УДК 004.5, 004.415

ПРОГРАММНЫЙ ФРЕЙМВОРК ДЛЯ РЕАЛИЗАЦИИ ВЗАИМОДЕЙСТВИЯ С ПОЛЬЗОВАТЕЛЬСКИМИ ИНТЕРФЕЙСАМИ IOS-ПРИЛОЖЕНИЙ НА ОСНОВЕ ОКУЛОГРАФИИ

Н. С. Афанасьев[1] [0000-0002-5306-9672]

Институт информационных технологий и интеллектуальных систем Казанского (Приволжского) федерального университета
ул. Кремлевская, 35, г. Казань, 420008

[1]reqenoxus@gmail.com

Аннотация

Использование технологий отслеживания взгляда для взаимодействия с пользовательским интерфейсом iOS-приложений существенно затруднено отсутствием унифицированного подхода к их интеграции. Существующие решения либо жестко ограничены своей предметной областью, либо представляют собой исключительно исследовательские проекты, непригодные для решения прикладных задач. В статье рассматривается создание фреймворка, осуществляющего отслеживание взгляда пользователя на экране Apple-устройств с использованием нативных технологий, а также предоставляющего унифицированный подход к разработке приложений, управляемых при помощи взгляда.

Ключевые слова*: gaze tracking, eye tracking, отслеживание взгляда, окулография, обработка жестов, TrueDepth, ARKit, SceneKit, UIKit, iOS, UX, UI*

ВВЕДЕНИЕ

Задача компьютерной окулографии не нова и имеет множество практических приложений. В их числе решения, помогающие людям с ограниченными возможностями взаимодействовать с компьютерными системами, в том числе с мо-

© Н. С. Афанасьев, 2022.
Данная статья распространяется на условиях международной лицензии Creative Commons License Attribution 4.0 International (CC BY 4.0).

бильными, посредством отслеживания их взгляда и положения головы [1, 2], решения в области разработки мобильных игр [3], оценки пользовательских предпочтений в области мобильного ритейла [4] и других сферах [5].

В связи с высокими темпами развития камер и датчиков, интегрируемых в мобильные устройства [6, 7], а также повсеместным их распространением [8], применять разработки в области окулографии становится возможным не только в специальных лабораторных условиях, но и в условиях повседневного использования персональных мобильных телефонов, что существенно расширяет возможный круг потребителей подобных технологий. В частности, существует ряд работ, посвященных исследованию этой темы в контексте устройств компании Apple. В их числе специализированные модели машинного обучения для предсказания положения взгляда [9–11], а также ряд прикладных разработок [12–-14], но описанные в них решения либо покрывают один конкретный сценарий использования, либо предназначены сугубо для исследовательских целей и не могут быть применимы в прикладных задачах. Иными словами, имеющиеся наработки не предлагают унифицированного подхода к построению приложений, управляемых посредством взгляда пользователя, и сейчас для каждого конкретного сценария использования разрабатывается свое собственное решение, что приводит к дороговизне и, как следствие, малой распространенности этой технологии на рынке мобильных приложений. К аналогичным выводам пришли специалисты из eBay [15], разрабатывая свое решение в смежной области – управление iOS-приложением при помощи движений головы.

Таким образом, целью данной работы стала разработка iOS-фреймворка для окулографии, предоставляющего конкурентоспособную точность распознавания и предлагающего единый подход к проектированию приложений, управляемых при помощи взгляда.

ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

Изучение окулографии ведется с 1960-х годов, и одним из первых подходов к решению задач в этой области стало использование метода электроокулографии – отслеживания движения глаз при помощи считывания электрических потенциалов с электродов, прикрепленных к уголкам глаз [16]. По мере увеличения

вычислительных мощностей компьютеров и развития технологий в области цифровой обработки изображений стали появляться менее инвазивные, значит, и более предпочтительные для широкого использования варианты, такие как Skyle [14] и Tobii EyeX [17], представляющие собой системы камер, специально предназначенных для отслеживания положения глаз. В «противовес» решениям, основанным на специализированных дополнительных устройствах, можно выделить противоположную группу – решения, основанные исключительно на программном обеспечении и использующие в качестве входных данные, полученные с камер общего назначения – фронтальной камеры смартфона или веб-камеры персонального компьютера или ноутбука [18]. С развитием камер, устанавливаемых в смартфоны, а также ростом популярности AR/VR технологий в мобильных приложениях [19], эту группу пополняют и мобильные решения [20, 21]. В частности, в эту же группу войдет решение, разрабатываемое в рамках данной работы. При этом для определения наиболее важных критериев, которым должно соответствовать разрабатываемое решение, необходимо провести сравнительный анализ непосредственных аналогов – решений для iOS, основанных на программном обеспечении:

- iTracker [9] – представляет собой сверточную нейронную сеть, обрабатывающую входной сигнал с фронтальной камеры телефона. Модель анализирует полное изображение лица для определения его положение относительно камеры, а также изображения каждого из глаз в отдельности для определения их направления относительно лица. Сопоставляя полученные данные, нейронная сеть предсказывает точку на экране, куда направлен взгляд пользователя. Заявленная средняя ошибка предсказанного положения взгляда (расстояние на плоскости между реальным и предсказанным положениями точки) составляет 1.71 см и 2.53 см на телефонах и планшетах без предварительной калибровки. С калибровкой ошибка уменьшается до 1.34 см и 2.12 см соответственно. Из недостатков решения можно выделить сравнительно низкую производительность – созданная модель может обрабатывать данные с камеры только с частотой 10–15 Гц. Также это решение не предоставляет API для интеграции с пользовательским интерфейсом: возможно лишь предсказать точку на экране.

- ScreenGlint [10] – нейронная сеть, предсказывающая положение взгляда пользователя на экране и использующая в качестве ориентира отражение экрана смартфона на поверхности глаза. По заявлению авторов модель достигает средней ошибки в диапазоне от 1.47 см до 1.72 см. Из недостатков можно выделить зависимость точности результатов от освещения, так как используются данные о положении блика от устройства на поверхности глаза, – авторы заявляют, что оценка результатов проводилась исключительно в закрытых помещениях, и при более сложном освещении результаты могут ухудшиться. Также стоит отметить необходимость калибровки для использования модели. Аналогично iTracker, ScreenGlint также не предоставляет возможности реализовывать взаимодействие с пользовательским интерфейсом, решая лишь задачу предсказания точки взгляда.
- SmartEye [11] – система отслеживания взгляда на экране с применением прототипа смартфона с инфракрасной камерой и сверточных нейросетей. Согласно исследованию [11], эта система обеспечивает возможность обрабатывать поток данных с частотой 30 кадров в секунду и на данный момент является самым точным решением среди трекеров взгляда на смартфонах. Главным недостатком решения является то, что оно основано на применении инфракрасного сенсора, что невозможно на iOS устройствах без серьезной модификации – камера TrueDepth, устанавливаемая на iPhone, начиная с модели X, обладает таким сенсором, но система не предоставляет возможности использовать его напрямую [21].
- Hawkeye Access [12] – проприетарное приложение-браузер, полностью управляемое взглядом пользователя. Для реализации отслеживания взгляда используется фреймворк ARKit, предоставляемый компанией Apple. Использование нативного фреймворка дает возможность добиться серьезного улучшения производительности (обработка сигнала с частотой 60 кадров в секунду) и снижения нагрузки на процессор за счет низкоуровневых оптимизаций. Согласно исследованиям [22], использование ARKit позволяет добиться средней ошибки в предсказании точки взгляда пользователя, равной 1.44 см, что соответствует уровню аналогов в этой области. Серьезным недостатком Hawkeye Access является его ограниченная область

применения – решение представляет собой веб-браузер и не может использоваться в других задачах. Access также не дает возможности использовать полученные наработки для интеграции технологии в собственные приложения, а авторы не раскрывают исходный код.

- EyeMU Interactions [13] – JavaScript-приложение, работающее на устройствах под управлением iOS в браузере Safari и представляющее собой комбинацию отслеживания взгляда пользователя с гироскопическими жестами. Приложение позволяет определить, на что пользователь в данный момент смотрит (это может быть, например, системное уведомление), после чего при выполнении определенного гироскопического жеста (например, смахивания влево или вправо) выполнить то или иное действие с объектом, на который направлено внимание пользователя. Распознавание точки взгляда пользователя в этом решении реализовано с помощью сверточной нейросети, по своим принципам схожей с таковой в iTracker и использующей тот же самый датасет для обучения. Заявленная авторами точность предсказания достигает 1.74 см и не требует калибровки при использовании. Стоит отметить, что реализация, по мнению самих разработчиков, является недостаточно эффективной для использования в приложениях – обработка сигналов происходит с частотой 20 Гц, при этом отмечается серьезная нагрузка на батарею смартфона из-за использования Javascript вместо технологий, нативных для платформы. Авторы также не предоставляют возможности реализовывать свою собственную логику для жестов, доступны лишь несколько демонстрационных решений, но нет возможности создавать свои.

На базе аналогов, описанных выше, можно сформулировать список критериев, которым должно соответствовать разрабатываемое решение:

1. Производительность – поддержка решением обработки данных в реальном времени (60 Гц).
2. Универсальность – решение не должно зависеть от конкретного сценария использования и быть применимо для различных задач окулографии.

3. Отсутствие специфических требований к устройству – решение должно быть применимо на устройствах без специальных модификаций или дополнительных аксессуаров.
4. Наличие интерфейса для интеграции в приложения – решение позволяет внедрять в приложения функционал, связанный с отслеживанием взгляда пользователя.
5. Точность – соответствие точности предсказания другим state-of-art решениям.

КОНЦЕПЦИЯ ПРОГРАММНОГО РЕШЕНИЯ

Так как перед разрабатываемым решением ставятся две основные задачи – отслеживание положения взгляда и взаимодействие с его помощью с элементами интерфейса, было принято решение разделить эту ответственность между двумя «слоями» со следующими обязанностями:

- Backend-слой:
  - получение и обработка данных с камеры;
  - вычисление координат точки взора;
  - отслеживание моргания.
- Frontend-слой:
  - преобразование данных о взгляде в события взаимодействия с пользовательским интерфейсом;
  - API для интеграции в клиентское приложение.

Нетрудно видеть, что Frontend-слой является «выходом» для Backend-слоя, а общую схему взаимодействия клиентского приложения и фреймворка можно изобразить следующим образом (рис. 1):

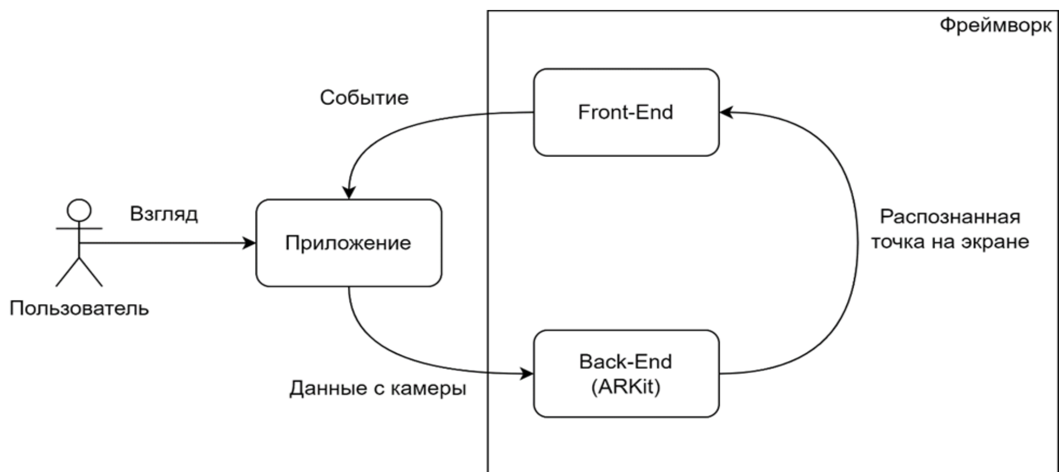

Рисунок 1. Взаимодействие клиента и разрабатываемого фреймворка

Так как задачи, решаемые слоями, независимы, их взаимодействие можно реализовать через абстракцию – Frontend-слой является делегатом Backend-слоя, что позволяет ему работать с данными о положении взгляда, не заботясь об их происхождении (рис. 2):

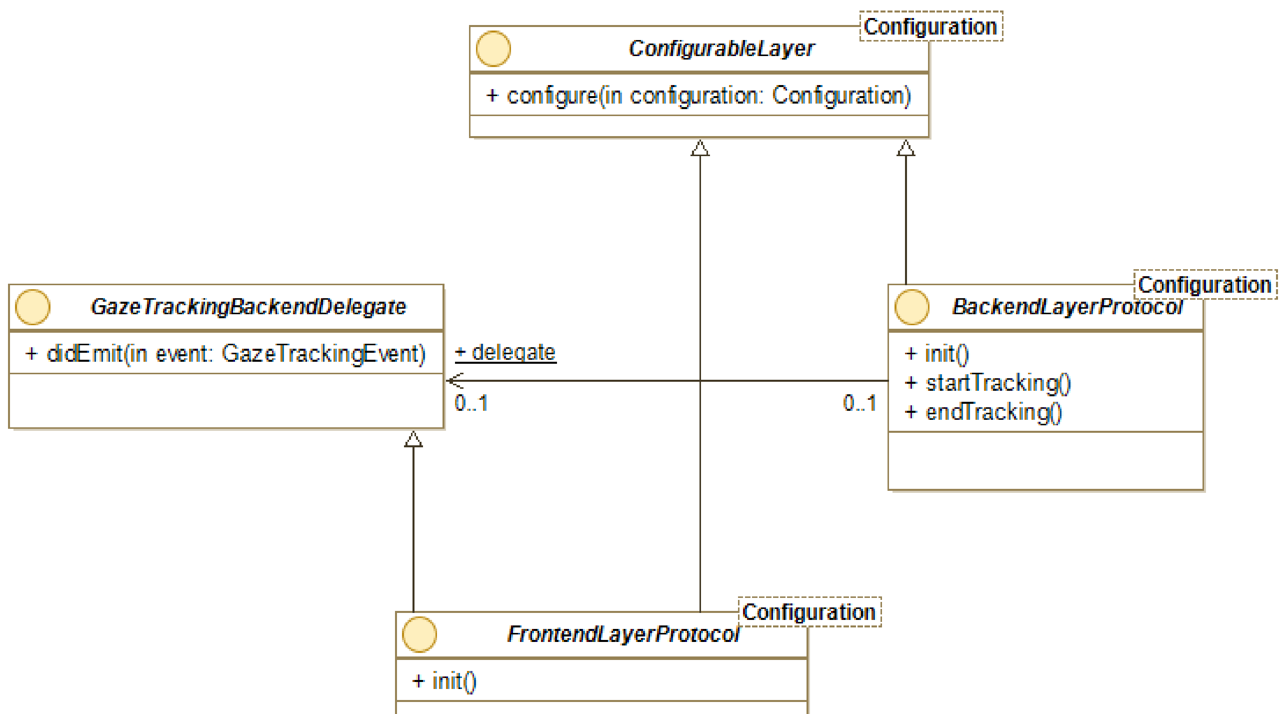

Рисунок 2. Диаграмма слоев решения

Такой подход к построению архитектуры слоев позволяет реализовать единую точку инициализации для создаваемой системы, использующую пару из конфигураций в качестве входного параметра и настраивающую связь между слоями (рис. 3):

