МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное
учреждение высшего образования
«Южный федеральный университет»

Институт математики, механики
и компьютерных наук им. И. И. Воровича

Денисов Илия Игоревич

РАЗРАБОТКА КРОСС-ПЛАТФОРМЕННОЙ БИБЛИОТЕКИ ДЛЯ АНАЛИЗА ФИНАНСОВЫХ ДАННЫХ

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
по направлению подготовки
02.04.02 – Фундаментальная информатика и информационные технологии,
направленность программы
«Разработка мобильных приложений и компьютерных игр»

Научный руководитель –
доц., к. ф.-м. н. Шабас Ирина Николаевна

Рецензент –
ст. преп. каф. ПМП Пучкин Максим Валентинович

Допущено к защите:
руководитель
образовательной программы _____ Демяненко Я. М.

Ростов-на-Дону – 2025

Оглавление

Введение

Анализ финансовых данных всегда представлял предмет повышенного интереса. Эпоха интернета принесла новые возможности в этой области – теперь каждый, имеющий доступ к глобальный сети, может получать актуальные данные бирж, анализировать их и практически моментально принимать решение о покупке или продаже различных активов на этих биржах и рынках. Существует богатый спектр приложений, предоставляющих возможность аналитики финансовых данных с помощью самых разнообразных инструментов, однако лишь малая их часть предоставляет эти инструменты в качестве открытого исходного кода с возможностью дальнейшей интеграции в другие системы в качестве библиотеки.

Данная работа посвящена разработке библиотеки визуальных инструментов анализа финансовых данных. В работе анализируются существующие кроссплатформенные решения и их недостатки, рассматривается архитектура и реализация библиотеки визуальных инструментов для анализа финансовых данных, разработанной на языке TypeScript с использованием библиотеки Lightweight Charts и графического интерфейса Canvas API.

Основу библиотеки составляют инструменты рисования на финансовых графиках, которые позволяют отмечать тренды и паттерны, проводить измерения и прогнозирование, рассчитывать уровни цен. Библиотека предоставляет возможность добавлять на финансовый график более 10 инструментов рисования, а также более 5 индикаторов.

1. Обзор предметной области

В данной главе рассматриваются предметная область аналитики финансовых данных и терминология, основные инструменты для анализа данных, описываются существующие библиотеки с инструментами анализа финансовых данных.

1.1. Аналитика финансовых данных

В биржевой торговле перед трейдером стоит задача поиска закономерностей, краткосрочных и долгосрочных трендов, прогнозирования движения цены, для выбора подходящего момента продажи или покупки актива.

Существующие торговые платформы предлагают трейдеру широкие возможности для исследования поведения актива. В их число входит:

1. Визуальный анализ с помощью добавления на график актива инструментов рисования для выявления паттернов.
2. Визуальный анализ с использованием технических индикаторов.
3. Создание и тестирование торговых стратегий на исторических данных актива с целью выявления оптимальных условий для покупки или продажи актива в будущем.

В последующих главах рассматриваются приведённые выше типы инструментов.

1.1.1. Паттерны технического анализа графика финансовых данных

В финансовом анализе паттерном (от англ. pattern — модель, образец) называют устойчивые повторяющиеся сочетания данных цены, объёма или

индикаторов. Анализ паттернов основывается на одной из аксиом технического анализа: «история повторяется» — считается, что повторяющиеся комбинации данных приводят к аналогичному результату [1].

Паттерны можно разделить на три основных категории:

- Неопределённые (могут вести и к продолжению, и к смене текущего тренда).
- Паттерны продолжения текущего тренда.
- Паттерны смены существующего тренда.

Паттерны определяются визуально на графике. Их обнаружению помогает использование различных инструментов рисования, специализированных под задачи обнаружения паттернов.

Ниже на приведён пример одного из паттернов «Бычий флаг» (от англ. Bullish flag) (рис. 1):

Рис. 1. Паттерн «Бычий флаг»

График на рис. 1 ограничен двумя отрезками, которые называются линия тренда (от англ. Trend line). Согласно теории паттернов, график цены актива, вошедший в состояние колебания между двумя параллельными отрезками из состояния роста/падения, наиболее вероятно продолжится в

направлении роста/падения соответственно. Таким образом, на основе этого паттерна трейдер может получить прогноз о дальнейшей динамике цены.

