Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Факультет інформатики
Кафедра інформатики

РОЗРОБКА ФРЕЙМВОРКУ ДЛЯ МОДИФІКАЦІЇ ВЗАЄМОДІЇ КОРИСТУВАЧІВ З МОБІЛЬНИМИ ПРИСТРОЯМИ

Текстова частина до магістерської роботи
за спеціальністю 122 «Комп'ютерні науки»

Керівник магістерської роботи
к. ф-м. н., доцент
Нагірна А. М.

_____
(підпис)
"____"_____ 2023 р.
Виконав студент МП КН-2
Сабадишин М.О.

_____
(підпис)

"____" _____ 2023 р.

Київ-2023

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав. Кафедри інформатики,
доцент, к.ф-м.н.
Гороховський С.С.
_____
(підпис)
“____” _____ 2023 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на магістерську роботу

Студента Сабадишина Максима Олександровича факультету інформатики 2 курсу

ТЕМА розробка фреймворку для модифікації взаємодії користувачів з мобільними пристроями

Зміст ТЧ до магістерської роботи:
Календарний план
Вступ
Аналіз предметної області та дослідження її актуальності
Теоретичні відомості щодо використаних технологій
Огляд практичної реалізації фреймворку
Висновки по роботі
Список джерел

Дата видачі “____” _____ 2023 р. Керівник _____
(підпис)

Календарний план виконання роботи:

Тема: розробка фреймворку для модифікації взаємодії користувачів з мобільними пристроями

№	Назва етапу	Термін виконання	Примітка
1.	Отримання теми роботи	30.11.2022	
2.	Ознайомлення з предметною областю	20.01.2023	
3.	Визначення способу вирішення задачі	01.02.2023	
4.	Пошук конкурентів, аналіз їх особливостей	15.02.2023	
5.	Збір та визначення технологій для розробки фреймворку	17.02.2023	
6.	Вивчення теоретичної частини	26.02.2023	
7.	Створення архітектури фреймворку	20.03.2023	
8.	Визначення додаткового функціоналу	05.04.2023	
9.	Розробка фреймворку	10.04.2023	
10.	Аналіз створеного рішення з точки зору ефективності та його впливу на систему	20.05.2023	
11.	Планування наступних кроків та аналіз зон для покращення	24.05.2023	
12.	Оформлення роботи	27.05.2023	

Студент Сабадишин М.О.

Керівник Нагірна А.М.

“____” _____

ЗМІСТ

Анотація

Метою кваліфікаційної роботи є розробка фреймворку, що полегшує створення додатків на базі iOS з урахуванням потреб людей з вадами здоров’я шляхом реалізації керування зором. Робота містить аналіз ринку для пошуку існуючих рішень, їх особливостей, сильних та слабких сторін. В результаті базуючись на розумінні існуючих інструментів запропоновано своє рішення, що дозволяє інтегрувати керування зором у нові та існуючі додатки.

Ключові слова: ARKit, iOS, керування зором, інклюзивність, фреймворк, Swift Package Manager, Swift, UIKit.

Вступ

Сьогодні мобільні пристрої посідають важливе місце в житті людей. Смартфони виконують задачі різного характеру і за різноманіттям функціонала є чи не найбільш багатим пристроєм серед тих, які люди щоденно використовують.

Враховуючи, що вони є невід'ємною частиною життя людей абсолютно різних груп, важливим є однаково рівне право кожної людини мати доступ до можливостей мобільних пристроїв.

Окремою категорією, яку варто розглядати, є люди з обмеженими можливостями. Розробка продуктів часто спрямована на загальну аудиторію, що не потребує додаткової уваги при створенні додатку, в той час як різні фізичні обмеження можуть ускладнювати користування звичними для нас речами. Мобільні пристрої не виключення.

Попри те що більшість людей з обмеженими можливостями використовує смартфони, багато додатків не є повноцінно адаптованими під використання ними. Велику частину адаптації та зручного користування смартфоном виконує Apple у своєму iPhone, проте все ще залишається частина роботи для розробників стороннього програмного забезпечення, яка часто залишається незавершеною.

Актуальність роботи полягає у великій кількості людей з вадами моторики руху, багато з яких використовують смартфони, та відсутності методів керування додатками для них. Дослідження проведене в даній роботі надає змогу розробляти мобільні застосунки стороннім розробникам з врахуванням даної категорії людей та збільшувати кількість інклюзивних додатків на платформі.

Метою роботи була поставлена розробка фреймворку, що модифікував би взаємодію з мобільними пристроями на iOS та полегшував досвід користування цьому сегменту користувачів.

Завданням роботи є аналіз існуючих рішень на ринку у схожій предметній області, дослідження та вибір модифікації взаємодії, а також розробка фреймворку, що реалізує обрану модифікацію.

Об’єктом дослідження є методи взаємодії користувачів з додатками на мобільних пристроях.

Методи дослідження складаються з порівняльного аналізу, експериментів, розрахунків та вимірювань, і опису.

Джерела дослідження включають в себе технічну документацію про використані технології та підходи, інтернет-ресурси, статистичні дані в предметній області, та звіти по стану ринку.

Наукова новизна полягає у тому, що створений фреймворк реалізує управління для підключеного до нього додатку без суттєвих ресурсів сторонніх розробників, а також надає допоміжні інструменти для кращої інтеграції.

Практичне значення роботи – це можливість інтегрувати керування зором у додатки під управлінням iOS та збільшувати кількість інклюзивних застосунків на платформі.

Робота складається з трьох розділів.

В розділі 1 оглянуто різні сфери використання керування зором, оглянуто цільову аудиторію такого способу взаємодії з пристроєм. Також досліджено як Apple працює з інклюзивними користувачами сьогодні та які можливості надає. Даний розділ також охоплює огляд ринку та існуючих рішень, їх плюси та мінуси.

Розділ 2 містить в собі вибір технологій для розробки, їх теоретичні відомості, аналіз технічних можливостей до розробки такого функціонала сьогодні.

Розділ 3 описує процес розробки фреймворку, його обмеження та можливості, аналіз готового рішення з точки використання ресурсів та впливу на систему, а також окреслює точки росту та подальші кроки в його розвитку.

Створено фреймворк, що модифікує взаємодію користувачів з мобільними пристроями додаючи можливість керування зором в сторонніх додатках.

Розділ 1. Аналіз предметної області та дослідження її актуальності

1.1 Важливість досліджень в предметній області

Станом на 2022 рік мобільні пристрої використовуються більшою частиною населення планети, маючи 8.6 мільярдів активних мобільних з'єднань, що можна співставити як 1 з'єднання на людину [1]. За такого глобального використання варто також зазначити, що вони вирішують широкий спектр різних проблем. До прикладу, розглянемо дослідження проведене Deloitte у 2020 році, у якому було порівняно різні сценарії використання смартфонів, ноутбуків, планшетів, а також настільних комп'ютерів [2]. Порівняння було проведено серед різних вікових категорій та статей. Можна побачити, що телефон є найпопулярнішим засобом для використання у майже всіх оглянутих категоріях серед усіх категорій користувачів. Виключенням є лише онлайн-покупки та онлайн пошук, який є більш використовуваним у віковій категорії 25-34.

Звісно, що серед сотень мільйонів користувачів мобільних пристроїв є також люди з обмеженими можливостями.

У 2016 році «Rehabilitation Engineering Research Center for Wireless Technologies» провели опитування серед людей з вадами здоров'я для того аби зрозуміти рівень використання мобільних пристроїв ними [3]. Згідно з результатами опитування, 84% з тих хто прийняв в ньому участь є користувачами смартфонів, і 70% використовує мобільні додатки. Враховуючи тренди та розвиток мобільних пристроїв можна припускати, що на сьогодні ще більший відсоток людей з вадами здоров'я користуються смартфонами.

Розглядаючи питання поширеності використання смартфонів людьми з вадами здоров'я в площині абсолютних значень можемо звернутись до дослідження 2018 року проведеного "American Community Survey" згідно з яким 12.6% американців, а саме 40.6 мільйонів мали ту чи іншу ваду здоров'я [4]. Всі ці дані вказують на те, що сегмент таких користувачів по світу є великим і варто враховувати його при розробці нового програмного забезпечення.

1.2 Інструменти, що надаються Apple для людей з обмеженими можливостями

Apple приділяє багато уваги людям з обмеженими можливостями при розробці свого програмного забезпечення. iPhone, мобільний пристрій, що розроблений Apple, працює під управлінням ОС iOS, яка містить в собі багато інструментів, що модифікують взаємодію з пристроєм та полегшують роботу з ним для різних категорій людей [5].

Однією з найбільш суттєвих функцій, що підтримуються Apple є VoiceOver. Дана технологія полегшує роботу з пристроєм для людей з вадами зору читаючи контент на дисплеї по натисканню на нього. Також вона додає в ОС нові жести для керування нею з урахуванням того, що просте натискання на екран спричиняє озвучення контенту. Іншим функціоналом для людей з вадами зору є можливість збільшувати шрифт та загалом контент на пристрої.

Для людей з проблемами фокусування було додано технологію Guided Access, що обмежує по часу користування пристроєм або ж «замикає» користувача в певному додатку.

У комбінації з навушниками від Apple люди з вадами слуху можуть на певному рівні компенсувати собі недостачу слуху, що також суттєво допомагає у повсякденному житті.

Увесь наведений вище функціонал закриває потреби різних категорій людей з обмеженими можливостями. Проте варто зазначити, що у згаданому раніше опитуванні серед людей з обмеженими можливостями, 25% респондентів мали труднощі з використанням рук або пальців [3]. Apple в iOS надає певний функціонал, що закриває потреби несуттєвих обмежень. До прикладу, AssistiveTouch (рис. 1.1). Ця технологія додає віртуальну кнопку на екран смартфона, тим самим полегшуючи доступ до певних його функцій, що полегшує роботу з ним людям, які мають обмеження в моториці кінцівок.

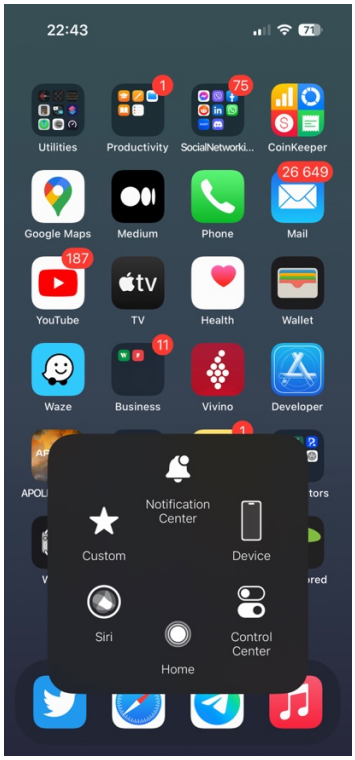


Рисунок 1.1 Приклад роботи AssistiveTouch

Попри те, що Apple закриває якісь з потреб людей, що мають проблеми зі звичайним підходом до управління мобільними пристроями, залишається велика кількість користувачів, що не отримують належної підтримки.

До прикладу, люди з травмами спинного мозку, церебральними паралічем, м’язовою дистрофією, з БАС, та іншими схожими хворобами не мають змоги користуватись телефоном у звичній для нас формі. Окрім цього, велика кількість людей зіштовхується з проблемами моторики рук, що також перешкоджає зручному користуванню смартфонами. Даний сегмент користувачів технології, що надаються Apple, до кінця не покривають, тим самим не надаючи можливість зручного та самостійного користування смартфоном.

Як один з доступних каналів комунікації між людьми та смартфоном присутній зоровий контакт. А враховуючи те, що сьогодні фронтальні камери смартфонів є досить потужними, потенційним розв’язаням проблеми користування мобільними пристроями для людей з вадами руху стало управління за допомогою зору.

З ймовірних причин відсутності керування зором як стандартної опції на iPhone можна виділити те, що попри якість сучасних камер та штучного інтелекту, точність контролю смартфона очима все ще далека від ідеальної. Враховуючи стандарти якості, які присутні в Apple, їх небажання вводити не до кінця відточений функціонал може блокувати розробку цієї технології в ОС. Іншою ймовірною причиною може бути потенційне зловживання технологією та її використання у небезпечних сценаріях, до прикладу, за кермом.

1.3 Аналіз наявних продуктів, що використовують трекінг погляду

Хоч на рівні ОС Apple не робила кроків по підтримці технології, певний рух в цю сторону був присутнім. Для iPhone жодних змін не було, проте користувачі iPad і відповідно iPadOS починаючи з версії 15.0 отримали змогу керувати пристроєм за допомогою зору. Дана підтримка присутня на системному рівні, проте крім самого iPad вимагає також стороннього пристрою – TD Pilot [6] (рис. 1.2).

Рисунок 1.2 TD Pilot з iPad [6]

Сам пристрій обладнаний інфрачервоними датчиками для відстежування руху очей, динаміком, та додатковим дисплеєм ззаду, і розроблений окремо для iPad. Також компанія що розробила його вже має експертизу у галузі й створювала схожі продукти для пристроїв під управлінням ОС Windows. Все це дало змогу за допомогою сторонньої компанії вирішити проблему управління

одного з пристроїв Apple людям, що не мають змоги керувати ним у звичний спосіб. Окрім них, компанія також займається розробкою програмного забезпечення для людей з вадами руху і має додаток TD Snap, що розроблений для полегшення комунікації між користувачами [7] (рис. 1.3).

Рисунок 1.3 Інтерфейс TD Snap

Така широка підтримка категорії користувачів, яку ми розглядаємо, є корисною для рівних можливостей керування мобільними пристроями. Проте у даної розробки є і недоліки. Вона не є доступною на iOS та лише підтримує iPad, також її немає у вільному для продажу доступі, а потенційним покупцям потрібно залишати заявку на придбання. Але найсуттєвішим недоліком такої системи є її фінансова доступність. Ціна набору може складати до $10.000, що є доволі великою сумою для середньостатистичного покупця.

Серед повноцінних інструментів, що надають можливість керування зором, є також продукт компанії eyeV під назвою Skyle [8]. Він є схожим на розглянутий раніше TDPilot і є не лише програмним забезпеченням, а і повноцінним фізичним продуктом. В ньому використані сторонні датчики для фіксації погляду, які йдуть інтегровані в чохол, що продається. Це дозволяє отримати вищу точність трекінгу, аніж за використання камери пристрою.

Skyle надає як можливість керування операційною системою iPad, так і власне програмне забезпечення. Основною цільовою аудиторією даного

продукту є люди з обмеженими можливостями, у яких немає можливості користуватись планшетом за допомогою рук. Це обумовлює те, що додаткові програми надають полегшений інтерфейс, а також функціонал, що є потрібним саме цій категорії користувачів. Окрему увагу було приділено освіті для дітей шкільного віку, тому в доступі є різні навчальні ігри.

Окрім власного програмного забезпечення, eyeV розробили SDK – інструментарій для сторонніх розробників. Він дозволяє розробляти програмне забезпечення для підтримки інструменту Skyle у сторонніх продуктах. Його вихідними даними є координати того куди в даний момент дивиться користувач. Дана розробка допомагає поширити можливості керування iPad за допомогою зору на інші програмні застосунки. Окрім підтримки Swift як мови програмування, розробники також створили інструментарій використовуючи мову програмування Dart. Також передбачено при цьому можливість використання будь-якої бажаної мови, адже у відкритий доступ надано протокол комунікації з Skyle.

На жаль, в процесі аналізу не було знайдено жодного програмного застосунку, що реалізував би у себе даний фреймворк.

Попри те, що ціна на Skyle є високою, у порівнянні з TDPilot варто відзначити, що він дешевший. Придбати Skyle можна за €3.000, що суттєво дешевше аніж його конкурент.

Також Skyle підтримує сторонні пристрої керування, як до прикладу фізичну кнопку. Це дозволяє використовувати його для наведення курсору, а фізичну кнопку для активації дії.

Іншим інструментом, де використовується керування зором є додаток Hawkeye Access [9] (рис. 1.4). Він надає інтерфейс веб-браузера для можливості користування ним користуючись лише зором. Сам додаток підтримує різні сценарії керування, можливості взаємодіяти з веб-сторінками на тому ж рівні, що і звичайний користувач.

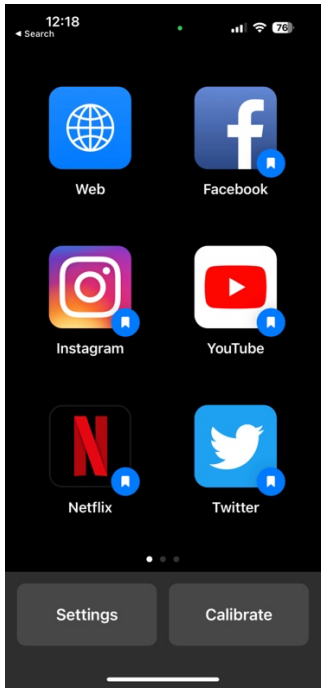


Рисунок 1.4 Інтерфейс HawkeyeAccess

Даний додаток є єдиним пропрацьованим рішенням, що присутнє в App Store, магазині додатків для смартфонів Apple.

Провівши аналіз досвіду користувача помітні певні недоліки даного застосунку, які були взяті до уваги при розробці власного рішення. Додаток має функціонал калібрації для того, щоб краще відстежувати точку в яку дивиться користувач. В середньому калібрація займає близько 30 секунд. Наявною проблемою є те, що вона необхідна перед кожним використанням додатка, що додає затримку та незручності для користувача. Інакшим знайденим нюансом є стандартні налаштування сервісу. За замовченням HawkeyeAccess як натискання на кнопку визначає погляд на неї протягом певного часу. Враховуючи те що система керування зором не є ідеальною часто доводиться витрачати певний час для того, щоб навести поглядом на необхідний елемент користувацького інтерфейсу. Неодноразово в користуванні додатком виникає ситуація, коли користувач в процесі наведення зором на елемент викликає спрацьовування дії цього елементу.

Така неінтуїтивна поведінка може призвести до великої кількості випадкових натискань на елементи екрану.

З сильних сторін HawkeyeAccess можна виділити те що додаток має багато способів конфігурації, насправді реалізує функціонал керування зором, та модифікує інтерфейс браузеру переписуючи стандартні елементи та додаючи нові задля більш зручного керування ними зором.

Також оглянутий додаток дозволяє використовувати різні елементи міміки як елементи управління, а саме посмішка чи кліпання очима.

Для підтримки рівня інклюзивності додатків HawkeyeAccess є дуже важливим елементом, проте даний проєкт вже не є в стадії активного розвитку. Згідно з інформацією присутньою на продуктовій сторінці в AppStore, останнє оновлення від розробників додатку відбулося два роки назад.

Використання технології слідкування за поглядом привертає увагу і бізнесу також. Розуміння як користувач сприймає контент на сайті чи додаткові може принести багато цінності в подальшому розвитку продукту, адже дає змогу проаналізувати його сильні та слабкі сторони. Одним з представників цієї категорії продуктів є SeeSo.

SeeSo [10] – платформа що дозволяє слідкувати за поглядом користувача, агрегувати зібрані дані, а також протягом сеансу користувача збирати іншу корисну для розробників аналітику. Даний сервіс – це потужний інструмент для маркетингу та продуктових команд, адже аналітика зібрана там найбільш корисна саме для них.

Як і HawkeyeAccess, оглянутий раніше, даний сервіс потребує калібрації для початку роботи, проте на відміну від HawkeyeAccess, калібрація займає мало часу і вимагає лише погляду по центру екрану.

Сам додаток SeeSoWebAnalysis, що присутній в AppStore, є скоріше прикладом можливостей продукту і дозволяє протестувати те як він працюватиме для вас, як потенційного клієнта.

Після перегляду обраної сторінки надається можливість отримати аналіз вашої поведінки на ній, а саме: кількість точок фокусу на екрані, теплову карту погляду, та графіки, що візуалізовано показують позицію погляду протягом часу користування, а також періоди фокусу (рис. 1.5).

Рисунок 1.5 Аналіз сторінки Wikipedia.com в додатку SeeSo. Зліва направо: загальна аналітика, теплова карта, графіки координат в часі

Проте основним продуктом є SDK – набір інструментів для розробників, що дозволяє інтегрувати даний сервіс до себе в додаток або на сайт та користуватись зібраною аналітикою серед своїх користувачів.

Вихідними даними роботи даного фреймворку є також координати куди дивиться користувач. Тому за бажання розробника даний аналітичний інструмент можна адаптувати та використовувати дані з нього для додавання можливості керування пристроєм.

Резюмуємо сильні сторони сервісу. По-перше, це надання глибокої аналітики по поведінковим шаблонам користувача, що дозволяє краще його зрозуміти та краще налаштовувати бізнес-процеси зі сторони продукту.

По-друге, вимоги до пристрою: SeeSo потребує iOS вище версії 11.0, що забезпечує підтримку всіх сучасних смартфонів від Apple, починаючи з iPhone 6s.

Також інтеграція даного сервісу у власний продукт описана в документації та не є складною. Крім цього, є підтримка й інших операційних систем, а також настільних комп'ютерів, що дозволяє розширити зону аналізу.

Звісно, крім сильних сторін, даний продукт має і свої слабкі місця. Одним з таких є те, що він не є безкоштовним. Присутня безкоштовна 30-денна версія, проте для повноцінного користування платформою необхідно сплачувати за підписку. Розробник не надає ціни у відкритому доступі. Також попри легкість інтеграції варто зазначити, що даний сервіс перш за все про аналітику користувачів, а не покращення інклюзивності, тому жодних змін в інтерфейсі вашого додатку чи керуванні пристроєм не є передбачено. Окрім цього, він вимагає від користувача постійного перебування в мережі інтернет, в офлайн режимі надання інформації не є можливим.

Розглянуто різні продукти що імплементують можливість слідкування за поглядом користувача, а також ті, які крім цього надають можливість управління додатком.

TDPilot надає високу якість трекінгу, а також повноцінну інтеграцію не тільки в окремих додатках, а й у всій операційній системі. Крім цього надається окреме програмне забезпечення, що полегшує комунікацію та інші процеси для людей з обмеженими можливостями. Недоліком даного продукту є його висока ціна та те, що він є присутнім лише на iPad.

Skyle є схожим за призначенням продуктом до TDPilot. Його виділяє значно нижча ціна, а також наявність відкритого фреймворку. Незважаючи на його доступність лише на одному з пристроїв Apple, за наявності даного iPad у користувача Skyle закриває функціонал керування використовуючи погляд, а підтримка інших сторонніх пристроїв управління дозволяє комбінувати та персоналізувати процес управління планшетом під себе.

HawkeyeAccess надає можливість керування зором і на пристроях під управлінням iOS. Багато переписаних і доданих нових елементів користувацького інтерфейсу полегшують керування зором, а наявність великої кількості конфігурацій дозволяє за бажання налаштувати продукт під себе. Дане рішення – найбільш універсальний наразі продукт для користування людям з обмеженими можливостями в рамках iOS. З його недоліків можна відзначити постійну необхідність калібрування, а також те, що налаштування за замовчуванням призводять до великої кількості фантомних натискань.

Інструмент від VisualCamp під назвою SeeSo використовує трекінг очей в іншому напрямкові проте також надає координати погляду користувача на екрані. Його сильною стороною є великий обсяг аналітики, що йде в супроводі з трекінгом. З мінусів можна виділити те, що це платний інструмент, а також, що він не створювався напряму для забезпечення хорошого користувацького досвіду людям з обмеженими можливостями.

1.4 Аналіз інших потенційних зон для використання керування поглядом

Можна побачити, що можливість зчитування погляду користувача застосовується не тільки для інклюзивності. Саме тому було оглянуто й інші потенційні місця застосування даного функціоналу.

В аналізі існуючих рішень на ринку було оглянуто SeeSo, що надає можливість слідкування за поглядом користувача через призму маркетингових досліджень та кращого розуміння того, як користувачі використовують той чи інший аспект продукту. В дослідженнях дані про погляд підкріплюють різного роду продуктові гіпотези та допомагають тим хто їх проводить.