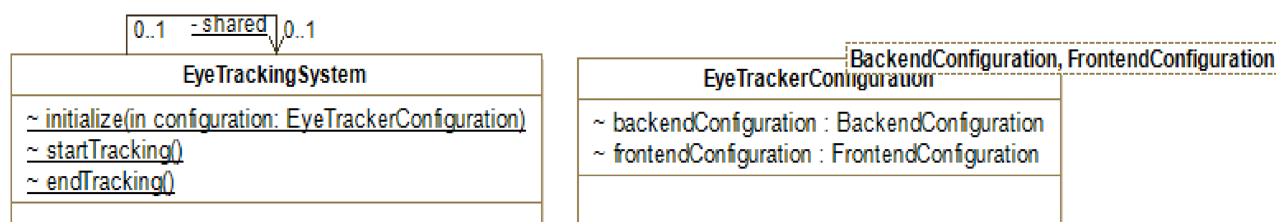

Рисунок 3. Конфигуратор системы отслеживания взгляда

Немаловажным следствием этой архитектурной особенности является возможность пользователя заменять реализации компонентов системы на собственные, если того требует ситуация.

РАСПОЗНАВАНИЕ ВЗГЛЯДА

Исходя из необходимости соответствия разрабатываемого решения критерию точности распознавания, было принято решение использовать фреймворк Apple ARKit для обработки данных с камеры. Согласно исследованию [22], этот инструмент пригоден для использования в задаче окулографии и имеет точность, сравнимую с аналогами. При этом, согласно тому же исследованию, ARKit показывает в данной задаче от 4-х до 6-кратного прироста производительности по сравнению с ранее упомянутым iTracker, что делает его среди рассмотренных самой предпочтительной технологией с точки зрения скорости обработки данных.

Анализируя изображения, получаемое с камеры TrueDepth, ARKit предоставляет для каждого кадра информацию о расположении в пространстве тех или иных объектов, представляя информацию о них в виде объектов ARAnchor. В частности, используя конфигурацию ARFaceTrackingConfiguration, можно получить информацию о расположении в сцене лица пользователя, а также отследить положение глаз и их поворот, характеризующий направление взгляда. Указанные данные о расположении рассчитываются относительно родительского объекта – лица пользователя. Помимо этого система также предоставляет информацию о

камере – ее расположение в пространстве, а также углы поворота. Помимо информации о пространственном положении объектов ARKit позволяет отслеживать и другую информацию – в случае конфигурации для трекинга лиц, в частности, доступна информация о степени «закрытости» глаза в виде значения от 0 до 1, что позволяет отслеживать моргание пользователя.

Расположение тех или иных объектов в пространстве в ARKit описывается с помощью матриц аффинного преобразования следующего вида (рис. 4):

$$M = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Рисунок 4. Матрица аффинного преобразования ARKit

Такой подход позволяет представить всю информацию о расположении объекта с помощью одной структуры (рис. 5 и 6):

$$T = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Рисунок 5. Матрица переноса по осям XYZ

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad R_y(\alpha) = \begin{pmatrix} \cos\alpha & 0 & -\sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ -\sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Рисунок 6. Матрицы вращения по осям XYZ

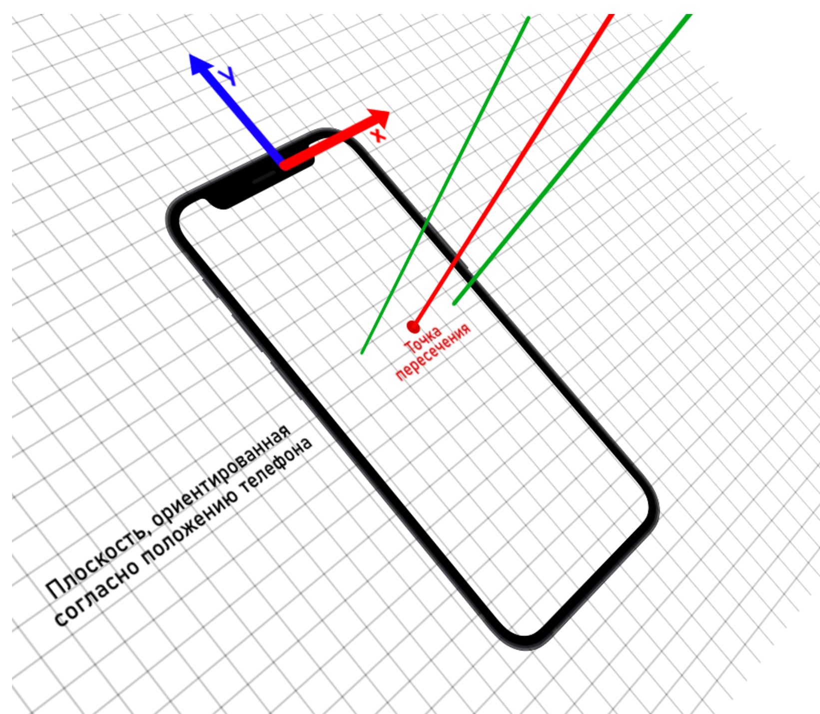

Рисунок 7. Виртуальная сцена для определения положения взгляда

Так как ARKit предоставляет информацию о смещении и повороте головы в координатах всей сцены, а также аналогичную информацию о глазах в координатах лица, появляется возможность сконструировать виртуальную трехмерную сцену (рис. 7), с помощью которой можно будет получить координаты точки взгляда на экране.

Для построения виртуальной сцены было решено воспользоваться другим нативным фреймворком Apple – SceneKit, позволяющим размещать в виртуальном пространстве элементы (Nodes). При этом каждый элемент может иметь свои дочерние элементы, координаты которых будут задаваться в координатной системе родительского элемента. Такой подход явно перекликается с представлением данных в ARKit и позволяет не задумываться о конвертации полученных из него данных в координатную систему всей сцены – будет достаточно лишь воссоздать внутри сцены ту же иерархию элементов. Кроме того, SceneKit оперирует той же системой представления расположения объектов в виде матриц, что опять же позволяет не задумываться о конвертации формата данных и не терять на этом производительность.

Итак, для распознавания положения взгляда строится сцена, состоящая из следующих объектов:

- *Квадратная плоскость со стороной 1 м* – плоскость имитирует собой поверхность экрана и используется для проекции на нее луча взгляда. Расположение плоскости в глобальных координатах, а также поворот должны совпадать с расположением камеры в сцене. Такой подход позволит в будущем нивелировать деградацию точности распознавания при поворотах и наклонах девайса. Размеры плоскости выбраны из соображений нормировки, чтобы не выполнять дополнительных вычислений при последующей конвертации координат из метрической системы в логическую.
- *Виртуальный элемент лица* – элемент, имеющий в координатах сцены то же расположение, что и лицо пользователя. Служит родительским элементом для следующих элементов.
- *Виртуальные элементы глаз* – элементы, являющиеся дочерними для элемента лица. Их расположение в координатной системе родителя равно вычисленному положению глаз относительно лица.
- *Начальная точка взгляда* – дочерний элемент «лицевого» элемента, расположение которого получается путем усреднения соответствующих матриц левого и правого глаз. Служит начальной точкой луча взгляда.
- *Конечная точка взгляда* – дочерний элемент начальной точки, смещенный от последней на 2 метра в положительном направлении по оси Z. При таком расположении начальная и конечная точки взгляда находятся по разные стороны вышеупомянутой плоскости, а прямая, их соединяющая, пересечет плоскость в тех случаях, когда пользователь смотрит на экран.

После построения вышеописанной конфигурации определение положения взгляда сводится к нахождению точки пересечения прямой, проходящей через начальную и конечную точки взгляда и плоскости. Вычисление координат этой точки возможно с помощью встроенного функционала фреймворка SceneKit, а именно, метода, возвращающего координаты точки пересечения в системе координат плоскости. Последним шагом алгоритма является конвертация полученных координат из метрической системы, используемой в SceneKit, в логическую, которой подчиняется UIKit, с помощью которого происходит построение интерфейсов

в iOS-приложениях. Кроме того, необходимо учитывать ориентацию устройства в пространстве, так как при ее смене в системе изменяется ориентация осей координат: положительное направление оси X всегда направлено вправо, а оси Y – вниз, а также изменяется смысл понятий «ширина» и «высота» – при портретной ориентации они совпадают с физическими измерениями устройства, а при альбомной физическая высота становится логической шириной, а физическая ширина, соответственно, логической высотой.

Обозначения, используемые при описании реализации алгоритма:

- *x* – x-координата точки пересечения прямой и плоскости в ее координатах и метрической системе измерения.
- *y* – y-координата точки пересечения прямой и плоскости в ее координатах и метрической системе измерения.
- *d* – диагональ экрана в дюймах, определяется спецификацией устройства.
- *h*, *w* – целочисленные доли высоты и ширины в соотношении сторон экрана; определяются спецификацией устройства.
- *pointWidth* – ширина экрана в логической системе измерения; предоставляется системой.
- *pointHeight* – высота экрана в логической системе измерения; предоставляется системой.
- *metricWidth* – ширина экрана устройства в метрах; вычисляется с использованием логической ширины и диагонали следующим образом: $$metricWidth = \frac{wd}{\sqrt{(h^2+w^2)}} * 0{,}0254;$$
- *metricHeight* – высота экрана смартфона в метрах; вычисляется с использованием логической высоты и диагонали следующим образом: $$metricHeight = \frac{hd}{\sqrt{(h^2+w^2)}} * 0{,}0254$$
- *resultX* – x-координата точки взгляда в логической системе координат.
- *resultY* – y-координата точки взгляда в логической системе координат.

Параметры алгоритма, зависящие от спецификации конкретного устройства, такие как соотношение сторон экрана, а также длина диагонали в дюймах предоставляются open-source библиотекой DeviceKit [23].

При помощи параметров, описанных выше, алгоритм вычисления итоговой точки взгляда пользователя представляет собой следующее преобразование координат для каждой конкретной ориентации устройства:

- Портретная ориентация (на рис. 8 красной границей выделена плоскость, на которую проецируется взгляд; масштаб уменьшен для наглядности):

$$resultX = \left(\frac{2x}{metricWidth}\right) * pointWidth + \frac{pointWidth}{2},$$
$$resultY = -\left(\frac{2y}{metricHeight}\right) * pointHeight$$

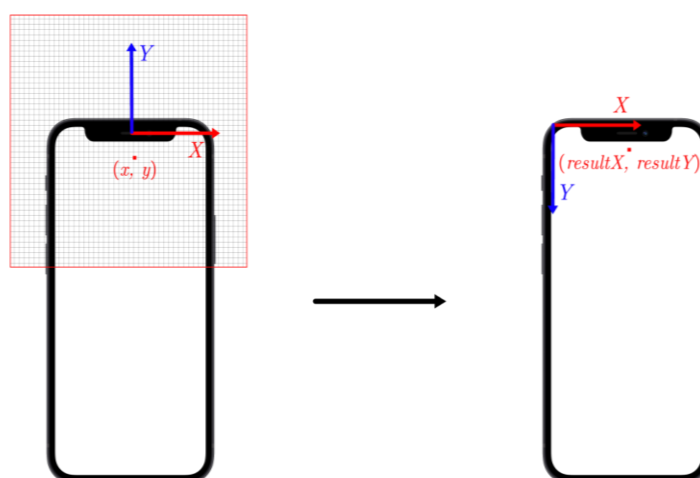

Рисунок 8. Конвертация координат в портретной ориентации

- Левая альбомная ориентация (рис. 9):

$$resultX = \left(\frac{2x}{metricHeight}\right) * pointWidth,$$
$$resultY = -\left(\frac{2y}{metricWidth}\right) * pointHeight + \frac{pointHeight}{2}$$

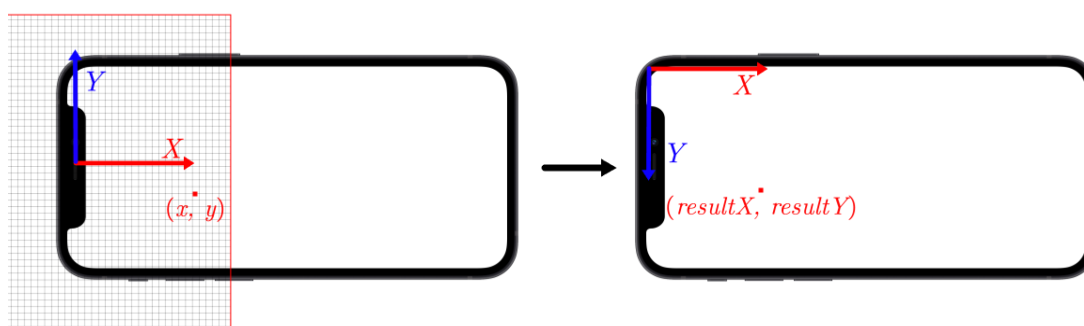

Рисунок 9. Конвертация координат в левой альбомной ориентации

- Правая альбомная ориентация (рис. 10):

$$resultX = \left(\frac{2x}{metricHeight}\right) * pointWidth + pointWidth,$$

$$resultY = -\left(\frac{2y}{metricWidth}\right) * pointHeight + \frac{pointHeight}{2}$$

