https://download.geofabrik.de/asia/kazakhstan.html.

Отлично — давай отправлять данные с ESP32 прямо в **Google Sheets** через Google Apps Script и Webhook. Это бесплатный способ логгировать любые данные в Google Таблицу **без серверов и баз данных**.

## ✅ Что ты получишь

ESP32 будет:

1. Каждые 5 минут забирать уровень пробок из JSON.
2. Отправлять этот уровень и метку времени в Google Таблицу.
3. Всё хранится в Google Sheets, можно строить графики и анализ.

## 📁 Шаг 1: Подготовь Google Таблицу

1. Создай новый Google Sheet: [https://sheets.new](https://sheets.new/)
2. Назови, например: Almaty Traffic Log
3. В **первую строку** добавь заголовки:

Time, Level

## 🧩 Шаг 2: Напиши Google Apps Script

1. В таблице нажми Extensions → Apps Script
2. Удали всё и вставь:

function doPost(e) {

var sheet = SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Лист1");

var data = JSON.parse(e.postData.contents);

var time = new Date();

var level = data.level;

sheet.appendRow([time, level]);

return ContentService.createTextOutput("OK");

}

1. Нажми **Deploy → New deployment**
2. Нажми **Select type → Web app**
3. Установи:
   * Execute as: **Me**
   * Who has access: **Anyone**
4. Нажми **Deploy**, разреши доступ → скопируй URL вида:

https://script.google.com/macros/s/AKf.../exec

## 🔐 Шаг 3: Код ESP32 для отправки в Google Sheets

Вот доработанный код с отправкой в Google:

#include <WiFi.h>

#include <HTTPClient.h>

const char\* ssid = "YOUR\_WIFI\_SSID";

const char\* password = "YOUR\_WIFI\_PASSWORD";

const char\* traffic\_url = "https://raw.githubusercontent.com/yourusername/traffic-data/main/almaty-traffic.json";

const char\* google\_webhook\_url = "https://script.google.com/macros/s/AKf.../exec"; // вставь свой URL

unsigned long interval = 5 \* 60 \* 1000;

unsigned long lastCheck = 0;

void setup() {

Serial.begin(115200);

WiFi.begin(ssid, password);

while (WiFi.status() != WL\_CONNECTED) delay(500);

Serial.println("WiFi connected.");

}

void loop() {

unsigned long now = millis();

if (now - lastCheck >= interval || lastCheck == 0) {

lastCheck = now;

if (WiFi.status() == WL\_CONNECTED) {

HTTPClient http;

http.begin(traffic\_url);

int httpCode = http.GET();

if (httpCode == 200) {

String payload = http.getString();

Serial.println("Traffic JSON: " + payload);

int idx = payload.indexOf("level");

int colon = payload.indexOf(":", idx);

int end = payload.indexOf("}", colon);

String levelStr = payload.substring(colon + 1, end);

levelStr.trim();

int trafficLevel = levelStr.toInt();

Serial.print("Traffic level: ");

Serial.println(trafficLevel);

// Отправляем в Google Sheets

HTTPClient httpPost;

httpPost.begin(google\_webhook\_url);

httpPost.addHeader("Content-Type", "application/json");

String jsonPayload = "{\"level\": " + String(trafficLevel) + "}";

int postCode = httpPost.POST(jsonPayload);

Serial.print("POST response: ");

Serial.println(postCode);

httpPost.end();

}

http.end();

}

}

delay(1000);

}

## ✅ Что можно улучшить

* Добавить **обработку ошибок / offline-режим**.
* Поддержка **SD-карты для бэкапа**.
* График на Google Sheets: просто вставь Line Chart по столбцу Level.

Хочешь — могу ещё автоматизировать создание графика, Telegram-бота с уведомлениями, или систему на ESP32 с дисплеем.

ID

AKfycbwJ3I56O3ciOp6uDYiHC1E70mrcyABTBYZbk1ArKzafGYB2FzGMfXGMn4C9Pwk4XkeO  
  
  
Deployment  
https://script.google.com/macros/s/AKfycbwJ3I56O3ciOp6uDYiHC1E70mrcyABTBYZbk1ArKzafGYB2FzGMfXGMn4C9Pwk4XkeO/exec

Отлично, начну проектирование комплексной ГИС-системы с GUI, которая будет визуализировать дорожные пробки на основе данных из shapefile и динамического парсинга трафика с карт. Я подберу архитектуру с высокой производительностью, мультиплатформенным GUI, поддержкой стеков C++/Rust, SQLite/PostGIS, и возможностью гибкой настройки визуализации, тем и языков интерфейса.

Дам тебе детальное техническое обоснование по:

* структуре проекта (модули, языки, зависимости);
* выбору подходящих библиотек для карты, GUI, БД и парсинга;
* организации работы с shapefile (и разбиением на отрезки по 10м);
* архитектуре системы обновления данных из веба с учётом точности и защиты от блокировок;
* возможностям расширения под glassmorphism и мультиязычность.

Ожидай готовый проектный план и стартовую реализацию в ближайшее время.