Керування очима корисне людям і без фізичних обмежень здоров’я, адже надає функціонал управління пристроєм у випадку коли можливість взаємодіяти з пристроєм руками є відсутньою. Часто такі ситуації виникають в побутових сценаріях і існування такого функціоналу допоможе вирішити їх. До прикладу, перемкнути пісню під час миття посуду, або відкрити наступний пункт рецепту під час приготування їжі. Іноді ж керування поглядом це питання зручності. До

прикладу, читаючи книгу чи веб-сайт отримувати автоматичне прогортання сторінки вниз у випадку, якщо погляд сфокусовано в нижній її частині.

Добре розроблена система по управлінню очима, що інтегрована в операційну систему, може бути використана і для водіїв. Її задача в цьому сценарії не дати можливість керування транспортним засобом під час руху, а навпаки, відслідковувати напрям зору водія і попереджати в разі якщо зафіксовано відволікання уваги від керування. Іншим сценарієм слідкування за зором може бути написання екзаменів у віддаленому форматі. До прикладу, на онлайн екзаменах часто присутні екзаменатори, що слідкують за студентом, який здає їх, на предмет потенційного списування. Наявність системи керування зором могла б повідомляти про потенційні сценарії списування і привертати до них увагу екзаменатора, таким чином частково автоматизуючи його роботу.

1.5 Висновки до розділу 1

В даному розділі було оглянуто предметну область та обґрунтовано її актуальність. Сьогодні присутня велика кількість людей з обмеженими можливостями руху кінцівок, а даний сценарій роботи не є розповсюдженим в додатках на базі iOS. На рівні операційної системи потреба не є закритою взагалі, і лише окремі додатки надають функціонал по керуванню зором. Додатки, що його надають, є спеціалізованими саме для цього сценарію використання і найчастіше модифікують лише доступ до веб-ресурсів. Спеціалізовані прилади для надання можливості керувати зором існують, проте не є фінансово доступними та підтримують переважно iPad.

В процесі огляду не було знайдено жодного популярного додатку на базі iOS що як альтернативне керування надавав би можливість використовувати зір. Незважаючи на поточну не ідеальність точності роботи даного рішення, для людей з обмеженими можливостями воно надавало б єдиний можливий сценарій роботи з популярними застосунками, а його відсутність унеможливлює керування для багатьох.

Також було оглянуто інші сценарії використання даного способу керування смартфоном. Було знайдено декілька різних випадків в яких наявність такого керування полегшила б виконання задач.

Підсумовуючи, розробка продукту, що дав би змогу користуватись смартфоном на базі iOS за допомогою зору є актуальною проблемою, вирішення якої допоможе в багатьох сценаріях роботи з пристроями.

Розділ 2. Теоретичні відомості щодо використаних технологій

2.1 Аналіз можливостей для імплементації слідкування за поглядом

В попередньому розділі було оглянуто різні імплементації управління зором для мобільних пристроїв. Пори те що розглянуті застосунки часто вирішували різні сценарії роботи користувача або ж використовувались для збору аналітики, всі вони містили в собі трекінг погляду користувача.

Враховуючи важливість точного та добре оптимізованого модулю, що виконує відслідковування погляду користувача, було досліджено наявні можливості для його імплементації з точки зору технологій та підходів.

Одним з найлегших варіантів з точки зору розробки є використання вже готового заліза та інструментарію. Такий, до прикладу, було оглянуто в попередньому розділі. Skyle надає відкритий доступ до свого інструментарію, що дозволяє реалізувати власне рішення базуючись на їх готовій інфраструктурі. Проте даний підхід має лише одну перевагу – простоту в розробці та точність слідкування. З іншого ж боку є дуже багато недоліків, такі як унеможливлення модифікації модуля по слідкуванню, підтримка лише одного пристрою, та необхідність стороннього обладнання.

Іншим підходом є використання власних моделей машинного навчання для естимації точки куди дивиться користувач. Даний підхід можна окремо поділити на дві категорії – зі зберіганням моделі на пристрої та зі зберіганням моделі в хмарі.

Розглянемо опцію зі зберіганням моделі локально. У 2017 році на своїй щорічній конференції для розробників Apple представила фреймворк під назвою CoreML [11]. Так як він є розробленим всередині Apple, то присутня широка підтримка операційних систем компанії, а саме: iOS, macOS, iPadOS, watchOS. Даний фреймворк доволі легко підключити в рамках екосистеми та він надає багато можливостей. Додаючи до переваг бути розробленим Apple є той факт, що його робота тісно інтегрована та оптимізована під пристрої, що створює перевагу в швидкості роботи. А моделі iPhone на базі процесорів A11 Bionic та

вище мають вбудований окремий компонент – процесор Neural Engine [12]. Його призначення в тому аби оптимізувати та пришвидшувати задачі, що пов’язані з нейронними мережами і відповідно покращувати ефективність моделей машинного навчання. Окремо від цього він працює як в режимі пікової продуктивності, так і в енергозберігаючому режимі, що дає змогу поєднувати різні формати роботи за потреби. Його наявність ще більше покращує швидкість та ефективність роботи моделей машинного навчання за умови використання CoreML.

Так як модель зберігається на пристрої, CoreML дозволяє обробляти всю інформацію локально на ньому, без використання з’єднання з мережею. Крім цього, всі дані користувача залишаються у нього на пристрої без передачі їх на сторонні сервери, що додає даним безпеки, а користувачам приватності.

Також CoreML дозволяє імпортувати попередньо треновані моделі з інших фреймворків, таких як TensorFlow [13]. Це відкриває розробникам велику кількість вже готових моделей для модифікації та покращень. Проте незважаючи на підтримку імпорту різних форматів, до недоліків фреймворку можна виділити те, що через обмеження до типів шару та розмірів моделі, CoreML не завжди дає можливість без додаткових модифікацій напряму імпортувати модель.

Проте загалом використання CoreML для імпорту на iPhone моделей машинного навчання навчених через TensorFlow чи PyTorch є зручним та оптимальним способом для розробки системи відслідковування погляду користувача на екран.

Іншим способом використання моделей є застосування фреймворку TensorFlow Lite [14]. Такий підхід дозволяє використовувати моделі TensorFlow без необхідності конвертувати їх в формат, що підходить для CoreML. Проте конвертацію все ж треба робити – у формат TensorFlow Lite. Завдяки ній зменшується розмір самої моделі та оптимізується її швидкодія на пристроях, що не мають багато ресурсів, такими як мобільні пристрої.

TensorFlow Lite має свій фреймворк, який можна підключити напряму до проекту iOS додатку, проте він підтримує лише підключення через такий

сторонній менеджер залежностей як CocoaPods [15] та систему збірки Bazel [16]. Поняття менеджерів залежностей та різницю між ними буде розглянуто в розділі 3. На даний момент CocoaPods залишається популярним рішенням, враховуючи велику кількість проектів які почали використовувати його до появи основного конкурента – Swift Package Manager [17]. Проаналізовано одне з опитувань 2022 року проведене JetBrains серед розробників, що використовують мови програмування Apple на предмет того який з двох продуктів розробники використовують [18] (рис. 2.1).

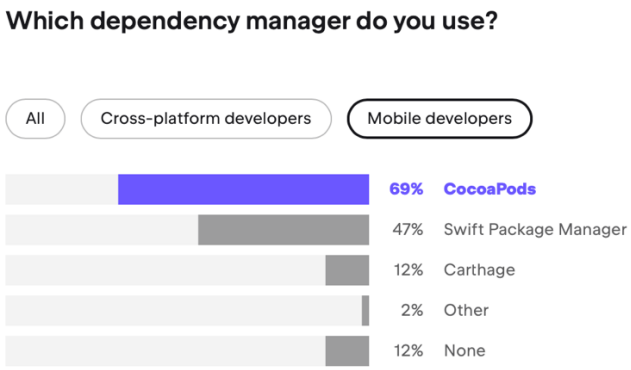


Рисунок 2.1 Статистика використання менеджерів залежностей [18]

Незважаючи на перевагу CocoaPods, багато розробників використовують Swift Package Manager, тому відсутність підтримки такого формату підключення фреймворку може викликати труднощі у великого відсотку користувачів.

З переваг даного підходу також можна виділити підтримку ним багатьох платформ, таких як Android, iOS, та вбудованих систем, а також надання API під різні мови програмування, до прикладу Swift, Objective-C, C++, Python. Це дає змогу, за наявності декількох продуктів, поєднувати підходи до розробки та перевикористовувати ті чи інші зміни.

Серед недоліків присутній той факт, що цей фреймворк не має такої великої бази користувачів, як до прикладу у повноцінного TensorFlow, що може ускладнити пошук проблеми пов’язаної конкретно з роботою фреймворку.

Також у 2019 році Google почав роботу над підтримкою повноцінного TensorFlow для Swift, що дозволяє повну інтеграцію фреймворку в мову, проте у

2021 році проект було офіційно закрито [19]. Тому при розробці функціоналу сьогодні даний розробники не зможуть використати переваги такої інтеграції.

Якщо розглядати підхід за якого модель буде зберігатись в хмарі, то питання вибору технології також лягає і на бекенд. Існує багато готових рішень з цього питання у великих гравців ринку, таких як Amazon, Google, Microsoft, Oracle. На сьогоднішній момент, Amazon є основним надавачем хмарних послуг, проте всі з вказаних вище сервісів забезпечують розробників достатньою інфраструктурою для виконання завдань в домені машинного навчання [20] (рис. 2.2).

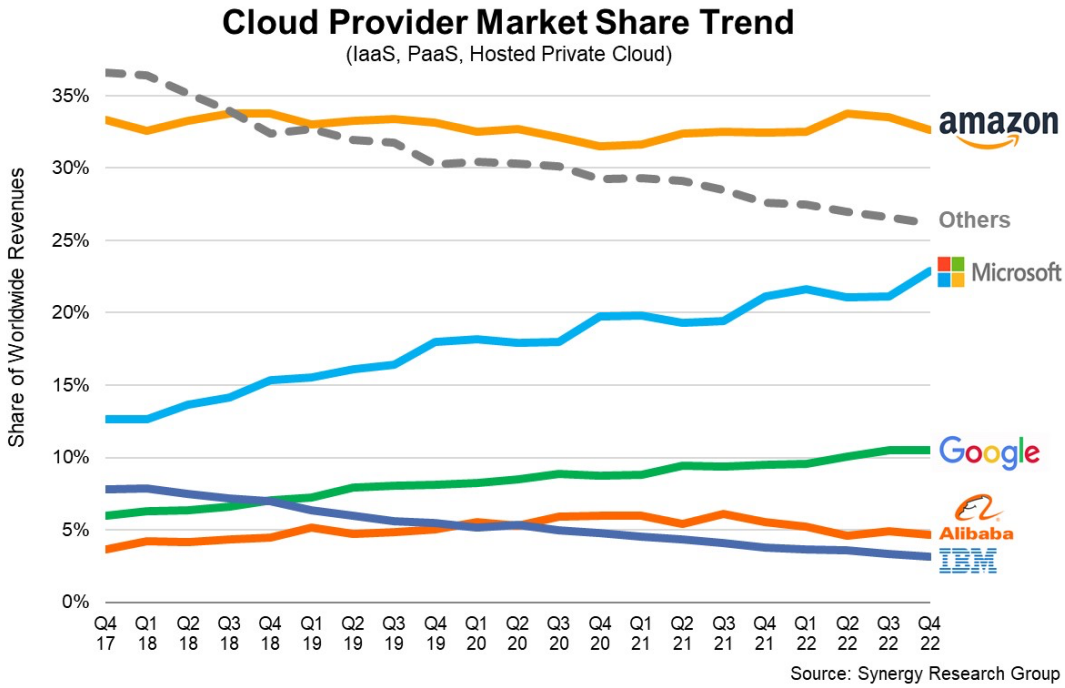


Рисунок 2.2 Розподіл ринку надавачів хмарних послуг [20]

Такі сервіси надають інфраструктуру, опрацьовування даних, їх зберігання, а також інструменти та фреймворки з машинного навчання. Це звільняє розробників від думок про розширення та підтримку інфраструктури і є зручним варіантом роботи.

Даний підхід має як свої переваги так і недоліки. З переваг можна виділити легку розширюваність. У разі якщо кількість запитів зростає або ж спадає, з хмарним рішенням можна швидко оптимізувати ресурси, що використовуються. З точки зору використання моделей машинного навчання даний підхід є

вигідним, адже у разі ускладнення моделі або збільшення кількості оброблюваних даних, обчислювальні потужності можна адаптувати.

Таке управління ресурсами добре впливає і на співвідношення ціни та ефективності. А саме, оплата йде лише за ресурси які використовуються. Окрім цього, хмарні рішення надають зручне місце для співпраці та можливості доступу до системи різним членам команди. Для команд з декількох людей це може значно полегшити процеси розробки.

Важливим з точки зору використання машинного навчання на смартфонах є також і те, що хмарні рішення надають можливість тренувати модель та вносити зміни без необхідності оновлень самого додатку.

Варто відзначити, що переваги хмарних рішень несуть за собою також і певні недоліки. Одним з таких є необхідність наявності інтернет підключення. Якість з’єднання впливає на те наскільки швидким є результат, що також може погіршувати якість роботи додатку. Крім цього, дані користувачів не є такими ж збереженими та приватними, адже необхідно надсилати їх на сервер для отримання результатів.

Отже, порівнюючи спосіб використання власних моделей машинного навчання бачимо, що хмарні технології дозволяють без нових версій додатку вносити зміни в модель, навчати її, але залежать від інтернет з’єднання та вимагають, хоч і оптимізованої, проте оплати за свої послуги. В той час збереження моделі на пристрої надає більш захищений спосіб взаємодії з даними користувача, відсутність залежності від інтернет з’єднання, проте вимагають більше ресурсів від самого пристрою, а також унеможливлюють зміну моделі та її тренування без релізу нової версії.

Комбінацією з точки зору ресурсів необхідних на розробку та можливості конфігурацій рішення слідкування за зором користувача є використання Apple ARKit [21]. Детальний огляд даного фреймворку буде надано пізніше в розділові, його основною функцією є надання можливості створювати додатки з підтримкою доповненої реальності. Він надає в собі функціонал, що дозволяє проектувати віртуальні об’єкти на потік відео з фронтальної або основної камер

девайсу. Окремі функції дозволяють отримувати інформацію про обличчя користувача, що дозволяє виділяти інформацію про стан погляду користувача та прогнозувати куди на екрані смартфона він дивиться в живому часі.

Даний підхід не вимагає роботи над моделлю машинного навчання, так само він не потребує її навчання та деплою в мобільний додаток. Все навчання було пророблено розробниками фреймворку, що дає змогу стороннім розробникам сфокусуватись на інших важливих деталях власних продуктів. Так само немає необхідності і у інтернет з'єднанні, всі операції відбуваються офлайн, а тому всі дані користувача увесь час залишаються локально на пристрої. Також даний фреймворк активно підтримується Apple та отримує оновлення з покращеною точністю видачі інформації і новими наборами даних.

Оглянувши всі три підходи щодо імплементації слідкування за зором користувача було вирішено використати можливості Apple ARKit для прогнозування поточного погляду користувача на екран смартфону. Таке рішення було прийнято виходячи з великої кількості даних на яких натренована модель, можливості широкої кастомізації вхідної інформації щодо стану погляду користувача, швидкості роботи, та безпеки даних.

2.2 Актуальні підходи в розробці під iOS та вибір мови програмування

Підходячи до написання фреймворку, що має працювати в програмах для iOS, постали різні запитання щодо того які саме функціональні обмеження на систему можуть бути, а які допустити не можна. Також стояло питання і основних технологій що використовуватимуться в самому рішенні.

Перше проаналізоване питання полягало в мінімальній версії, що має підтримуватись, а також пристроях, які мають бути включені в підтримку. Так як в огляді вже існуючих рішень по керуванню зором робився наголос на те, що деякі з них підтримують малу частку пристроїв, було важливим покрити максимально велику кількість девайсів. Також важливо було проаналізувати це питання через призму того, що опція з використанням ARKit вже сама ставить

певні обмеження на мінімально підтримувану версію iOS, а саме iOS 11.0. Було проаналізовано аналітику щодо розподілу версій, надану Apple [22] (рис. 2.3).

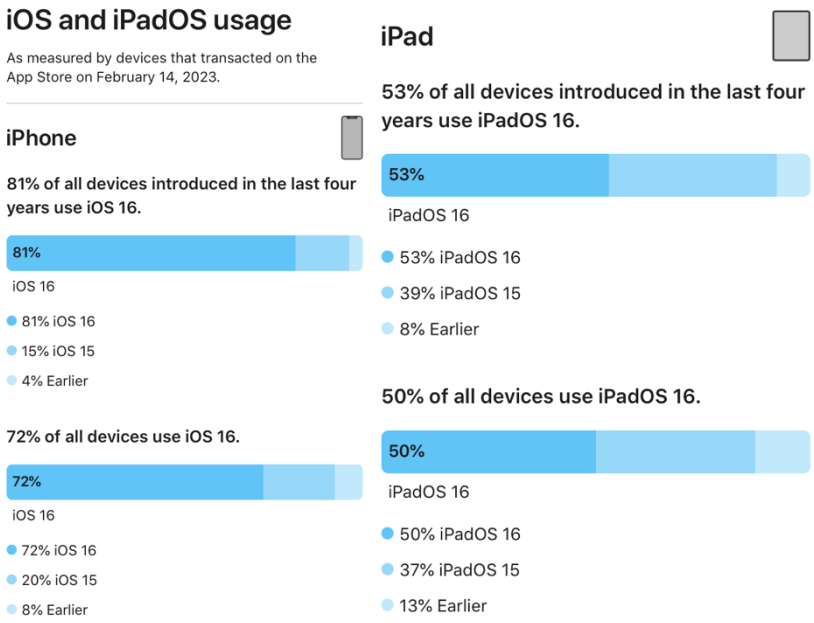


Рисунок 2.3 Статистика користування версіями iOS та iPadOS [22]

Згідно з нею бачимо, що серед всіх пристроїв на iOS 82% використовують iOS 16 та iOS 15. Серед тих 8% що залишились, скоріше за все більша частка використовує iOS 14 та iOS 13 згідно тренду. Схожа ситуація і на iPadOS. Якщо розглянути розподіл версій на пристроях, що були вироблені протягом останніх чотирьох років, то бачимо, що покриття актуальними версіями iOS та iPadOS там ще більше. Вважаємо, що згідно з розподілом версій iOS охоплюється більше 95% користувачів, що є прийнятним.

Також існує залежність і від моделі пристрою. Для відстеження очей фронтальна камера, що використовується, має бути з підтримкою технології TrueDepth [23], принцип роботи якої описано далі в розділі. Було оглянуто відповідну аналітику. Проаналізовано було графік наданий Mixpanel, згідно якого єдиним пристроєм серед найбільш вживаних сьогодні, який був вказаний в аналітиці та не підтримує технологію TrueDepth, є iPhone SE 2 Generation [24]. Його частка складає 2.75% за останні 3 місяці даних. Велику частину складає категорія Other, в яку було занесено різні менш використовувані пристрої. Серед

них також є ті, що підтримують TrueDepth, проте більш детальної частки вивести не вдалося (рис. 2.4).

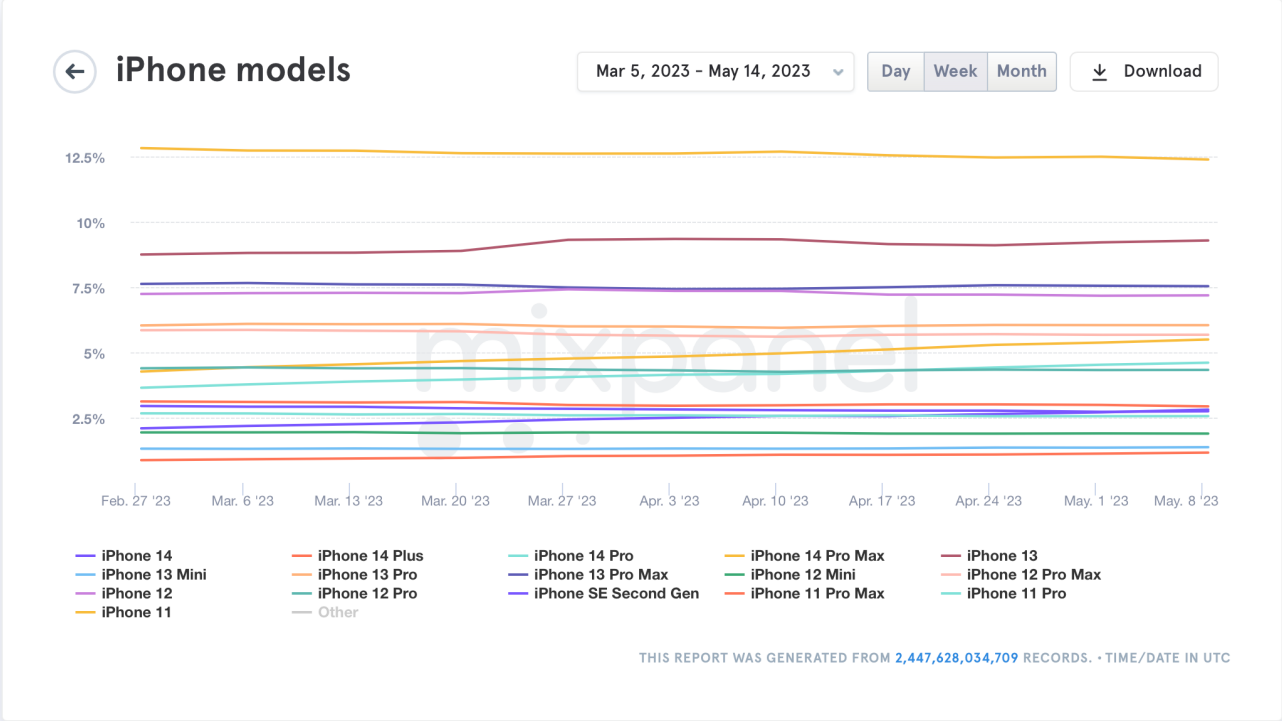


Рисунок 2.4 Звіт щодо частки найпопулярніших пристроїв [24]

Загалом, технологія TrueDepth присутня у iPhone починаючи з iPhone X, який було випущено в 2017 році. Враховуючи цей факт та оглянуту аналітику можна вважати, що і згідно розподілу по пристроям використання ARKit є виправданим та охоплює переважну більшість використовуваних смартфонів.

Наступним у визначенні підходів було використання мови програмування. Перед розробкою програмного забезпечення на iOS серед нативних мов програмування, а саме таких, що було створено та є підтримувано для конкретної платформи, вибір стоїть між Objective-C [25] та Swift [26].

Objective-C – це об’єктно-орієнтована мова загального призначення, що загально використовується для розробки програмного забезпечення під платформи Apple. Вона була розроблена в 1980 роках як розширення мови C з компонентами об’єктно-орієнтованого програмування. Однією з особливостей Objective-C є його динамічність, що дозволяє маніпулювати класами та об’єктами протягом роботи застосунку. На Objective-C написана велика кодова

база, він повністю інтегрується з кодом C та C++, а також інтероперабельний зі Swift.

Проте з недоліків присутній вищий за середній поріг входу, синтаксис мови не є спільним до більшості сучасних аналогів. Також відсутнє автоматичне управління пам’яттю, що може призвести до певних проблем та витоків пам’яті у разі некоректно написаного функціоналу.

З точки зору поширеності використання, то останні роки вона зменшується. Незважаючи на це, все ще дуже багато проектів написаних на Objective-C підтримуються, проте нові проекти найчастіше з’являються з підтримкою іншої мови програмування – Swift.