На рис. 2 представлены основные паттерны в техническом анализе:

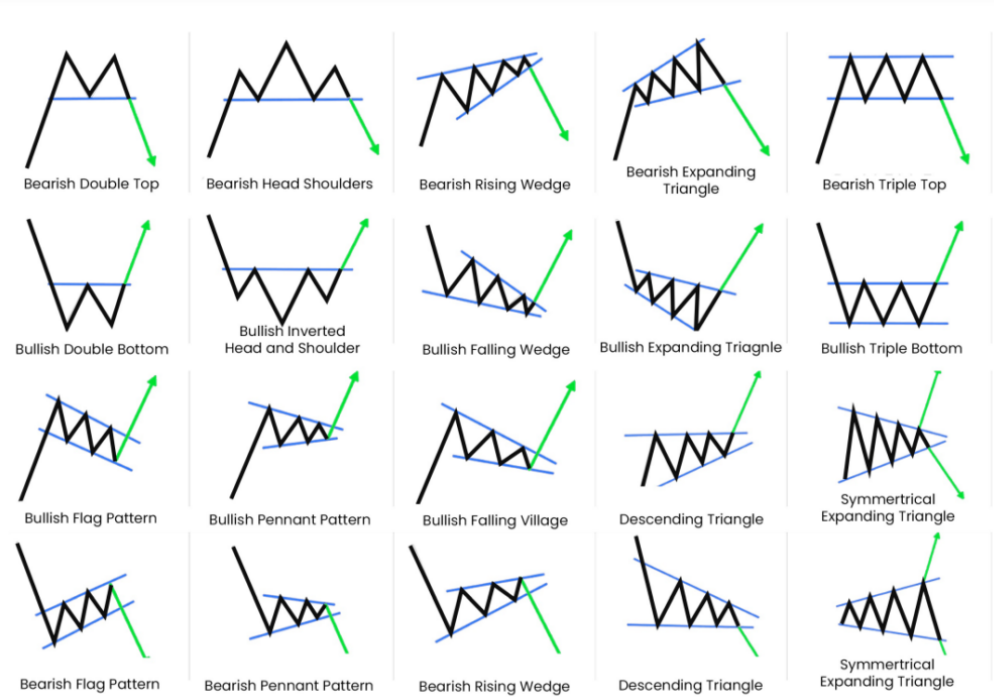

Рис. 2. Основные паттерны технического анализа

Прикладная значимость паттернов в теории технического анализа определяется эмпирически.

1.1.2. Инструменты рисования для выявления паттернов

Как можно заметить, на рис. 2 для идентификации паттернов используются такие инструменты рисования, как линии тренда. Среди трейдеров популярны и многие другие инструменты рисования:

- Линии:
  - Луч
  - Горизонтальная линия
  - Вертикальная линия

    - Параллельный канал
    - Тренд регрессии
    - Виллы
- Инструменты Фибоначчи
    - Каналы Фибоначчи
    - Коррекция Фибоначчи
    - Клин Фибоначчи
    - Спираль Фибоначчи
    - Временные периоды Фибоначчи
- Инструменты Ганна
    - Коробка Ганна
    - Веер Ганна
- Паттерны
    - Паттерн XABCD
    - Паттерн ABCD
    - Паттерн «Голова и плечи»,
    - Паттерн «Треугольник».
- Фигуры
    - Прямоугольник
    - Треугольник

- Ломаная линия
- Дуга
- Эллипс
- Кривая

Инструменты рисования добавляются на чарт таким образом, чтобы точки, по которым они строятся, находились на значениях цены актива. На рис. 3 представлены добавленные инструменты рисования на графике в трейдинговой платформе TradingView [2] (слева-направо: прямоугольник, ломаная линия, кривая, коррекция Фибоначчи, клин Фибоначчи):