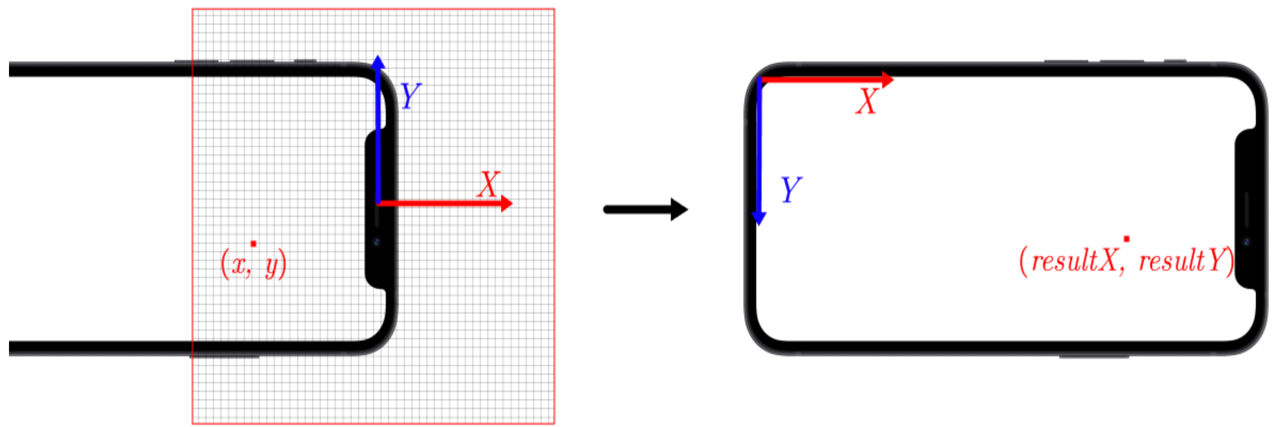

Рисунок 10. Конвертация координат в правой альбомной ориентации

Описанный алгоритм применяется для каждого обработанного кадра в реальном времени (рис. 11).

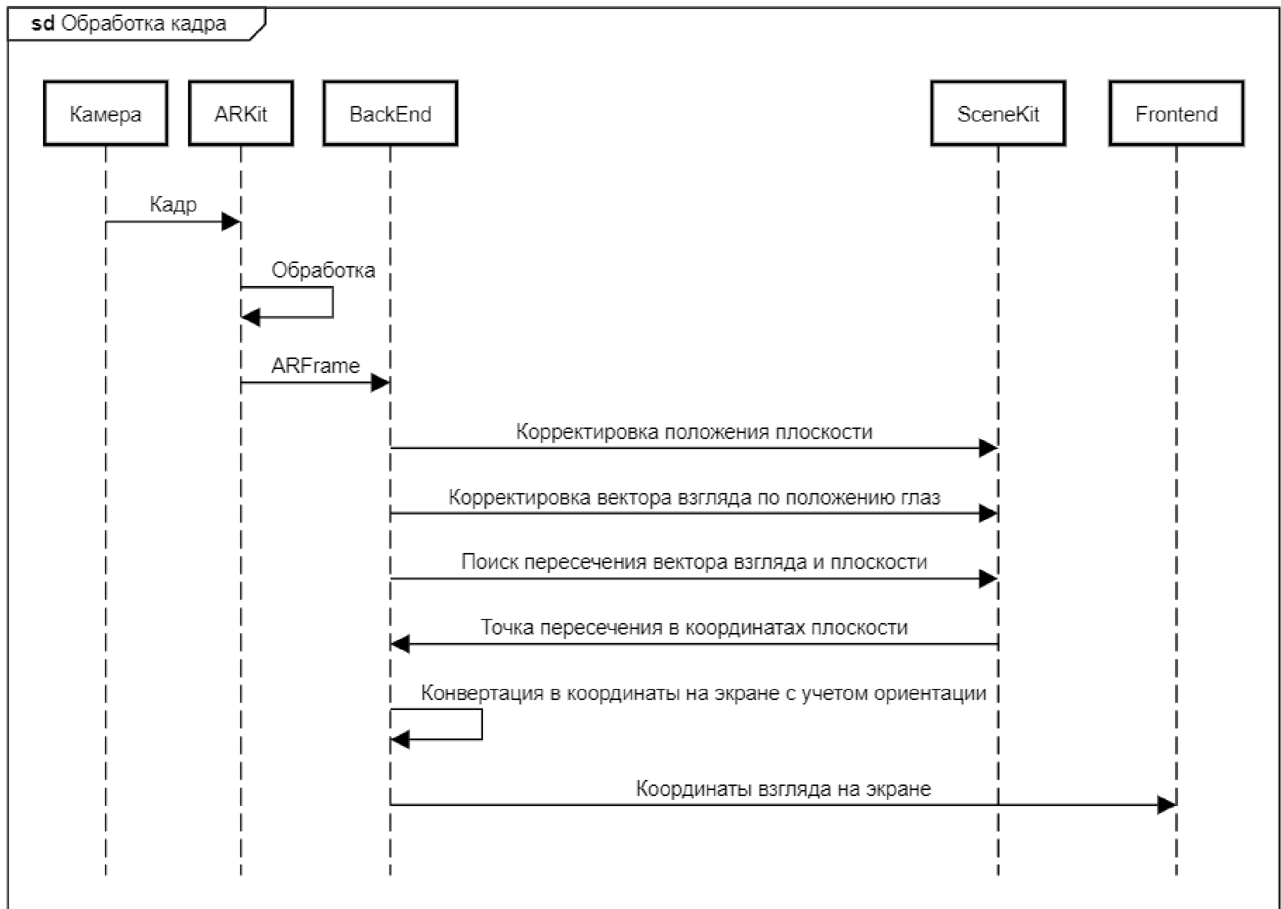

Рисунок 11. Диаграмма последовательности для описанного алгоритма

Помимо отслеживания взгляда пользователя необходимо отслеживать события моргания. Это позволит получить больше вариантов для взаимодействия с элементами интерфейса. Как уже упоминалось ранее, ARKit предоставляет информацию об особенностях выражения лица в виде словаря blendShapes [24]. В частности, этот словарь содержит «степень закрытости» левого и правого глаз в отдельности, выраженную в виде числа в диапазоне от 0 до 1. При превышении порогового значения, определенного заранее, моргание будет считаться состоявшимся, а алгоритм получит возможность проверки вышеуказанных величин вместе с определением положения взгляда. При этом необходимо учитывать, что при моргании направление взгляда кратковременно смещается вниз, поэтому при определении точки, при взгляде на которую произошло моргание, необходимо использовать данные, полученные несколько кадров назад. Этот параметр, как и пороговое значение, задается при конфигурации Backend-слоя. Полный список параметров:

- *blinkFrameOffset* – число кадров, на которые необходимо «вернуться», чтобы получить данные о точке, на которую смотрел пользователь перед тем, как моргнуть.
- *leftEyeBlinkThreshold* – пороговое значение, при превышении которого система считает, что произошло моргание левым глазом.
- *rightEyeBlinkThreshold* – пороговое значение, при превышении которого система считает, что произошло моргание правым глазом.
- *coordinateMapper* – ссылка на конкретную реализацию интерфейса, определяющего функцию перехода между координатами метрической и логической систем; по умолчанию используется реализация алгоритма, приведенного выше.
- *sceneView* – опциональная ссылка на объект ARSCNView, позволяющий вывести на экран визуализацию сцены, описанной выше; используется для целей отладки и демонстрации решения.

КОНЦЕПЦИЯ ОБРАБОТЧИКОВ ЖЕСТОВ

Одной из главных задач нашей работы является разработка публичного API для разработчика, позволяющего использовать данные о положении взгляда пользователя и моргании для реализации произвольной логики. Самым главным приоритетом при решении этой задачи являлось соответствие предлагаемого API нативному подходу к построению пользовательского взаимодействия в iOS.

Для соответствия разрабатываемого решения традиционному подходу к разработке iOS-приложений было принято решение реализовывать внешний API для разработчика как расширение концепции распознавателей жестов (UIGestureRecognizer) – части фреймворка UIKit, унифицирующей событийно-ориентированный подход к проектированию взаимодействия с пользователем в iOS-приложениях.

UIGestureRecognizer представляет собой объект, инкапсулирующий логику обработки определенных событий (чаще всего событий касания экрана). При успешном распознавании того или иного жеста (как правило – совокупности касаний) обработчик с помощью механизма target-action [25] отправляет ассоциированным с ним объектам target сообщение о необходимости выполнения действия

– action. Один или несколько таких объектов можно ассоциировать с любым элементом отображения (наследником UIView), тем самым достигается возможность единообразно добавлять любому элементу на экране возможность обрабатывать те или иные виды нажатий. Внутреннее устройство такого объекта представляет собой конечный автомат из нескольких состояний, переходы между которыми осуществляются при обработке получаемых событий. Существую два основных типа обработчиков – дискретные и непрерывные.

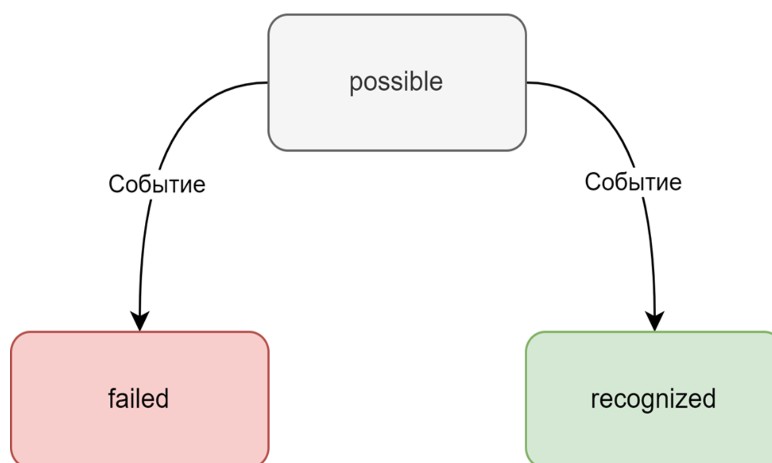

Рисунок 12. Диаграмма состояний дискретного обработчика жестов

Как видно на рис. 12, для дискретных обработчиков используются три состояния:

- *possible* – состояние по умолчанию, означающее, что обработчик готов к получению и обработке событий;
- *recognized* – состояние, означающее, что ожидаемое событие успешно распознано; при достижении этого состояния обработчик вызывает ассоциированную с ним логику через механизм target-action, после чего возвращается в состояние possible;
- *failed* – состояние, означающее, что ожидаемое событие не было распознано; по достижении этого состояния вызов ассоциированного action не происходит, а обработчик возвращается в состояние possible.

К дискретным обработчикам жестов в стандартной библиотеке относятся такие обработчики, как:

- *UITapGestureRecognizer* – обработчик одного или нескольких последовательных нажатий (одним или несколькими пальцами);
- *UILongPressGestureRecognizer* – обработчик длительного касания;
- *UISwipeGestureRecognizer* – обработчик «свайпа» в том или ином направлении.

В случае непрерывных обработчиков число возможных состояний становится больше (рис. 13).

Состояния *possible*, *recognized* и *failed* работают аналогично дискретным обработчиком, но добавляется еще ряд новых:

- *began* – состояние, означающее начало распознавания жеста; при переходе в это состояние происходит вызов соответствующего target-action обработчика;
- *changed* – состояние, означающее очередное изменение в процессе распознавания жеста, еще не приводящее его к завершению, но и не приводящее к ошибке; при переходе в это состояние происходит вызов соответствующего target-action обработчика;