Swift – нова мова загального призначення, що так як і Objective-C створена Apple для розробки додатків для iOS, macOS, watchOS, та інших операційних систем продуктів компанії. Вперше Swift було представлено в 2014 році на WWDC – щорічній конференції для розробників від Apple, а реліз першої версії відбувся рік потому – в 2015. Ціль створення Swift була у виправленні фундаментальних недоліків Objective-C, а також в збільшенні популярності iOS розробки. Як зазначалось раніше в огляді Objective-C, його синтаксис створював доволі високий поріг входу новим розробникам, що не зовсім співвідноситься до продуктової стратегії Apple, яка передбачає в собі простоту. Swift, з іншого боку, був наділений більш зрозумілим синтаксисом, автоматичним керуванням виділеною пам’яттю, та строгою типізацією. Окремою сильною стороною Swift є те, що з моменту свого першого релізу він є мовою з відкритим програмним кодом, що дозволяє розробникам з усього світу долучатись до її покращень, а також надає можливість компілювати програмний код поза середовищем операційних систем Apple. Зараз Swift підтримується на всіх популярних ОС, існують фреймворки що дозволяють розробляти, до прикладу, бекенд функціонал використовуючи його.

Окрім загального порівняльного аналізу було поставлено задачу оцінити частку використання обох представників між собою. Наразі не існує прямого порівняння між цими двома мовами, тому для розуміння популярності було

використання індексу TIOBE [27]. Індекс TIOBE, що є акронімом до «The Importance Of Being Earnest», є системою виміру популярності мов програмування, який враховує як відносну одна одній популярність, так і їх частку ринку.

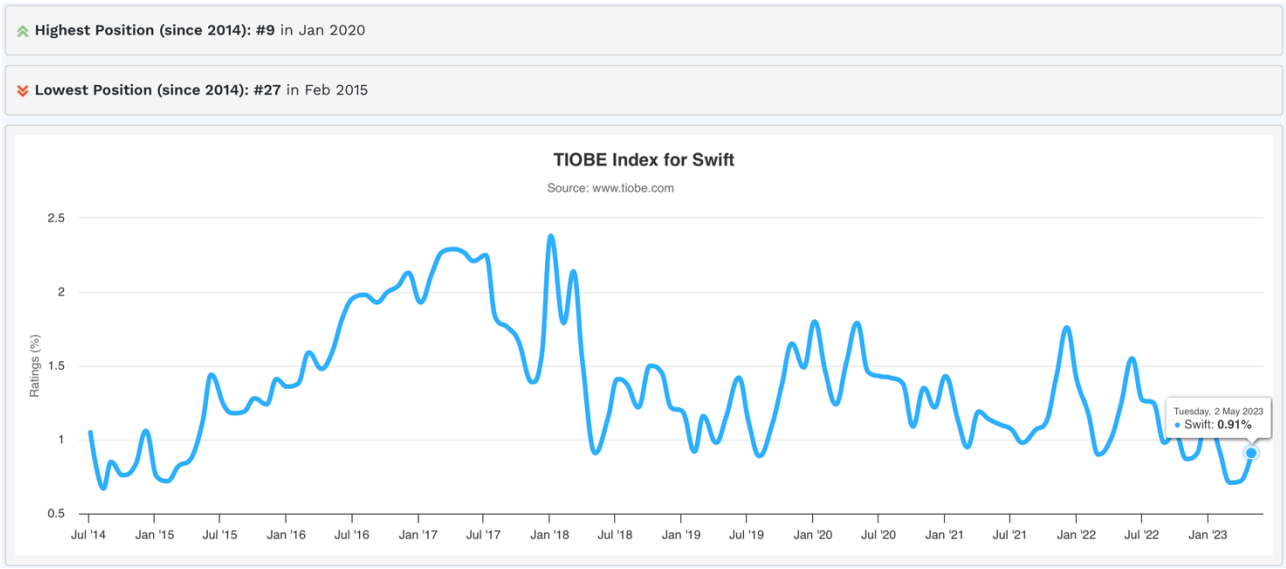


Рисунок 2.5 Індекс TIOBE для Swift [28]

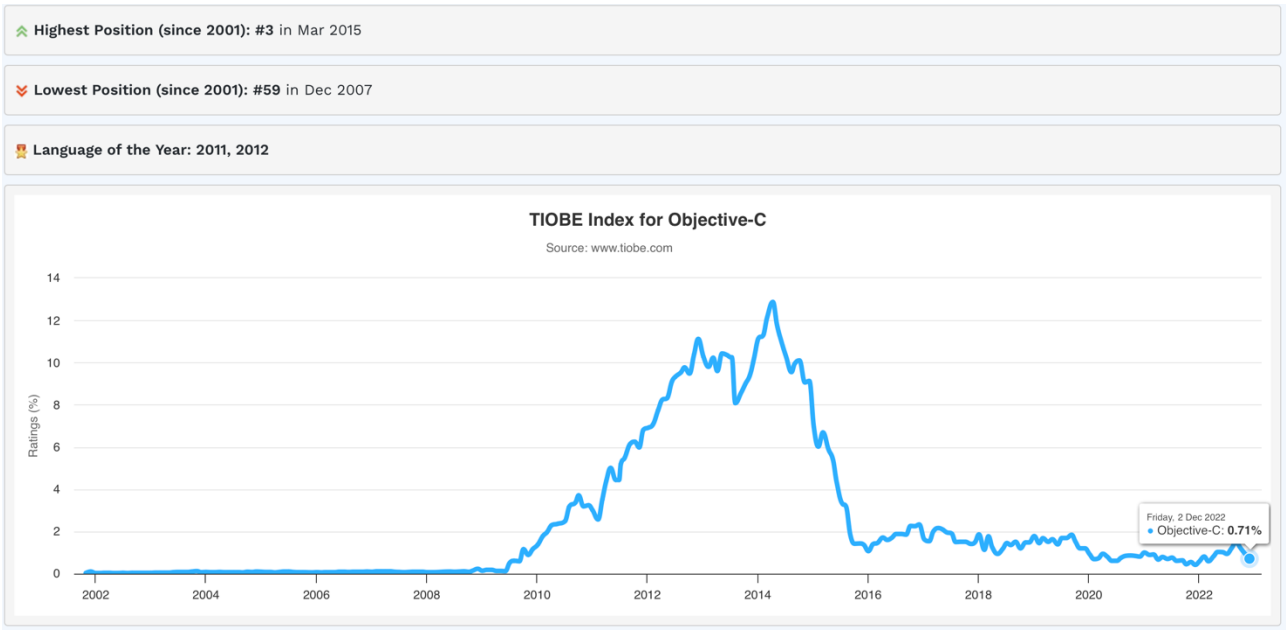


Рисунок 2.6 Індекс TIOBE для Objective-C [29]

Вище наведено графіки індексу TIOBE для обох мов програмування. З них можемо бачити, що в свій час Objective-C був дуже розповсюдженим, адже будучи доступним лише для Apple платформ посів 3 місце в 2015 році, а також був мовою року, яка визначається як мова з найбільшою швидкістю приросту в

популярність, в 2011 та 2012 роках (рис. 2.5). Незважаючи на це, оцінюючи сьогоднішні показники, Swift перегнав Objective-C і з останнього зрізу даних для обох мов Swift має 0.91% в той час як Objective-C – 0.71%, що приблизно на 12% більше (рис. 2.6).

Зважаючи на те, що Swift надає зручніший синтаксис для роботи, наразі є більш популярним вибором та займає більшу частку ринку, а також те, що Objective-C не надає жодних технологічних переваг з точки зору написання фреймворків чи розробки функціоналу трекінгу очей, було обрано працювати зі Swift.

Останнім з важливих рішень був вибір на чому базувати частину фреймворку, що відповідатиме за взаємодію з частиною користувацького інтерфейсу клієнта. На сьогоднішній день існує два основних способи написання коду інтерфейсів – фреймворки UIKit [30] та SwiftUI [31].

UIKit був та залишається основним фреймворком що постачає набір візуальних елементів та їх логіки для побудови інтерфейсів ще з появою першого iPhone в 2007 році. Протягом всіх цих років він отримував постійні оновлення та покращення, залишаючись актуальним інструментом для розробки користувацьких інтерфейсів. Для керування елементами екрану UIKit використовує ієрархію екрану в яку покладено такі елементи як кнопки, текстові поля, та інші. Всі вони організовані та скомпоновані в ієрархію для зручної роботи. Використовуючи модель ланцюжку відповідальності, UIKit реалізує можливість передачі факту про зроблену користувачем дію. Для розміщення елементів на екрані використовується Auto Layout, який позиціонує їх відносно один одного та незалежно від розміру екрану.

На зображенні наведеному нижче можна побачити як UIKit обробляє повний цикл взаємодії користувача з пристроєм – починаючи з фізичного натиснення пальцем на екран, та закінчуючи надсиланням інформації про конкретну взаємодію в код клієнта (рис. 2.7).

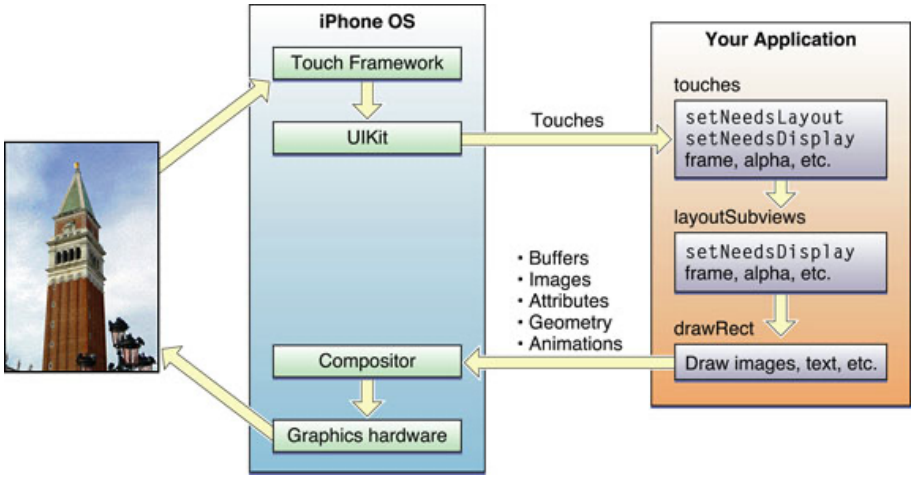


Рисунок 2.7 Діаграма взаємодії UIKit з користувацьким інтерфейсом [32]

Враховуючи те, що він присутній на платформах доволі давно, існує велика кількість ресурсів та документацій по розробці користувацьких інтерфейсів використовуючи даний фреймворк. Також переважна частина спільноти розробників має експертизу в ньому, що створює велику спільноту для отримання консультацій. Додатково до джерела експертизи, вона також стала рушієм до створення великої кількості сторонніх бібліотек, що полегшують роботу з UIKit.

В якості способів розробки за допомогою UIKit є написання інтерфейсів в коді, а також використання Storyboard, що дозволяє будувати екрани у графічному середовищі (рис. 2.8).

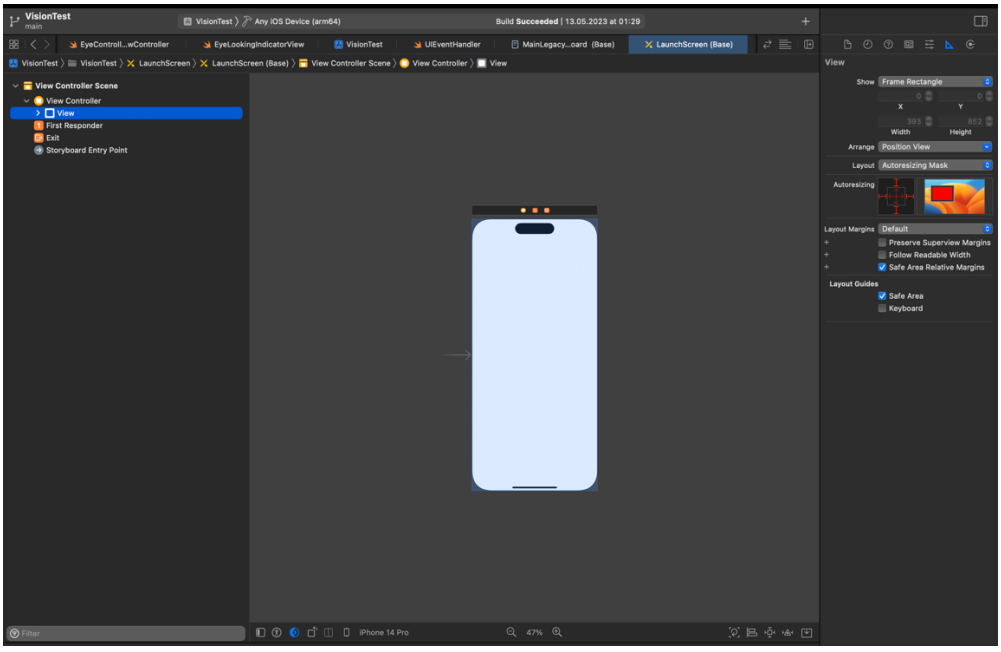


Рисунок 2.8 Інтерфейс Storyboard

В 2019 році Apple було випущено новий фреймворк для роботи з користувацьким інтерфейсом – SwiftUI. На відміну від UIKit, SwiftUI пропагує декларативний підхід до розробки інтерфейсів. Філософія підходу базується на описанні бажаного стану інтерфейсу, а не покрокових дій щоб його досягти. Це дозволяє покращувати зрозумілість коду та його легкість до підтримування. Також SwiftUI підтримується лише з коду на відміну від UIKit, що також дозволяє проектувати інтерфейси у графічному середовищі.

Серед переваг які SwiftUI приніс в порівнянні з UIKit можна також виділити менший обсяг коду необхідного для написання екрану. Порівнюючи два однакові елементи інтерфейсу написані за допомогою UIKit та SwiftUI видно різницю у кількості необхідних інструкцій (рис. 2.9).

```swift
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

        let button = UIButton(type: .system)
        button.setTitle("Button Action", for: .normal)
        button.titleLabel?.font = UIFont.boldSystemFont(ofSize: 20)
        button.tintColor = .white
        button.backgroundColor = .blue
        button.layer.cornerRadius = 12
        button.translatesAutoresizingMaskIntoConstraints = false
        button.addTarget(self, action: #selector(buttonTapped),
            for: .touchUpInside)

        view.addSubview(button)

        NSLayoutConstraint.activate([
            button.centerXAnchor.constraint(equalTo:
                view.centerXAnchor),
            button.topAnchor.constraint(equalTo:
                view.centerYAnchor),
            button.widthAnchor.constraint(equalToConstant: 150),
            button.heightAnchor.constraint(equalToConstant: 50)
        ])
    }

    @objc func buttonTapped() {
        // This is called when button is being clicked
    }
}
```

```swift
struct ContentView: View {
    var body: some View {
        VStack {
            Button(action: {
                // This is called when button is being clicked
            }) {
                Text("Button Action")
                    .font(.title)
                    .padding()
                    .foregroundColor(.blue)
                    .background(Color.yellow)
                    .cornerRadius(12)
            }
        }
    }
}
```

Рисунок 2.9 Різниця у кількості коду для однакового інтерфейсу UIKit та SwiftUI

Близько вдвічі менше коду було написано на SwiftUI для досягнення еквівалентного результату.

Серед недоліків, присутніх у SwiftUI, варто відзначити обмеженість по версіям, а саме від iOS 13.0 та вище. Також багато нового функціоналу з’являлось пізніше, що ще більше обмежує підтримувані версії. Окрім цього, SwiftUI менш стабільний фреймворк, аніж UIKit, та має меншу користувацьку спільноту, що може ускладнити пошук потенційної проблеми у разі її виникнення.

Серед критеріїв вибору основного підтримуваного фреймворку для побудови інтерфейсів найбільш значимим критерієм було обрано поширеність в існуючих додатках, що додатково контролювалося б швидкістю росту. А саме, щоб не виникла ситуація за якої кандидат, що вже є застарілим та не використовується, проте з великою існуючою базою, був би обраний. Проте, на жаль, немає жодної статистики щодо частки використання SwiftUI чи UIKit в додатках присутніх в App Store. Протягом дослідження було знайдено частку використання UIKit та SwiftUI в iOS 16.0 (рис. 2.10). Це дає змогу приблизно зрозуміти частку використання обох технологій сьогодні.

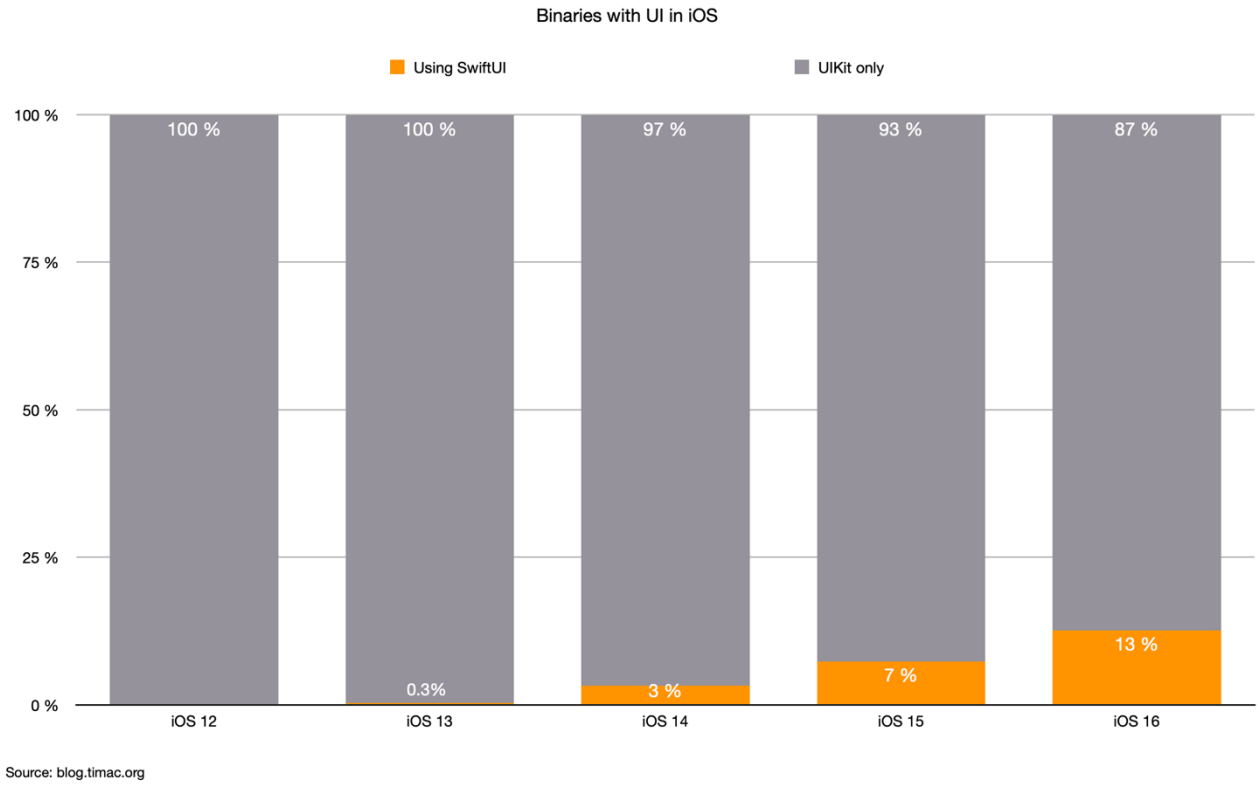


Рисунок 2.10 Частка використання UIKit та SwiftUI в iOS [33]

Згідно статистики бачимо, що з моменту появи SwiftUI його частка використання поступово збільшується, проте все ще займає дуже мало у порівнянні з UIKit. Враховуючи дану статистику, а також те, що UIKit все ще широко використовується та має свої переваги і велику спільноту користувачів, було прийнято рішення забезпечити роботу фреймворку по керуванню поглядом базуючись саме на ньому.

2.3 Теоретичні відомості про ARKit як про фреймворк доповненої реальності

Враховуючи те, що в якості підходу, що використовуватиметься для реалізації трекінгу погляду буде фреймворк від Apple ARKit, детальніше розглянемо його особливості та аспекти варті уваги.

Представлений він був в 2017 році на щорічній конференції для розробників від Apple. Як було сказано раніше, основна задача ARKit полягає в наданні функціоналу доповненої реальності на пристрої під управлінням iOS. Для цього він використовує набір датчиків, що існують вбудовані в сучасні смартфони: сенсори руху, LiDAR, що є сенсором, який за допомогою лазера будує 3D мапу оточення, камеру пристрою. Інформацію отриману від них можна використовувати для накладання віртуальних об’єктів на вхідні дані з камери пристрою в реальному часі або ж для власних подальших обрахунків.

Перед роботою з ARKit було досліджено технічні деталі фреймворку. Для того щоб правильно співвідносити реальний простір з віртуальним, слідкувати за рухом пристрою, та надавати якісну доповнену реальність, використовується техніка візуально-інерційної одометрії. Вона використовує отримані з камери зображення середовища та позначає на них визначні точки, за якими в майбутньому відслідковуються зміни. Паралельно з цим процесом датчики пристрою визначають рух його в просторі. Поєднуючи інформацію з датчиків та зміну зображень, техніка візуально-інерційної одометрії прогнозує траєкторію пристрою та його позицію відносно початкового місця в тривимірному просторі.

Серед можливостей фреймворку також присутнє відслідковування та розуміння так званої «сцени», що можна інтерпретувати як оточення. Це включає в себе ідентифікацію горизонтальних та вертикальних поверхонь, об’єктів та зображень, а також слідкування за їх позиціями та переміщеннями. Такий функціонал дозволяє додавати віртуальні об’єкти на реальні поверхні і інтегрувати віртуальний контент в реальний світ. Для цього ARKit має та використовує імплементовану систему якорів. Вони функціонують як точки закріплення об’єктів і дозволяють точно визначати позицію віртуального контенту у світі та тримати її впродовж сесії.

Для того щоб закріплені об’єкти були максимально інтегровані у реальний світ, ARKit надає вбудовану підтримку для аналізу та обрахунку світла у середовищі. Надання інформації про освітлення дозволяє відмальовувати віртуальні об’єкти з правильним власним освітленням та тінями.

Починаючи з версії ARKit 2.5, Apple було додано можливість доступу до інформації про риси обличчя користувача, зокрема очі та зіниці. Прямої підтримки слідкування за поглядом користувача в ARKit немає, проте надані розробникам дані дозволяють розглянути можливість імплементації за допомогою ARKit. Для коректного зчитування інформації про обличчя користувача використовується технологія TrueDepth в камері iPhone (рис. 2.11).