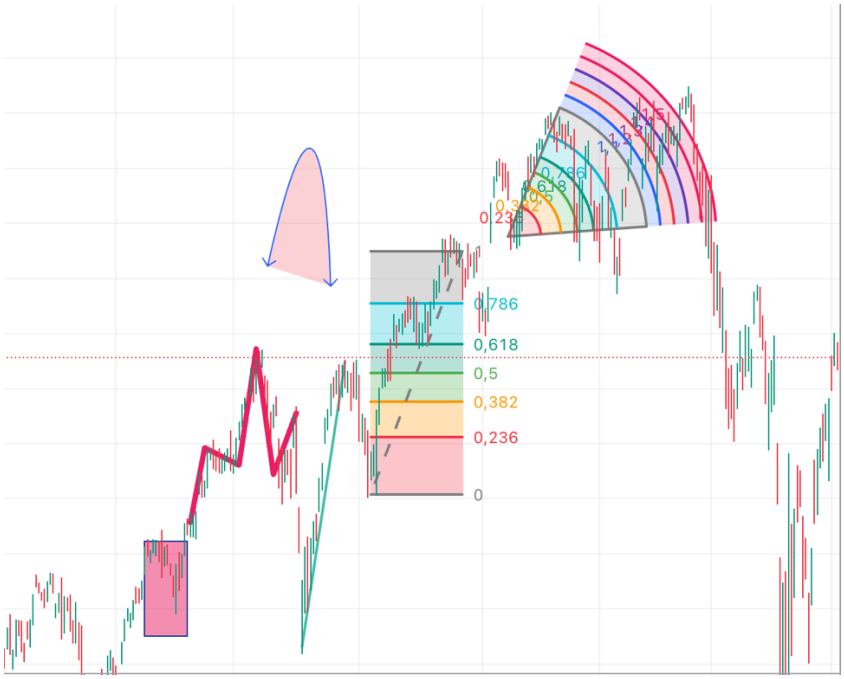

Рис. 3. Инструменты рисования

Данные инструменты позволяют редактировать свой стиль, местоположение точек на графике, форму и другие параметры при их наличии.

1.1.3. Технические индикаторы

В финансовом анализе технический индикатор — это математический расчёт, основанный на исторической цене, объёме или (в случае фьючерсных

контрактов) информации об открытом интересе, целью которого является прогнозирование направления финансового рынка [2].

Технические индикаторы являются важной частью анализа финансовых данных и обычно визуально отображаются на графике рядом с основным графиком цены актива, предоставляя трейдеру информация о возможном направлении тренда в будущем, объёме текущей торговле, а также о многих другие важных показателях (рис. 4).

Рис. 4. График, на котором совмещены графики цены с графиком стохастического осциллятора.

Существует два основных типа технических индикаторов [3]:

- Индикаторы наложения, которые используют тот же масштаб, что и цены, наносятся поверх цен на графике актива. Примерами являются скользящие средние и полосы Боллинджера.
- Осцилляторы, которые колеблются между локальным минимумом и максимумом, наносятся выше или ниже графика цен. Примерами

являются стохастический осциллятор (рис. 4) или RSI (RSI от англ. relative strength index — индекс относительной силы).

Наиболее популярными индикаторами являются:

- RSI.
- MACD (MACD от англ. moving average convergence divergence — схождение/расхождение скользящих средних).
- Линии Боллинджера.
- Уровни Фибоначчи.
- Средний истинный диапазон.
- Стохастический осциллятор.
- Индикатор Ишимоку.
- Профиль объёма.

1.1.4. Торговые стратегии

Многие современные приложения для анализы финансовых данных предлагают также возможность создавать алгоритмические стратегии торговли. Это означает, что трейдер может в виде скриптового кода описать, при каких условиях нужно совершать покупки/продажи активов. В данной работе не описывается детали имплементации таких скриптовых систем, однако необходимо заметить, что подбор параметров для вышеупомянутых скриптовых программ автоматической торговли, выполняемый не в реальном времени с реальными сделками на бирже, а симулируя сделки на исторических данных, можно смело отнести к одному из способов анализа финансовых данных. На листинге 1 приведён пример кода для автоматической торговли на языке PineScript [4]:

Листинг 1

```
strategy("MA Strategy", overlay=true)
ma = ta.sma(close, 10)
plot(ma)
```

```
strategy.entry("Buy", strategy.long, when=close > ma)
strategy.close("Buy", when=close < ma)
```

1.2. Библиотеки для анализа финансовых данных

В данной главе описываются существующие библиотеки с инструментами анализа финансовых данных.