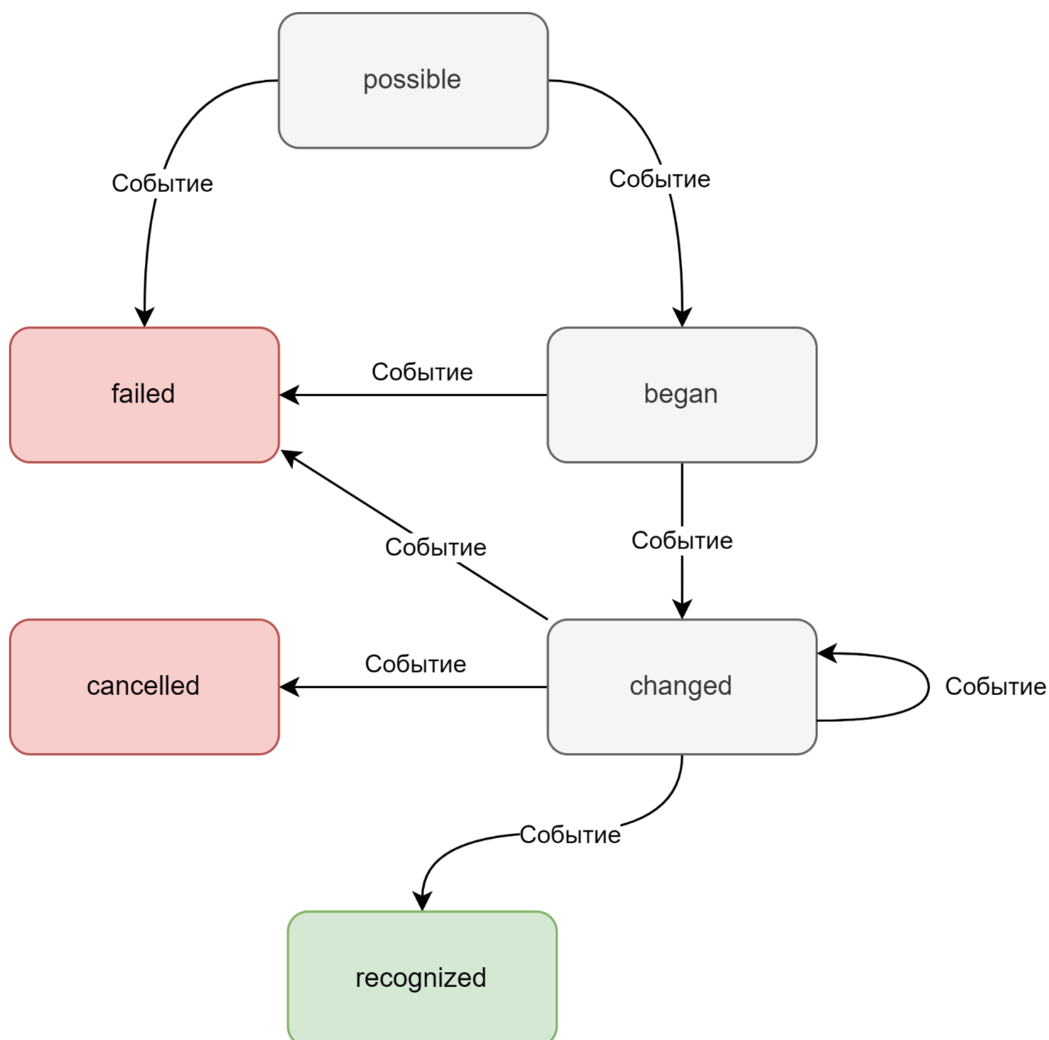

Рисунок 13. Диаграмма состояний непрерывного обработчика жестов

- *cancelled* – состояние, означающее, что в процессе распознавания жеста произошло какое-то событие, которое прервало этот процесс, например, пользователю в этот момент пришел вызов; переход в это состояние прерывает распознавание и сбрасывает состояние обработчика.

ОБОБЩЕНИЕ КОНЦЕПЦИИ НА ОБРАБОТКУ ВЗГЛЯДА

Описанная выше модель состояний универсальна и не зависит от конкретного типа взаимодействия. Для обобщения подхода на «жесты», реализуемые с помощью взгляда, необходимо спроектировать сущность, описывающую то или иное событие, отслеживаемое с помощью окулографии, и использовать его для изменения состояния обработчика.

Обработчики из стандартной библиотеки используют для этой цели объект UITouch, автоматически генерируемый системой при касании экрана. Использование такого же объекта для окулографических жестов невозможно по двум причинам:

- корректное создание объекта UITouch невозможно с помощью открытого API;
- использование этих объектов привело бы к ситуации, когда помимо окулографических обработчиков на них бы реагировали и обычные, так как формат события бы не отличался.

Для этих целей был разработан следующий формат сообщения, позволяющий описать любое событие, связанное с распознаванием взгляда (рис. 14).

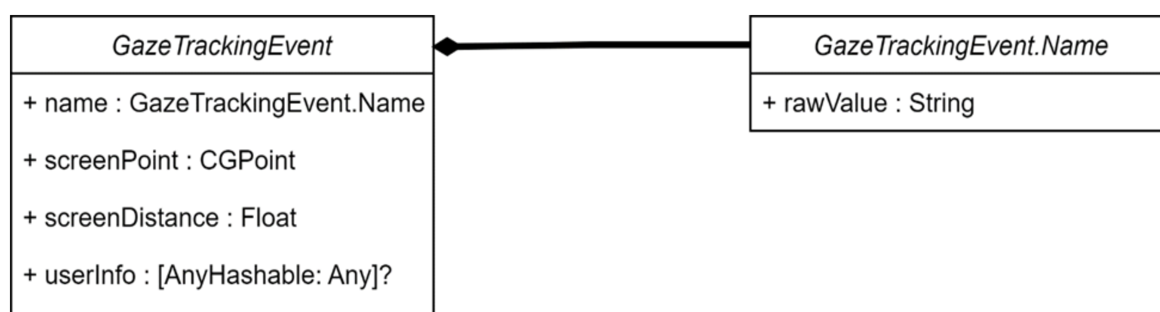

Рисунок 14. Формат событий

Такая модель события позволяет передать обработчику координаты точки взгляда в координатной системе всего экрана, дистанцию от лица пользователя до экрана, а также, опционально, любые другие параметры, переданные в виде словаря. Такой подход позволяет разработчику Backend-слоя добавить дополнительные параметры в событие без нужды изменять основную его модель. Также объект содержит поле Name, задача которого состоит в определении конкретного типа события.

Разрабатываемая реализация включает в себя следующие типы:

- *gazePositionChanged* – изменение положения взгляда;
- *leftEyeBlink* – моргание левым глазом;
- *rightEyeBlink* – моргание правым глазом;
- *bothEyesBlink* – одновременное моргание обоими глазами.

Реализация типа Name кодируется строкой, а не представляется перечислением с целью дать возможность добавлять свои собственные типы, если это понадобится пользователю.

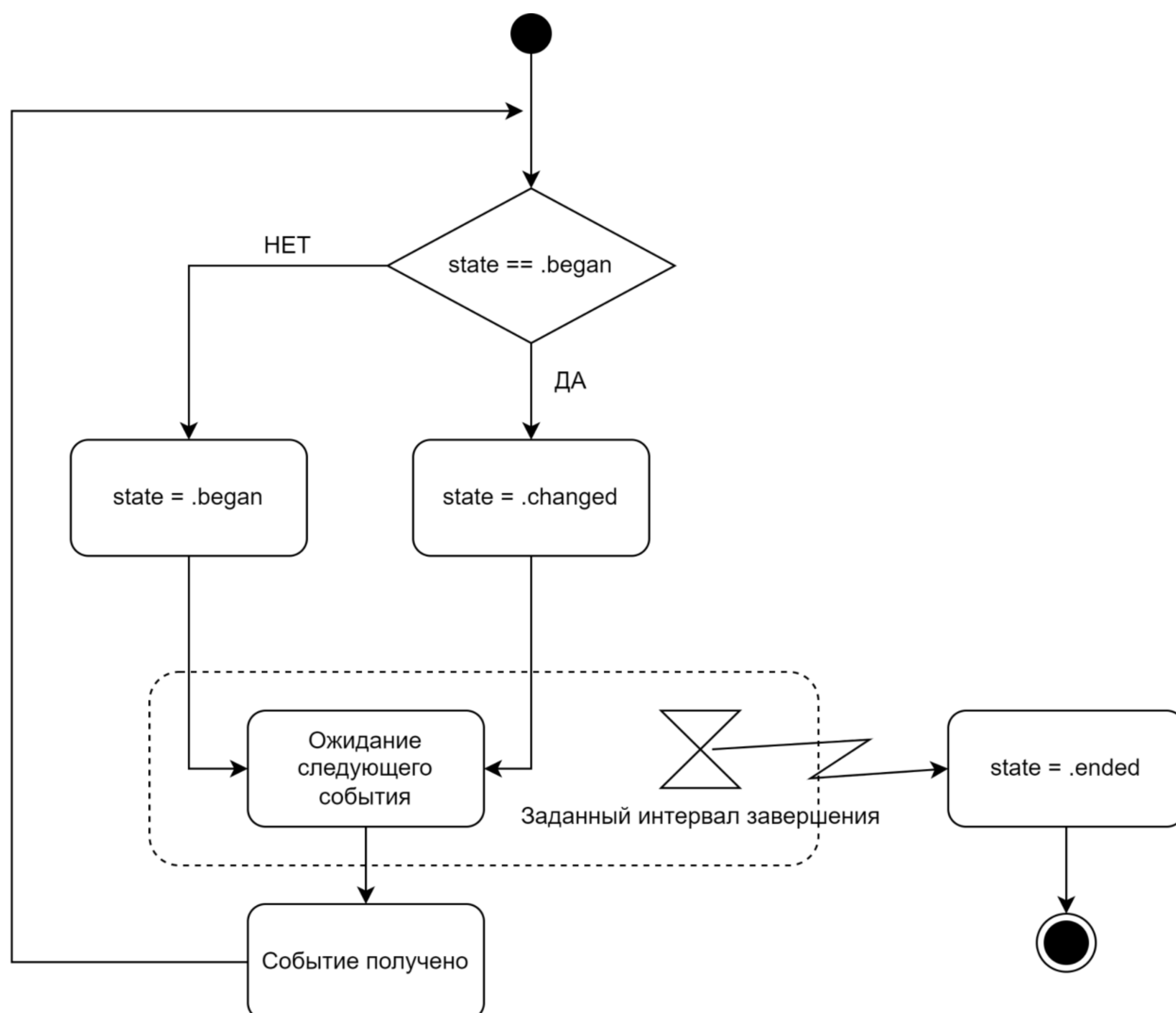

Рисунок 15. Обработка события в *GazeGestureRecognizer*

Таким образом, появляется возможность расширить число системных обработчиков жестов дополнительными вариантами, предназначенными для обработки событий взгляда и использующие вышеописанную модель для изменения своего состояния. В число новых обработчиков входят:

- *GazeGestureRecognizer* – непрерывный жест взгляда, аналог *UIPanGestureRecognizer*. Распознается до тех пор, пока пользователь фокусирует взгляд на определенном элементе отображения. Имеет настраиваемый интервал, в течение которого жест не считается завершенным, даже

если при этом взгляд пользователя переместился – это необходимо для возможности борьбы со случайным переводом взгляда или морганием. Алгоритм обработки события данным обработчиком представлен на рисунке 15.

- *BlinkGestureRecognizer* – дискретный жест моргания, аналог *UITapGestureRecognizer*. Позволяет распознавать одно или несколько морганий подряд при взгляде на определенный интерфейс интерфейса. Допускается распознавание моргания левым и правым глазами в отдельности и одновременного моргания обоими глазами. Кроме того, допускается настройка числа ожидаемых морганий (blinkCount) и максимально возможного временного интервала между морганиями в случае, когда их больше одного. Обработка события представлена на рисунке 16.

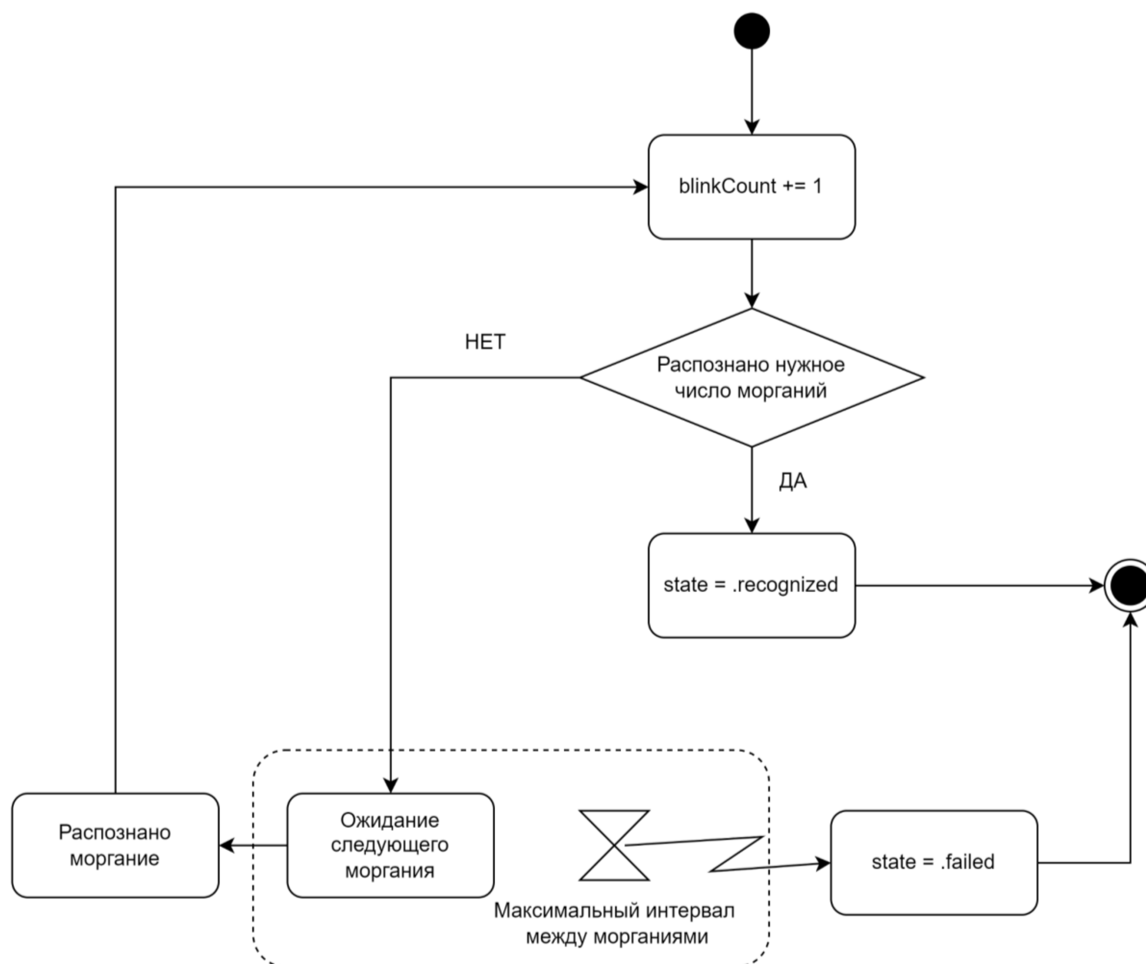

Рисунок 16. Обработка события в *BlinkGestureRecognizer*

- *LongGazeGestureRecognizer* – дискретный жест долгого взгляда на элемент, аналог *UILongPressGestureRecognizer*. Распознается при фокусировке взгляда на элементе в течение заданного периода времени. Также имеется возможность задать интервал времени, в течение которого жест не считается прерванным, даже если фокус взгляда был переведен – в целях борьбы со случайной сменой фокуса или морганием. При этом такой интервал обязательно должен быть меньше, чем интервал, соответствующий распознаванию. Реализация обработки события представлена на рисунке 17 и состоит из двух таймеров. Первое полученное событие активирует оба, а последующие перезапускают таймер, отвечающий за интервал, в течение которого жест не считается нераспознанным. Если он завершится первым, значит, пользователь отвел взгляд на длительное время, тем самым прервав жест. В противном случае первым завершится второй таймер, что будет означать успешное распознавание.