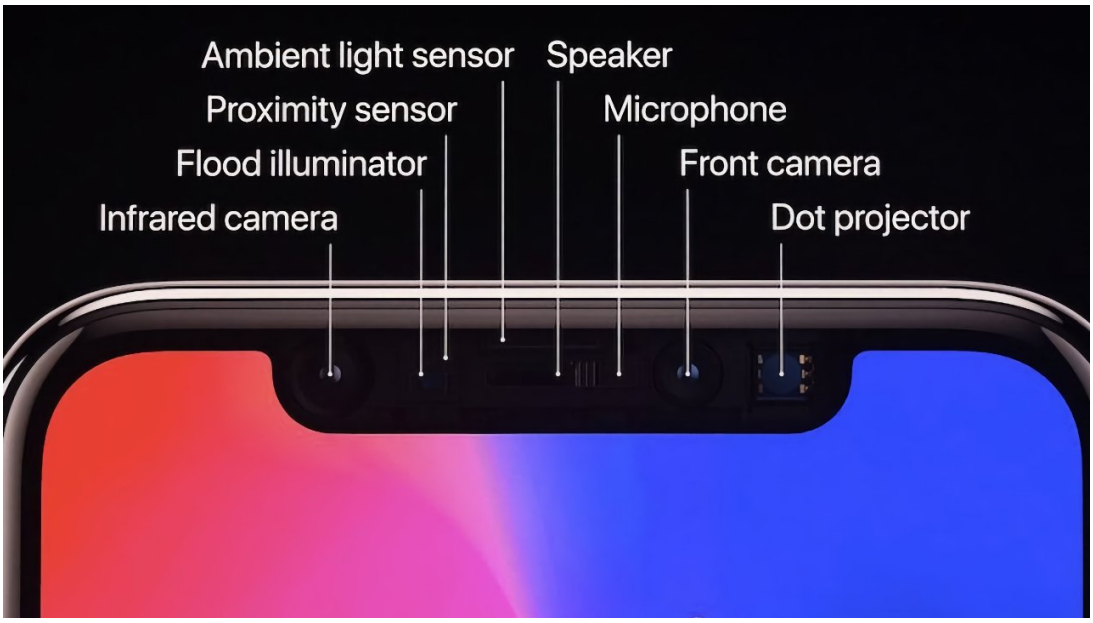


Рисунок 2.11 Система сенсорів TrueDepth у iPhone [34]

Дана система була створена для пошуку та сканування обличчя користувача і містить шість різних сенсорів. Процес роботи складається з взаємодії кожного з них. За пошук обличчя відповідає інфрачервона камера, і у разі його знаходження відбувається обрахунок необхідного освітлення для того аби почати процес ідентифікації. У системі TrueDepth за роботу зі світлом відповідають сенсор навколишнього освітлення та сенсор наближеності. Після цього сенсор під назвою «Flood illuminator» знімає двовимірні зображення обличчя користувача, які в комбінації з проекцією точок на обличчя користувача використовуються для побудови вже тривимірної мапи його обличчя. Дане

відображення і використовується ARKit для ідентифікації змін на обличчі. В рамках iOS TrueDepth використовується в основному у зв’язці технології FaceID, що дозволяє безпечно виконувати дії в межах операційної системи.

Загалом, ARKit надає доступ до багатьох даних та дозволяє створювати дійсно високоякісну доповнену реальність. Проте варто відзначити, що якість досвіду сильно залежить від освітлення, відсутності рефлективних поверхонь, та того наскільки складним є оточення, адже в роботі ARKit покладається на якість вхідних даних.

2.4 Аналіз можливостей для дистрибуції коду в рамках iOS

За мету роботи було поставлено покращення досвіду користування пристроями під управлінням iOS людям з обмеженими можливостями. В якості конкретизації цільової аудиторії було обрано людей, що мають проблеми з моторикою та загалом керуванням смартфоном за допомогою рук. Було знайдено потенційний спосіб керування пристроєм для неї, а саме керування зором. Незважаючи на визначеність того як саме буде вирішуватись дана проблема технічно, існує невизначеність щодо форми в якій дане рішення буде надаватись.

Згідно аналізу потенційних конкурентів можна побачити, що керування зором надається або у вигляді повної інтеграції в інтерфейс зі стороннім приладом для трекінгу очей, або як окремо виділений додаток. Перший описаний варіант не є оптимальним, адже вимагає розробки власного фізичного приладу. Окрім цього, в такому випадку підтримуваною операційною системою скоріше за все буде лише iPadOS, а ціна та логістика унеможливить покриття великої кількості аудиторії.

Було розглянуто вирішення проблеми написанням власного додатку. Він слугував би вхідною точкою для користувача та надавав йому інтерфейс для користування смартфоном. Саме такий концепт взяв за основу HawkeyeAccess описаний раніше. В такому випадку написана програма розглядалась як така, що надає доступ до користування веб-браузером, який в свою чергу дає доступ до безлічі сайтів. Такий спосіб реалізації давав би можливість людям з вадами зору

користуватись смартфоном, проте як веб-переглядачем, а не мобільним пристроєм. Також як актуальний функціонал розглянуто було імплементацію окремих важливих застосунків в рамках власного додатку. Перелік таких застосунків включає в себе, до прикладу: калькулятор, погоду, таймер. За даної імплементації додаток можна було б сприймати як окрему операційну систему що надавала б власні рішення з можливістю керування зором.

Використання саме окремого додатку має свої переваги та недоліки. З переваг можна виділити той факт, що окремий застосунок дозволяє повністю використати функціонал керування зором та налаштувати програмне забезпечення під власні потреби так як код пишеться відразу з підтримкою відслідковування погляду користувача. Також створення додатку передбачає його публікацію в App Store – магазин додатків на iOS. Це відкриває можливості для його знаходження потенційними користувачами, адже всі власники пристроїв під управлінням iOS мають встановлений App Store на своєму мобільному девайсі. Також як загальна практика встановлено так званий «фічеринг» додатків – його просування всередині App Store редакційною командою Apple (рис. 2.12).

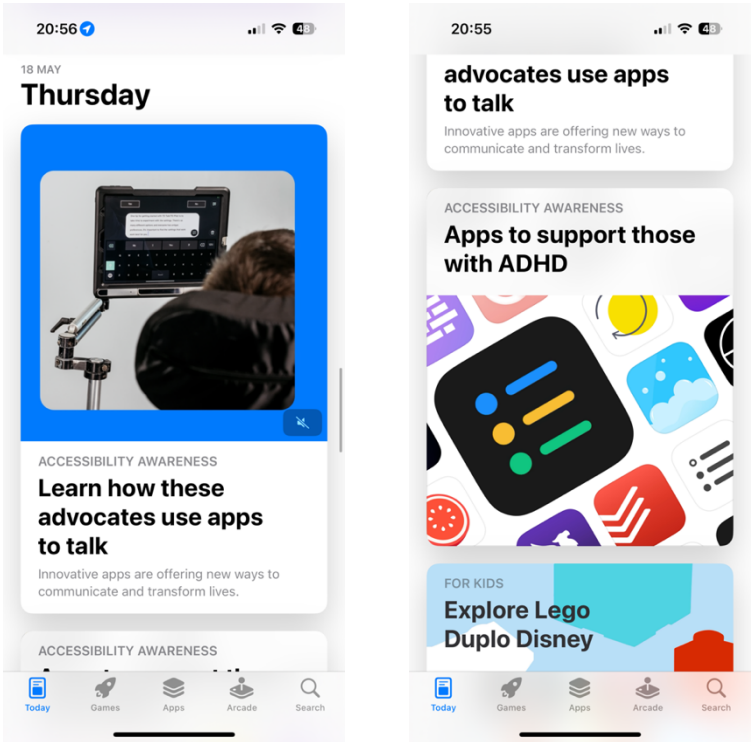


Рисунок 2.12 Приклади "фічерингу" від Apple

До прикладу, на зображеннях вище Apple окремою сторінкою просуває матеріали про те як люди з обмеженими можливостями використовують додатки під управлінням iOS, а також окремо додатки, що допомагають боротись з синдромом дефіциту уваги з гіперактивністю. Таким чином вони отримують велику кількість переглядів та потенційних завантажень без витрат на маркетинг. Маючи окремий додаток що вирішує потреби людей з проблемами моторики кінцівок, також можна було б претендувати на просування від Apple та показ потенційній цільовій аудиторії, допомагаючи при цьому великій кількості людей. Також доступ до встановлення додатку є дуже простим, для цього використовується лише інфраструктура Apple, а процес встановлення однаковий для всіх додатків під управлінням iOS. Окремий застосунок легше просувати і поза App Store, адже є простір для того аби створити та закріпити бренд. З точки зору того щоб стати доступним якомога більшій кількості людей дана особливість є доволі важливою.

Проте є і доволі суттєві недоліки в такому вирішенні поставленої проблеми. Незважаючи на перевагу власного додатку в тому, що його можна повністю конфігурувати та надати йому саме той функціонал, який потрібно, наявність будь-якого функціоналу вимагає попередньої розробки та релізу. Написання функціоналу калькулятору чи погоди є відносно тривіальною задачею для мобільної розробки, проте наявність, до прикладу, навігації чи певних інших можливостей є вже складною задачею. Це призводить до того, що скільки б часу не виділялось під застосунок, його функціональні можливості все рівно залишатимуться малими в порівнянні з широким набором існуючих на сьогодні додатків. Основним за набором можливостей буде функціонал веб-переглядача, проте найбільшої зручності користування мобільним пристроєм можна досягти використовуючи нативні додатки, а не веб-версії продуктів.

Загалом, підтримка додатку з великим набором вбудованих в нього менших застосунків вимагає розподілення уваги на кожен з них та великої кількості виділеного ресурсу. Такий підхід зменшує час на підтримку та покращення основного функціоналу, який забезпечує вирішення початкової

проблеми – а саме відслідковування погляду користувача та управління застосунком за допомогою нього.

Описані недоліки виявились достатньо суттєвими щоб в процесів дослідження оглянути інші способи надання користувачам можливості використовувати зір як спосіб для керування додатками в рамках iOS.

Іншим підходом, що дозволяє розповсюджувати програмне забезпечення є створення фреймворку. Фреймворк – це зібрана колекція коду, що надає певний набір функціоналу його клієнтам – розробникам додатків, які цей фреймворк під'єднують до власних продуктів. У їх складі присутні компоненти для використання, стандартизовані інтерфейси, що надають доступ до внутрішнього функціоналу, та певна попередньо визначена архітектура. В контексті iOS розробки, фреймворки енкапсулюють в собі набір розробленого функціоналу або технології, що дозволяє користувачам уникати повторної власної розробки, а лише підключити готове рішення. Окрім цього, їх використання дозволяє отримувати переваги від модульної архітектури додатку. Модульна архітектура передбачає розбиття застосунку на окремі модулі, кожен з яких відповідає окремий домен чи функціонал. Мета фреймворку – надати рішення актуальної проблеми в певному конкретному домені. ARKit, оглянутий раніше, також є фреймворком. Розроблений в Apple, він вирішує реалізацію технології доповненої реальності та надає інтерфейс для її використання у сторонніх додатках. Іншою перевагою використання фреймворку є його оновлення та підтримка. Адже поширюючи таким чином код, всі його оновлення стають доступними для всіх клієнтів відразу.

З недоліків підходу знайдено необхідність клієнтам керувати цією залежністю, вирішувати потенційні проблеми з сумісністю з власним додатком, а також слідкувати за оновленням версії. Також розробники фреймворку не можуть гарантувати однакової його імплементації в кожному з продуктів і мають покладатись на те, що клієнти правильно використовуватимуть розроблений ними функціонал.

В контексті поставленої в роботі задачі, фреймворк може енкапсулювати функціонал слідкування за поглядом користувача, та передавати інформацію про отримані дані його клієнтам, стороннім додаткам. Таким чином, розробникам сторонніх додатків потрібно використовувати отриману інформацію для застосування тих чи інших дій від користувача у межах власного продукту. Також як опцію розглянуто і більш тісну інтеграцію фреймворку в додаток, що передбачає частковий чи повний контроль над діями користувача в підключеному застосункові відразу з фреймворку, без додаткових дій сторонніх розробників.

Порівнюючи підхід дистрибуції через додаток з дистрибуцією фреймворком варто відзначити, що додаток дозволяє максимально інтегрувати керування зором у продукт, адже і частина керування зором, і інші аспекти застосунку є написані в одному проекті, і відповідно повністю сумісні один з одним. В той час як фреймворк не гарантує максимально якісної інтеграції керування зором в продукт, адже функціонал написаний та випущений окремо, і вже потім під'єднується до незалежних від нього застосунків. Окрім цього, у аспекті просування технології та збільшення обізнаності цільової аудиторії про існування продукту, додаток також надає кращу у цьому платформу. Можливість просування від Apple у рамках App Store та наявність власного додатку з брендом полегшує вищеописану задачу, в той час як фреймворк не є повноцінним продуктом і його поширення залежить від рівня зацікавленості іншими розробниками та інтеграції його в продукти.

З іншого боку, важливим недоліком додатку є його обмежений функціонал. Фреймворк децентралізує поширення функціоналу по керуванню зором та дозволяє будь-кому додати його собі в застосунок. Розробнику ж фреймворку достатньо лише раз його розробити та підтримувати надалі. В той час як функціонал додатку повністю залежить від розробника та є централізованим. В такому випадку фреймворк може значно ширше забезпечити людей з обмеженими можливостями у альтернативному способі управління, а керування зором може з'явитись у багатьох популярних додатках.

Враховуючи важливість саме поширення такого функціоналу серед існуючих та майбутніх додатків під управлінням iOS було вирішено розробляти можливість управління поглядом саме у вигляді фреймворку, при цьому врахувавши потенційні проблеми з якісною інтеграцією.

2.5 Висновки до розділу 2

В розділі ретельно проаналізовано необхідні технології для реалізації керування зором на пристроях під управлінням iOS. Розглянуто технології для слідкування за поглядом користувача, серед яких виділено використання вже готового апаратного рішення та інструментарію, виявлено переваги такі як зменшення часу на розробку алгоритму, та недоліки, такі як мала кількість підтримуваних пристроїв та необхідність купівлі додаткового заліза. Також проаналізовано можливість створення власної моделі машинного навчання та спосіб збереження її локально на пристрої та в хмарі. Останнім було проаналізовано використання фреймворку ARKit від Apple. Серед оглянутих способів після аналізу всіх переваг та недоліків було обрано використання фреймворку ARKit.

Розділ також містить огляд можливих підходів в розробці під iOS. Проаналізовано відмінності у мовах Swift та Objective-C, поширеність у використанні та динаміку росту. Після аналізу було обрано використання Swift як сучасного та більш популярного варіанту.

Після вибору ARKit як фреймворку для імплементації слідкування за поглядом було детально оглянуто теоретичні відомості про нього, як саме технічно надається можливість слідкування за обличчям користувача. В рамках даного дослідження було оглянуто технологію TrueDepth та її роль в цьому процесі.

Також розділ описує аналіз можливих способів дистрибуції коду. Було оглянуто створення власного додатку, що спостерігається у багатьох із конкурентів, та дистрибуцію за допомогою стороннього фреймворку. Серед переваг додатку було виділено тісну інтеграцію з функціоналом керування зором

та легкість розповсюдження, в той час як серед недоліків знайдено відсутність можливості покрити багато функціоналу в одному продукті. Підхід дистрибуції шляхом створення фреймворку також має певні недоліки, такі як відсутність можливості повної інтеграції та необхідність бути універсальним рішенням. З переваг виділено можливість більш широко вплинути на ринок додатків, адже він є доступним для всіх розробників платформи. Порівнюючи переваги та недоліки було обрано дистрибуцію шляхом створення окремого фреймворку.

Підсумовуючи, даний розділ містить процес вибору технологій та їх порівняльну характеристику, а також окреслює детальні відомості про ARKit як про рішення, що використовуватиметься у слідкуванні за поглядом користувача.

Розділ 3. Огляд практичної реалізації фреймворку

3.1 Загальний огляд побудови фреймворку та спосіб його дистрибуції

Перед початком розробки постало питання вибору способу дистрибуції фреймворку. Серед поточних лідерів на ринку присутні Swift Package Manager, CocoaPods та Carthage [37]. Swift Package Manager виділяється наявністю повноцінної підтримки в XCode [38], основній IDE для розробки програмного забезпечення під iOS, та легкістю інтеграції у проект. CocoaPods має перевагу у тому, що він давно на ринку та сформував спільноту, а також підтримує Objective-C. Carthage також підтримує Objective-C та фокусується на простоті роботи з залежностями. Проте враховуючи що створений в роботі фреймворк підтримує лише Swift, переваги Swift Package Manager у повній інтеграції з інструментами Apple, та підхід орієнтований на Swift проекти привів до вибору саме його як шляху дистрибуції.

Оглядаючи структуру створеного проекту варто відзначити що з початку роботи він був створений як Swift Package. Однією з особливостей є створення файлу Package.swift (рис. 3.1). Це файл маніфесту, що містить в собі конфігурацію по налаштуванню та керуванню пакетом.

```
// swift-tools-version: 5.7.1
// The swift-tools-version declares the minimum version of Swift required to build this package.

import PackageDescription

let package = Package(
    name: "VisionARy",
    platforms: [
        .iOS(.v13)
    ],
    products: [
        .library(
            name: "VisionARy",
            targets: ["VisionARy"]),
    ],
    dependencies: [
    ],
    targets: [
        .target(
            name: "VisionARy",
            dependencies: []),
        .testTarget(
            name: "VisionARyTests",
            dependencies: ["VisionARy"]),
    ]
```

Рисунок 3.1 Файл маніфесту

Обрана назва фреймворку – VisionARy, що поєднує в собі основну мету та способи виконання, які з нею поєднані. Vision – про зір, ключовий інструмент за допомогою якого реалізується можливість керувати смартфоном людям з вадами руху. AR – як технічний інструмент за допомогою якого дана задача виконана. А повністю слово visionary означає візіонер, тобто той, хто дивиться в майбутнє з ідеями про те, як воно може виглядати. Можливість кожному незалежно від стану здоров'я мати доступ до технологій є однією з ключових задач, що має бути вирішеною в майбутньому.

Повна структура фреймворку наведена в додатку 1. Логічно проект розділено на такі типи: об'єкти даних, об'єкти, що виконують допоміжну роль, об'єкти бізнес-логіки, та об'єкти елементів користувацького інтерфейсу. Ті з них, які передбачаються для використання іншим розробникам мають модифікатор доступу open, що у Swift дає можливість доступу з іншого проекту, до прикладу того, який під'єднує фреймворк.

3.2 Огляд реалізації слідкування за поглядом користувача

Розглянемо окремо частину функціоналу, що забезпечує слідкування за поглядом користувача. Ізольована діаграма класів, що стосуються лише його, представлена в додатку №2. В реалізації трекінгу погляду користувача основним інструментом був ARKit. Опишемо алгоритм на високому рівні. Спочатку створюються всі необхідні об'єкти для створення AR-сцени. В них входять представлення обличчя та окремий елемент, який буде розміщено на AR-сцені та що буде виконувати роль уловлювача погляду користувача. Після інформації що користувач дивиться на екран, використовуються локальні координати погляду користувача на об'єкт, які конвертуються в значення координат в контексті самого екрану. Отримавши поточні координати погляду, використовуються алгоритми для їх згладжування, а також додатково фіксуються зміни в міміці користувача. Всі отримані параметри записуються у відповідні структури та надсилається сигнал про те, що відбулося їх оновлення.

Опишемо алгоритм детальніше. Було проведено ініціалізацію архітектури AR-сцени. Загальний код налаштування наведено нижче (рис. 3.2) .

```
func configureNodes() {
    let plane = SCNPlane(width: 1, height: 1)
    plane.firstMaterial?.isDoubleSided = true
    plane.firstMaterial?.diffuse.contents = UIColor.green

    hitTestNode = SCNNode(geometry: plane)
    hitTestNode.position.z = 0.1

    let leftEyeScnCone = SCNCone(topRadius: 0.008, bottomRadius: 0, height: 0.3)
    leftEyeScnCone.firstMaterial?.diffuse.contents = UIColor.blue

    let leftEyeNode = SCNNode()
    leftEyeNode.eulerAngles.x = -.pi / 2
    leftEyeNode.geometry = leftEyeScnCone
    leftEyeNode.position.z = 0.1

    let leftContainerNode = SCNNode()
    leftContainerNode.addChildNode(leftEyeNode)
    leftEyeSCNNode = leftContainerNode

    let rightEyeScnCone = SCNCone(topRadius: 0.008, bottomRadius: 0, height: 0.3)
    rightEyeScnCone.firstMaterial?.diffuse.contents = UIColor.blue

    let rightEyeNode = SCNNode()
    rightEyeNode.eulerAngles.x = -.pi / 2
    rightEyeNode.geometry = rightEyeScnCone
    rightEyeNode.position.z = 0.1

    let rightContainerNode = SCNNode()
    rightContainerNode.addChildNode(rightEyeNode)
    rightEyeSCNNode = rightContainerNode
}

func setupTracking(configuration: ARFaceTrackingConfiguration) {
    faceArView.automaticallyUpdatesLighting = true
    faceArView.scene.rootNode.addChildNode(faceNode)
    faceArView.scene.rootNode.addChildNode(hitTestNode)
    faceNode.addChildNode(leftEyeSCNNode)
    faceNode.addChildNode(rightEyeSCNNode)
    leftEyeSCNNode.addChildNode(lookingPositionOfLeftNode)
    rightEyeSCNNode.addChildNode(lookingPositionOfRightNode)

    lookingPositionOfLeftNode.position.z = 2
    lookingPositionOfRightNode.position.z = 2

    faceArView.session.run(configuration, options: [.resetTracking, .removeExistingAnchors])
}
```

Рисунок 3.2 Програмний код створення архітектури AR-сцени

Спочатку створено екземпляр ARSCNView. Даний об’єкт використовується для відмальовування сцен доповненої реальності та є підкласом SCNView фреймворку SceneKit. Його можливості по роботі з тривимірним простором, інтеграція з камерою, та сумісність з ARKit забезпечують подальшу роботу алгоритму.

Наступним кроком в налаштуванні початкового стану було додавання необхідних вузлів в ARSCNView. Вузол представлений типом SCNNode що є фундаментальним в SceneKit. Кожен з них представляє окремі елементи в AR-сцені. В ARKit він є елементом ієрархічної структури графа сцени. Граф сцени є

структурою, що визначає віртуальні об’єкти та їх відносини один з одним на AR-сцені. Вузли в ньому організовані у деревоподібну структуру та мають зв’язок батько-дитина. Кожен батько може мати декілька дочірніх вузлів, що дозволяє використовувати композицію та мати окремі атомарні вузли, що відповідають за свої менші частини об’єкту. SCNNode також надає можливості для визначення позиції та орієнтацію елемента на AR-сцені, має пов’язану з ним геометрію та відображення.

У фреймворку було додано faceNode як вузол, що відповідає за представлення обличчя, а також hitTestNode. hitTestNode утворено з простого SCNPlane, що є об’єктом, який надає плоску прямокутну фігуру в тривимірному просторі, а задається через ширину та висоту, що визначає його розмір на сцені. Додатково до вузла faceNode додано два вузла leftEyeSCNNode та rightEyeSCNNode, що використовуватимуться для слідкування за поглядом, а також будучи представленими геометрією типу SCNCone використовуються для візуалізації точки куди дивиться користувач. SCNCone є геометричним примітивом, що представляє конус. Налаштування окремих вузлів для очей включає задання верхнього та нижнього радіусів, кольору, розміщення їх перед очима, а також Ейлерівського кута для конусів, що визначає обертання об’єкта у власному просторі координат. Так як в ARKit об’єкти SCNPlane розміщені вертикально за замовченням, відповідно використано значення ейлерівського кута -pi / 2 для коректного застосування елемента. Для завершення архітектури сцени до leftEyeSCNNode та rightEyeSCNNode додано дочірні вузли lookingPositionOfLeftEye та lookingPositionOfRightEye. Їм надана відстань у 2 для позиції по z осі координат. У просторі ARKit одиницею виміру відстані є метри, відповідно надання параметру position.z значення 2 ставить відповідні вузли 2 метри від центру очей. Необхідність даної архітектури буде обґрунтовано в описі роботи механізму слідкування за поглядом користувача. Візуально створена AR-

сцена надана на зображенні нижче, де зеленим позначено hitTestNode, а два сині конуси відповідають за leftEyeSCNNode та rightEyeSCNNode (рис. 3.3).