В результате исследования были выделены несколько библиотек:

1. Lightweight Charts [5]

Библиотека от компании TradingView, предоставляющая API для создания графиков и добавления на сторонних данных для заполнения. Библиотека реализована на языках Typescript. Имеет открытый исходный код и распространяется по лицензии Apache 2.0, что позволяет использовать ее в любой сфере при условии упоминания ее происхождения. Данная библиотека не предоставляет инструментов анализа, однако может послужить основной для создания подобных инструментов.

2. Plotty [6]

Данная библиотека имеет сразу на нескольких языках программирования: Python, R, JavaScript, Julia, MATLAB. Plotty имеет широкий функционал и поддерживает отрисовку финансовых графиков, нескольких видов японских свечей, диаграмм, некоторых специфических индикаторов, Библиотека распространяется по лицензии MIT, что позволяет свободно ее свободно модифицировать и переиспользовать в любых приложениях. К недостаткам библиотеки можно отнести недостаточную стилизируемость графика, отсутствие API для рисования на некоторых частях графика, таких как оси абсцисс

и ординат, язык библиотеки – JavaScript, который позволяет легче допускать ошибки в виду, в частности, отсутствия статической типизации, что является недостатком в сравнении с, например, TypeScript.

3. Go-chart [7]

Библиотека Go-Chart реализована на языке Go и предоставляет API для рисования графиков и некоторых графических примитивов. Результат рисования сохраняется в формате SVG, что является неоптимальным в ситуации постоянного обновления графика. Более того, библиотека не поддерживается с 2024 года, что крайне снижает ее стабильность в будущем. Как и библиотека Plotty, Go-Chart распространяется по лицензии MIT.

4. TA-Lib [8]

TA-Lib – библиотека для технического анализа без графического интерфейса, имплементированная на C++ и Python. Она реализует более 200 функций расчёта различных индикаторов. Распространяется по лицензии BSD. К недостаткам библиотеки можно отнести необходимость интеграции с другим приложением, реализующим визуализацию. С точки зрения производительности лучше всего было бы реализовать такое приложение так же на C++, однако стоит сказать, что данная технология имеет высокий порог вхождения и время разработки на ней порой кратно выше, чем у других технологий.

1.3. Анализ технологий для разработки библиотеки анализа финансовых данных

Для анализа финансовых данных необходимо выбрать такой набор технологий разработки, благодаря которому можно разработать

кроссплатформенную, легко интегрируемую и современную библиотеку. При анализе существующих библиотек было выяснено, что одними из самых востребованных технологий сегодня являются веб-технологии. Использование веб-приложений не требует установки и доступно из любого современного браузера. К тому же, разработка, отладка и тестирование таких приложений выполняется быстрее и проще. Однако одним из главных существенных недостатков веб-технологий является их быстродействие, так как используемые тут языки – интерпретируемые. Очевидно, что они всегда будут уступать по быстродействию компилируемым языкам.

При исследовании библиотек, описанных выше, было установлено, что практически ни одна из них не предоставляет реализованных инструментов рисования для выявления паттернов технического анализа, описанных в главе 1.1.1. Поэтому было принято решение использовать библиотеку Lightweight Charts [5] для рендеринга графика актива и на ее базе реализовать основные инструменты рисования и некоторые технические индикаторы. Данная библиотека имеет гибкое API, визуально привлекательна, имеет вес всего 35 килобайт. Следствием такого выбора является использование веб-технологий, а именно языков TypeScript и технологии Vue для создания демонстрационного приложения.

В связи с выбором библиотеки Lightweight Charts для рендеринга графиков, было бы разумно использовать те же средства для отрисовки примитивов. В данном случае речь идёт о Canvas API [9]. Это хорошо зарекомендовавшая себя технология, которая позволяет отрисовывать разнообразные двумерные примитивы. Более того, одним из ее преимуществ можно назвать поддержку аппаратного ускорения, что является немаловажным при наличии необходимости отрисовки сразу многих графических примитивов на графике.

В качестве системы контроля версий был выбран Git, а в качестве среда разработки – Visual Studio Code.