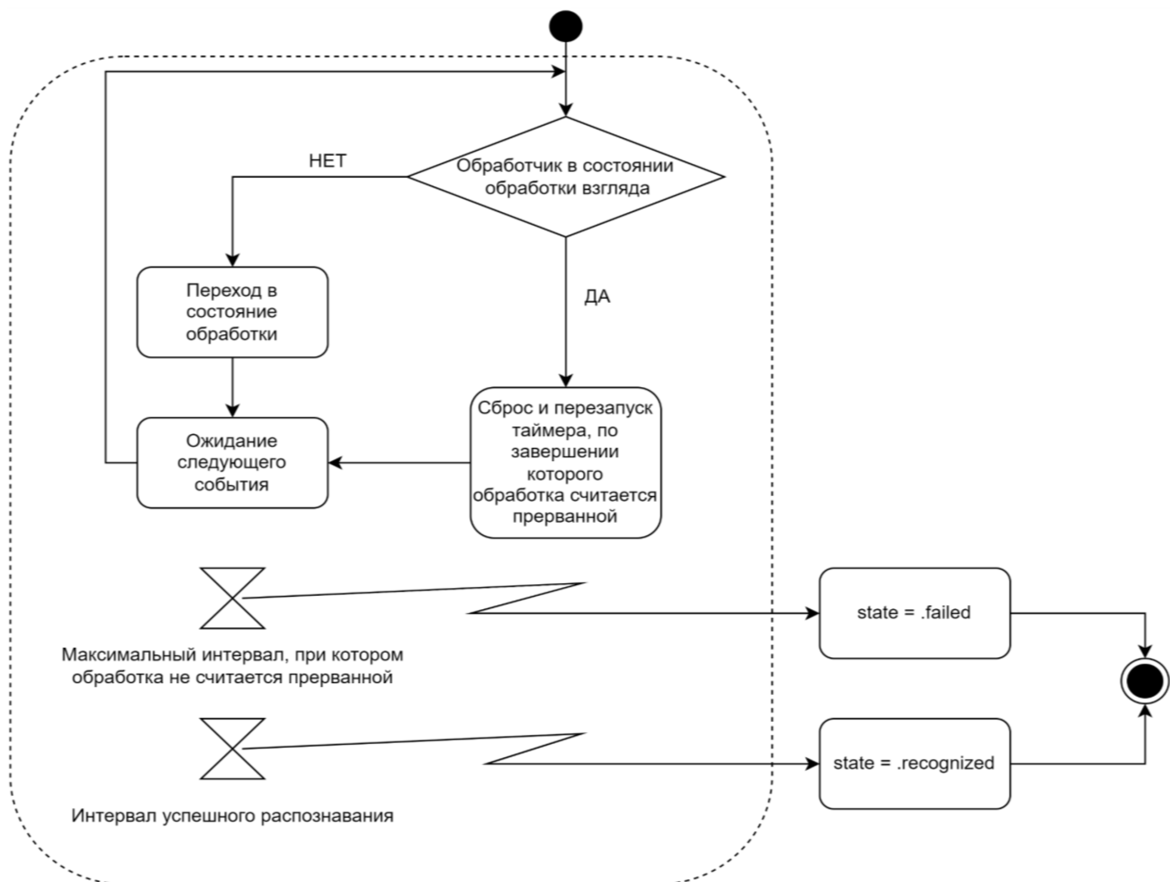

Рисунок 17. Обработка события в *LongGazeGestureRecognizer*

ДИСПЕТЧЕРИЗАЦИЯ СОБЫТИЙ МЕЖДУ ОБРАБОТЧИКАМИ

Помимо реализации самих обработчиков событий, было необходимо реализовать систему, способную доставлять события до этих обработчиков и обеспечивать их взаимосвязанное функционирование. Это является прямой обязанностью Frontend-слоя, который, будучи делегатом Backend-слоя, оповещается о каждом новом распознанном событии. Поскольку разрабатываемое решение является расширением встроенного функционала обработчиков жестов, логика решения этой задачи должна соответствовать таковой в UIKit.

Так как при реализации используется собственная модель события, возможность использовать для этой задачи встроенную логику UIKit отсутствует, и необходимо реализовывать ее отдельно. На верхнем уровне алгоритм обработки события внутри Frontend-слоя выглядит следующим образом (рис. 19).

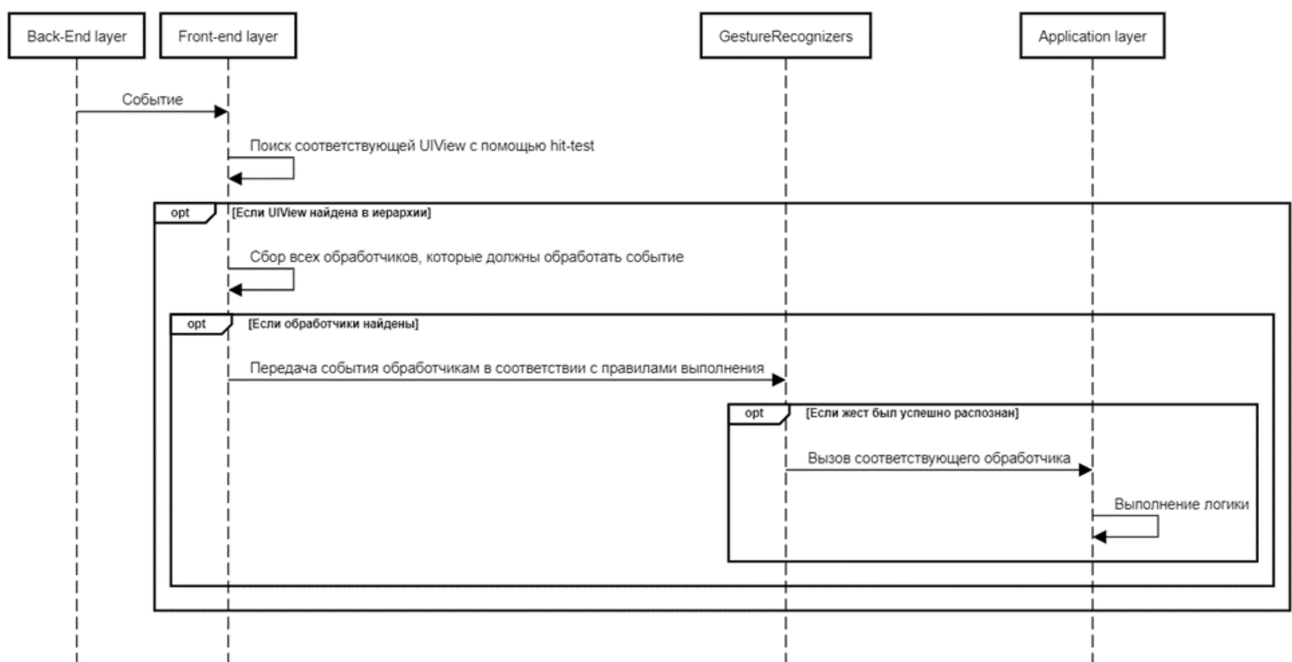

Рисунок 19. Обработка события Frontend-слоем

Первый шаг – поиск элемента, на котором сфокусирован взгляд пользователя, выполняется с помощью поиска в глубину по дереву элементов, начиная от родительского объекта UIWindow – «окна», в котором все визуальные элементы находятся в качестве дочерних с разной степенью вложенности. Все дочерние

элементы окна, будучи наследниками UIView, имеют два вспомогательных метода для выполнения этого поиска:

- `pointInisde(_ point: CGPoint, with event: UIEvent) -> Bool` – метод принимает на вход точку с координатами, выраженными в координатной системе самого элемента, а также объект события и возвращает *true* или *false*, в зависимости от того, находится ли точка внутри границ элемента.
- `hitTest(_ point: CGPoint, with event: UIEvent) -> UIView?` – метод принимает на вход точку с координатами, выраженными в координатной системе самого элемента, а также объект события и возвращает либо сам элемент, если он самый глубокий в иерархии, либо одного из своих потомков, если точка принадлежит ему, либо *nil*, если обход на этом узле должен прекратиться.

С использованием методов, описанных выше, алгоритм поиска нужного элемента реализуется стандартными средствами UIKit и заключается в вызове метода *hitTest* на объекте UIWindow с передачей ему точки в координатах экрана.

На следующем шаге необходимо получить все обработчики, которые должны получить событие. Алгоритм их нахождения полностью аналогичен таковому из UIKit и визуально представлен на рисунке 20.

После того, как все нужные обработчики найдены, нужно определить, каким образом они могут взаимодействовать между собой. По умолчанию в UIKit может происходить распознавание лишь одного жеста за раз, но это поведение можно переопределить с помощью объекта делегата обработчика жеста. Для динамического контроля за взаимодействием различных обработчиков делегат предлагает для реализации следующие методы:

- `gestureRecognizer(UIGestureRecognizer, shouldRecognizeSimultaneouslyWith: UIGestureRecognizer) -> Bool` – метод позволяет динамически определять, должны ли два обработчика обрабатывать события одновременно. При этом даже если один из этой пары вернет в этом методе *false*, это еще не гарантирует отсутствие одно-

временного распознавания, так как оставшийся обработчик может получить от своего делегата значение *true* – в этом случае жесты обработаются одновременно.

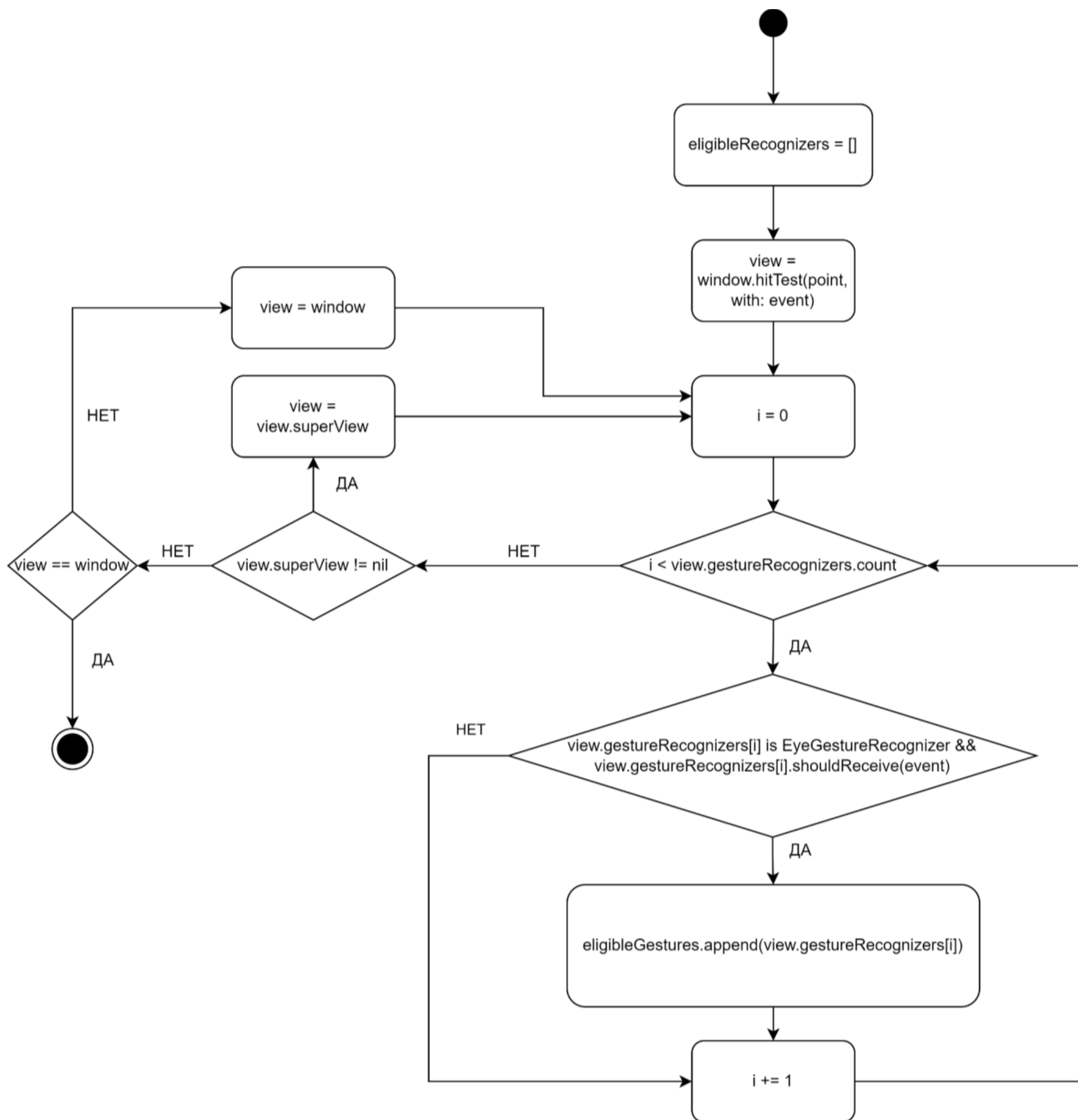

Рисунок 20. Получение всех обработчиков жестов, основанных на отслеживании взгляда