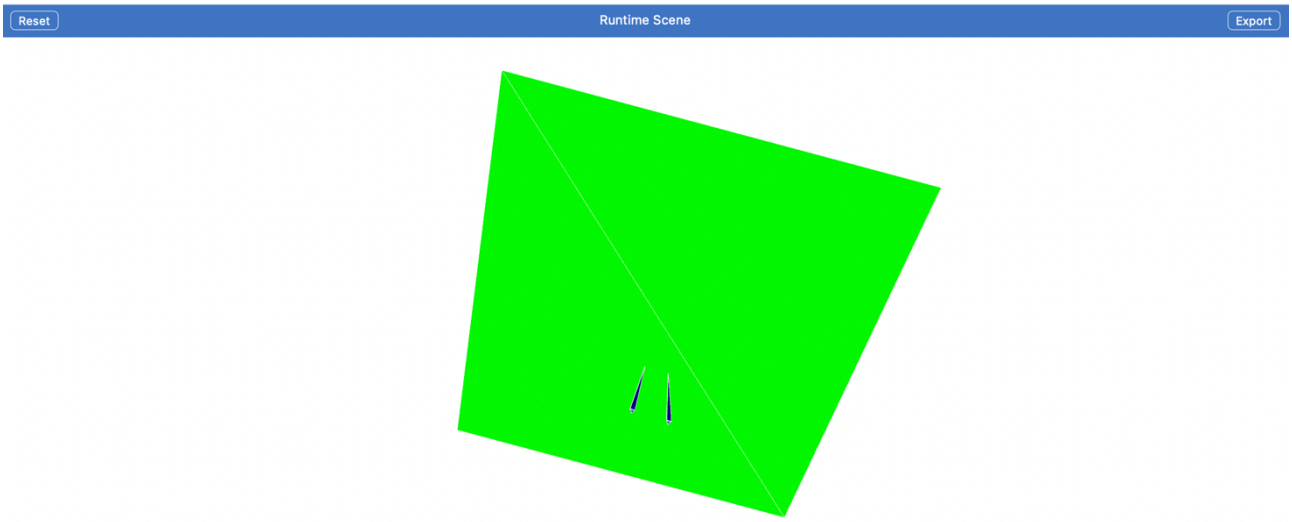


Рисунок 3.3 Створена AR-сцена

Таким чином отримано готову архітектуру для доповненої реальності. Вона розгортається на екземплярах UIViewController, що виконують протокол EyeControllableViewControlling. Алгоритм знаходження координати точки погляду користувача реалізується об'єктом EyeTrackingManager, що реалізує протокол EyeTrackingManaging (рис. 3.4). Користувачі функціоналу слідкування за поглядом працюють через протокол, не знаючи конкретної імплементації. EyeTrackingManaging включає об'єкт faceData типу FaceData, що містить повну інформацію про відслідковане обличчя та функцію track, що приймає всі вузли створені на етапі архітектури сцени.

```
protocol EyeTrackingManaging {
    var faceData: FaceData { get }
    func track(withInfo info: (hitTest: SCNNode, face: SCNNode, leftEye: SCNNode, rightEye: SCNNode, lookAtLeftEye: SCNNode,
        lookAtRightEye: SCNNode, faceView: ARSCNView))
}
```

Рисунок 3.4 Декларація протоколу EyeTrackingManager

Функція track запускає сесію доповненої реальності та робить себе її делегатом. Шаблон делегування широко розповсюджений у iOS та використовується як сторонніми розробниками, так і у самій системі. Даний підхід дозволяє одному об'єкту вести комунікацію з іншим через певний визначений протокол. Стаючи делегатом, об'єкт має надати реалізацію необхідних за протоколом функцій, що дозволяє викликати їх з іншого об'єкта

за необхідності. З точки зору даної реалізації, ставши делегатом, EyeTrackingManager має змогу отримувати оновлення в трекінгу від ARSCNView.

Після вищеописаних налаштувань виконується метод session.run(configuration, options: [.resetTracking, .removeExistingAnchors]), що запускає сесію для faceArView з опціями по новому запуску слідкування та видаленню вже існуючих, якщо вони є, якорів на сцені. Також в запуск передається попередньо створена конфігурація. Вона передається з об'єкту ViewController, що тримає посилання на даний елемент. Конфігурація визначає налаштування з якими працює доповнена реальність та задається типом ARConfiguration. ARConfiguration має різні підкласи, такі як ARWorldTrackingConfiguration, що відслідковує позицію пристрою в площині реального світу, або ж ARImageTrackingConfiguration, що відмальовує віртуальний контент відносно реального світу та ідентифікує і відслідковує за зображеннями. Так як в рамках роботи досліджується керування зором, то для доповненої реальності необхідним є інший нащадок ARConfiguration - ARFaceTrackingConfiguration. Він надає функціонал для знаходження та слідкування за рухами і мімікою користувача в реальному часі, що дозволяє створювати власні надбудови для подальшої імплементації. Перед використанням виконується перевірка ARFaceTrackingConfiguration.isSupported на предмет наявності можливості використовувати трекінг.

Після запуску сесії доповненої реальності EyeTrackingManager як делегат спочатку отримує один раз інформацію про додавання вузла обличчя до ARAnchor у методі renderer(_:didAdd:for), а всі наступні рази отримує інформацію про оновлення вузла у методі renderer(_:didUpdate:for). Код, що виконується, однаковий для обох методів (рис. 3.5). Трансформація, що відбулась у вузлі, переноситься на faceNode, відбувається перевірка того чи faceAnchor, що передався у метод є екземпляром класу ARFaceAnchor, що відповідає за представлення обличчя, а також викликається метод update(withFaceAnchor:).

```
func renderer(_ renderer: SCNSceneRenderer, didAdd node: SCNNode, for anchor: ARAnchor) {
    faceNode.transform = node.transform

    guard let faceAnchor = anchor as? ARFaceAnchor else { return }

    update(withFaceAnchor: faceAnchor)
}

func renderer(_ renderer: SCNSceneRenderer, didUpdate node: SCNNode, for anchor: ARAnchor) {
    faceNode.transform = node.transform

    guard let faceAnchor = anchor as? ARFaceAnchor else { return }
    update(withFaceAnchor: faceAnchor)
}
```

Рисунок 3.5 Реалізація методів делегату

Функція update(withFaceAnchor:) містить логіку обрахування поточної координати погляду користувача. Спочатку eyeRNode та eyeLNode роблять ідентичну трансформацію до тієї, яку має вхідний ARAnchor для правого та лівого ока відповідно.

На даний момент архітектура AR-сцени складається з hitTestNode, що є пласким прямокутником розміщеним на ній перед користувачем, eyeRNode та eyeLNode, що є вузлами що позначають точки очей, а також lookAtTargetRNode та lookAtTargetLNode, що є точками куди спрямований погляд. Для розуміння того куди дивиться користувач виконується підхід «кидання променів». За допомогою нього визначається пересічення певного заданого променю з бажаним об’єктом у тривимірному просторі. Ідея полягає в тому щоб симулювати шлях променю світла в заданому напрямі і знаходити перетини з об’єктами. З точки зору поточної задачі, промінь задається як відрізок від eyeRNode та eyeLNode до lookAtTargetRNode та lookAtTargetLNode відповідно. Елемент, пересічення з яким є шуканим, є hitTestNode. На малюнку графічно зображено представлення променів та пошук пересічення (рис. 3.6).

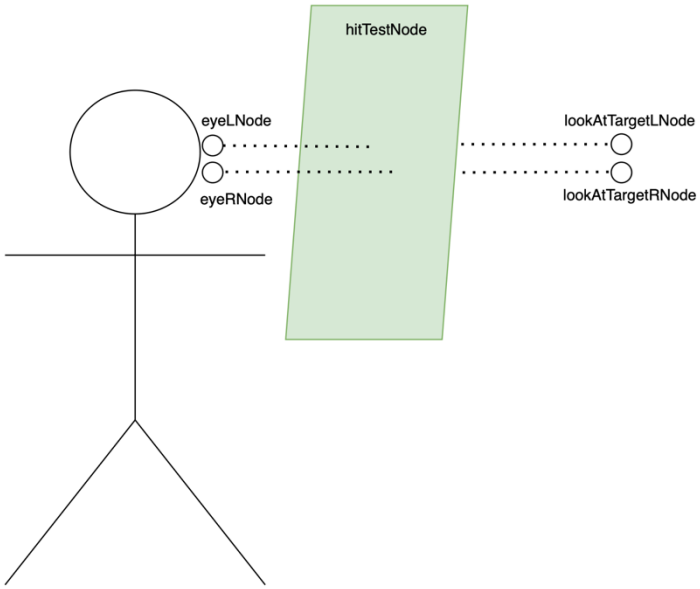


Рисунок 3.6 Графічне зображення пошуку пересічення

З точки зору технічної імплементації ідеї, об’єкти SCNNode мають метод hitTestWithSegment, в який було передано параметром from lookAtTargetEyeRNode.worldPosition та параметром to eyeRNode.worldPosition для правого ока та відповідні значення для лівого. Вузли у ARKit мають два представлення їх позиції – worldPosition та localPosition. Від обраного представлення залежить чи надані координати будуть мати позицію відносно реального світу чи відносно батьківського вузла. У випадку даного алгоритму, цікавить саме позиція в просторі реального світу.

У разі знаходження пересічення з hitTestNode обраховуються координати погляду очей по осі x та по осі y. Для цього застосовуються формули:

$$eyeLookAtX_{left/right} = \frac{intersectionResult.x}{phoneScreenSize.x * \frac{1}{2}} * screenBounds.width$$

Формула 3.1 Обрахунок координати погляду на осі x

та

$$eyeLookAtY_{left/right} = \frac{intersectionResult.y}{phoneScreenSize.y * \frac{1}{2}} * screenBounds.height + heightComp$$

Формула 3.2 Обрахунок координати погляду на осі y

Розглянемо кожну з формул. Для знаходження горизонтальної координати поточного погляду користувача беремо значення перетину з hitTestNode по осі x

та ділимо його на реальний розмір смартфону поділений на 2. Обидва значення надані у метрах. Отримавши значення конвертоване до розміру телефону, множимо його на ширину екрану в пікселях. Отримане значення є координатами по осі x. Для знаходження вертикальної координати виконуємо ту ж формулу з однією відмінністю. Емпіричним шляхом знайдено, що координата по осі y конвертована в розміри екрану девайсу не відповідає коректному значенню, а працює з певним сталим для кожної з ітерацій відхиленням. Знайдено, що відхилення для пристроїв складає близько $\frac{screenBounds.height}{2.5}$. Відповідно до знайденої координати додається дане значення.

Під час знаходження координати погляду користувача фреймворком фіксуються і зміни в його міміці. Інформація про зміни на обличчі користувача отримується через зчитування параметру blendShapes у вхідного anchor, що має тип ARFaceAnchor.BlendShapeLocation. Параметр не обмежений ними, проте включає такі дані як eyeLeftOpen, eyeLeftBlink, brownDownLeft, tongueOut, що дозволяє гнучко налаштовувати досвід доповненої реальності згідно рис обличчя. Формат даних – число в діапазоні від 0.0 до 1.0. До прикладу, в eyeLeftOpen значення 0.0 відноситься до закритого лівого ока, а збільшення числа, що прямує до 1.0, позначає ступінь відкритості ока. Узагальнюючи, кожен з параметрів при значенні 0.0 ідентифікує, що його умова не виконується, а збільшення до 1.0 показує наскільки до виконання умови є наближеним поточний стан обличчя користувача. При цьому, умова може бути виконана і при значенні менше 1.0 згідно фізіологічних особливостей користувача. Так, mouthLeftSmiled може позначати посмішку лівої частини обличчя користувача вже при значенні 0.5. В рамках даної роботи відбувається збір таких даних щодо рис обличчя та міміки користувача як: відкритість лівого ока, відкритість правого ока, усмішка лівої частини обличчя, та усмішка правої частини обличчя, а також відкритість щелепи. Для візуалізації побудови моделі для фреймворку наведено діаграму (рис. 3.7).

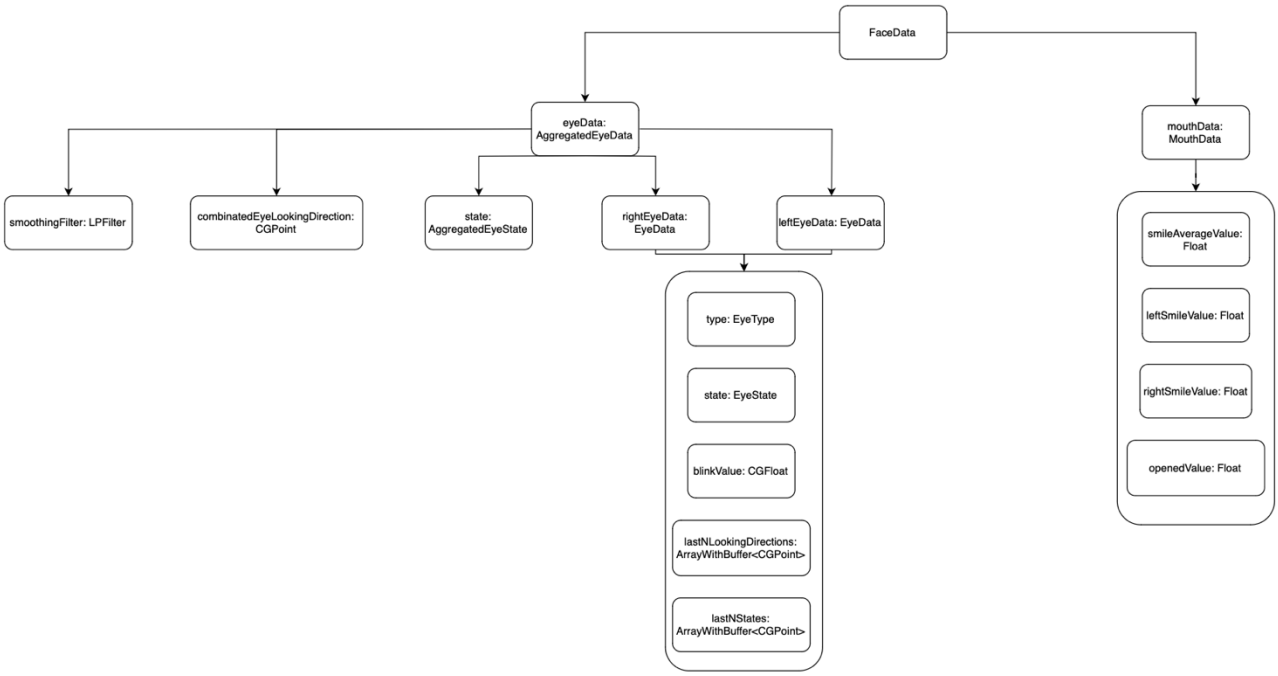


Рисунок 3.7 Архітектура шару моделі фреймворку

Уся зібрана інформація щодо користувача зберігається у окремо створеній структурі даних FaceData. Підхід до створення даної структури полягав в композиції різних елементів обличчя. Саме тому об’єктами, що входять до FaceData є AggregatedEyeData та MouthData. AggregatedEyeData зберігає в собі інформацію про поточний погляд очей та їх стан. Стан очей надається типом AggregatedEyeState та може набувати таких значень як закрите ліве око, закрите праве око, відкриті очі, закриті очі, ліве око моргнуло, праве око моргнуло, обидва ока моргнули. А параметр combinatedEyeLookingDirection, в свою чергу, повертає поточну координату погляду користувача. AggregatedEyeData як структура надає інформацію про стан очей виходячи з стану кожного ока окремо, дані про яких зберігаються у EyeData класі. Також зберігаються дані і про стан рота користувача, що включають в себе наскільки значення загальної посмішки, посмішки лівої та правої частин обличчя, а також наскільки відкритою є щелепа. Параметри набувають значень від 0.0 до 1.0. Опис використання зібраної інформації буде описано пізніше в розділі.

В рамках тестування точності визначення позиції погляду користувача, вищеописаний підхід показав певні недоліки у вихідних даних. Одним з таких є

висока чутливість до зміни погляду, що призводить до постійного коливання значень у суттєвих діапазонах навіть при концентрації погляду на одній точці. Іншою проблемою є неповноцінна зона слідкування за поглядом, що обмежувалась лише центральною частиною екрану. Було розглянуто способи зменшити дані ефекти.

Для більш точного розуміння динаміки в сторону покращення чи погіршення було створено тестовий інтерфейс в якому розміщено точку, на яку має бути сфокусовано погляд. Взято заміри погляду в центральну точку для розуміння відхилення в кожному з оглянутих способів.

Було оглянуто та досліджено способи збільшення точності вихідних даних та взято два для тестування.

Перший з них базується на згладжуванні даних шляхом взяття середнього значення на заданому діапазоні з останніх n результатів. Таким чином згідно гіпотези братиметься найбільш ймовірна точка куди дивиться користувач.

Так як природа предметної області обробки аудіо передбачає велику кількість шумів, було розглянуто способи вирішення цієї проблеми в ній. Одним зі знайдених методів є використання фільтру низьких частот. Даний підхід часто використовується в різних задачах, де стоїть потреба прибрати шуми. Окрім обробки аудіо, він застосовується в роботі з зображеннями для згладжування вхідних даних, в задачах з домену біології, до прикладу для попередньої обробки і фільтрування шумів чи артефактів у електрокардіограмі.

В рамках розробки фреймворку було реалізовано обидва методи та оглянуто їх продуктивність з різними налаштуваннями.

Для реалізації згладжування шляхом отримання середнього числа в структурі Eye було створено lastNLookingDirections, що містить останні n точок, які надійшли як локація куди дивиться дане око. Swift не має вбудованого типу даних масив, що містить лише фіксовану кількість елементів. Для реалізації такого типу даних було створено структуру, що відповідає протоколу RangeReplacableCollection. Даний протокол у Swift визначає необхідні для реалізації методи, що створюють колекцію, елементи в яку можуть додаватись,

видалятись, та редагуватись. Реалізація даного протоколу дозволяє створювати власні колекції, що можуть заміняти масив. Створений тип даних ArrayWithBuffer дозволяє на моменті ініціалізації визначати буфер, максимальну кількість елементів, яка може бути в ньому в один момент. При переповненні буферу, новий елемент заміщує найстарішого в колекції. Такий підхід дозволяє енкапсулювати логіку заміщення старих елементів новими та високорівнево керувати колекцією. В структурі Eye параметр lastNLookingDirections має тип ArrayWithBuffer та при надходженні нової точки зору користувача додає її в параметр. AggregatedEyeData отримуючи запит на читання параметру combinatedEyeLookingDirection у реалізації з вищеописаним способом згладжування утворює масив середніх значень агрегований з обох очей для горизонталі та вертикалі, та бере середнє значення в ньому. Таким чином отримується згладжене значення, що базується на останніх n. Число n визначається експериментально для пошуку оптимального значення.

Реалізація фільтру низьких частот полягала у створенні окремої структури даних LPFilter, що містить в собі імплементацію фільтру. Для ініціалізації надходить коефіцієнт згладжування, як число у діапазоні від 0.0 до 1.0. Наближення значення коефіцієнта до одиниці посилює ефект згладженості. Алгоритм тримає значення попереднього елементу та при кожному новому вхідному значенні обраховує згладжене за такими формулами:

$$smoothedValue.x = oldValue.x + (smoothCoefficient * (newValue.x - oldValue.x))$$

Формула 3.3 Обрахунок згладженого значення координати на осі x

та

$$smoothedValue.y = oldValue.y + (smoothCoefficient * (newValue.y - oldValue.y))$$

Формула 3.4 Обрахунок згладженого значення координати на осі x

де для відповідних осей $smoothedValue$ позначають нові значення, $oldValue$ – останнє згенероване згладжене значення, а $smoothCoefficient$ – коефіцієнт згладження.

Екземпляр класу LPFilter зберігається у AggregatedEyeData та використовується при читанні параметру currentEyeLookingDirection (рис. 3.7).

Було проведено тест на точність концентрації погляду та площу покриття слідкування за зором. Кожен тест проводився з новою збіркою додатку, до якого підключено фреймворк. Перед початком тестування обличчя не було видиме фронтальній камері смартфону, а початкова точка погляду була спрямована в лівий верхній кут. Кінцевий результат тесту – спроба сконцентрувати погляд в центрі екрану, що позначено кольоровим квадратом. Набори параметрів, на яких проведено тест:

- Імплементація без згладжування даних
- Імплементація з використанням згладжування шляхом взяття середнього значення. Буфер масиву – 10
- Імплементація з використанням згладжування шляхом взяття середнього значення. Буфер масиву - 15
- Імплементація з використанням фільтру низьких частот. Коефіцієнт згладжування – 0.5
- Імплементація з використанням фільтру низьких частот. Коефіцієнт згладжування – 0.8
- Імплементація з використанням згладжування шляхом взяття середнього значення з буфером 10 в комбінації з використанням фільтру низьких частот, коефіцієнт згладжування – 0.5

Відповідно до порядку тестів на зображенні результаті застосовувались такі кольори: червоний, жовтий, зелений, рожевий, коричневий та помаранчевий (рис. 3.8).

З результатів тесту отримано такі результати: відсутність жодного згладжування унеможливила концентрацію віртуального курсору в центр екрану та не є достатньо надійною для інтеграції у фреймворк. Наявність згладжування шляхом взяття середнього числа останніх n координат надала значно кращі результати. Проте зі збільшенням буферу зменшилась чутливість курсору до руху зіниць, що ускладнило покриття країв екрану без руху голови. В свій час використання фільтру низьких частот хоч і є кращим, ніж відсутність

згладжування, надає більшу чутливість що ускладнює концентрацію зору на потрібній точці. Змішаний алгоритм не показав кращих результатів та проявив себе як більш нечіткий.

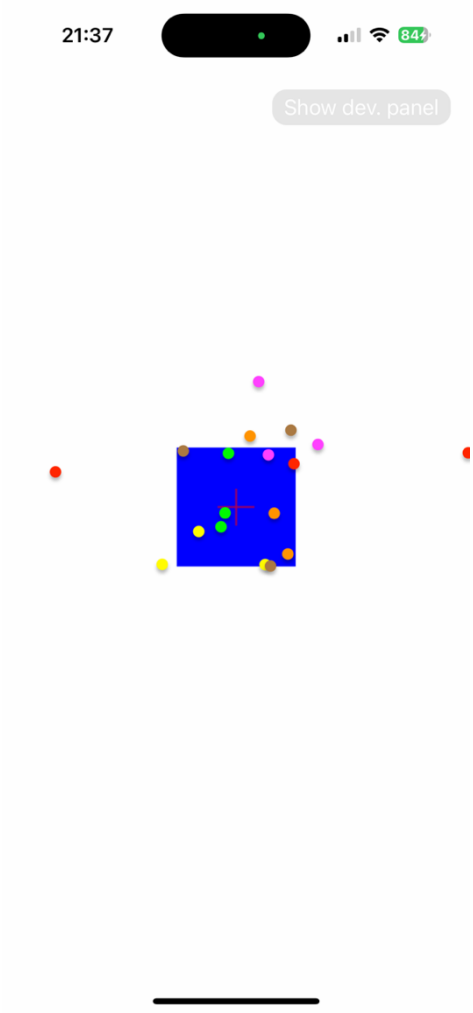


Рисунок 3.8 Результати тесту погляду в центр екрану

Враховуючи результати тестів було обрано у фінальну частину фреймворку інтегрувати метод згладжування базуючись на останніх 15 значеннях. Незважаючи на його обмеженість суто використовуючи рух зіниць, він продемонстрував найкращу точність серед інших претендентів.

3.3 Огляд реалізації керування зором за допомогою фреймворку

Отримання координат поточного погляду користувача дає велику платформу для реалізації управління зором в додатках сторонніх розробників проте вимагає великої роботи по парсингу отриманих координат у дії в

середовищі додатку. Додаткова робота в цьому напрямку потенційно зменшить кількість програмного забезпечення, розробники якого будуть інтегрувати фреймворк до себе. Тому в рамках роботи було заплановано реалізацію взаємодії користувача зором з основним функціоналом мобільного застосунку на iOS зі сторони фреймворку та без залучення великих ресурсів розробників програмного забезпечення, що інтегрують фреймворк в систему.

Перед початком роботи окреслено функціональні вимоги. Користувач має лише за допомогою зору працювати з елементами керування на екрані, мати змогу переходити на нові екрани, гортати існуючі, та надавати ввідні дані у програму, до прикладу текстовий ввід. Окрім цього, користувач має бачити поточну ідентифіковану позицію свого погляду та обраний об’єкт, а також елементи взаємодії з об’єктами керування мають бути зручними у використанні.

Ідея реалізації керування зором полягає у створенні батьківського класу UIViewController, який створюватиме архітектуру AI-сцени та зв’язок з іншими елементами фреймворку. Для розробника, що імплементує даний функціонал до себе в додаток, необхідно наслідувати свої UIViewController’и не від стандартного UIViewController, а від EyeControllableViewController. Таким чином ініціалізуватиметься робота фреймворку. Нижче наведено діаграму взаємодії класів в рамках імплементації керування зором (рис. 3.9).