TypeScript –язык программирования, являющийся расширением языка JavaScript. Он привносит поддержку типов и вместе с этим позволяет исключить многие ошибки, которые связаны с использованием неправильных типов, ещё на этапе разработки и компиляции. Также благодаря поддержке типов, код, написанный на TypeScript, может анализироваться средой разработки, автоматически генерировать документацию и давать подсказки.

Vue — это прогрессивный JavaScript-фреймворк с открытым исходным кодом, предназначенный для построения пользовательских интерфейсов и одностраничных приложений (SPA). Vue ориентирован на плавную адаптацию, позволяя использовать его как библиотеку для создания отдельных виджетов, так и как полноценный фреймворк с широким набором инструментов.

2. Разработка библиотеки

2.1. Проектирование архитектуры библиотеки графических инструментов для анализа финансовых данных

Исходя из проведённого исследования, был определено ряд технических требований к разрабатываемой библиотеке:

1. Библиотека должна быть реализована на языке TypeScript с использованием фреймворка Vue.
2. Библиотека будет зависеть от библиотеки Lightweight Charts и использовать для отрисовки примитивов Canvas API.
3. Библиотека должна предоставлять следующие важные инструменты для поиска паттернов технического анализа:

   Инструменты рисования:

   - Прямоугольник
   - Треугольник
   - Линия тренда
   - Горизонтальная линия
   - Вертикальная линия
   - Ломаная линия
   - Спираль Фибоначчи
   - Клин Фибоначчи
   - Коррекция Фибоначчи

- Кривая

Должна быть реализована возможность добавления инструментов рисования на график и их перетаскивания на другую часть графика в будущем при желании. Также при изменении масштаба графика, фигуры, нарисованные с помощью инструментов рисования, должны менять свой масштаб соответственно.

Стили инструментов должны быть заранее заданы и синхронизированы друг с другом.

Сериализация и десериализация на данном этапе не предусмотрены, однако архитектура библиотеки должна быть построена таким образом, чтобы такой функционал был возможен в будущем.

Технические индикаторы:

- Линии Боллинджера
- Скользящее среднее

Изменение параметров индикаторов аналогично изменению параметров инструментов рисования не предполагает изменения, так как для этого требуется дополнительная экстенсивная разработка UI, что не является основной темой данной работы.

4. Разработанная библиотека должна быть интегрирована в тестовое приложение и опубликована в интернете на платформе GitHub Pages.

2.2. Проектирование интерфейса

Разработанная библиотека предоставляет возможность создавать на графике актива новые фигуры с помощью инструментов рисования, а также добавлять на него технические индикаторы.

Так как основной своей зависимостью библиотека имеет другую библиотеку Lightweight Charts, отвечающую за отрисовку графика, то инструменты рисования необходимо спроектировать таким образом, чтобы они удовлетворяли некоторым архитектурным особенностям Lightweight Charts. В частности, есть два объекта интерфейса, которые используются для создания инструментов рисования и технических индикаторов:

Листинг 2. Интерфейсы графика и серии

```
interface IChartApiBase<HorzScaleItem = Time>;
interface ISeriesApi;
```

С помощью этих интерфейсов добавляемые в данной работе инструменты анализа:

1. Подписываются на события мыши, которые обрабатываются в объектах интерфейса $IChartApiBase$;
2. Подписываются на события обновления графика, которые вызываются при изменении масштаба, изменении временного или ценового диапазона на графике, обновлении цены в объектах интерфейса $IChartApiBase$;

Также классы инструментов должны реализовывать интерфейс примитива на графике $ISeriesPrimitveBase$ (Листинг 3), который определяет базовую функциональность и структуру примитива:

Листинг 3. Интерфейс графического примитива в Lightweight Charts

```
export interface ISeriesPrimitiveBase<TSeriesAttachedParameters = unknown> {
    updateAllViews?(): void;
    priceAxisViews?(): readonly ISeriesPrimitiveAxisView[];
    timeAxisViews?(): readonly ISeriesPrimitiveAxisView[];
    paneViews?(): readonly IPrimitivePaneView[];
    priceAxisPaneViews?(): readonly IPrimitivePaneView[];
    timeAxisPaneViews?(): readonly IPrimitivePaneView[];
```

```
    autoscaleInfo?(
        startTimePoint: Logical,
        endTimePoint: Logical
    ): AutoscaleInfo | null;

    attached?(param: TSeriesAttachedParameters): void;
    detached?(): void;

    hitTest?(x: number, y: number): PrimitiveHoveredItem | null;
}
```