# Архитектура кроссплатформенного ГИС‑приложения для карты пробок Алматы

## Структура проекта и стек технологий

Проект следует разделить на модули, обеспечивающие сбор данных о пробках, хранение и обработку этих данных, а также визуализацию и интерфейс. Рассмотрим **язык и стек технологий** для каждого компонента:

* **Язык и производительность:** Наиболее подходящими являются **C++** или **Rust**, так как они компилируемые, высокопроизводительные и надёжные (без сборщика мусора). Rust обеспечивает гарантии безопасности памяти и потоков на этапе компиляции ( [Rust Programming Language](https://www.rust-lang.org/" \l ":~:text=Reliability) ), что снижает риск ошибок, а C++ обладает зрелой экосистемой библиотек (например, Qt для GUI). Оба языка способны эффективно обрабатывать большие объёмы геоданных в реальном времени. Можно даже сочетать их (например, основную логику на Rust, а GUI на Qt/C++ через биндинги) для лучшего баланса безопасности и наличия готовых инструментов.
* **GUI (кросс-платформенная):** Рекомендуется **Qt** (Qt 6) как фреймворк для настольного GUI. Qt – открытое решение (LGPL) с высокой производительностью и поддержкой Windows/Linux/macOS, оно предлагает богатые возможности для кастомизации интерфейса и готовую поддержку i18n. Qt Quick (QML) использует аппаратное ускорение через OpenGL для отрисовки интерфейса, обеспечивая плавную работу даже с большим числом графических элементов. Альтернативы: Рассматривается **Slint** (ранее SixtyFPS) – современная лёгкая кросс-платформенная GUI-библиотека с поддержкой Rust и C++ ([Slint 1.4 released: lightweight cross-platform GUI toolkit for Rust, C++ and JavaScript • DEVCLASS](https://devclass.com/2024/02/01/slint-1-4-released-lightweight-cross-platform-gui-toolkit-for-rust-c-and-javascript/" \l ":~:text=Slint%20is%20an%20open%20source,js%20or%20Deno)). Slint тоже не требует проприетарных компонентов и может рендерить UI через OpenGL или Skia, однако экосистема Qt более зрелая (например, есть Qt Location, Qt Linguist и др.). В целом стек может выглядеть так: **Rust/C++ (core)** + **Qt/QML (UI)** + **GDAL/OGR** для работы с геоданными, **Selenium/OpenCV** для парсинга пробок (подробнее ниже), и **SQLite/PostGIS** для хранения данных.
* **Отсутствие проприетарности:** Все выбранные компоненты должны быть с открытой лицензией. Qt, GDAL/OGR, SQLite, PostGIS – все открытые. **Не использовать проприетарные SDK** (например, закрытые картографические движки или коммерческие БД). Для рендеринга карты можно использовать OpenGL напрямую или через Qt Quick Scene Graph – это обеспечит контроль над производительностью без зависимостей на проприетарные движки.

**Структура проекта** может быть организована по шаблону MVC/MVVM:

* **Модель (Model/Data):** слой, отвечающий за хранение дорожной сети и информации о пробках. Реализуется в виде менеджера данных, работающего с локальной БД (или in-memory структурами) и предоставляющего API для выборки сегментов по условиям (например, получить сегменты для отображения в текущем масштабе).
* **Парсинг и синхронизация (Traffic Fetcher):** отдельный модуль/поток, который раз в минуту собирает информацию о пробках с онлайн-карт и обновляет Model. Должен работать асинхронно, чтобы не блокировать UI.
* **Визуализация (View/Map Renderer):** компонент, отвечающий за отрисовку карты Алматы из локального SHP (разбитого на сегменты) и раскрашивание сегментов в соответствии с уровнем пробки. Может быть реализован как кастомный виджет (например, унаследованный от QQuickItem или QGraphicsView), который запрашивает у Model текущие данные и рендерит их.
* **Контроллер/Логика (Controller):** связывает всё вместе – реагирует на действия пользователя (фильтрация, масштабирование карты, выбор источника данных), вызывает обновление данных и перерисовку. В Qt это частично реализуется через сигналы/слоты и QML-движок, в Rust+Slint – через встроенные механизмы сигналов DSL.

Такая структура обеспечит модульность: можно заменять компоненты (например, переключиться с SQLite на PostGIS, или добавить новый источник данных пробок) без серьёзных изменений остальных частей.

## Архитектура парсинга и синхронизации с картами

Сбор данных о пробках будет происходить путем парсинга публично доступной информации с нескольких онлайн-карт (например, 2ГИС, Яндекс.Карты, Google Maps). Ключевые элементы архитектуры парсера:

* **Используемые инструменты:** Для автоматизированного доступа к картам можно применять **Selenium** (WebDriver) для загрузки веб-версий карт, а затем анализировать DOM или делать скриншоты с наложенными пробками. Если данные выдаются графически (цветные линии на дорогах), понадобится **компьютерное зрение (OpenCV)** или анализ пикселей изображения; если удаётся получить числовые данные (например, API возвращает скорости), достаточно парсить JSON/XML. Также пригодится **OCR** для распознавания текстовых меток на картах (например, если на карте отображаются значения скорости или сообщения о ДТП).
* **Мульти-источники и переключение:** Парсер должен уметь обращаться к разным источникам. Реализуется абстракция TrafficSource с единым интерфейсом (например, метод getTrafficData(roadSegments)), а конкретные реализации – TwoGISTrafficSource, YandexTrafficSource, GoogleTrafficSource. Приложение может опрашивать их по очереди или параллельно. **Автоматическое переключение**: если один источник недоступен или заметно отдаёт некорректные данные, используется другой. Например, по умолчанию брать данные 2ГИС, но если 2ГИС не отвечает – переключиться на Яндекс. Также можно комбинировать: для повышенной надёжности сверять данные – если один источник показывает красный (сильный затор), а другой зелёный на том же участке, пометить данные этого участка как подозрительные и не отображать крайние значения.
* **Парсинг 2ГИС (пример):** 2ГИС позволяет открыть карту города с пробками через URL с параметром (например, https://2gis.kz/almaty?...&traffic). В браузере это отобразит карту Алматы с выделением пробок цветом. Далее:
  1. **Headless-загрузка:** Selenium в безголовом режиме открывает эту страницу (parse.py) (parse.py). Скриптом JavaScript скрываем лишние элементы интерфейса (панели, маркеры) чтобы не мешали анализу (parse.py).
  2. **Получение изображения:** Делаем скриншот видимой области карты (например, 1080x1080 px) (parse.py). При необходимости, можно увеличить разрешение (скриншотить в более высоком масштабе или увеличить изображение) для повышения точности распознавания мелких сегментов (parse.py).
  3. **Сопоставление с сегментами:** Зная географические координаты наших дорожных сегментов (из SHP), нужно определить, какие пиксели на скриншоте им соответствуют. Можно использовать **проекцию Web Mercator (EPSG:3857)** – все онлайн-карты (Google, Яндекс, 2ГИС) используют схожую проекцию меркатора. Алгоритм: получаем текущий центр карты и масштаб (zoom) из URL или известной формулы, переводим координаты сегментов (широта/долгота) в пиксельные координаты на изображении (parse.py) (parse.py). В прототипе можно применить приближённую формулу с cos(φ) для пересчёта метров в пиксели, как сделано в примере с Selenium (parse.py).
  4. **Измерение цвета сегмента:** Для каждого сегмента берём пиксельные координаты его начала и конца, выбираем несколько точек между ними и считываем цвет пикселя (parse.py) (parse.py). Усреднив цвет, определяем степень пробки. Например, если много красного – значит затор, зелёного – свободно. В 2ГИС и Яндекс пробки обозначаются цветами: **зелёный** = свободно, **жёлтый** = затруднения, **красный** = пробка ([Точные пробки в Алматы сразу в двух системах](https://findh.org/3455-onlain-probki.html?region=almaty" \l ":~:text=%D0%9E%D0%B1%D0%BE%D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%B8%D0%B5%20%D0%BF%D1%80%D0%BE%D0%B1%D0%BE%D0%BA%20%D1%86%D0%B2%D0%B5%D1%82%D0%BE%D0%BC%20%D0%BD%D0%B0%20%D0%BA%D0%B0%D1%80%D1%82%D0%B0%D1%85%3A) ). Можно ввести градации (например, 5 уровней цвета от зелёного к красному или численно 1–10) (parse.py). В нашем случае хватит 3–4 градаций (зелёный, жёлтый, красный, возможно бордовый для застойных пробок). Полученный цвет сегмента сразу сохраняется как показатель трафика.
  5. **OCR/дополнение:** Если источник (например, Google) отображает определённые метки (иконки ДТП, текст "ДТП" или скорость на участке), можно дополнительно распознавать их. Однако для начальной реализации достаточно цветовой информации.
* **Парсинг Яндекс.Карт:** Яндекс тоже отображает пробки цветными линиями. Можно аналогично загрузить их веб-карты через Selenium. Возможно, потребуется имитировать нажатие кнопки "Пробки" через JS, если прямым URL не включить. После отображения – та же процедура: снять скрин, вычислить цвета. У Яндекса шкала также зелёный/жёлтый/красный, плюс они дают **балл пробок** (1–10) для всего города, который можно парсить из DOM (но он глобальный, не для сегментов).
* **Парсинг Google Maps:** Google сложнее, так как веб-версия может требовать API ключ или отказываться загружаться полноценно под автоматизацией. Тем не менее, можно попробовать использовать Static Maps API с параметром стиля, либо тоже Selenium. Если удастся получить изображения с Traffic Layer, анализ аналогичен. В случае Google Maps API, можно запросить дорожный граф с данными о пробках (скорость на сегменте) через Directions API или Roads API, но это потребует API ключ и может нарушать условие неиспользования проприетарных решений (да и вызовы платные). Поэтому предпочтём методы, не требующие закрытых API.
* **Частота запросов:** Необходимо соблюдать разумную частоту. Раз в минуту для каждого источника – это 60 запросов/час, что в пределах допустимого для картографических сервисов (обычно в пользовательских условиях ограничений на просмотр карты нет, но агрессивный парсинг мог бы вызвать капчу). Чтобы снизить нагрузку, можно:
  1. Кешировать результаты: если за предыдущую минуту данные не изменились (например, ночью пробки не меняются быстро), то лишний раз не перезапрашивать.
  2. Чередовать источники: например, в чётные минуты брать 2ГИС, в нечётные – Яндекс, тем самым каждый источник дергается раз в 2 минуты.
  3. Использовать минимально необходимый объем данных: загружать только карту нужного района. 2ГИС позволяет задать bounding box/центр и зум, аналогично можно для Яндекса.
* **Защита от ложных данных:** Как упомянуто выше, слияние данных разных провайдеров поможет фильтровать ошибки. Если один источник внезапно показал пробку (красный) на сегменте, а остальные нет – возможно это артефакт (либо локальная аномалия). Можно либо игнорировать одиночное значение, либо отметить сегмент особым цветом (например, синим – данные не подтверждены). В дальнейшем, накопив исторические данные, можно обучить модель обнаружения выбросов: сравнивать текущую скорость с типичной в это время дня, чтобы понять, не ошибка ли.
* **Интеграция с основным приложением:** Парсер работает в отдельном потоке или процессе (в фоновом режиме). Раз в минуту он получает данные пробок (например, в виде словаря {segment\_id: traffic\_level}). Затем через сигнал/сообщение передаёт их UI-части. UI, получив новые данные, обновляет цвета сегментов на карте. Благодаря этому обновление происходит асинхронно и плавно.

Архитектурно, данный парсер-модуль изолирован: его можно запускать и отлаживать отдельно. Это позволит в будущем легко заменить его реализацию (например, перейти на официальный API, если он станет доступен, или добавить новые источники – Waze, OpenStreetMap Traffic и т.п.).