- `gestureRecognizer(UIGestureRecognizer, shouldRequireFailureOf: UIGestureRecognizer) -> Bool` – метод позволяет определить зависимость одного жеста от другого по следующему правилу: если делегат обработчика, передаваемого в метод первым параметром, вернул true, то этот обработчик получит возможность распознать жест только в том случае, если обработчик, переданный вторым параметром в процессе распознавания, перейдет в состояние *failed*. Такое поведение позволяет избегать ситуаций, при которых один из обработчиков перекрывает возможность работы для другого. Пример такой ситуации – элемент имеет обработчик единичного касания и двойного касания одновременно. При такой конфигурации при двойном нажатии не произойдет распознавание соответствующего жеста, вместо этого дважды будет распознан жест единичного касания. Используя этот метод, можно создать правило, по которому жест единичного касания будет распознаваться только тогда, когда жест двойного касания перейдет в состояние *failed.*
- `gestureRecognizer(UIGestureRecognizer, shouldBeRequiredToFailBy: UIGestureRecognizer) -> Bool` – этот метод является обратным к предыдущему и позволяет создать требование, при котором обработчик из второго параметра получит событие лишь при переходе первого в состояние *failed.*

Кроме того, аналоги последних двух методов есть также и у самих обработчиков жестов, что позволяет задавать такие требования на уровне всего подкласса. Пользуясь двумя последними методами делегата и их классовыми аналогами, можно построить между объектами обработчиков граф зависимостей в виде словаря, где ключом является обработчик, а значением – массив обработчиков, неудача распознавания которых должна привести к получению обработчиком-ключом соответствующего события. При построении графа каждый обработчик проверяется на наличие зависимостей только со всеми последующими в списке. При этом проверяется как зависимость первого от второго, так и наоборот. Стоит заметить, что если первый обработчик зависит от второго, то обратная за-

висимость уже не проверяется. С помощью этих мер удается избегать циклических зависимостей между обработчиками. Обработчик A считается зависящим от обработчика B, если выполняются оба следующих условия:

- B не зависит от A.
- Хотя бы одно из следующих условий:
  - `A.delegate?.gestureRecognizer?(A, shouldRequireFailureOf: B) == true`;
  - `B.delegate?.gestureRecognizer?(B, shouldBeRequiredToFailBy: A) == true`;
  - `A.shouldRequireFailure(of: B) == true`;
  - `B.shouldBeRequiredToFail(by: A) == true`.

Из условий, приведенных выше, следует, что приоритет при построении графа зависимостей отдается значению true: если делегат B возвращает false в методе делегата, определяющем зависимость A от B, а делегат A, в свою очередь, такую зависимость разрешает, – зависимость будет создана.

Учитывая все вышеописанное, становится возможным описать общий алгоритм диспетчеризации событий, учитывающий возможность обработчиков выполняться одновременно, а также зависеть от состояния других обработчиков. Терминальными состояниями в алгоритме, изображенном на рис. 22, названы состояния *began*, *changed* и *ended*, то есть все те, что вызывают срабатывание метода action, закрепленного за обработчиком.

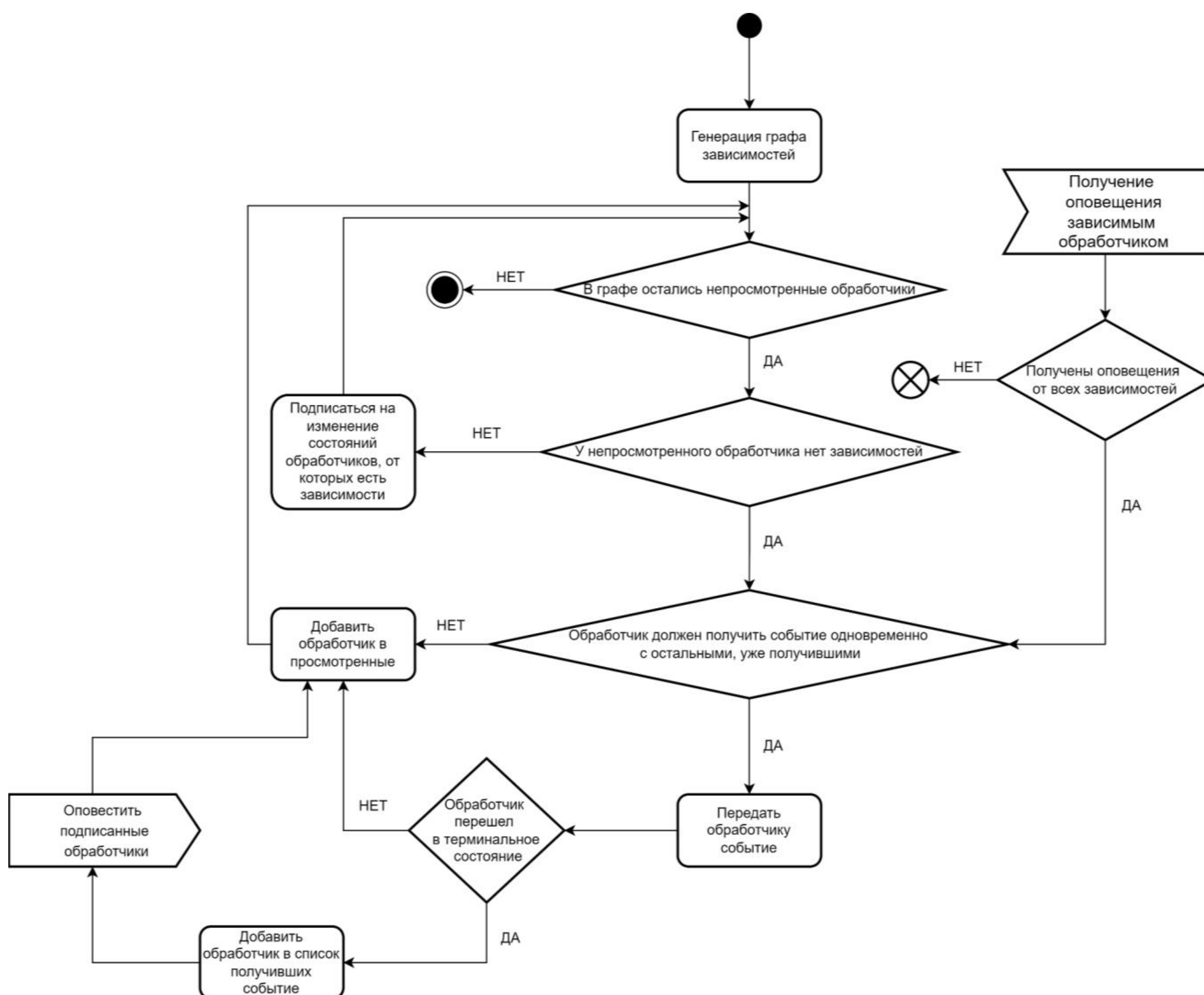

Рисунок 22. Диспетчеризация событий в системе обработчиков

Указанный выше алгоритм завершается в тот момент, когда в системе не остается непросмотренных обработчиков. В силу того, что зависимости удовлетворяются асинхронно, система диспетчеризации сохраняется в памяти до того момента, пока все зависимости не будут тем или иным образом удовлетворены.

ПРИМЕР ИСПОЛЬЗОВАНИЯ

Для подключения разработанного решения к приложению разработчику необходимо выполнить следующую последовательность действий:

1. Подключить фреймворк к проекту приложения с помощью менеджера зависимостей SPM [26].
2. Произвести инициализацию системы при старте приложения:

a. для использования конфигурации по умолчанию достаточно вызвать следующий метод:

```
EyeTrackingSystem.initializeWithDefaultConfiguration()
```

b. если необходимо задать пользовательские параметры для конфигурации, можно воспользоваться следующим методом:

```
EyeTrackingSystem.initializeWithCustomConfiguration<B:
BackendLayerProtocol, F: FrontendLayerProtocol>(_ configura-
tion: EyeTrackingConfiguration<B, F>)
```

при использовании встроенной реализации Frontend и Backend-слоев, вызов метода выше может принимать вид:

```
EyeTrackingSystem.initializeWithCustomConfiguration(
    EyeTrackingConfiguration<ARKitTracker, EventDispatcher>
        .builder()
        .backend(
            config: ARKitTrackerConfiguration(
                 blinkFrameOffset: 20,
                 leftEyeBlinkThreshold: 0.3,
                 rightEyeBlinkThreshold: 0.3
            )
        )
        .frontend(
            config: EventDispatcherConfiguration(
                windowDetectionMethod: .automatic,
                displayGazeLocation: true
            )
        )
        .build()
)
```

Для конфигурации встроенной реализации Frontend-слоя доступны два параметра: windowDetectionMethod, позволяющий передать системе ссылку на объект UIWindow, начиная от которого будет производиться поиск элемента, к которому относится событие, либо предоставить системе возможность определить этот объект самостоятельно, а также параметр displayGazeLocation, отвечающий за отображение предсказываемой точки взгляда на экране. Кроме конфигурации встроенных реализаций слоев, разработчик имеет возможность

передать в этом методе объекты собственных реализаций этих компонентов, если того требует ситуация.

3. Вызвать следующий статический метод в момент, когда нужно начать отслеживание взгляда:

   `EyeTrackingSystem.startTracking().`

4. Вызвать следующий статический метод в момент, когда отслеживание нужно приостановить (например, для экономии ресурсов): `EyeTrackingSystem.endTracking().`

После выполнения описанных выше шагов система будет готова к работе. Далее, необходимо добавить нужные обработчики жестов к элементам пользовательского интерфейса таким же образом, как при использовании системных обработчиков из UIKit. Пример добавления обработчика события моргания двумя глазами к кнопке button:

```
    let blinkGestureRecognizer = BlinkGestureRecognizer(
    target: self,
    action: #selector(self.handleBlink)
)
blinkGestureRecognizer.blinkType = .bothEyes
blinkGestureRecognizer.blinkCount = 1
button.addGestureRecognizer(blinkGestureRecognizer)
```

Пример реализации target-action логики для обработчика выше, при которой в результате моргания при взгляде на кнопку в консоль будет выведена строка “hello world”:

```
@objc func handleBlink() {
    print(“hello world”)
}
```

ОЦЕНКА ТОЧНОСТИ РАСПОЗНАВАНИЯ ВЗГЛЯДА

Для оценки эффективности разработанного решения необходимо рассмотреть две основных метрики: точность распознавания взгляда и производительность. Их оценка производится с помощью серии экспериментов и позволяет проверить соответствие решения изначально заявленным критериям. Для оценки точности приведенного алгоритма использовались две метрики – евклидово расстояние между реальной и предсказанной точками в сантиметрах и отклонение в градусах (рис. 23).

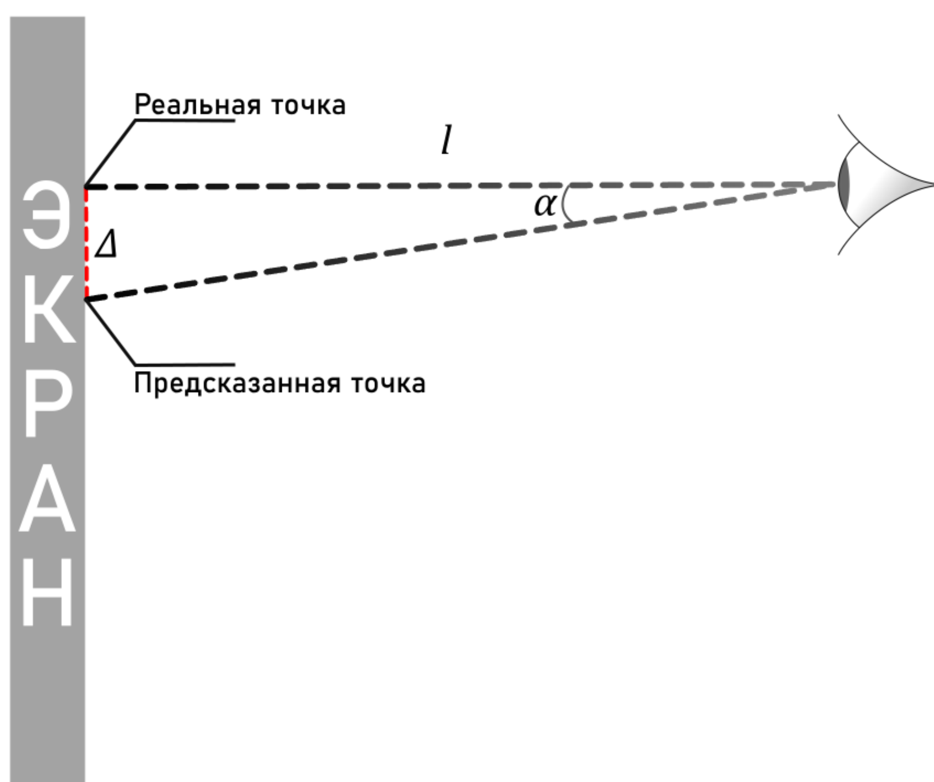

Рисунок 23. Оценка точности распознавания

На рис. 23 использованы следующие обозначения:

- Δ – пространственное отклонение (евклидово расстояние), выраженное в сантиметрах,
- α – угол отклонения, выраженный в градусах,
- *l* – расстояние от лица до экрана, выраженное в метрах.

Вычисление указанных метрик производится следующим образом:

$$\Delta = \sqrt{\left(x_{real} - x_{predicted}\right)^2 + \left(y_{real} - y_{predicted}\right)^2},$$

где x_{real} – x-координата реальной точки на экране; $x_{predicted}$ – x-координата предсказанной точки взгляда на экране; y_{real} – y-координата реальной точки на экране; $y_{predicted}$ – y-координата предсказанной точки взгляда на экране; $\alpha = atan\left(\frac{\Delta}{l}\right)$.