Lightweight Charts вызывает следующие функции-геттеры (если они определены) для получения ссылок на определённые примитивом представления (от англ. Views - представления) для соответствующей части графика (в будущем будем называть часть графика – панелью):

- $paneViews$
- $priceAxisPaneViews$
- $timeAxisPaneViews$
- $priceAxisViews$
- $timeAxisViews$

Первые три представления позволяют рисовать на соответствующих панелях (панель основного графика, панель шкалы цен и панель времени) с использованием *CanvasRenderingContext2D* из Canvas API и должны реализовывать интерфейс $IPrimitivePaneView$ (Листинг 4):

Листинг 4. Интерфейс $IPrimitivePaneView$

```
export interface IPrimitivePaneView {
    zOrder?(): PrimitivePaneViewZOrder;
    renderer(): IPrimitivePaneRenderer | null;
}
```

Метод $zOrder$ должен вернуть (при наличии) то, где в порядке наложения [10] находится объект, соответствующий представлению (Листинг 5):

Листинг 5. Класс, описывающий нахождение объекта в порядке наложения

```
export type PrimitivePaneViewZOrder = 'bottom' | 'normal' | 'top';
```

Метод $renderer$ возвращает объект интерфейса $IPrimitivePaneRenderer$, вызывая метода $draw$ у которого, библиотека инициирует отрисовку примитива (Листинг 6):

Листинг 6. Интерфейс $IPrimitivePaneRenderer$

```
export interface IPrimitivePaneRenderer {
    draw(target: CanvasRenderingTarget2D): void;
}
```

Представления, возвращаемые методами $priceAxisViews$ и $timeAxisViews$, должны реализовывать интерфейс $ISeriesPrimitiveAxisView$ (Листинг 7) и использоваться для определения меток, которые будут отрисованы на соответствующих шкалах или панелях графика (Рис. 5).

Интерфейс $ISeriesPrimitiveAxisView$ можно использовать для определения метки на оси ординат или абсцисс. Этот интерфейс предоставляет несколько методов для определения внешнего вида и положения метки, например, метод $coordinate$, который должен возвращать желаемую координату для метки на оси. Он также определяет необязательные методы для установки фиксированной координаты ($fixedCoordinate$), текста ($text$), цвета текста ($textColor$), цвета фона ($backColor$) и видимости метки ($visible$, $tickVisible$).

Листинг 7. Интерфейс $ISeriesPrimitiveAxisView$

```
export interface ISeriesPrimitiveAxisView {
    coordinate(): number;
    fixedCoordinate?(): number | undefined;
    text(): string;
    textColor(): string;
    backColor(): string;
    visible?(): boolean;
    tickVisible?(): boolean;
}
```

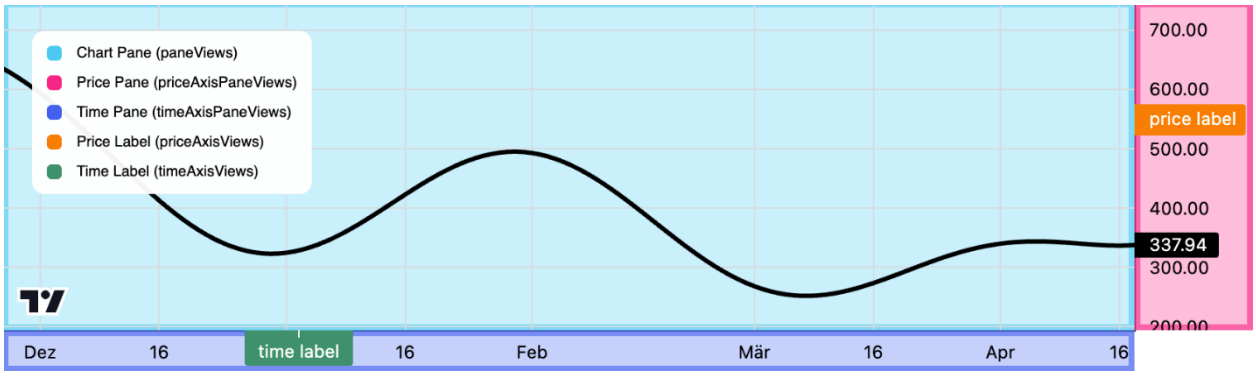

Рис.5. Голубым обозначена основная панель, синим и розовым панель ось абсцисс и ординат соответственно, зелёным и оранжевым обозначены метки на осях абсцисс и ординат соответственно

Раскрыв основные интерфейсы, использующиеся в нашей библиотеке из Lightweight Charts, можно перейти к описанию интерфейса рассматриваемой в данной работе библиотеки.