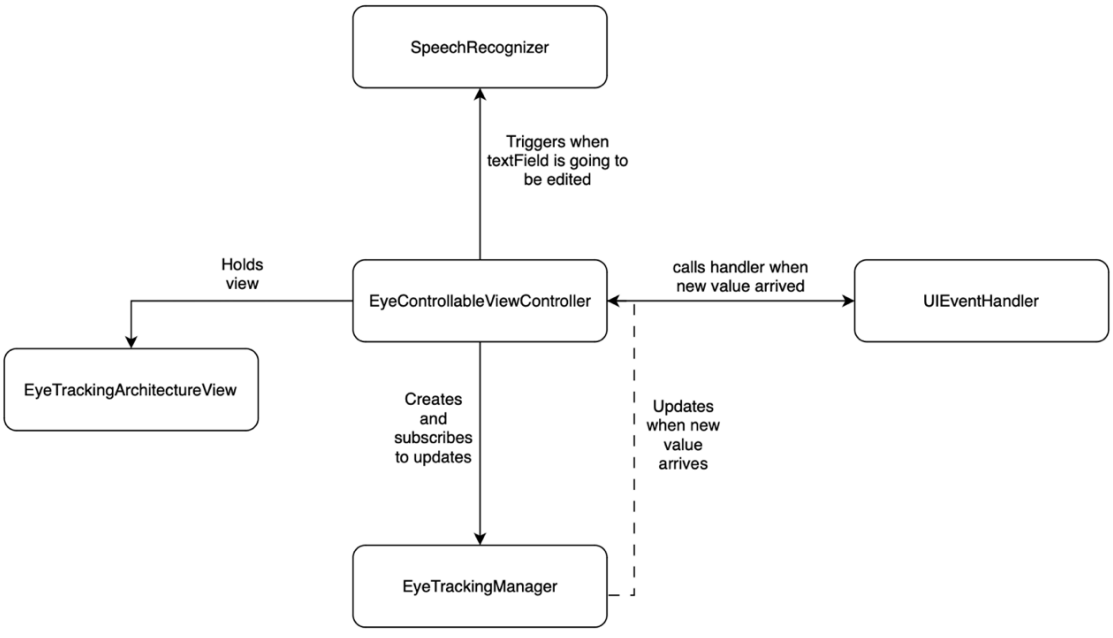


Рисунок 3.9 Діаграма взаємодії класів функціоналу керування зором

EyeControllableViewController створює об’єкти EyeControllableManaging та стає слухачем нових об’єктів FaceData. Для імплементації функціоналу очікування нових змін, що відбуваються в іншому класі, було обрано фреймворк Combine [35]. Він розроблений Apple та надає можливості реактивного програмування і роботи з асинхронним потоком даних в програмному забезпеченні написаному під Apple платформи. Об’єктом зміни якого відслідковуються є FaceData. Під час розробки даного функціоналу постала проблема вкладеності типів та слідкування за їх змінами. З діаграми на рисунку 3.7 бачимо, що FaceData має вкладені типи, що відповідають за стан очей та нижньої щелепи. Функціонал по слідкуванню за очима змінює вкладені об’єкти FaceData, проте не сам даний об’єкт. Відповідно базова імплементація слухання за станом FaceData не надсилає сповіщення про оновлення. Для вирішення проблеми було реалізовано виклики objectWillChange що в інтерфейсі Combine повідомляє про зміни. Виклики було реалізовано у функції, що оновлює стан об’єкту, що в свою чергу дозволило вручну слідкувати за змінами та надсилати сповіщення про них. FaceData об’єднувала сповіщення objectWillChange з MouthData та AggregatedEyeData і у разі надходження будь-якого з них також надсилала слухачам своїх змін objectWillChange (рис. 3.10). Це дозволило зберегти вкладеність та забезпечити підтримку Combine.

```
init() {
    Publishers.Merge(
                self.eyeData.objectWillChange,
                self.mouthData.objectWillChange
            )
            .sink(receiveValue: self.objectWillChange.send)
            .store(in: &self.cancellables)
}
```

Рисунок 3.10 Об'єднання сповіщень про оновлення вкладених об'єктів у FaceData

Після отримання нових даних FaceData у тілі EyeControllableViewController відбувається оновлення даних статистики, зміна позиції візуального індикатору точки куди дивиться користувач, та за допомогою функції hitTest здійснюється пошук найбільш релевантного

елементу, на якому зупинився курсор користувача. Функція hitTest в iOS розробці виконує функцію пошуку елементу на екрані з яким найбільш ймовірно взаємодіяв користувач. Для цього алгоритм проходить по ієрархії візуальних елементів на екрані перевіряючи чи лежить точка в площині елементу. У разі якщо так, то алгоритм рухається далі по дочірнім об'єктам елементу. Якщо ні, шукає інший. В кінці роботи алгоритму знаходиться візуальний елемент, з яким взаємодіяв користувач. При розробці фреймворку вручну запускається алгоритм hitTest за допомогою якого і отримується потенційний елемент взаємодії. Робота з ним передається класу EventHandler.

Даний клас відповідає за опрацювання вхідний подій ініційованих зором користувача. Основна ідея в обробці подій полягає в тому щоб симулювати натискання для знайденого елементу, якщо він є елементом взаємодії, таким як кнопка, слайдер, чи перемикач. Початкова ідея реалізації базувалась на використанні Responder chain. В системі iOS події передаються та обробляються екземплярами класу UIResponder, такими як UIView, UIApplication, та інші. UIKit керує тим яким чином подія доходить то елементу, що її обробляє. Реалізація Responder Chain полягає у зв'язному списку, де у кожного UIResponder є nextResponder і у разі якщо поточний об'єкт не готовий прийняти в обробку подію, вона передається об'єкту nextResponder.

Початкова концепція полягала у тому щоб в момент дії користувача створювати всередині фреймворку подію та передавати її екземпляру UIApplication, що в свою чергу використовуватиме функцію sendEvent(), яка на вхід приймає екземпляр об'єкту UIEvent, що створений через ініціалізатор використовуючи UITouch екземпляр, що міститиме локацію натискання на екран. За такої імплементації система б отримувала інформацію про натискання та обробляла подію правильно. В ході досліджень було виявлено, що написання такого функціоналу вимагає використання закритих API та окрім заборони на публікацію такого коду в App Store він не має гарантій у працездатності з новими оновленнями і може викликати неочікувану поведінку. Через це було оглянуто

та реалізовано більш комплексний спосіб керування та обробляти різні типи елементів у окремо виділеному класі, яким став EventHandler.

Важливим було покрити взаємодію зі UIScrollView, що відповідає за контейнер для візуальних елементів, який по висоті є більшим за висоту екрану та надає можливість гортати зміст. Важливість полягає в тому, що на відміну від набору таких елементів контролю, як кнопки, на одній локації перебуває лише один елемент з яким відбувається взаємодія. В цей час, UIScrollView перебуває під усіма елементами і тому майже завжди виникає ситуація, де на одній локації є і UIScrollView і інший елемент керування додатком. Для даного елемента фреймворк реалізує гортання екрану. Для реалізації скролу виділяється верхня ¼ частини UIScrollView та нижня ¼ частини компоненту. Після цього відбувається перевірка чи є поточна точка погляду користувача спрямована на одну з частин. Якщо так, то параметру lookingDirection надається значення top або ж bottom відповідно до зони погляду. Для уникнення випадкового гортання було оглянуто можливі зміни на обличчі, які можна було б фіксувати у зв’язці з поглядом для ініціалізації скролу. Таким було обрано посмішки лівої частини обличчя. Відповідно, коли користувач дивиться у верхню чи нижню частину екрану та робить усмішку лівою стороною обличчя, UIScrollView ініціалізує скрол на ¼ свого розміру. Обраний саме такий спосіб активації для уникнення випадкових натискань. Переважно всі елементи візуального інтерфейсу є нащадками UIScrollView, тому робота з таким універсальним об’єктом дає змогу закрити багато різних сценаріїв використання.

Після цього фреймворком перевіряються зміни у стані очей користувача на предмет виконаної дії. У рамках імплементації було проведено огляд можливих елементів контролю очима. Після тестування було виділено, що певні дії такі як блимання очима, є недостатньо надійними, адже трапляються у людей несвідомо та можуть вести до випадкових дій. Одними із тестів також була реалізація керування через блимання лише одного з очей. Даний спосіб показав себе краще на тестуваннях та був обраний як потенційний фінальний варіант. Проте було проведено дослідження, що одним з симптомів тиків є моргання

одного з очей [36]. Враховуючи наявність інших сценаріїв взаємодії було оглянуто і їх. Одним з розглянутих варіантів був такий, де дія з елементами контролю буде активуватись після ідентифікації лівого чи правого закритого ока. Закритість ока визначається у випадку, якщо стан блимання триває протягом останніх 10 подій. Таким чином мінімізується ймовірність випадкового натиснення на елемент та все ще зберігається зручність керування. Такий підхід є зручним для користувача та мінімізує ймовірність випадкової активації якоїсь з дій, проте поточна імплементація слідкування за поглядом в момент закриття ока ідентифікує його зсув, що часто призводить до зміщення позиції з бажаної. Через це було відхилено розгляд керування зором та проаналізовано можливості інших елементів міміки обличчя. Серед варіантів відкриття щелепи як функція активації показав найкращий з результатів. При відкритті щелепи позиція очей не змінюється, а також даний рух не є таким, що постійно виконується людиною за використання додатку.

Для легкого повернення на попередній екран окремо було додано використання закритого правого ока для виклику функції закриття поточного екрану, адже дана дія не вимагає наведення на конкретний елемент.

Розглянемо обробку відкриття щелепи, що симулює процес натискання на елемент. Під час підготовки до написання даного функціоналу було оглянуто існуючі елементи, що можуть обробляти натискання користувача та виділено основні, що зустрічаються в інтерфейсах користувача та які є можливим імплементувати, а також ті, імплементувати які напряму не є можливим. Серед можливих для імплементації було виділено стандартну UIButton, UISwitch, UISlider, UITextField. Багато елементів інтерфейсу є нащадками вищезазначених класів, що дозволяє реалізувавши функціонал для базових типів також покрити і їх модифіковані версії. Серед тих типів, реалізація яких не була можлива напряму, є UIStepper, UISegmentControl, та UITabBar. Принцип покладений в роботу EventHandler полягає в тому, що UIView, яка подана на вхід, покроково виконує конвертацію у ті типи даних обробка яких підтримується. У разі якщо

жоден тип не було знайдено або знайдений тип не підтримує обробку події, то вона ігнорується.

UIButton – стандартний тип, що представляє кнопку у UIKit, є найпоширенішим елементом для обробки подій. Він може набувати різних відображень, та має однакову систему обробки натискання користувачем. У випадку керування зором, якщо знайдена UIView є типу UIButton, то у неї викликається функція sendActions(for: .touchUpInside), що симулює подію натискання.

UISwitch – окремий клас елементів керування, що є перемикачем та може мати лише 2 стани – увімкнений та вимкнений. Реалізація натискання виконана через надання перемикачеві стану, який є протилежним до поточного. Таким чином користувач натиснувши на елемент надасть йому протилежного значення – перемкне.

Реалізація керування UISlider не стала настільки ж очевидною. Даний елемент відповідає за слайдер, значення якого задається пересуванням повзунка пальцем. Для того аби керування зором було можливе з даним елементом, було розроблено окремий алгоритм, в якому спочатку знаходимо мінімальне та максимальне значення в координатах для слайдера на осі x. Далі, маючи координату погляду користувача, знаходимо на якому відсотку слайдера він знаходиться за формулою:

$$percentageSelected = \frac{(currentLookingX - sliderMinX)}{(sliderMaxX - sliderMinX)}$$

Формула 3.5 Обрахунок обраного відсотку в слайдері

Верхня частина формули знаходить поточну позицію відносно початку та ділиться на загальну ширину слайдера. Знайшовши відносне до діапазону можливих значення слайдеру, обране користувачем, з’являється можливість знайти нове значення у абсолютних величинах, яке можна передати UISlider для оновлення його відображення. Використано наступну формулу:

$$value = (sliderMaxValue - sliderMinValue) * percentageSelected + sliderMinValue$$

Формула 3.6 Конвертація відсоткового значення в абсолютне

Даний підхід дозволяє користувачеві однією дією перетягнути елемент у бажане місце.

UITextField у Swift надає поле для вводу тексту користувачем. Загальним методом вводу є системна клавіатура. Так як логіка керування зором є окремо прописаним функціоналом, що реалізований як звичайний UIViewController, то його елементи, включаючи необхідні для взаємодії з користувацьким інтерфейсом, не мають доступу до системної клавіатури та її відображення. Що призводить до того, що ввід тексту через керування зором не є можливим. Для того щоб надати користувачам керування зором можливість вводити текст було оглянуто можливості для використання голосового вводу. Проте, незважаючи на наявність даного підходу в iOS, його використання обмежене системою, а можливість викликати клавіатуру відразу з голосовим вводом відсутня. Через відсутність можливості вирішити питання вводу системним функціоналом, було реалізовано власний підхід до введення інформації голосом. У випадку якщо знайдена UIView на яку відбулося натиснення є типом UITextField, EventHandler викликає функцію becomeFirstResponder() у неї. Виклик даної функції призводить до того, що поле UITextField стає активним та за стандартної поведінки з’являється клавіатура. Для приховування клавіатури, враховуючи відсутність необхідності у ній, адже керування зором її не підтримує, та наступних дій по обробці взаємодії з UITextField, базовий клас EyeTrackableViewController було зроблено делегатом UITextField. Це надало можливості вносити зміну в поведінку під час різних подій життєвого циклу UITextField. Для реалізації вводу голосом необхідно два методи з делегату, а саме textShouldBeginEditing(_ textField: UITextField) -> Bool та textDidEndEditing(_ textField: UITextField) -> Bool. Метод textShouldBeginEditing ініціалізує запуск ідентифікації голосу, стає слухачем результатів ідентифікації, та робить видимим об’єкт speechKeyboardView у eyeTrackingArchitectureView.

Окрім цього, даний метод повертає завжди false, що блокує показ системної клавіатури. Для імплементації голосового введення тексту було створено окремий клас SpeechRecognizer, який енкапсулює в собі логіку по роботі з ідентифікацією голосового вводу та повертає результат в текстовому форматі. SpeechKeyboardView з'являється внизу екрану як індикатор початку запису, а натиснення на нього зупиняє запис голосового вводу. Метод textDidEndEditing зупиняє ідентифікацію голосового введення та приховує SpeechKeyboardView (рис. 3.11).

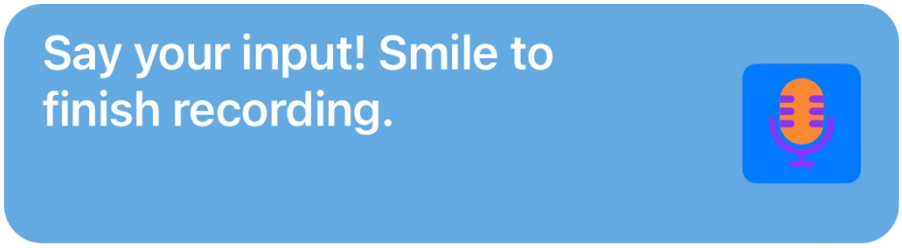


Рисунок 3.11 SpeechKeyboardView

SpeechRecognizer використовує фреймворк Speech, розроблений Apple, для розпізнавання вимовлених користувачем слів та AVFoundation для запису голосу. При створенні об'єкту SpeechRecognizer відбувається запит на дозвіл використання користувачем мікрофону. При виклику функції startRecording() першим чином відбувається перевірка доступності об'єкту SFSpeechRecognizer з фреймворку Speech. У разі якщо він доступний, відміняються всі минулі запити на розпізнавання голосу, створюються необхідні для цього класи, та ініціалізується SFSpeechRecognitionTask, об'єкт, що відповідає за процес розпізнавання мовлення. Для роботи йому передається SFSpeechAudioBufferRecognitionRequest, що сприймає аудіо на вхід у форматі буферів та передає SFSpeechRecognitionTask на обробку та розпізнавання. Вхідне аудіо для нього надходить з методу AVAudioInputNode installTap, який встановлює доріжку з параметрами налаштування, такими як розмір буферу та іншими. Таким чином аудіо проходить від мікрофону до задачі розпізнавання. Отримавши розпізнаваний текст, йде його присвоєння параметру currentRecognizedText, а у разі якщо даний текст є фінальним, то і зупинка

процесу розпізнавання. Так як користувачеві важливо бачити сказаний ним текст ще в процесі його створення, SpeechRecognizer та його складові налаштовані таким чином, що текст розпізнається паралельно з прослуховуванням мікрофону, що дає змогу відразу його генерувати. Це стає можливим за допомогою значення true параметру shouldReportPartialResults у voiceRequest. Повний код розпізнавання вводу від користувача надається в додатку 3.

Розглядаючи типи об’єктів, реалізація взаємодії з якими була неможлива напряму, було проаналізовано шляхи вирішення керування. Найчастіше така проблема виникала через приватні типи об’єктів, які відповідають за взаємодію, що унеможливило їх точну ідентифікацію та активацію з коду через відсутність доступу.

До прикладу, UIStepper. Даний об’єкт тримає певне значення, яке можна збільшити або зменшити на задану одиницю. Приклад UIStepper наведено на зображенні нижче (рис. 3.12).

Рисунок 3.12 Приклад UIStepper

У випадку UIStepper, елемент не дає можливості доступу до окремих елементів додавання та віднімання. Для реалізації керування очима даним елементом було створено об’єкт UIStepperWrapper, що є обгорткою над стандартною реалізацією. Обгортка містить параметр action, що може набувати значення up у випадку збільшення числа та down у випадку його зменшення. У функції viewDidAppear класу EyeControllableViewController, що викликається після появи екрану, відбувається пошук об’єктів для яких задіяна непряма реалізація взаємодії поглядом. Так як відомо, що UIStepper має лише два дочірніх елементи, у разі його знаходження у ієрархії створюються два екземпляри об’єкту UIStepperWrapper відповідно з action top та action bottom, що накладаються поверх відповідних візуальних елементів UIStepper. EventHandler в свою чергу не шукає елемент UIStepper, а вже UIStepperWrapper. При

знаходженні даного елементу, EventHandler збільшує або зменшує, залежно від параметру action у UIStepperWrapper, значення UIStepper.

Схожий підхід застосовується і для UISegmentedControl. Даний візуальний елемент схожий за будовою на UIStepper, проте використовується для переключення різних сегментів, кількість яких може бути більшою, ніж дві. Приклад вигляду UISegmentedControl наведено на зображенні нижче (рис. 3.13).

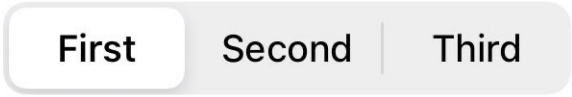


Рисунок 3.13 Приклад UISegmentControl

Було створено UISegmentedControlWrapper, що містить в собі цілочисельний параметр index. У разі знаходження UISegmentedControl серед елементів екрану обраховується ширина кожного окремого сегменту. Для цього загальна ширина UISegmentedControl ділиться на кількість сегментів. Після на кожен з них накладається UISegmentedControlWrapper з присвоєним параметром index, що відповідає індексу відповідного сегменту та шириною, що була обчислена для кожного з них. В EventHandler у разі потрапляння зору саме на UISegmentedControlWrapper виконується присвоєння параметру selectedSegmentIndex відповідного індексу, що був збережений у обгортку.

UITabBar - елемент контролю, що надає функціонал переключення між різними екранами чи задачами шляхом кнопок в нижній секції екрану. Найчастіше в iOS розробці UITabBar є використовуваним у зв’язці з UITabBarController, що є нащадком UIViewController та слугує контейнером для дочірніх екранів, які можна переключати за допомогою UITabBar. Наразі існуючий EyeTrackableViewController є нащадком UIViewController, а не UITabBarController, що унеможливлює використання керування зором у разі застосування UITabBarController. Для надання функціоналу керування зором на базі UITabBarController було створено окремий клас EyeTrackableTabViewController, що є нащадком від UITabBarController та забезпечує передачу UITabBar у EventHandler. Його інший функціонал є ідентичним до такого в EyeTrackableViewController, а тому опис роботи з

останнім в даній роботі також відноситься і до EyeTrackableTabViewController. Для реалізації керування зором UITabBar через приватність типу, що використовується для окремих кнопок, був використаний ідентичний підхід до того, які є в UIStepper та UISegmentedControl. Створено окремний клас UITabBarButtonWrapper, що має параметр index, який відповідає за порядок кнопки в UITabBar та розміщується поверх UITabBar. У випадку натискання зором на UITabBarButtonWrapper, EventHandler системно викликає зміну індексу UITabBar на той, який міститься у натисненому UITabBarButtonWrapper.

В рамках роботи над фреймворком було опрацьована ймовірність того, що сторонні додатки в якості кнопок, що мають стандартний тип UIButton, використовуватимуть певний власний тип даний або ж UIView. Така ситуація трапляється коли виконуються задачі по розробки елементів з налаштованим зовнішнім виглядом, що суттєво відрізняється від того як може виглядати UIButton. Було створено протокол EyeTappableViewProviding, який зобов’язує такі класи мати topTappableView, що має тип EyeTappableView. EyeTappableView – клас, що EventHandler ідентифікує при пошуку отримувача події. Знайшовши його, EventHandler викликає метод tap(type: UIControl.Event), що в свою чергу викликає функцію onTap, яка має налаштовуватись вже розробниками додатку, адже onTap визначає дію, що відбудеться внаслідок натискання.

Для уникнення обробки подій зайвий раз у фреймворку прописані умови, що дозволяють уникати дуплікації подій користувачем шляхом перевірки нової події з попередньої на предмет ідентичності. Таким чином одна дія поглядом користувача еквівалентна одній дії і в самому додатку.

Для кращого досвіду користувача на екрани де присутнє керування зором додається індикатор погляду, що показує де саме алгоритм визначає точку погляду користувача (рис. 3.14). Окрім цього, червоним кольором виділяється елемент на екрані, який буде опрацьований у разі натискання. Таким чином користувач чітко розуміє як програма сприймає його погляд. Індикатор створений як підклас UIView та змінює свою позицію у

EyeControllableViewController при отриманні нового стану FaceData, а червоне виділення надається об’єкту у EventHandler шляхом зміни параметру borderColor.

Рисунок 3.14 Зовнішній вигляд індикатору

3.4 Огляд додаткового функціоналу для фреймворку

Окрім функціоналу передбаченого для вирішення першочергової задачі роботи, а саме надання людям з обмеженими можливостями доступ до керування телефоном за допомогою зору, розроблений фреймворк також надає декілька інструментів для розробників, що інтегрують його до себе в застосунок.

Одним з таких є можливість логування. Під час відлагоджування та тестування програмного забезпечення, розробникам важливо розуміти поточний стан роботи програми. Окрім цього, при виникненні помилок іноді необхідно мати збереженими записи про стан та роботу програми в ітерації, коли проблема виникла. Такі потреби і вирішує логування. Логування – це практика запису інформації під час роботи програмного забезпечення. Apple надає стандартні інструменти для його здійснення, а саме os_log та починаючи з iOS 14.0 Logger. Порівнюючи вищевказані способи з стандартним print() та debugPrint() існують різного роду переваги. По-перше, os_log та Logger способи є оптимізовані компілятором, що забезпечує швидке виконання, в той час як print() та debugPrint() ні, що у високонавантажених програмах відіграє негативну роль. По-друге, print() та debugPrint() надають лише можливість вивести повідомлення в консоль, в той час як os_log та Logger мають змогу зберігати повідомлення на пристрої, а також надають повідомленням тип, джерело в коді звідки дане повідомлення згенеровано, а також час коли подія трапилась. Це дозволяє краще шукати необхідні повідомлення та більш швидко відлагоджувати програму.