## Рекомендации по формату и предобработке SHP-файлов

**Формат исходных данных:** Дорожная сеть Алматы будет загружена из локального **Shapefile (SHP)**. Shapefile – распространённый формат ГИС-данных (набор файлов .shp, .dbf, .shx и др.), поддерживающий хранение геометрии и атрибутов. Однако у него есть ограничения: например, имена атрибутов не более 10 символов, нет поддержки топологии, размер файла до 2 ГБ ([What is a shapefile? .shp, .dbf and .shx • Max Dietrich](https://mxd.codes/articles/what-is-a-shapefile-shp-dbf-and-shx" \l ":~:text=,No%20genuine%203D%20support)). Для нашей задачи shapefile приемлем, но стоит рассмотреть более современные альтернативы. **GeoPackage (GPKG)** или **PostGIS** предлагают больше возможностей и хранят всё в одном файле/БД без этих ограничений ([What is a shapefile? .shp, .dbf and .shx • Max Dietrich](https://mxd.codes/articles/what-is-a-shapefile-shp-dbf-and-shx" \l ":~:text=Exploring%20Alternatives%20to%20Shapefiles)). GeoPackage – это открытый формат на базе SQLite, а PostGIS – расширение к PostgreSQL, делающее его полнофункциональной пространственной БД ([SpatiaLite: SpatiaLite](https://www.gaia-gis.it/fossil/libspatialite/index" \l ":~:text=SpatiaLite%20is%20smoothly%20integrated%20into,equivalent%20to%20PostgreSQL%20%2B%20PostGIS)). Оба могут надежно хранить наши сегменты дорог и исторические данные пробок. Например, PostGIS широко используется для дорожных сетей благодаря масштабируемости и богатому набору гео-функций ([Applications of PostGIS and PostgreSQL in Modern Geospatial Analysis](https://www.ve3.global/applications-of-postgis-and-postgresql-in-modern-geospatial-analysis/" \l ":~:text=PostGIS%20is%20an%20open,and%20PostgreSQL%20across%20different%20industries)).

**Предобработка дорожной сети (разбиение на 10-метровые сегменты):** В исходном shapefile дороги скорее всего представлены полилиниями целых улиц или крупных отрезков. Нам нужно **разрезать каждую линию на сегменты длиной 10 м**. Это можно сделать следующими способами:

* **ГИС-инструменты:** Использовать QGIS или GDAL. В GDAL есть утилита ogr2ogr с возможностью разбиения, либо подключить скрипт на Python с использованием OGR API. Также GRASS GIS содержит модуль v.split.length, который разбивает линии на сегменты заданной длины ([Split polylines at a specific distance in QGIS / ogr2ogr](https://gis.stackexchange.com/questions/258216/split-polylines-at-a-specific-distance-in-qgis-ogr2ogr" \l ":~:text=ogr2ogr%20gis,does%20exactly)). Например, можно загрузить shapefile в GRASS, запустить v.split.length input=roads output=roads\_10m length=10 – на выходе получится новый файл, где каждая дорога разбита на ~10-метровые отрезки.
* **PostGIS:** Загрузить исходные линии в PostGIS и использовать функцию **ST\_Segmentize** или ST\_Split. ST\_Segmentize(geom, 10) добавит вертексы каждые 10 м на геометрии, а затем ST\_Dump или ST\_Split можно применить для разделения по этим точкам. Так мы программно получим множество коротких LineString.
* **Скрипт вручную:** Написать скрипт, который читает координаты полилинии и шагает по ним с накоплением длины, создавая новые сегменты при превышении 10 м. Это можно сделать, но лучше опереться на готовые гео-библиотеки (меньше риска ошибок в геодезии).

**Атрибуты сегментов:** При разрезании важно сохранить идентификатор улицы и порядок сегментов. Например, присвоить каждому сегменту уникальный ID вида "StreetName\_seg001". В выходном наборе данных будет колонка с названием улицы и номером сегмента. Так можно будет фильтровать или подсвечивать целиком улицу, собрав все сегменты с одинаковым именем. Также можно хранить длину сегмента (в идеале ~10 м, крайние могут быть короче если длина улицы не делится ровно).

**Проекция данных:** Убедиться, что координаты согласованы с теми, что ожидает наш визуализатор и парсер пробок. Оптимально использовать **WGS84** (широта/долгота) для хранения, поскольку источники пробок (Яндекс, Google) работают в широте/долготе. Однако для отрисовки на плоскости лучше проектировать координаты в метры (например, Web Mercator EPSG:3857 или местная проекция). Можно хранить два поля геометрии: исходный WGS84 и спроецированный вариант. В PostGIS, к примеру, можно завести Geometry(Point, 4326) и Geometry(Point, 3857) для каждой точки сегмента. Если храним только WGS84, то при рендеринге будем проектировать на лету. Влияние на производительность минимально, если использовать простую проекцию, но можно предвычислить для ускорения.