Получение данных, необходимых для вычисления отклонений, производилось с помощью эксперимента, в рамках которого испытуемому предлагалось следить взглядом за точкой на экране. Каждый раз, когда система распознает новую точку взгляда на экране, производится расчет евклидового расстояния между ней и «идеальной» точкой, движущейся по экрану. Кроме того, средствами ARKit фиксируется приблизительное расстояния от лица до экрана смартфона. По окончании эксперимента сохраняются среднее пространственное отклонение и среднее расстояние от лица до экрана. На основе этих данных производится расчет угла отклонения.

Траектория точки показана на рис. 24. Движение начинается от точки 1, а прохождение каждой из стрелок занимает две секунды. Последняя точка располагается в центре экрана. Так как система поддерживает альбомную ориентацию, траектория также автоматически пропорционально подстраивается под текущую ориентацию устройства.

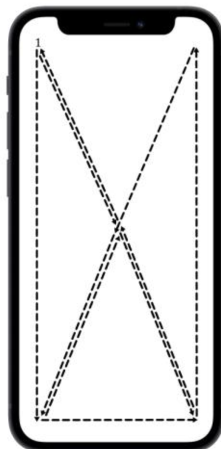

Рисунок 24. Траектория «идеальной» точки на экране

Полный эксперимент занимает 18 секунд и выполняется с частотой обновления экрана в 60 кадров в секунду. Таким образом, одно испытание позволяет получить усредненное отклонение в градусах и сантиметрах на основе около 1080 отдельных измерений.

При проведении эксперимента смартфон размещается на специальном штативе, позволяющем регулировать углы наклона по осям X и Y (рис. 25). Голова испытуемого неподвижна и расположена на заданном расстоянии на уровне экрана смартфона.

Рисунок 25. Установка для фиксации устройства

Измерения точности производились с использованием устройства Apple iPhone 12 Mini под управлением iOS 15.5. Для каждой из ориентаций (портретная, левая альбомная, правая альбомная) рассматривались пять случаев:

- устройство расположено перпендикулярно взгляду;
- устройство наклонено на 15 градусов в положительном направлении оси X;
- устройство наклонено на 15 градусов в отрицательном направлении оси X;
- устройство наклонено на 15 градусов в положительном направлении оси Y;
- устройство наклонено на 15 градусов в отрицательном направлении оси Y.

Для каждого из случаев производилось пять независимых испытаний. Таким образом, серия экспериментов содержит 75 независимых измерений, каж-

дое из которых, как было указано ранее, содержит около 1080 отдельных измерений отклонения. Всего были проведены две серии экспериментов в разных условиях. В первом случае испытания проводились без отображения предсказанной точки на экране, а во втором случае она была видимой. Результаты первой серии представлены в таблицах 1–3:

Таблица 1. Усредненные показатели точности в портретной ориентации

	Δ, pt.	Δ, см.	α, град.	*l*, м.
Перпендикулярно	121.5198449	1.863738925	3.058346105	0.348917902
15° по оси X	126.1530491	1.93479796	3.196714584	0.345608822
-15° по оси X	119.4069968	1.831334363	2.949938691	0.355282272
15° по оси Y	113.2929082	1.737563137	2.810618648	0.35404134
-15° по оси Y	112.8362141	1.73055886	2.820730189	0.351573338

Таблица 2. Усредненные показатели точности в левой альбомной ориентации

	Δ, pt.	Δ, см.	α, град.	*l*, м.
Перпендикулярно	99.2640869	1.521467504	2.46230048	0.35409145
15° по оси X	99.32916445	1.522464977	2.45649227	0.354626564
-15° по оси X	100.4491829	1.539632028	2.564976492	0.343888084
15° по оси Y	94.02865067	1.441221501	2.324039943	0.354389908
-15° по оси Y	103.6312298	1.588404763	2.526351547	0.360188454

Таблица 3. Усредненные показатели точности в правой альбомной ориентации

	Δ, pt.	Δ, см.	α, град.	*l*, м.
Перпендикулярно	106.9752863	1.639660695	2.610194765	0.359367904
15˚ по оси X	105.3752049	1.615498835	2.625566731	0.351983119
-15˚ по оси X	107.3729781	1.602190452	2.571041685	0.357779693
15˚ по оси Y	111.0812991	1.649823521	2.717085377	0.347994089
-15˚ по оси Y	111.0624266	1.570941563	2.601909458	0.346306784

Из приведенных результатов замеров можно сделать следующие выводы:

- В среднем по всей серии ошибка системы составляет около 108 точек (1.65 см) или 2.68˚.
- Стандартное отклонение в зависимости от ориентации устройства составляет около 9.61 точек (0.15 см) или 0.25˚, что позволяет заключить, что точность алгоритма существенным образом не зависит от ориентации устройства.
- Стандартное отклонение в зависимости от поворота устройства в каждой ориентации представлено в таблице 4 и позволяет заключить, что поворот устройства в заданном диапазоне по осям X и Y существенным образом не влияет на точность алгоритма.

Таблица 4. Стандартное отклонение в зависимости от поворота

	$\sigma(\Delta)$, pt.	$\sigma(\Delta)$, см.	$\sigma(\alpha)$, град.
Портретная	5.648074951	0.08662401718	0.1637479851
Левая альбомная	3.45909131	0.05301912493	0.09177681367
Правая альбомная	2.574311632	0.03132238018	0.05509940652

Результаты второй серии экспериментов (с предсказываемой точкой, видимой испытуемому) представлены в таблицах 5–-7.

Таблица 5. Усредненные показатели точности при видимой точке в портретной ориентации

	Δ, pt.	Δ, см.	α, град.	*l*, м.
Перпендикулярно	64.93241289	0.995862572	1.608413	0.354550966
15˚ по оси X	65.60818226	1.006226786	1.622489135	0.355389242
-15˚ по оси X	61.48929135	0.943055727	1.513964562	0.356741428
15˚ по оси Y	68.0074467	1.04302408	1.665302015	0.358886768
-15˚ по оси Y	65.0814779	0.998148769	1.62415837	0.352011556

Таблица 6. Усредненные показатели точности при видимой точке в левой альбомной ориентации

	Δ, pt.	Δ, см.	α, град.	*l*, м.
Перпендикулярно	64.66490306	0.991149484	1.586634077	0.357860568
15˚ по оси X	67.7229802	1.038022076	1.654347969	0.359592622
-15˚ по оси X	66.66243389	1.024849087	1.679440628	0.350282642
15˚ по оси Y	65.91835626	1.028582535	1.601117292	0.368524345
-15˚ по оси Y	68.2537732	1.015429365	1.612060424	0.360982671

Таблица 7. Усредненные показатели точности при видимой точке в правой альбомной ориентации

	Δ, pt.	Δ, см.	α, град.	*l*, м.
Перпендикулярно	67.8631078	1.040169878	1.718798882	0.346977592
15˚ по оси X	70.09962755	1.074450071	1.701589532	0.361679694
-15˚ по оси X	67.4719812	1.034174896	1.686856696	0.351101388
15˚ по оси Y	69.77300248	1.069443735	1.792034267	0.34183665
-15˚ по оси Y	70.56571451	1.055339606	1.794716585	0.337062920

Результаты второй серии экспериментов позволяют сделать следующие выводы:

- Средняя ошибка по всей серии составила около 67 точек (1.02 см) или 1.65˚. Таким образом, по сравнению с первой серией точность была увеличена на 61.4%. Это, вероятнее всего, связано с возможностью испытуемого видеть результат работы системы в реальном времени и влиять на него для улучшения результата. Исходя из такого ощутимого прироста точности, было принято решение встроить в реализацию Frontend-слоя опцию включения отображения «курсора».
- Аналогично первой серии, в результатах второй сохраняется малая зависимость точности распознавания от ориентации – стандартное отклонение составляет около 2 точек (0.02 см) или 0.06˚. Аналогичный вывод можно сделать и для зависимости от поворота по осям X и Y в диапазоне от -15˚ до 15˚.

В обеих сериях экспериментов измерения производились при среднем расстоянии между лицом испытуемого и экраном в 35 см. Имеет смысл провести дополнительные испытания с целью выявления зависимости между точностью алгоритма и расстоянием. С этой целью были дополнительно проведены две серии по 5 замеров на расстояниях в 25 см и 45 см соответственно. Замеры производились в портретной ориентации и при включенном отображении предсказываемой точки. Результаты представлены в таблице 8.

Таблица 8. Усредненные показатели ошибки в портретной ориентации в зависимости от расстояния

	Δ, pt.	Δ, см.	α, град.
0.25 м.	63.31681119	0.9710842346	2.107622504
0.35 м.	64.9324129	0.9958625725	1.608413
0.45 м.	67.43337716	1.034219636	1.317330087

Стандартное отклонение по всей выборке составляет 8 точек или около 0.12 см, что позволяет сделать вывод о незначительности зависимости точности алгоритма от расстояния в диапазоне от 25 до 45 см. В приведенной таблице отклонение, выраженное в градусах, изменяется значительно сильнее остальных метрик, так как она напрямую зависит от расстояния, и близкое евклидово отклонение на различных дистанциях дает более сильное различие в градусной мере.

Также необходимо провести сравнение разработанного решения с рассмотренными state-of-art аналогами [9–13]. Согласно данным из соответствующих работ и проведенным замерам можно видеть, что разработанное решение при отображении указателя в реальном времени опережает все рассмотренные аналоги (рис. 26). При отключенном отображении указателя решение опережает два из четырех рассмотренных аналогов. Учитывая, что проведенные в рамках данной работы замеры могут не в полной мере быть аналогичны таковым в аналогичных работах, из полученных данных можно сделать вывод, что решение обладает точностью, как минимум сравнимой с state-of-art реализациями, что позволяет говорить о выполнении критерия точности, заявленного в начале работы.

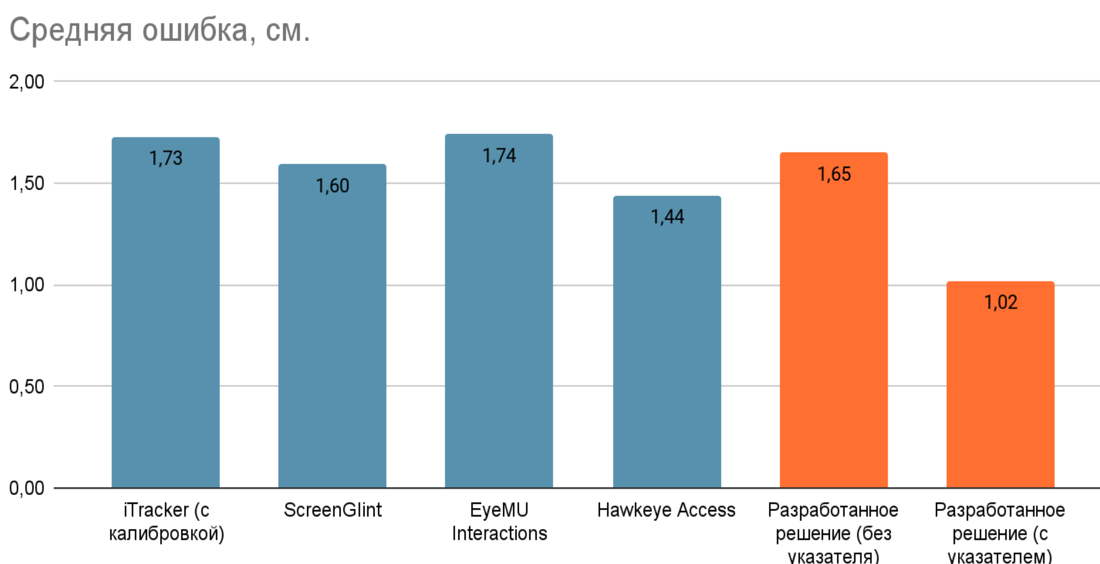

Рисунок 26. Сравнение точности с state-of-art аналогами

ОЦЕНКА ПРОИЗВОДИТЕЛЬНОСТИ

Помимо соответствия точности разрабатываемого решения state-of-art аналогам необходимо учесть также критерий производительности. Производительность решения складывается из скорости обработки данных в каждом из слоев:

- Backend-слой выполняет единственную задачу – предсказание точки взгляда на экране по алгоритму, описанному в главе 3. Этот процесс не зависит от конкретного сценария, шаги алгоритма всегда одинаковы, поэтому он поддается оценке в абсолютных единицах. На устройстве Apple iPhone 12 Mini под управлением iOS 15.5 получены следующие данные на основе выполнения алгоритма для 5000 кадров:
    - Среднее время обработки кадра – 0.00045 с.
    - Минимальное время обработки кадра – 0.000083 с.
    - Максимальное время обработки кадра – 0.0075 с.

  С учетом полученных данных можно заключить, что алгоритм достаточно производителен для обработки сигнала с частотой в 60 Гц, что согласуется с характеристиками камеры *TrueDepth*, используемой в качестве устройства ввода. Аналогичные выводы о производительности сделаны авторами статьи о применимости ARKit к задаче окулографии [22].
- Frontend-слой выполняет задачу преобразования данных, сформированных Backend-слоем, и диспетчеризации событий между обработчиками. Очевидно, что оценить производительность слоя в абсолютных единицах не представляется возможным, так как реальное число шагов алгоритма различается в зависимости от конфигурации каждого конкретного экрана приложения. Абсолютное время обработки каждого события линейно зависит от глубины иерархии элементов, так как выполняется их обход в глубину, а также квадратически от числа используемых обработчиков в силу необходимости обработки зависимостей между ними. Так как алгоритм концептуально повторяет таковой для системных обработчиков, имеются основания полагать, что в силу того, что UIKit поддерживает работу с интерфейсом с частотой кадров 60 Гц и более [27], тем же свойством обладает и Frontend-слой разрабатываемого решения.