Для фреймворку було створено загальний протокол LogProviding, що визначає необхідні методи для створення власного логеру, а також протокол

AdditionalLogProviding, який мають наслідувати додаткові логери, якщо такі є (рис. 3.15).

```
protocol LogProviding {
    func logMessage(_ message: String, level: OSLogType)
    func appendAdditionalLoggers(customLoggers: [AdditionalLogProviding])
}

protocol AdditionalLogProviding {
    func logMessage(_ message: String, level: OSLogType)
}
```

Рисунок 3.15 Протоколи для створення логеру

Функція logMessage(_ message: String, level: OSLogType) логує надане повідомлення з відповідним рівнем. Загалом існує 5 різних рівнів логування: default, error, fault, info, debug. Повідомлення рівня default, error, та fault завжди зберігаються на пристрої, рівня info лише у випадку появи fault рівня під час виконання програми, а debug завжди присутні лише під час роботи програми. ’

У випадку наявності інших точок куди програма надсилає логи створено протокол AdditionalLogProviding. Створені за цим протоколом екземпляри можна надавати у функцію appendAdditionalLoggers(customLoggers: [AdditionalLogProviding]) і їх функція logMessage буде виконана при виклику аналогічної у екземпляра LogProviding. Таким чином одним викликом функції логування можна виконувати його на різні точки, до прикладу на сервер.

Екземплярами LogProviding протоколу було створено класи LogProvider та LogCompatibilityProvider. Враховуючи, що фреймворк підтримує версію iOS і нижче iOS 14.0 неможливо було використовувати новий Logger клас. Для того щоб не підвищувати мінімальну підтримувану версію було розроблено дві реалізації: одну для iOS 14.0 та вище, та іншу для версій нижче, яка використовує os_log. Повна версія реалізації логування надана в додатку 4.

Іншим важливим функціоналом для відлагодження додатку що має підтримку керування зором є можливість під час тестування бачити поточний стан обличчя на екрані додатку. Для надання такого функціоналу була створена панель розробника (рис. 3.16).

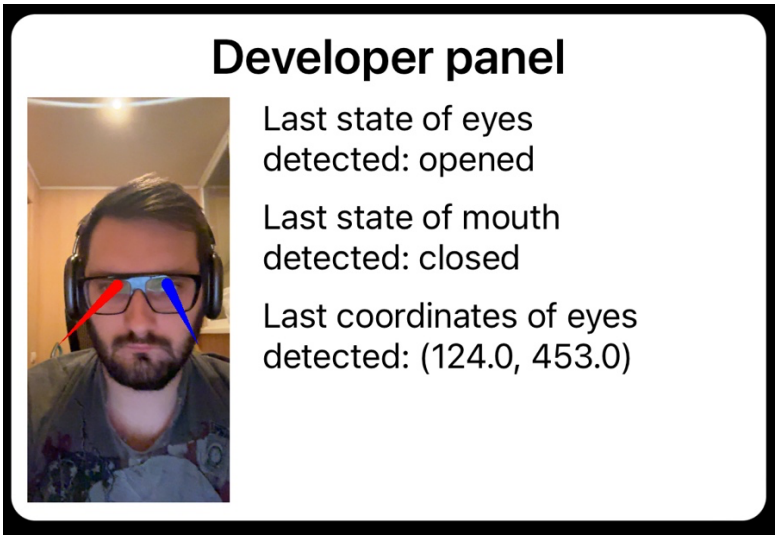


Рисунок 3.16 Панель розробника

Вона містить зображення з фронтальної камери на якому розміщено два проміні з кожного з очей, що спрямовані в сторону погляду користувача, а також інформацію про те який зараз EyeState, MouthState, та координати погляду. Таким чином розробник може тестувати додаток та отримувати більше контексту щодо того яким чином працює керування зором на поточному екрані. Даний елемент зберігається у EyeTrackingArchitectureView та з'являється поверх всіх інших екранів. Його поява можлива лише на тестових збірках, щоб унеможливити появу даного елементу для справжніх користувачів та активується закриттям очей.

Також після дослідження конкурентів взято до уваги було можливості налаштування поведінки фреймворку. Для цього створено структуру EyeTrackingConfig, що містить в собі статичні параметри, які дозволяють конфігурувати певні точки у роботі фреймворку (рис. 3.17).

```
struct EyeTrackingConfig {
    static var isDevModeEnabled: Bool = true
    static var deviceDimensions: CGRect = UIScreen.main.bounds
    static var shouldLoggingBeEnabled: Bool = false
    static var shouldHighlightFocusedViews: Bool = true
    static var subsystem: String = "com.framework.visionary"
    static var smoothFilterCoefficient: CGFloat = 1.0
}
```

Рисунок 3.17 Структура EyeTrackingConfig

Серед обраних для конфігурації параметрів є isDevModeEnabled, що у разі значення true не додає до архітектури екрану панелі розробника,

deviceDimensions, що визначає які розміри екрану треба використовувати для обрахування координати погляду. Даний параметр має значення за замовченням, що дорівнює реальним показникам розміру екрану, проте може змінюватись за бажанням розробників у випадку якщо використання керування зором обмежене до певної зони. Іншими параметрами для конфігурування є shouldLoggingBeEnabled, що активує або деактивує логування для фреймворку, shouldHighlightFocusedViews контролює чи використовується підсвітка для елементів екрану, на яких сконцентрований погляд користувача. Параметр subsystem належить до конфігурації логування. Він використовується для ідентифікації системи яка генерує логи, зазвичай на його місці використовується ідентифікатор додатку. Сторонні розробники мають змогу на старті додатку налаштувати його під свій продукт. Останнім серед параметрів конфігурації є smoothFilterCoefficient, що надає змогу увімкнути фільтр низьких частот для алгоритму. Значення більше 0.0 вмикає його використання та що впливає на те, наскільки сильним буде згладжування у фільтрі низьких частот.

3.5 Аналіз ефективності роботи функціоналу та впливу на систему

Важливим з точки зору розробки фреймворку є аналіз впливу його наявності на систему. Враховуючи, що система керування зором є постійно активною під час роботи програми та використовує фронтальну камеру смартфону, поставлена гіпотеза що споживання ресурсів пристрою збільшується та заряд батареї падає значно швидше. Для перевірки гіпотези було проведено тест. Смартфон моделі iPhone 14 Pro у його звичному для користуванні стані, а саме з увімкненим WiFi, без режиму енергозбереження, було використано у тестуванні додатку з та без підключеного керування зором протягом 60 хвилин з фіксацією рівня заряду батареї кожні 10 хвилин. Даний тест дозволив зрозуміти динаміку використання заряду при використанні створеного фреймворку. Результати на графіку нижче (рис. 3.18).

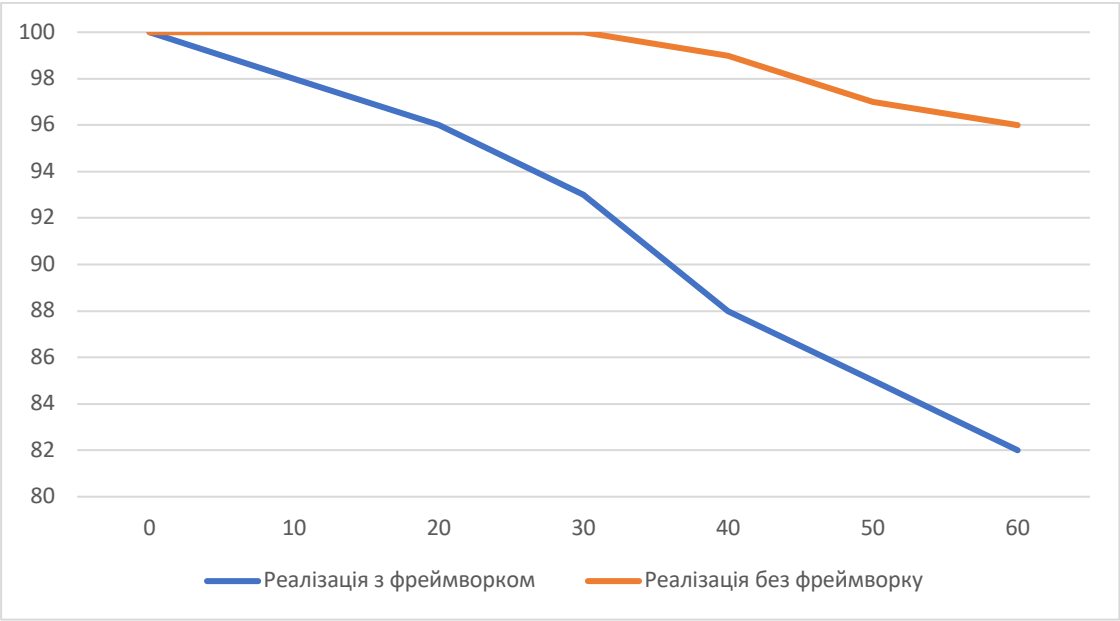


Рисунок 3.18 Порівняння рівня акумулятора у % з та без використання керування зором

З результатів чітко видно, що наявність фреймворку у застосункові негативно впливає на рівень заряду акумулятора. За годину роботи додатку, рівень заряду у версії, де був підключений фреймворк та активоване керування зором упав на 18% зі 100% до 82%. Версія без підключеного фреймворку вплинула на рівень заряду менш негативно і відбувся спад лише на 4% – з 100% до 96%. В результаті версія з підключеним фреймворком впливає на 127% гірше на довговічність смартфону.

Окремо було розглянуто показники завантаженості системи під час роботи додатку, такі як завантаженість процесору та оперативної пам’яті, а також енергетичний вплив з точки зору ОС (рис. 3.19, рис. 3.20).

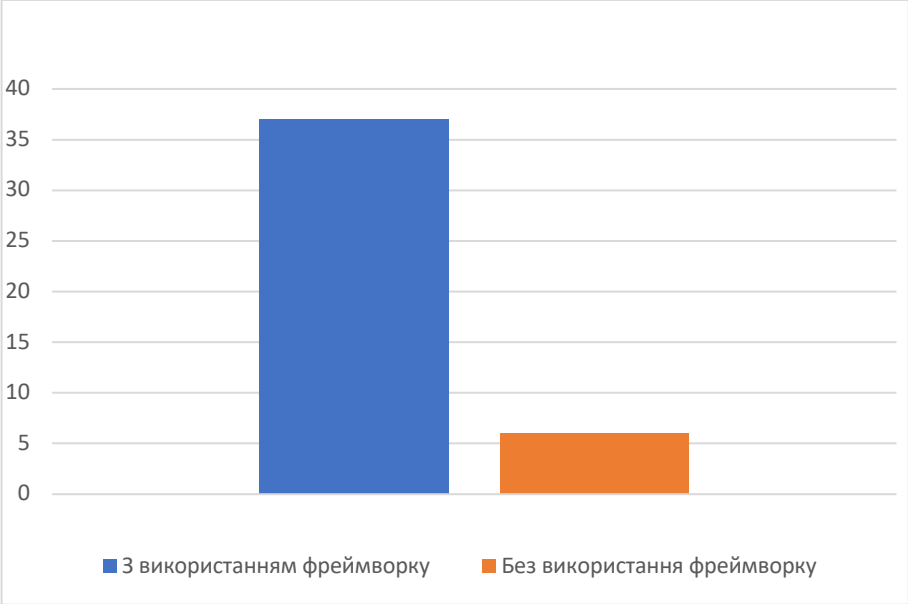


Рисунок 3.19 Середнє використання ресурсу процесору у %

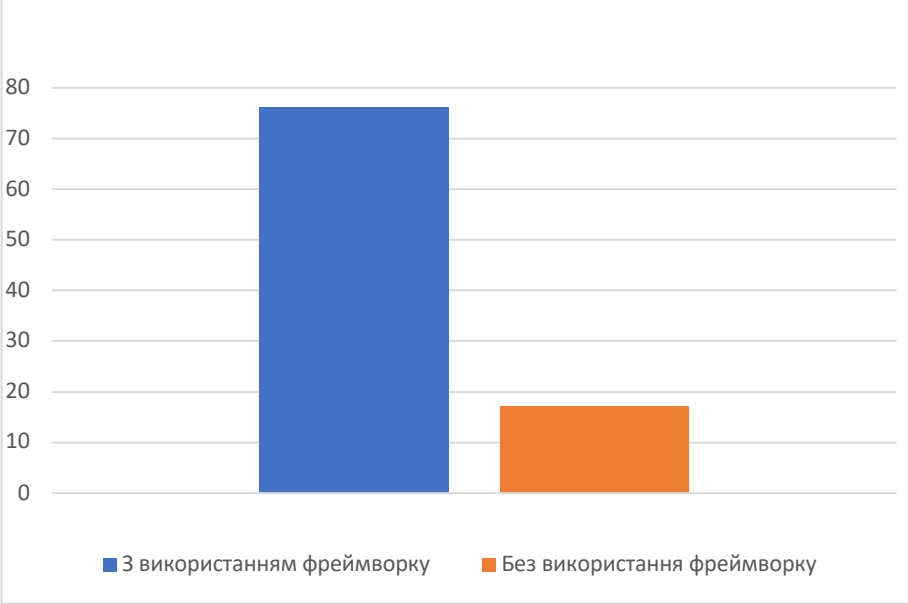


Рисунок 3.20 Середнє використання оперативної пам'яті у мегабайтах

Взявши середні значення показників отримано, що з використанням фреймворку процесор завантажений на 37%. Нижня межа 24% та верхня 50%. За цим же показником, відсутність фреймворку дозволяє завантажувати процесор на 6% з нижньою межею в 0% та верхньою в 12%. Схожа різниця спостерігається і у використанні оперативної пам’яті. У запуску з фреймворком отримано 76 мегабайт використання пам’яті та 17 мегабайт без нього. Щодо енергетичного ресурсу, ОС сприймає роботу з фреймворком на рівні між низьким та високим споживанням енергії, в той час як відсутність фреймворку завжди тримає

енерговикористання на низькому рівні, що підтверджується тестами на заряд батареї описані вище.

Загалом бачимо, що наявність фреймворку суттєво вимогливіша до ресурсів смартфону, а також більше енерговитратніша, аніж робота з його відсутністю. Наступні кроки по роботі включають в себе дослідження на предмет оптимізації витрати ресурсів під час роботи фреймворку.

3.6 Зони для покращення фреймворку та наступні кроки

Після розробки фреймворку, його інтеграції в проект та досвіду використання, було проаналізовано та виділено зони для покращення, а також наступні кроки в його розвитку.

Згідно аналізу впливу фреймворку на систему, його наявність значно погіршує час роботи смартфону, а також негативно впливає на завантаженість системи. Незважаючи на те, що за час роботи додатку з фреймворком температура телефону не підіймалась, а продуктивність з точки зору досвіду користувача не погіршилась, факт такої взаємодії з системою може негативно впливати і на бажання сторонніх розробників підключати фреймворк. Виходячи з цього, оптимізація його роботи є одним з найважливіших наступних кроків для успішного просування. Потенційними зонами, які можуть бути покращені, є дослідження щодо можливості вимкнення зайвих процесів у період коли користувач не дивиться на екран або ж менш часте оновлення даних локації.

Досліджуючи конкурентів було виявлено те, що вони орієнтуються переважно на iPad, планшети компанії Apple, а також надають власне залізо для слідкування за очима. Розробивши та протестувавши фреймворк на iPhone з використанням лише вбудованих сенсорів спостерігається похибка у слідкування за рухом саме зіниць. Такий ефект трапляється через ту причину, що фізична площа екрану смартфону є маленькою і відповідно граничні точки руху зіниць не завжди є достатніми для точної ідентифікації місця куди дивиться користувач. В поточній імплементації дана проблема вирішується тим, що у ситуації де рух зіниці не дозволяє довести точку до необхідного місця невеликий

рух голови у потрібну сторону. Пошук інших алгоритмів для оптимізації координати погляду, окрім розглянутих, може допомогти покращити користувацький досвід.

Окрім вирішення наявних проблем, також окреслено наступні кроки в розвитку фреймворку. Одним з таких є пошук каналів маркетингу для поширення обізнаності про фреймворк та збільшення потенційної частки додатків, що мають його підключеним. Також важливим кроком є розвиток окремої платформи, що буде консолідувати знання про всі додатки, що мають підтримку керування зором. Таким чином користувачам, які мають необхідність в керуванні додатками за допомогою погляду, буде просто знайти ті застосунки, що вже підтримують таку технологію. Для розробників застосунків це слугуватиме, окрім підтримки інклюзивності, також додатковим розміщенням продукту на сторонньому ресурсі, що дозволить збільшити охоплення потенційними користувачами. Окремо від наведених вище кроків стоїть тестування додатку на реальних користувачах шляхом проведення юзер тестів та збору якісних даних про їх досвід. Це дозволить зрозуміти окремі переваги та недоліки присутні у такому вирішенні проблеми, а також з упевненістю випускати наявне програмне забезпечення на всіх користувачів.

3.7 Висновки до розділу 3

В даному розділові було оглянуто практичну реалізацію фреймворку. Перед розробкою проаналізовано можливості для розповсюдження фреймворків призначених для пристроїв Apple та серед оглянутих обрано Swift Package Manager через його інтеграцією з інструментами Apple, мовою програмування Swift, а також легкість взаємодії.

Після вибору способу дистрибуції було описано технічну реалізацію системи слідкування за поглядом користувача, її залежності та можливості. Після отримання інформації про координату погляду користувача, фреймворк обробляє потенційну подію та надсилає її застосункові, що його підключив.

Третій розділ фокусується на описі того як саме обробляються дії користувача та яким чином відбувається керування додатком лише за допомогою зору.

Фреймворк також надає різні інструменти для покращення досвіду роботи з ним, що описані в огляді додаткового функціоналу. Розглянуто панель розробника, можливість конфігурувати поведінку через структуру налаштувань, та логування подій.

Після закінчення розробки та проведення тестів було оцінено вплив фреймворку на систему. Результати показали більші витрати заряду батареї та ресурсів смартфону, ніж без використання фреймворку, що створило зону для майбутнього покращення. Окрім цієї, в розділові також було розглянуто і інші точки росту фреймворку. Серед виділених, було сформовано потенційні зони для вирішення наявних проблем, а також окреслено інші з кроків по розвитку проекту.

Висновки по роботі

Результатом роботи є створений на мові Swift фреймворк, що реалізує функціонал керування зором в сторонніх додатках на пристроях під управлінням iOS. Перевагами фреймворку є легкість підключення, реалізація обробки подій зором на стороні фреймворку з малою залученістю розробників додатку, який його підключає. Також з переваг варто відмітити можливість відключити обробку подій та лише отримання координат для власної обробки. Окрім цього, фреймворк надає функціонал логування подій, додавання власних логерів для єдиної точки їх обробки, панель розробника для зручного відлагодження роботи фреймворку та застосунку, а також можливість налаштовувати окремі його параметри.

В першому розділі було оглянуто предметну область, досліджено її на предмет актуальності. Окрім цього, проведено дослідження конкурентів, що вже надають схожий функціонал на ринку. Враховуючи велику кількість людей з вадами моторики кінцівок, дороговизну наявних інструментів, що надають можливість керування зором для них, а також підтримку лише пристроїв iPad, було визначено актуальною роботу, метою якої стане модифікація взаємодії зі смартфоном для даного сегменту користувачів. В розділі також було оглянуто і інші потенційні сценарії користування смартфоном лише зором.

Другий розділ містить огляд та дослідження теоретичних відомостей про технології та підходи, що необхідні для реалізації поставленої задачі. Також в розділові покрито їх аналіз та вибір конкретних для використання у написанні фреймворку. Обрано підхід для реалізації слідкування за поглядом користувача за допомогою ARKit. Також в розділові обґрунтовано вибір Swift як мови програмування фреймворку та UIKit як фреймворку для інтерфейсу користувача. Окремо поставлено питання формату в якому буде вирішена задача: а саме переваги та недоліки створення окремого додатку або ж написання фреймворку.

В розділі три описана практична реалізація проекту, окреслено переваги та недоліки вже готового продукту, а також оцінено його вплив на систему. Також

розділ містить огляд існуючих можливостей для дистрибуції фреймворків та вибір серед них, їх переваги та недоліки. Серед найпопулярніших систем залежностей було обрано Swift Package Manager. Розділ містить опис архітектури фреймворку, реалізації окремих функціональних одиниць. Після описаної практичної імплементації, розділ фокусується на аналізі готового рішення, його впливі на систему, та подальших кроків по його розвитку.

Підсумовуючи, написане рішення реалізує можливість керування зором в сторонніх додатках під управлінням iOS, дозволяє легко створювати як новий додаток, так і підключати до існуючого. Дана розробка допомагає зробити простір додатків на пристрої під управлінням iOS більш інклюзивним, та надати більшій кількості людей можливість користуватись застосунками, саме тому ціль роботи вважається виконаною. Існуючі проблеми та зони для покращення окреслені в третьому розділі та будуть виправлені протягом життєвого циклу фреймворку з наступними оновленнями.

Список джерел

1. The Mobile Economy 2022 [Електронний ресурс] // GSMA. – 2022. – Режим доступу до ресурсу: 1. https://www.gsma.com/mobileeconomy/wp-content/uploads/2022/02/280222-The-Mobile-Economy-2022.pdf. – Назва з екрана.
2. 2018 Global Mobile Consumer Survey: US Edition [Електронний ресурс] // Deloitte. – 2018. – Режим доступу до ресурсу: https://www2.deloitte.com/tr/en/pages/technology-media-and-telecommunications/articles/global-mobile-consumer-survey-us-edition.html. – Назва з екрана.
3. Morris J. T., Sweatman W. M., Jones M. L. Smartphone Use and Activities by People with Disabilities: User Survey 2016. Journal on Technology and Persons with Disabilities. 2017. Режим доступу до ресурсу: https://scholarworks.csun.edu/bitstream/handle/10211.3/190202/JTPD-2017-p50-66.pdf?sequence=1.
4. American Community Survey [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://data.census.gov/table?t=Disability&tid=ACSST1Y2018.S1810&hidePreview=true. – Назва з екрана.
5. Get started with accessibility features on iPhone [Електронний ресурс] – Режим доступу до ресурсу: https://support.apple.com/en-gb/guide/iphone/iph3e2e4367/ios. – Назва з екрана.
6. TD Pilot [Електронний ресурс] // Tobii Dynavox Global. – Режим доступу: https://www.tobiidynavox.com/pages/tdpilot. – Назва з екрана.
7. TD Snap communication platform - AAC apps/software [Електронний ресурс] // Tobii Dynavox US. – Режим доступу: https://us.tobiidynavox.com/pages/td-snap. – Назва з екрана.
8. Skyle 2 for iPad – eyeV – The world's first eye tracker for the iPad. [Електронний ресурс] // eyeV – The world's first eye tracker for the iPad. –

Eyetracking for iPad and Windows. – Режим доступу: https://eyev.de/en/ipad/. – Назва з екрана.

9. Hawkeye Access | Control your iOS device using your eyes [Електронний ресурс] // Hawkeye | Learn where people look in your products. – Режим доступу: https://www.usehawkeye.com/accessibility. – Назва з екрана.
10. SeeSo::The gaze tracker [Електронний ресурс] // Eye Tracking Software: Optimize Your marketing | SeeSo. – Режим доступу: https://seeso.io. – Назва з екрана.
11. Core ML | Apple Developer Documentation [Електронний ресурс] // Apple Developer Documentation. – Режим доступу: https://developer.apple.com/documentation/coreml. – Назва з екрана.
12. Everything we actually know about the Apple Neural Engine (ANE) [Електронний ресурс] // GitHub. – Режим доступу: https://github.com/hollance/neural-engine. – Назва з екрана.
13. TensorFlow [Електронний ресурс] // TensorFlow. – Режим доступу: https://www.tensorflow.org. – Назва з екрана.
14. TensorFlow Lite | ML for Mobile and Edge Devices [Електронний ресурс] // TensorFlow. – Режим доступу: https://www.tensorflow.org/lite/. – Назва з екрана.
15. CocoaPods.org [Електронний ресурс] // CocoaPods.org. – Режим доступу: https://cocoapods.org. – Назва з екрана.
16. Bazel [Електронний ресурс] // Bazel. – Режим доступу: https://bazel.build. – Назва з екрана.
17. Swift Package Manager [Електронний ресурс] // Swift.org. – Режим доступу: https://www.swift.org/package-manager/. – Назва з екрана.
18. Swift and Objective-C Programming - The State of Developer Ecosystem in 2022 Infographic [Електронний ресурс] // JetBrains: Developer Tools for Professionals and Teams. – Режим доступу: https://www.jetbrains.com/lp/devecosystem-2022/swift-objc/. – Назва з екрана.

