recognition
JPEG	Joint Photographic Experts Group
IDE	Integrated Development Environment
GUI	Graphical User Interface
API	Application Programming Interface
ARC	Automatic Reference Counting
RIA	Rich Internet application
GPS	Global Positioning System
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
AR	Augmented Reality
VR	Virtual Reality
IoT	Internet of things
CSS	Cascading Style Sheets
RTF	Rich Text Format
HTML	HyperText Markup Language
CSV	Comma Separated Values
MVVM	Model-View-ViewModel
KVO	Key-Value Observing

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

PŘÍLOHA P I. PŘILOŽENÉ CD

Přiložené CD obsahuje:

- diplomovou práci ve formátu PDF
- zdrojové kódy aplikace