Для создания одного из инструментов рисования, перечисленного в главе 1.1.2., нужно создать следующие классы инструментов рисования и передать в их конструкторы объекты интерфейсов $IChartApiBase$, $ISeriesApi$, описанные выше:

- $RectangleDrawingTool$ – класс инструмента «Прямоугольник».
- $TriangleDrawingTool$ – класс инструмента «Треугольник».
- $FibChannelDrawingTool$ – класс инструмента «Коррекция Фибоначчи».
- $FibSpiralDrawingTool$ – класс инструмента «Спираль Фибоначчи»;

- $FibWedgeDrawingTool$ – класс инструмента «Клин Фибоначчи»;
- $CurveDrawingTool$ – класс инструмента «Кривая».
- $TrendLineDrawingTool$ – класс инструмента «Линия тренда»;
- $TimeLineDrawingTool$ – класс инструмента «Вертикальная прямая»;
- $PolylineDrawingTool$ – класс инструмента «Ломаная линия»;

Для создания одного из технических индикаторов, перечисленного в главе 1.1.2., нужно создать следующие классы технических индикаторов, передав эти классы после создания соответствующих классов в функцию $attachPrimitive$ у класса, реализующего интерфейс $ISeriesApi$:

- $SMAIndicator$ – индикатор «Скользящее среднее»;
- $BandsIndicator$ – индикатор «Линии Боллинджера»;

На листинге 8 приведён пример создания инструмента рисования и индикатора:

Листинг 8. Создание инструмента рисования $TriangleDrawingTool$ и индикатора $SMAIndicator$

```
const triangleDrawingTool = new TriangleDrawingTool(chart, series);

const smaIndicator = new SMAIndicator();
series.attachPrimitive(smaIndicator)
```

2.3. Реализация инструментов рисования

Любой инструмент рисования добавляет на график геометрические примитивы посредством позиционирования определённого количества точек на графике. Так, например, для добавления треугольника, необходимо добавить 3 точки, для линии тренда – 2 точки.

Также любой инструмент рисования имеет свой стиль, включающий набор цветов линий, цвета и прозрачность зарисовки некоторых зон геометрический линий, толщина линий и т. д.

Последней, но не менее важной составляющей любого инструмента рисования данной библиотеки является обработка событий мыши на графике, таких как:

- Клик мыши
- Зажатие мыши
- Снятие зажатия мыши
- Движение мыши

Учитывая вышеперечисленные общие черты всех инструментов рисования, можно выделить шаблонный тип для инструментов рисования $DrawingToolBase$. В качестве шаблонных параметров этот тип принимает:

- Класс соответствующего геометрического примитива $TDrawing$;
- Класс временного соответствующего геометрического примитива, который показывается пользователю лишь во время создания очередного примитива, $TPreviewDrawing$;
- Класс параметров инструмента рисования $TOptions$;

Внутри себя этот класс хранит временный массив точек $\_pointsCache$, массив уже созданных геометрический примитивов $\_drawings$, а также несколько обработчиков событий мыши: $\_clickHandler$, $\_dblClickHandler$, $\_moveHandler$.

Для геометрического примитива, соответствующего инструменту рисования был также выделен общий шаблонный тип $DrawingBase$ (Листинг 9).