ЗАКЛЮЧЕНИЕ

Разработанный фреймворк предоставляет разработчикам интерфейс, дающий возможность реагировать на данные о текущем положении взгляда на экране смартфона и выполнять на его основе произвольные действия. Созданное решение абстрагировано от конкретной предметной области и может быть применено для решения задач в любой из вышеперечисленных областей. В то же время, оно реализовано исключительно программными средствами и не требует каких-либо дополнительных аксессуаров, что позволяет увеличить потенциальный охват пользователей технологий окулографии при одновременном снижении трудозатрат и, как следствие, стоимости их внедрения.

Программный код разработанного фреймворка опубликован в открытом доступе в репозитории GitHub [28].

СПИСОК ЛИТЕРАТУРЫ

1. *Esiyok C. et al.* Novel hands-free interaction techniques based on the software switch approach for computer access with head movements // Universal Access in the Information Society. 2020. P. 1–15. https://doi.org/10.1007/s10209-020-00748-1

2. *Roig-Maimó M.F. et al.* Evaluation of a mobile head-tracker interface for accessibility // International Conference on Computers Helping People with Special Needs. Springer, Cham, 2016. P. 449–456.
https://doi.org/10.1007/978-3-319-41267-2_63

3. *Abbaszadegan M., Yaghoubi S., MacKenzie I.S.* TrackMaze: A comparison of head-tracking, eye-tracking, and tilt as input methods for mobile games // International Conference on Human-Computer Interaction. Springer, Cham, 2018. P. 393–405. https://doi.org/10.1007/978-3-319-91250-9_31

4. *Tupikovskaja-Omovie Z., Tyler D.* Clustering consumers' shopping journeys: eye tracking fashion m-retail // Journal of Fashion Marketing and Management: An International Journal. 2020. Т. 24. №. 3. P. 381–398.
https://doi.org/10.1108/JFMM-09-2019-0195

5. *Garbutt M. et al.* The embodied gaze: Exploring applications for mobile eye tracking in the art museum // Visitor Studies. 2020. Vol. 23. No. 1. P. 82–100.
https://doi.org/10.1080/10645578.2020.1750271

6. *Vogt M., Rips A., Emmelmann C.* Comparison of iPad Pro®'s LiDAR and TrueDepth capabilities with an industrial 3D scanning solution // Technologies. 2021. Vol. 9. No. 2. P. 25. https://doi.org/10.3390/technologies9020025

7. *Breitbarth A. et al.* Measurement accuracy and dependence on external influences of the iPhone X TrueDepth sensor // Photonics and Education in Measurement Science 2019. International Society for Optics and Photonics, 2019. Vol. 11144. P. 1114407. https://doi.org/10.1117/12.2530544

8. Number of smartphone users worldwide from 2016 to 2023 // Statista – The Statistics Portal for Market data, Market Research and Market Studies. URL: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/

9. *Krafka K. et al.* Eye tracking for everyone // Proceedings of the IEEE Conference On Computer Vision And Pattern Recognition. 2016. P. 2176–2184. https://doi.org/10.1109/CVPR.2016.239

10. *Huang M. X. et al.* Screenglint: Practical, in-situ gaze estimation on smartphones // Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems. 2017. P. 2546–2557. https://doi.org/10.1145/3025453.3025794

11. *Brousseau B., Rose J., Eizenman M.* Smarteye: An accurate infrared eye tracking system for smartphones // 2018 9th IEEE Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON). IEEE, 2018. P. 951–959. https://doi.org/10.1109/UEMCON.2018.8796799

12. Hawkeye Access | Control your iOS device using your eyes. URL: https://www.usehawkeye.com/accessibility

13. *Kong A. et al.* EyeMU Interactions: Gaze+ IMU Gestures on Mobile Devices // Proceedings of the 2021 International Conference on Multimodal Interaction. 2021. P. 577–585. https://doi.org/10.1145/3462244.3479938

14. Skyle 2 for iPad – eyeV. URL: https://eyev.de/en/ipad/

15. *Cicek M. et al.* Mobile head tracking for ecommerce and beyond //Electronic Imaging. 2020. Vol. 2020. No. 3. P. 303-1–303-12. https://doi.org/10.48550/arXiv.1812.07143

16. *Kaufman A.E., Bandopadhay A., Shaviv B.D.* An eye tracking computer user interface // Proceedings of 1993 IEEE Research Properties in Virtual Reality Symposium. IEEE, 1993. P. 120–121. https://doi.org/10.1109/VRAIS.1993.378254

17.*Gibaldi A. et al.* Evaluation of the Tobii EyeX Eye tracking controller and Matlab toolkit for research // Behavior Research Methods. 2017. Vol. 49. No. 3. P. 923–946. https://doi.org/10.3758/s13428-016-0762-9

18.*Xu P. et al.* Turkergaze: Crowdsourcing saliency with webcam based eye tracking //arXiv preprint arXiv:1504.06755. 2015. https://doi.org/10.48550/arXiv.1504.06755

19.*Qiao X. et al.* A new era for web AR with mobile edge computing // IEEE Internet Computing. 2018. Vol. 22. No. 4. P. 46–55.
https://doi.org/10.1109/MIC.2018.043051464

20.*Taaban R.A., Croock M.S., Korial A.E.* Eye Tracking Based Mobile Application // International Journal of Advanced Research in Computer Engineering & Technology (IJARCET). 2018. Vol. 7. No. 3. P. 246–250.

21.*Heryadi Y. et al.* Mata: An Android Eye-Tracking Based User Interface Control Application // Journal of Games, Game Art, and Gamification. 2016. Vol. 1. No. 1. P. 35–40. https://doi.org/10.21512/jggag.v1i1.7249

22.*Greinacher R., Voigt-Antons J.N.* Accuracy Assessment of ARKit 2 Based Gaze Estimation // International Conference on Human-Computer Interaction. Springer, Cham, 2020. P. 439–449. https://doi.org/10.1007/978-3-030-49059-1_32

23.devicekit/DeviceKit: DeviceKit is a value-type replacement of UIDevice. URL: https://github.com/devicekit/DeviceKit

24.blendShapes | Apple Developer Documentation. URL: https://developer.apple.com/documentation/arkit/arfaceanchor/2928251-blendshapes

25.init(target:action:) | Apple Developer Documentation.
URL: https://developer.apple.com/documentation/uikit/uigesturerecognizer/1624211-init

26.Swift.org – Package Manager. URL: https://www.swift.org/package-manager/

27.Optimizing ProMotion Refresh Rates for iPhone 13 Pro and iPad Pro | Apple Developer Documentation.
URL: https://developer.apple.com/library/archive/technotes/tn2460/_index.html

28.GitHub – ReQEnoxus/gaze-tracker: UIGestureRecognizer extension based on GazeTracking. URL: https://github.com/ReQEnoxus/gaze-tracker

SOFTWARE FRAMEWORK FOR IMPLEMENTING USER INTERFACE INTERACTION IN IOS APPLICATIONS BASED ON OCULOGRAPHY

Nikita Afanasev[1[0000-0002-5306-9672]]

Institute of Information Technology and Intelligent Systems, Kazan Federal University, 5 Kremlevskaya str., Kazan, 420008

[1]reqenoxus@gmail.com

Abstract

Usage of gaze tracking technologies for the purpose of user interface interaction in iOS applications is significantly hampered by the absence of a unified approach to their integration. Current solutions are either strictly limited to their own use-case or made solely for research purposes and thus inapplicable to real-world problems. The focus of this article is the development of a software framework that performs gaze tracking using native technologies and suggests a unified approach to the development of gaze-driven iOS applications.

***Keywords**: gaze tracking, eye tracking, oculography, gesture recognizers, TrueDepth, ARKit, SceneKit, UIKit, iOS, UX, UI*

REFERENCES

1. *Esiyok C. et al.* Novel hands-free interaction techniques based on the software switch approach for computer access with head movements // Universal Access in the Information Society. 2020. P. 1–15. https://doi.org/10.1007/s10209-020-00748-1
2. *Roig-Maimó M.F. et al.* Evaluation of a mobile head-tracker interface for accessibility // International Conference on Computers Helping People with Special Needs. Springer, Cham, 2016. P. 449–456. https://doi.org/10.1007/978-3-319-41267-2_63

3. *Abbaszadegan M., Yaghoubi S., MacKenzie I.S.* TrackMaze: A comparison of head-tracking, eye-tracking, and tilt as input methods for mobile games // International Conference on Human-Computer Interaction. Springer, Cham, 2018. P. 393–405. https://doi.org/10.1007/978-3-319-91250-9_31

4. *Tupikovskaja-Omovie Z., Tyler D.* Clustering consumers' shopping journeys: eye tracking fashion m-retail // Journal of Fashion Marketing and Management: An International Journal. 2020. T. 24. №. 3. P. 381–398.
https://doi.org/10.1108/JFMM-09-2019-0195

5. *Garbutt M. et al.* The embodied gaze: Exploring applications for mobile eye tracking in the art museum // Visitor Studies. 2020. Vol. 23. No. 1. P. 82–100.
https://doi.org/10.1080/10645578.2020.1750271

6. *Vogt M., Rips A., Emmelmann C.* Comparison of iPad Pro®'s LiDAR and TrueDepth capabilities with an industrial 3D scanning solution // Technologies. 2021. Vol. 9. No. 2. P. 25. https://doi.org/10.3390/technologies9020025

7. *Breitbarth A. et al.* Measurement accuracy and dependence on external influences of the iPhone X TrueDepth sensor // Photonics and Education in Measurement Science 2019. International Society for Optics and Photonics, 2019. Vol. 11144. P. 1114407. https://doi.org/10.1117/12.2530544

8. Number of smartphone users worldwide from 2016 to 2023 // Statista – The Statistics Portal for Market data, Market Research and Market Studies.
URL: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/

9. *Krafka K. et al.* Eye tracking for everyone // Proceedings of the IEEE Conference On Computer Vision And Pattern Recognition. 2016. P. 2176–2184.
https://doi.org/10.1109/CVPR.2016.239

10. *Huang M. X. et al.* Screenglint: Practical, in-situ gaze estimation on smartphones // Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems. 2017. P. 2546–2557. https://doi.org/10.1145/3025453.3025794

11. *Brousseau B., Rose J., Eizenman M.* Smarteye: An accurate infrared eye tracking system for smartphones // 2018 9th IEEE Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON). IEEE, 2018. P. 951–959.
https://doi.org/10.1109/UEMCON.2018.8796799

12.Hawkeye Access | Control your iOS device using your eyes. URL: https://www.usehawkeye.com/accessibility

13.*Kong A. et al.* EyeMU Interactions: Gaze+ IMU Gestures on Mobile Devices // Proceedings of the 2021 International Conference on Multimodal Interaction. 2021. P. 577–585. https://doi.org/10.1145/3462244.3479938

14.Skyle 2 for iPad – eyeV. URL: https://eyev.de/en/ipad/

15.*Cicek M. et al.* Mobile head tracking for ecommerce and beyond //Electronic Imaging. 2020. Vol. 2020. No. 3. P. 303-1–303-12. https://doi.org/10.48550/arXiv.1812.07143

16. *Kaufman A.E., Bandopadhay A., Shaviv B.D.* An eye tracking computer user interface // Proceedings of 1993 IEEE Research Properties in Virtual Reality Symposium. IEEE, 1993. P. 120–121. https://doi.org/10.1109/VRAIS.1993.378254

17.*Gibaldi A. et al.* Evaluation of the Tobii EyeX Eye tracking controller and Matlab toolkit for research // Behavior Research Methods. 2017. Vol. 49. No. 3. P. 923–946. https://doi.org/10.3758/s13428-016-0762-9

18.*Xu P. et al.* Turkergaze: Crowdsourcing saliency with webcam based eye tracking //arXiv preprint arXiv:1504.06755. 2015. https://doi.org/10.48550/arXiv.1504.06755

19.*Qiao X. et al.* A new era for web AR with mobile edge computing // IEEE Internet Computing. 2018. Vol. 22. No. 4. P. 46–55. https://doi.org/10.1109/MIC.2018.043051464

20.*Taaban R.A., Croock M.S., Korial A.E.* Eye Tracking Based Mobile Application // International Journal of Advanced Research in Computer Engineering & Technology (IJARCET). 2018. Vol. 7. No. 3. P. 246–250.

21.*Heryadi Y. et al.* Mata: An Android Eye-Tracking Based User Interface Control Application // Journal of Games, Game Art, and Gamification. 2016. Vol. 1. No. 1. P. 35–40. https://doi.org/10.21512/jggag.v1i1.7249

22.*Greinacher R., Voigt-Antons J.N.* Accuracy Assessment of ARKit 2 Based Gaze Estimation // International Conference on Human-Computer Interaction. Springer, Cham, 2020. P. 439–449. https://doi.org/10.1007/978-3-030-49059-1_32

23.devicekit/DeviceKit: DeviceKit is a value-type replacement of UIDevice. URL: https://github.com/devicekit/DeviceKit

24.blendShapes | Apple Developer Documentation. URL: https://developer.apple.com/documentation/arkit/arfaceanchor/2928251-blendshapes

25.init(target:action:) | Apple Developer Documentation. URL: https://developer.apple.com/documentation/uikit/uigesturerecognizer/1624211-init

26.Swift.org – Package Manager. URL: https://www.swift.org/package-manager/

27.Optimizing ProMotion Refresh Rates for iPhone 13 Pro and iPad Pro | Apple Developer Documentation. URL: https://developer.apple.com/library/archive/technotes/tn2460/_index.html

28.GitHub – ReQEnoxus/gaze-tracker: UIGestureRecognizer extension based on GazeTracking. URL: https://github.com/ReQEnoxus/gaze-tracker

---

СВЕДЕНИЯ ОБ АВТОРЕ

АФАНАСЬЕВ Никита Станиславович – бакалавр Института информационных технологий и интеллектуальных систем КФУ, г. Казань.

Nikita AFANASEV – undergraduate student of the Institute of Information Technologies and Intelligent Systems, Kazan (Volga region) Federal University, Kazan.

e-mail: reqenoxus@gmail.com

ORCID: 0000-0002-5306-9672

Материал поступил в редакцию 25 июня 2022 года