**Хранение и доступ к данным:** В начальной реализации, объем данных – вся дорожная сеть крупного города (Алматы). Это десятки тысяч сегментов (например, если суммарная длина дорог ~2000 км, то ~200k сегментов по 10 м). Такого объёма вполне достаточно держать в оперативной памяти для быстрого доступа. Но для долговременного хранения и истории лучше использовать БД:

* **SQLite + SpatiaLite:** Локальная файл-БД, не требующая установки сервера. Может хранить геометрию (через SpatiaLite-плагин) и атрибуты. Поддерживает пространственные индексы R-Tree для быстрой выборки сегментов по области. Преимущество – простота развёртывания (файл БД рядом с приложением) ([SpatiaLite: SpatiaLite](https://www.gaia-gis.it/fossil/libspatialite/index" \l ":~:text=SQLite%20is%20intrinsically%20simple%20and,lightweight)) ([SpatiaLite: SpatiaLite](https://www.gaia-gis.it/fossil/libspatialite/index" \l ":~:text=SpatiaLite%20is%20smoothly%20integrated%20into,equivalent%20to%20PostgreSQL%20%2B%20PostGIS)). Можно сразу поставлять предзаполненный SQLite файл с сегментами дорог Алматы.
* **PostgreSQL + PostGIS:** Более тяжеловесное решение (нужен сервер), но актуально, если планируется расширение (например, собираем данные пробок централизованно и несколько клиентов подключаются). PostGIS даёт богатый SQL-интерфейс для запросов (например, выбрать все сегменты на определенной улице, или все сегменты внутри полигона района). Он надёжен для хранения истории на больших промежутках времени и несколькими клиентами ([Applications of PostGIS and PostgreSQL in Modern Geospatial Analysis](https://www.ve3.global/applications-of-postgis-and-postgresql-in-modern-geospatial-analysis/" \l ":~:text=PostGIS%20is%20an%20open,and%20PostgreSQL%20across%20different%20industries)). Если приложение одиночное и офлайн, PostgreSQL избыточен; но если в перспективе делать веб-сервис или общую базу пробок – стоит заложить возможность.
* **Формат файлов:** Shapefile можно продолжать использовать для хранения предобработанной сети, но обновлять shapefile неудобно, и хранить историю пробок в нём невозможно. Поэтому лучше конвертировать финальный SHP в **GeoPackage** – который позволит хранить и геометрии, и связанные таблицы (например, таблицу истории трафика) в одном файле. GeoPackage – открытый формат OGC, фактически SQLite с SpatiaLite внутри, поддерживается QGIS и др. ([What is a shapefile? .shp, .dbf and .shx • Max Dietrich](https://mxd.codes/articles/what-is-a-shapefile-shp-dbf-and-shx" \l ":~:text=Exploring%20Alternatives%20to%20Shapefiles)).

**Резюме по предобработке:** Подготовить **набор данных сегментов 10 м** с необходимыми атрибутами (улица, ID, координаты). Убедиться, что данные чистые: нет дублей, все сегменты упорядочены вдоль улицы, известны начала/концы (можно добавить координаты центра сегмента для удобства). Этот процесс делается однократно и затем приложение работает с готовыми сегментами.

## План обеспечения высокопроизводительной визуализации

Для **быстрой отрисовки** десятков тысяч отрезков дорог с обновлением каждую минуту необходимы оптимизации на нескольких уровнях: хранение, выборка и непосредственный рендеринг.

* **Аппаратное ускорение графики:** Использовать GPU для рисования. Например, Qt Quick автоматически рендерит элементы через OpenGL (или Vulkan/DirectX в новых версиях), разгружая CPU. Это позволит отрисовать большое число сегментов (линий) с цветовой заливкой быстро. В отличие от традиционного QPainter на QWidget (который на CPU), Qt Quick Scene Graph объединяет примитивы и рисует их пачками на GPU. При правильной организации сцены можно добиться 60 FPS при панорамировании/масштабировании карты. **Вариант:** реализовать собственный рендеринг через OpenGL: загрузить координаты всех сегментов в вершинный буфер и использовать шейдер для отрисовки линий нужного цвета. Тогда обновление пробок сведётся к изменению цвета (можно передавать массив цветов в шейдере, или перекрашивать вершины) – это очень быстро, т.к. геометрия не меняется, а меняются лишь атрибуты. Однако это более сложный путь; Qt Quick по сути делает нечто подобное под капотом.
* **Пространственный индекс и фильтрация:** Нет необходимости всегда отрисовывать все сегменты – достаточно те, что видимы в текущем окне карты. Реализуем **фильтрацию по области просмотра (viewport)**. Каждое перерисовывание должно запрашивать у модели только сегменты, попадающие в текущий экранный bounding box. Для этого:
  + Предварительно построить **R-Tree индекс** по координатам сегментов. В SQLite/SpatiaLite такой индекс можно создать для геометрии; в C++ можно использовать Boost.Geometry R-tree, хранящий минимальные прямоугольники сегментов.
  + При изменении области (пан, зум) выполнять запрос: выбрать все сегменты, чьи bounding box пересекает экранный bounding box. Благодаря индексу, это очень быстрый запрос (логарифмическая зависимость).
  + Отбрасывать невидимые сегменты уменьшает нагрузку на рендер: например, на крупном масштабе в окне окажется лишь несколько тысяч сегментов вместо десятков тысяч по всему городу.
* **Упрощение геометрии на малых масштабах:** Если пользователь сильно отдалил карту (видит весь город), каждый 10-метровый сегмент может быть неразличим. Можно в таких случаях объединять сегменты в более крупные (генерализация). Например, на уровне города отображать дорогу целиком одним отрезком окрашенным в худший из её сегментов. Это уменьшит количество объектов для отрисовки, повысив скорость. Такая генерализация может быть реализована путем хранения параллельно упрощённого слоя (например, второй слой полилиний – каждая улица цельная). Тогда рендерер в зависимости от масштаба выбирает, рисовать ли подробные сегменты или обобщённые. Однако даже без этого, GPU справится с отрисовкой многих тысяч коротких линий.
* **Оптимизация отрисовки линий:** Для повышения FPS можно настроить параметры рисования:
  + Отрисовывать сегменты как **GL\_LINES** (линии без толщины в пикселях) – они очень быстры. Но возможно нужно рисовать линии с толщиной, чтобы на экране были заметны. Толщина 2-3 пикселя и сглаживание (anti-aliasing) улучшат видимость сегментов.
  + **Группировка отрисовки по цвету:** сначала нарисовать все зелёные, потом жёлтые, потом красные сегменты. Это минимизирует смену состояния рендеринга (переключение цвета пера).
  + Отключить лишнее: тени, градиенты нам не нужны для линий дорог – достаточно сплошного цвета.
  + Использовать **вершинные буферы:** загрузить координаты линий один раз. Например, создать массив вершин, где каждые 2 вершины – концы одного сегмента. Для рисования обновлять только массив цветов или униформу шейдера. Если используем Qt Quick, можно написать кастомный QQuickItem, который в методе updatePaintNode обновляет QSGGeometry (набор вершин) и QSGMaterial (цвета).
* **Обновление раз в минуту:** Частота 1 мин означает, что нет жёстких требований к анимации изменений. Мы можем просто по таймеру (QTimer) раз в 60 секунд перезагружать цвета сегментов и вызывать update(). Пользователь в это время может свободно двигать карту. Если же понадобится плавно анимировать изменение цвета (например, постепенный переход зелёного в красный), это можно реализовать через Qt’s PropertyAnimation, но необходимости особой нет. Главное – чтобы момент обновления не подвисал UI. Поэтому обновление цветов стоит делать либо в фоновом потоке, либо разбить на шаги (например, обновлять сегменты по частям). Однако 200k сегментов обновить – не проблема для C++ за доли секунды, а передача на GPU 200k вершин – тоже порядка миллисекунд, что на 60 сек вполне допустимо.
* **Тестирование производительности:** Следует профилировать приложение на разных масштабах. В узком масштабе нагрузка на CPU от выборки (тысячи сегментов) мала, на широком масштабе нагрузка на GPU (отрисовать десятки тысяч тонких линий) тоже должна укладываться в кадр. Если выявляются узкие места, можно переключиться на ещё более низкоуровневый рендеринг. Но ожидается, что связка C++/Qt Quick + OpenGL справится с задачей. Для примера, движок Qt Quick легко тянет размытие на GPU в реальном времени ([Qt Quick and Blurred Panels](https://www.qt.io/blog/qt-quick-and-blurred-panels" \l ":~:text=The%20main%20conclusion%20is%20that,be%20performant%20on%20modern%20GPUs)), а отрисовка простых линий куда менее ресурсна.