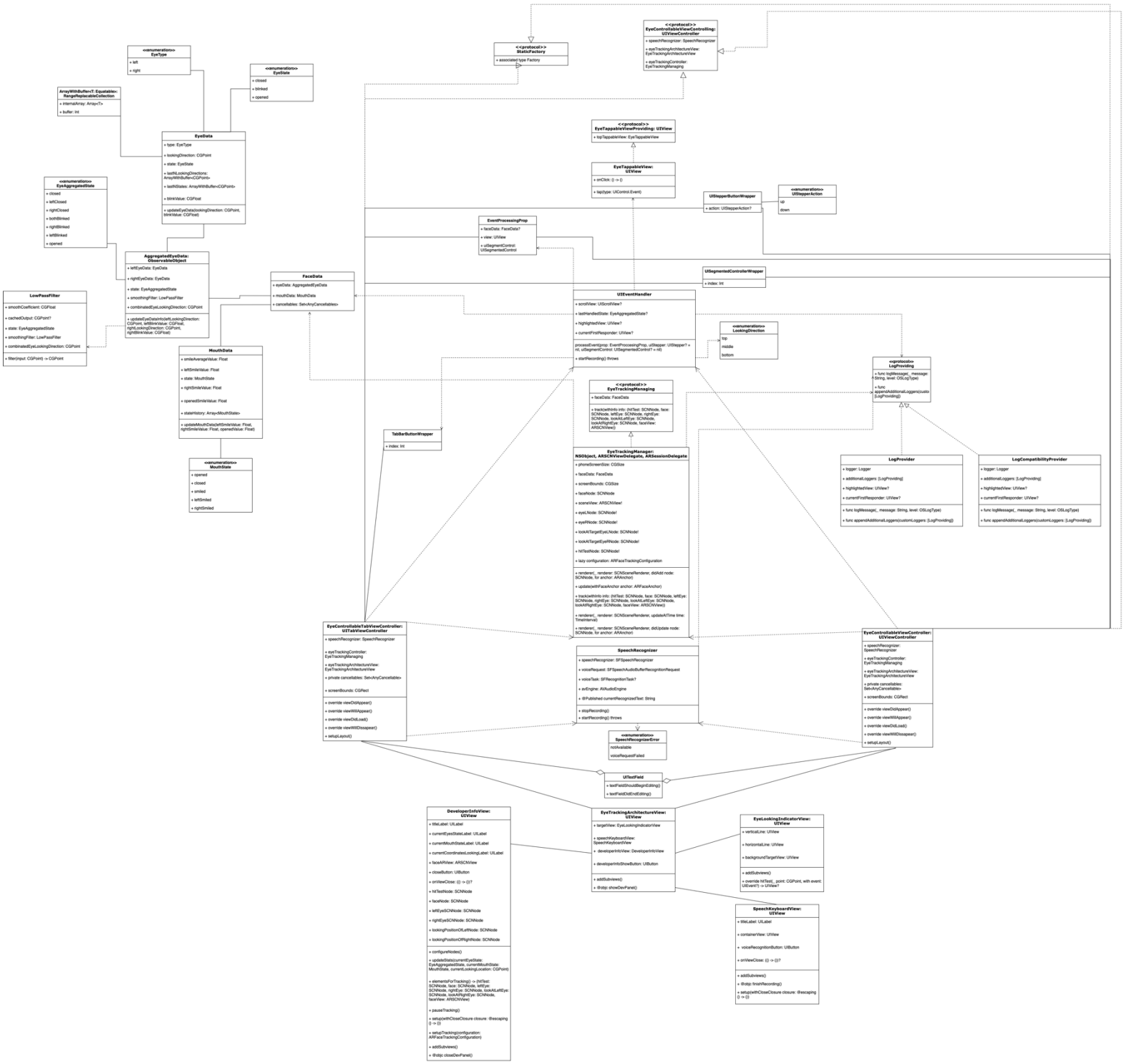
19. Swift for TensorFlow (In Archive Mode) [Електронний ресурс] // TensorFlow. – Режим доступу: https://www.tensorflow.org/swift/guide/overview. – Назва з екрана.
20. Cloud Spending Growth Rate Slows But Q4 Still Up By $10 Billion from 2021; Microsoft Gains Market Share | Synergy Research Group [Електронний ресурс] // Synergy Research Group | Strategic Market Intelligence for Emerging IT & Cloud. – Режим доступу: https://www.srgresearch.com/articles/cloud-spending-growth-rate-slows-but-q4-still-up-by-10-billion-from-2021-microsoft-gains-market-share. – Назва з екрана.
21. ARKit 6 - Augmented Reality - Apple Developer [Електронний ресурс] // Apple Developer. – Режим доступу: https://developer.apple.com/augmented-reality/arkit/. – Назва з екрана.
22. App Store - Support - Apple Developer [Електронний ресурс] // Apple Developer. – Режим доступу: https://developer.apple.com/support/app-store/. – Назва з екрана.
23. Про інноваційну технологію Face ID [Електронний ресурс] // Apple Support. – Режим доступу: https://support.apple.com/uk-ua/HT208108. – Назва з екрана.
24. Mixpanel Trends - Mixpanel | Product Analytics [Електронний ресурс] // Mixpanel: Event Analytics for Mobile, Web & More. – Режим доступу: https://mixpanel.com/trends/#report/iphone_models. – Назва з екрана.
25. About Objective-C [Електронний ресурс] // Apple Developer. – Режим доступу: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html. – Назва з екрана.
26. Swift.org [Електронний ресурс] // Swift.org. – Режим доступу: https://www.swift.org. – Назва з екрана.

27. TIOBE Index - TIOBE [Електронний ресурс] // TIOBE. – Режим доступу: https://www.tiobe.com/tiobe-index/. – Назва з екрана.
28. The Swift Programming Language [Електронний ресурс] // TIOBE. – Режим доступу: https://www.tiobe.com/tiobe-index/swift/. – Назва з екрана.
29. The Objective-C Programming Language [Електронний ресурс] // TIOBE. – Режим доступу: https://www.tiobe.com/tiobe-index/objective-c/. – Назва з екрана.
30. UIKit | Apple Developer Documentation [Електронний ресурс] // Apple Developer Documentation. – Режим доступу: https://developer.apple.com/documentation/uikit. – Назва з екрана.
31. SwiftUI Overview - Xcode - Apple Developer [Електронний ресурс] // Apple Developer. – Режим доступу: https://developer.apple.com/xcode/swiftui/. – Назва з екрана.
32. View and Window Architecture [Електронний ресурс] // Apple Developer. – Режим доступу: https://developer.apple.com/library/archive/documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/WindowsandViews/WindowsandViews.html. – Назва з екрана.
33. Apple's use of Swift and SwiftUI in iOS 16 [Електронний ресурс] // Timac. – Режим доступу: https://blog.timac.org/2022/1005-state-of-swift-and-swiftui-ios16/. – Назва з екрана.
34. AppleInsider. Face ID | iPhone, iPad, Masks [Електронний ресурс] / AppleInsider // AppleInsider. – Режим доступу: https://appleinsider.com/inside/face-id. – Назва з екрана.
35. Combine | Apple Developer Documentation [Електронний ресурс] // Apple Developer Documentation. – Режим доступу: https://developer.apple.com/documentation/combine. – Назва з екрана.
36. NHS website. Tics [Електронний ресурс] / NHS website // nhs.uk. – Режим доступу: https://www.nhs.uk/conditions/tics/. – Назва з екрана.

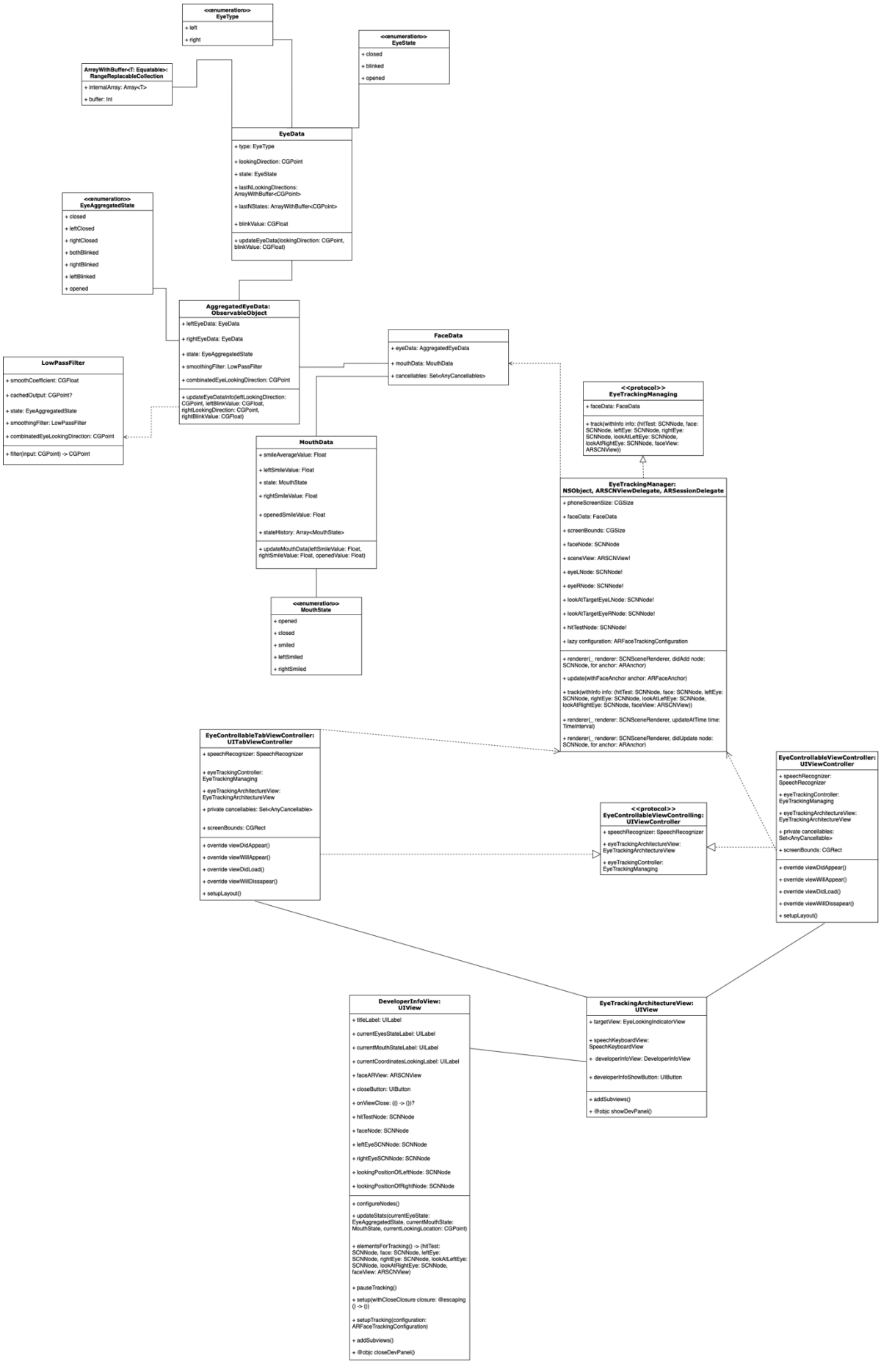
37. Carthage: A simple, decentralized dependency manager for Cocoa [Електронний ресурс] // GitHub. – Режим доступу: https://github.com/Carthage/Carthage. – Назва з екрана.
38. Xcode 14 - Apple Developer [Електронний ресурс] // Apple Developer. – Режим доступу: https://developer.apple.com/xcode/. – Назва з екрана.

Додатки

Додаток №1. Посилання на повний розмір зображення: https://drive.google.com/file/d/1nkpNhWXu_9wgxCFk-lbGEkQvnRxIjHwU/view?usp=sharing



Додаток №2


<<enumeration>>
EyeType
+ left
+ right
<<enumeration>>
EyeState
+ closed
+ blinked
+ opened
ArrayWithBuffer<T: Equatable>:
RangeReplacableCollection
+ internalArray: Array<T>
+ buffer: Int
EyeData
+ type: EyeType
+ lookingDirection: CGPoint
+ state: EyeState
+ lastNLookingDirections:
ArrayWithBuffer<CGPoint>
+ lastNStates: ArrayWithBuffer<CGPoint>
+ blinkValue: CGFloat
+ updateEyeData(lookingDirection: CGPoint,
blinkValue: CGFloat)
<<enumeration>>
EyeAggregatedState
+ closed
+ leftClosed
+ rightClosed
+ bothBlinked
+ rightBlinked
+ leftBlinked
+ opened
AggregatedEyeData:
ObservableObject
+ leftEyeData: EyeData
+ rightEyeData: EyeData
+ state: EyeAggregatedState
+ smoothingFilter: LowPassFilter
+ combinatedEyeLookingDirection: CGPoint
+ updateEyeDataInfo(leftLookingDirection:
CGPoint, leftBlinkValue: CGFloat,
rightLookingDirection: CGPoint,
rightBlinkValue: CGFloat)
FaceData
+ eyeData: AggregatedEyeData
+ mouthData: MouthData
+ cancellables: Set<AnyCancellables>
LowPassFilter
+ smoothCoefficient: CGFloat
+ cachedOutput: CGPoint?
+ state: EyeAggregatedState
+ smoothingFilter: LowPassFilter
+ combinatedEyeLookingDirection: CGPoint
+ filter(input: CGPoint) -> CGPoint
MouthData
+ smileAverageValue: Float
+ leftSmileValue: Float
+ state: MouthState
+ rightSmileValue: Float
+ openedSmileValue: Float
+ stateHistory: Array<MouthState>
+ updateMouthData(leftSmileValue: Float,
rightSmileValue: Float, openedValue: Float)
<<enumeration>>
MouthState
+ opened
+ closed
+ smiled
+ leftSmiled
+ rightSmiled
<<protocol>>
EyeTrackingManaging
+ faceData: FaceData
+ track(withInfo info: (hitTest: SCNNode, face:
SCNNode, leftEye: SCNNode, rightEye:
SCNNode, lookAtLeftEye: SCNNode,
lookAtRightEye: SCNNode, faceView:
ARSCNView))
EyeTrackingManager:
NSObject, ARSCNViewDelegate, ARSessionDelegate
+ phoneScreenSize: CGSize
+ faceData: FaceData
+ screenBounds: CGSize
+ faceNode: SCNNode
+ sceneView: ARSCNView!
+ eyeLNode: SCNNode!
+ eyeRNode: SCNNode!
+ lookAtTargetEyeLNode: SCNNode!
+ lookAtTargetEyeRNode: SCNNode!
+ hitTestNode: SCNNode!
+ lazy configuration: ARFaceTrackingConfiguration
+ renderer(_ renderer: SCNSceneRenderer, didAdd node:
SCNNode, for anchor: ARAnchor)
+ update(withFaceAnchor anchor: ARFaceAnchor)
+ track(withInfo info: (hitTest: SCNNode, face: SCNNode, leftEye:
SCNNode, rightEye: SCNNode, lookAtLeftEye: SCNNode,
lookAtRightEye: SCNNode, faceView: ARSCNView))
+ renderer(_ renderer: SCNSceneRenderer, updateAtTime time:
TimeInterval)
+ renderer(_ renderer: SCNSceneRenderer, didUpdate node:
SCNNode, for anchor: ARAnchor)
EyeControllableTabViewController:
UITabViewController
+ speechRecognizer: SpeechRecognizer
+ eyeTrackingController:
EyeTrackingManaging
+ eyeTrackingArchitectureView:
EyeTrackingArchitectureView
+ private cancellables: Set<AnyCancellable>
+ screenBounds: CGRect
+ override viewDidAppear()
+ override viewWillAppear()
+ override viewDidLoad()
+ override viewWillDissapear()
+ setupLayout()
<<protocol>>
EyeControllableViewControlling:
UIViewController
+ speechRecognizer: SpeechRecognizer
+ eyeTrackingArchitectureView:
EyeTrackingArchitectureView
+ eyeTrackingController:
EyeTrackingManaging
EyeControllableViewController:
UIViewController
+ speechRecognizer:
SpeechRecognizer
+ eyeTrackingController:
EyeTrackingManaging
+ eyeTrackingArchitectureView:
EyeTrackingArchitectureView
+ private cancellables:
Set<AnyCancellable>
+ screenBounds: CGRect
+ override viewDidAppear()
+ override viewWillAppear()
+ override viewDidLoad()
+ override viewWillDissapear()
+ setupLayout()
DeveloperInfoView:
UIView
+ titleLabel: UILabel
+ currentEyesStateLabel: UILabel
+ currentMouthStateLabel: UILabel
+ currentCoordinatesLookingLabel: UILabel
+ faceARView: ARSCNView
+ closeButton: UIButton
+ onViewClose: (() -> ())?
+ hitTestNode: SCNNode
+ faceNode: SCNNode
+ leftEyeSCNNode: SCNNode
+ rightEyeSCNNode: SCNNode
+ lookingPositionOfLeftNode: SCNNode
+ lookingPositionOfRightNode: SCNNode
+ configureNodes()
+ updateStats(currentEyeState:
EyeAggregatedState, currentMouthState:
MouthState, currentLookingLocation: CGPoint)
+ elementsForTracking() -> (hitTest:
SCNNode, face: SCNNode, leftEye:
SCNNode, rightEye: SCNNode, lookAtLeftEye:
SCNNode, lookAtRightEye: SCNNode,
faceView: ARSCNView)
+ pauseTracking()
+ setup(withCloseClosure closure: @escaping
() -> ())
+ setupTracking(configuration:
ARFaceTrackingConfiguration)
+ addSubviews()
+ @objc closeDevPanel()
EyeTrackingArchitectureView:
UIView
+ targetView: EyeLookingIndicatorView
+ speechKeyboardView:
SpeechKeyboardView
+ developerInfoView: DeveloperInfoView
+ developerInfoShowButton: UIButton
+ addSubviews()
+ @objc showDevPanel()

Додаток №3

```
import Foundation
import Speech
import AVFoundation
import Combine

enum SpeechRecognizerError: Error {
    case notAvailable
    case voiceRequestFailed
}

final class SpeechRecognizer {
    let speechRecognizer = SFSpeechRecognizer()!
    var voiceRequest: SFSpeechAudioBufferRecognitionRequest?
    var voiceTask: SFSpeechRecognitionTask?
    let avEngine = AVAudioEngine()
    let logProvider: LogProviding

    @Published var currentRecognizedText: String = ""

    init(logProvider: LogProviding) {
        self.logProvider = logProvider
        SFSpeechRecognizer.requestAuthorization { (status) in
            switch status {
            case .notDetermined: logProvider.logMessage("Speech
recognizer. Authorization status not determined", level: .error)
            case .restricted: logProvider.logMessage("Speech
recognizer. Authorization status restricted", level: .error)
            case .denied: logProvider.logMessage("Speech
recognizer. Authorization status denied", level: .error)
            case .authorized: logProvider.logMessage("Speech
recognizer. Authorization status authorized", level: .info)
            @unknown default: logProvider.logMessage("Speech
recognizer. Authorization status unknown", level: .info)
            }
        }
    }

    func stopRecording() {
        logProvider.logMessage("Stopped recording", level: .info)

        avEngine.stop()
        avEngine.inputNode.removeTap(onBus: 0)
        voiceRequest = nil
        voiceTask = nil
    }

    func cancelCurrentTask() {
        voiceTask?.cancel()
        voiceTask = nil
```

```
        voiceRequest = nil
        currentRecognizedText = ""
    }

    func startRecording() throws {
        logProvider.logMessage("Started recording", level: .info)
        guard speechRecognizer.isAvailable else {
            logProvider.logMessage("Speech recognizer not
available", level: .error)
            throw SpeechRecognizerError.notAvailable
        }

        cancelCurrentTask()
        voiceRequest = SFSpeechAudioBufferRecognitionRequest()
        voiceRequest?.shouldReportPartialResults = true
        voiceRequest?.requiresOnDeviceRecognition = true

        guard let voiceRequest = voiceRequest else {
            logProvider.logMessage("Speech recognizer failed at
voice request", level: .error)
            throw SpeechRecognizerError.voiceRequestFailed
        }

        let audioSession = AVAudioSession.sharedInstance()
        try audioSession.setActive(true, options:
.notifyOthersOnDeactivation)
        try audioSession.setCategory(.record, mode: .measurement,
options: .duckOthers)

        let inputNode = avEngine.inputNode

        voiceTask = speechRecognizer.recognitionTask(with:
voiceRequest) { result, err in
            var isCompletedMessage = false

            if let result = result {
                isCompletedMessage = result.isFinal
                self.currentRecognizedText =
result.bestTranscription.formattedString
            }

            if err != nil || isCompletedMessage {
                inputNode.removeTap(onBus: 0)
                self.avEngine.stop()
                self.cancelCurrentTask()
            }
        }
        avEngine.inputNode.removeTap(onBus: 0)
        inputNode.installTap(onBus: 0, bufferSize: 1024, format:
inputNode.outputFormat(forBus: 0)) { (buffer: AVAudioPCMBuffer,
when: AVAudioTime) in self.voiceRequest?.append(buffer)
```

```
        }

        avEngine.prepare()
        try avEngine.start()
    }
}

extension SpeechRecognizer: StaticFactory {
    enum Factory {
        static var `default`: SpeechRecognizer {
            let logProviding: LogProviding
            if #available(iOS 14.0, *) {
                logProviding = LogProvider(subcategory: "Event
handler")
            } else {
                logProviding =
LogCompatibilityProvider(subcategory: "Event handler")
            }

            return SpeechRecognizer(logProvider: logProviding)
        }
    }
}
```

Додаток №4

```
import Foundation
import os

protocol LogProviding {
    func logMessage(_ message: String, level: OSLogType)
    func appendAdditionalLoggers(customLoggers:
[AdditionalLogProviding])
}

protocol AdditionalLogProviding {
    func logMessage(_ message: String, level: OSLogType)
}

@available(iOS 14.0, *)
open class LogProvider: LogProviding {
    private let logger: Logger
    private var additionalLoggers: [AdditionalLogProviding] = []

    public init(subcategory: String) {
        logger = Logger(subsystem: EyeTrackingConfig.subsystem,
category: subcategory)
    }

    open func logMessage(_ message: String, level: OSLogType) {
        logger.log(level: level, "\(message)")

        additionalLoggers.forEach { customLogger in
            customLogger.logMessage(message, level: level)
        }
    }

    func appendAdditionalLoggers(customLoggers:
[AdditionalLogProviding]) {
        additionalLoggers.append(contentsOf: customLoggers)
    }
}

final class LogCompatibilityProvider: LogProviding {
    private let logger: OSLog
    private var additionalLoggers: [AdditionalLogProviding] = []

    init(subcategory: String) {
        logger = OSLog(subsystem: EyeTrackingConfig.subsystem,
category: subcategory)
    }

    func logMessage(_ message: String, level: OSLogType) {
        os_log("%@", log: .default, type: level, message)
```

```
        additionalLoggers.forEach { customLogger in
            customLogger.logMessage(message, level: level)
        }
    }

    func appendAdditionalLoggers(customLoggers:
[AdditionalLogProviding]) {
        additionalLoggers.append(contentsOf: customLoggers)
    }
}
```