**Дополнительные моменты:**

* **Фоновая карта:** Если нужен фон (например, схема города или спутниковый снимок), можно подложить тайловую карту OSM. Но это потребует загрузки тайлов и повысит нагрузку. В минимальной версии фон не обязателен – дороги сами дают карту. В перспективе можно использовать библиотеку Qt Location с плагином Mapbox/OSM для фоновой карты, а поверх рисовать наши сегменты как overlay.
* **Взаимодействие с пользователем:** При наведении/клике на сегмент можно его подсвечивать или показывать информацию (например, улица, текущая скорость). Это означает, что нам нужно уметь быстро определять, на какой сегмент кликнули. С Spatial Index это просто: берем координату клика (в гео-пространстве), находим ближайший сегмент. Или в Qt можно использовать QGraphicsItem с установленной геометрией сегмента и включённым флагом ItemContainsShape, но создание 200k QGraphicsItem нецелесообразно. Лучше свой поиск с использованием индекса.
* **Использование нескольких потоков:** Рендеринг в Qt Quick выполняется в отдельном OpenGL-потоке, UI – в главном. Это хорошо, т.к. тяжёлый отрисовочный код не тормозит отклик интерфейса. Можно также вынести выборку сегментов из БД в поток. Однако нужно следить за потокобезопасностью доступа к данным (например, копировать нужные данные перед передачей в поток отрисовки).

Итогом, план визуализации такой: заранее загружаем геометрию всех сегментов (например, в массив или в GPU-буфер). При изменении вида фильтруем и рендерим их. При обновлении трафика – меняем цвета и перерисовываем. С современным железом это позволит иметь интерактивную карту пробок без лагов.

## Концепт интерфейса (дизайн с эффектом glassmorphism и мультиязычность)

**Дизайн интерфейса:** Приложение должно выглядеть современно и аккуратно, вписываться в стилистику glassmorphism. Это означает использование полупрозрачных размытых панелей, сглаженных углов, а также динамических эффектов при взаимодействии. Основное окно – карта, занимающая большую часть экрана. Поверх карты могут располагаться плавающие панели настроек и информации.

([Qt Quick and Blurred Panels](https://www.qt.io/blog/qt-quick-and-blurred-panels)) Пример: интерфейс с размытыми полупрозрачными панелями. В нашем приложении подобные панели могут использоваться для отображения легенды, фильтров или меню поверх карты, сохраняя эффект "стекла" (фон карты просвечивает размыто) ([Qt Quick and Blurred Panels](https://www.qt.io/blog/qt-quick-and-blurred-panels" \l ":~:text=The%20main%20conclusion%20is%20that,be%20performant%20on%20modern%20GPUs)). Благодаря возможностям Qt Quick MultiEffect, добиться такого blur-эффекта можно напрямую средствами Qt, который выполняется на GPU и не сильно влияет на производительность даже при анимациях.

Конкретное расположение и элементы интерфейса:

* **Верхняя панель/Заголовок:** Может быть нестандартным – например, прозрачным заголовком с названием приложения, кнопками свернуть/закрыть, переключателями языка. При желании можно сделать окно **безрамочным** и стилизовать самостоятельно (Qt позволяет убирать стандартную рамку окна и обрабатывать перемещение/resize вручную). Тогда можно придать окну закруглённые углы и эффект акрила на Windows 10/11. Однако это потребует нативного кода для реализации системного блюра (Windows Composition API). Проще – имитировать glassmorphism внутри окна, как в примере выше.
* **Панель инструментов (Toolbox):** Например, слева или сверху может быть кнопка открытия меню фильтров, кнопка выбора источника пробок (если надо переключаться между 2ГИС/Яндекс вручную), кнопка обновления вручную и переключатель темной/светлой темы. Эти элементы можно сделать иконками поверх полупрозрачного фона.
* **Окно фильтров/настроек:** При нажатии, выезжает боковая панель (drawer) или появляется модальное окно с настройками. Здесь пользователь может фильтровать отображение:
  + Выбор улицы или набора улиц для отображения (все остальные скрыть). Можно сделать выпадающий список улиц или поле поиска по названию – тогда на карте останутся только сегменты выбранных улиц.
  + Фильтр по уровню пробок: например, показывать только заторы (красные) чтобы сразу видеть проблемные места, или наоборот, убрать зеленые (несущественно свободные).
  + Переключение отображения: “только выбранные участки” (возможно, пользователь сможет сам ткнуть два конца отрезка, и отобразятся только сегменты между ними – хотя это сложнее), или “отображать улицы целиком” (можно реализовать режим, когда выделяется любая улица, и вся она подсвечивается, невзирая на сегменты – для общего представления).
  + Настройки внешнего вида: ползунки для прозрачности панелей, включение/выключение размытия (если на слабом ПК – можно отключить blur), выбор цветовой схемы (разные оттенки для пробок, например, некоторые предпочитают красно-черную схему для дальтоников).
* **Легенда:** Небольшая плавающая карточка, объясняющая цвета (зелёный = свободно, жёлтый = затруднено, красный = пробка). Она тоже оформляется в стиле полупрозрачной подсказки, возможно с иконками.
* **Масштабирование и навигация:** Справа может быть стандартный виджет масштабирования (+/−). Панорамирование – перетаскиванием карты мышью (реализуется отслеживанием событий мыши на виджете карты). Также можно поддержать масштаб колесом мыши или мультитач жесты (Qt это поддерживает из коробки).
* **Мультиязычность UI:** Интерфейс должен поддерживать казахский, русский, английский. В Qt это достигается использованием механизма переводов: все текстовые строки оборачиваются в tr()/qsTr() и внешние \*.ts файлы содержат переводы ([Make your application multilingual | System on Module](https://www.e-consystems.com/blog/system-on-module-som/make-your-application-multilingual/" \l ":~:text=This%20paper%20describes%20how%20to,lingual%20dialog%20box)). Мы подготовим файлы переводов для трех языков. Пользователь может переключать язык динамически (через меню настроек "Язык: [Русский | Қазақша | English]"), при этом с помощью QTranslator загружается соответствующий .qm-файл, и все видимые строки обновляются на выбранный язык ([Make your application multilingual | System on Module](https://www.e-consystems.com/blog/system-on-module-som/make-your-application-multilingual/" \l ":~:text=Introduction)). Важно учесть поддержку казахского алфавита (Qt с этим проблем не имеет). Также возможно автопереключение – при старте приложения определить язык системы и установить его по умолчанию.
* **Локализация данных:** Названия улиц в Алматы есть на русском и казахском. Если в исходном SHP присутствуют оба (часто в атрибутах могут быть Name\_ru и Name\_kz), можно в интерфейсе отображать название улицы согласно выбранному языку. Однако это усложнит данные. Проще – пока отображать названия как есть (возможно, на русском или латиницей). Но для полного мультиязычного опыта, можно подключить справочник улиц двуязычный. Это можно реализовать в будущем: иметь таблицу соответствий или использовать OpenStreetMap данные, где у улиц есть name:kk и name:ru.

**Стилистика:** В цветовой гамме приложения преобладают нейтральные тона для фона (например, темно-синий/серый полупрозрачный фон панелей), чтобы основная цветовая информация (пробки) контрастно выделялась. Эффект glassmorphism означает наличие **размытия заднего фона** под полупрозрачными элементами. Это особенно хорошо для отображения поверх пестрой карты, т.к. текст на панелях останется читаемым. Qt Quick, как уже упоминалось, дает инструменты для реализации размытия (ShaderEffect, MultiEffect). Например, можно применять **Gaussian Blur** к фону панели при отрисовке ([Qt Quick and Blurred Panels](https://www.qt.io/blog/qt-quick-and-blurred-panels" \l ":~:text=The%20main%20conclusion%20is%20that,be%20performant%20on%20modern%20GPUs)). Нужно лишь быть осторожным с обновлением: размывать фон можно один раз и кэшировать текстуру, чтобы не делать это каждый кадр.

**Иконки и шрифты:** Используем векторные иконки (например, набор Material Icons или свои SVG), чтобы они красиво выглядели на любом DPI. Поддерживаем HiDPI экраны. Шрифт интерфейса – нейтральный, поддерживающий казахские и русские символы (например, Open Sans, Roboto или системный Segoe UI/Ubuntu).

**Пример сценария использования интерфейса:** Пользователь запускает приложение – видит карту Алматы с наложением пробок в реальном времени. В верхнем углу – индикатор «Источник: 2ГИС (обновлено 15:30)». Справа – легенда цветов. Пользователь может приблизить колесом к своему району, навести на улицу – всплывет подсказка «пр. Абая – Затор, скорость ~5 км/ч». Через меню он переключает язык на английский – все надписи меняются (улицы остаются транслитерированы или на английском, если есть перевод). Вечером пользователь открывает панель фильтра и выбирает показать только пробки балла 4 и выше – карта отображает только красные сегменты, позволяя оценить самые загруженные улицы.

## План расширения и обновлений

Заложенная архитектура и стек рассчитаны на будущие улучшения. Вот **планируемые направления развития** приложения:

* **Поддержка новых регионов:** Изначально ориентируемся на Алматы, но решение легко масштабируется на другие города. Нужно предусмотреть загрузку другого SHP (или подключение к базе с данными нужного города). Архитектурно для этого достаточно, чтобы модуль данных мог переключаться на другой набор сегментов. В UI можно добавить выбор города (и соответствующего источника пробок). Хранение нескольких городов можно организовать в одной БД, разделяя сегменты по городу.
* **Больше источников данных:** Добавление **народных данных о пробках**. Например, интеграция с API Яндекс.Пробок, если станет возможным, либо с **Waze** (у них есть программа Connected Citizens, предоставляющая данные городским службам). Это позволит получать более точные данные и информацию о происшествиях (ДТП, перекрытия). Архитектура plugin-like для источников уже позволит добавить, скажем, WazeTrafficSource без изменения остального кода.
* **События на дорогах:** Помимо заторов, важно отображать сопутствующую информацию: **ДТП, дорожные работы, перекрытия**. Многие карты (Яндекс, Google) помечают значками эти события. В перспективе можно парсить и их (например, определять значок и геолокацию события через компьютерное зрение или публичные RSS-ленты дорожных служб). Приложение могло бы показывать значок на карте и учитывать событие в оценке достоверности пробки (например, если пробка красная и на сегменте отмечено ДТП – это объясняет затор).
* **Аналитика и история:** Собирая данные каждую минуту и сохраняя их, через некоторое время получится ценная база исторических пробок. Планируется реализовать **режим воспроизведения истории**: пользователь выбирает дату и время (или диапазон), и карта отображает состояние пробок на тот момент. Это полезно для анализа (например, сравнить загруженность утром и вечером, либо посмотреть влияние праздников/карантинов и т.д.). Технически, для этого нужно сохранять историю в БД (например, таблица traffic\_history с полями segment\_id, timestamp, level). UI: слайдер времени или календарь для выбора.
* **Прогнозирование пробок:** На основе накопленной истории можно внедрить простейший прогноз – например, график ожидаемого трафика на ближайший час для выбранной улицы (с учётом типичного суточного цикла). Либо использовать модели машинного обучения для предсказания пробок (это отдельная сложная задача, но с историческими данными реально). Архитектура хранения данных и модульная структура позволит добавить сервис прогноза как ещё один модуль, который будет выдавать “прогнозируемые цвета” и отображаться, скажем, пунктирной линией.
* **Оптимизация парсинга:** В будущем, чтобы снизить зависимость от скриншотов, можно переходить на прямое получение данных. Например, если 2ГИС или другие предоставят открытое API трафика (пусть даже неофициально), переписать источник на использование API, что значительно уменьшит задержки и ресурсы. Также, можно распараллелить сбор данных – напр., брать несколько мелких скриншотов разных районов одновременно, чтобы повысить разрешение без увеличения времени.
* **Выход на мобильные платформы:** Заложенный стек Qt и/или Slint теоретически позволяет собрать приложение под Android/iOS. Это естественное развитие – сделать мобильное приложение "Пробки Алматы офлайн". Нужно будет адаптировать UI под сенсорное управление и маленький экран (упростить интерфейс, использовать нижние навигационные панели в стиле мобильных гайдлайнов). Придётся учесть энергопотребление (минутные обновления в фоне) и возможные ограничения мобильных ОС на фоновые задачи.
* **Интеграция с навигацией:** Еще один вектор – добавление функций навигатора. Например, построение маршрутов с учётом пробок. Это потребует иметь граф дорог с весами (скорость по сегменту на данный момент). С помощью алгоритма Dijkstra/A\* пользователь мог бы проложить путь через город и приложение подсветит маршрут с расчетным временем. Эта функция усложнит систему (нужен роутинг-движок, граф связности), но вписывается в общую архитектуру (граф можно построить из тех же сегментов, храня индексы соседей). База пробок по сегментам уже даёт веса для графа. Это можно запланировать на будущее, возможно используя библиотеку маршрутизации (например, OSRM, GraphHopper – но они Java, либо свой простой).
* **Обновления технологического стека:** Следует отслеживать выход новых версий Qt, библиотек, обновлять приложение для совместимости. Например, Qt 7 может принести улучшения в рендеринге или новые компоненты. Rust тоже развивается – возможно появятся ещё более удобные GUI-фреймворки. Благодаря модульности, можно будет частично переписать компоненты на новые технологии без полного рефакторинга. Например, если спустя время решат, что полностью перейти на Rust + графическая библиотека (вместо C++/Qt), то можно реализовать отображение через Skia/Vulkan, сохранив логику парсинга и базу.
* **Сообщество и открытость:** Если проект будет открытым, принимать вклад от сообщества: новые языки интерфейса (например, добавить поддержка узбекского или китайского, если актуально), новые города, отчёты об ошибках. Организация кода по модулям и чистая архитектура поможет сторонним разработчикам быстрее разобраться.

Таким образом, начав с конкретной задачи (карта пробок Алматы офлайн, обновляемая раз в минуту), мы строим прочный фундамент, который позволит эволюционировать продукт в полноценную мультиплатформенную систему мониторинга трафика и навигации, не зависящую от проприетарных сервисов.