Листинг 9. Шаблонный класс $DrawingBase$.

```
export class DrawingBase<DrawingOptions> extends ChartInstrumentBase {
  _options: DrawingOptions;
  _points: Point[];
  _bounds: DrawingBounds;

  constructor(
    points: Point[],
    defaultOptions: DrawingOptions,
    options: Partial<DrawingOptions> = {}
  ) {
      /* инициализация */
  }
  public addPoint(p: Point) {
      /* добавление новой точки */
  }
  public updatePoint(p: Point, index: number) {
      /* обновление точки с заданным индексом */
  }
  applyOptions(options: Partial<DrawingOptions>) {
      /* применение опций стиля */
  }
  hitTest(x: number, y: number): PrimitiveHoveredItem | null {
      /* реализации функции проверки коллизии */
  }
  protected _updateDrawingBounds(point: Point) {
      /* обновления границ графического примитива */
  }
}
```

Данный класс наследуется от общего для всех инструментов в данной библиотеке класса $ChartInstrumentBase$, который в свою очередь является небольшой обёрткой над вышеописанным интерфейсом $ISeriesPrimitiveBase$.

Необходимо отметить наличие ограничивающего прямоугольника $\_bounds: DrawingBounds$. Его необходимость основана в желании отрисовывать на панелях времени и цены прямоугольное пространство,

демонстрирующее границы, в которых содержится геометрический примитив (Рис. 6).

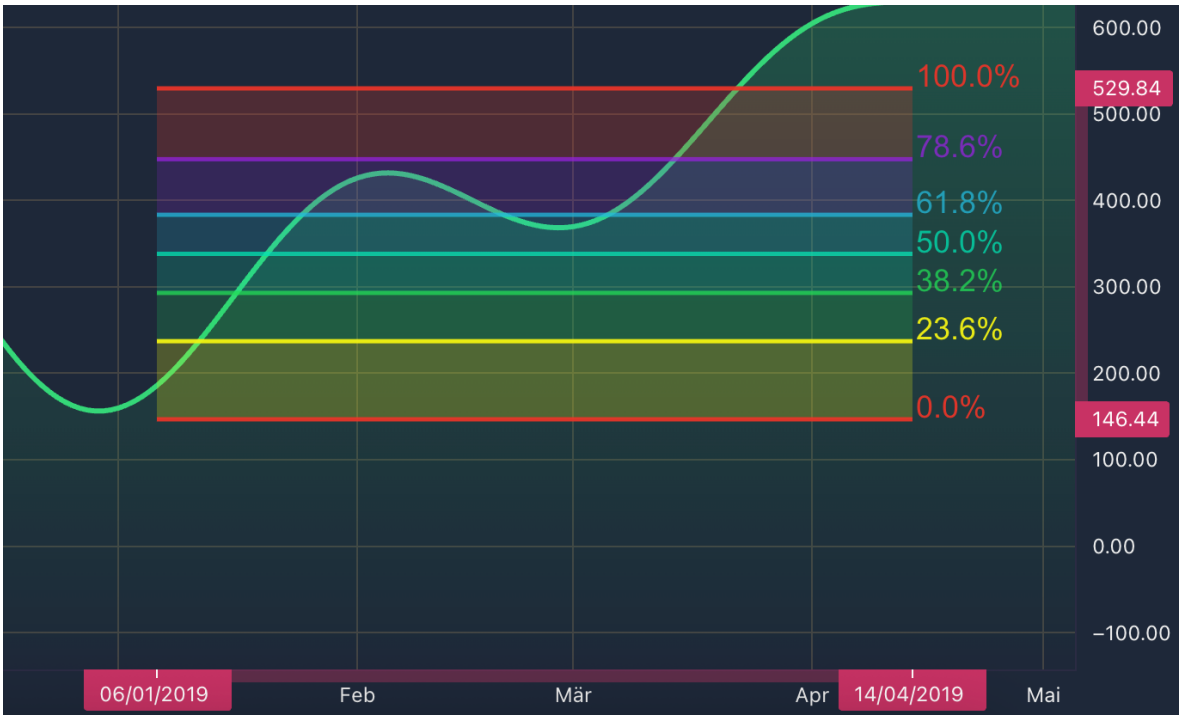

Рис. 6. На панелях цены и времени можно заметить розовые прямоугольные, показывающие границы области, в которой расположен геометрический примитив.

Каждый инструмент рисования представляет собой конечный автомат с несколькими состояниями (рис. 7):

1. IDLE – неактивное состояние.
2. ADD – добавление нового геометрического примитива.
3. MOVE – перемещение на графике одного из существующих геометрических примитивов, относящихся к данному инструменту.

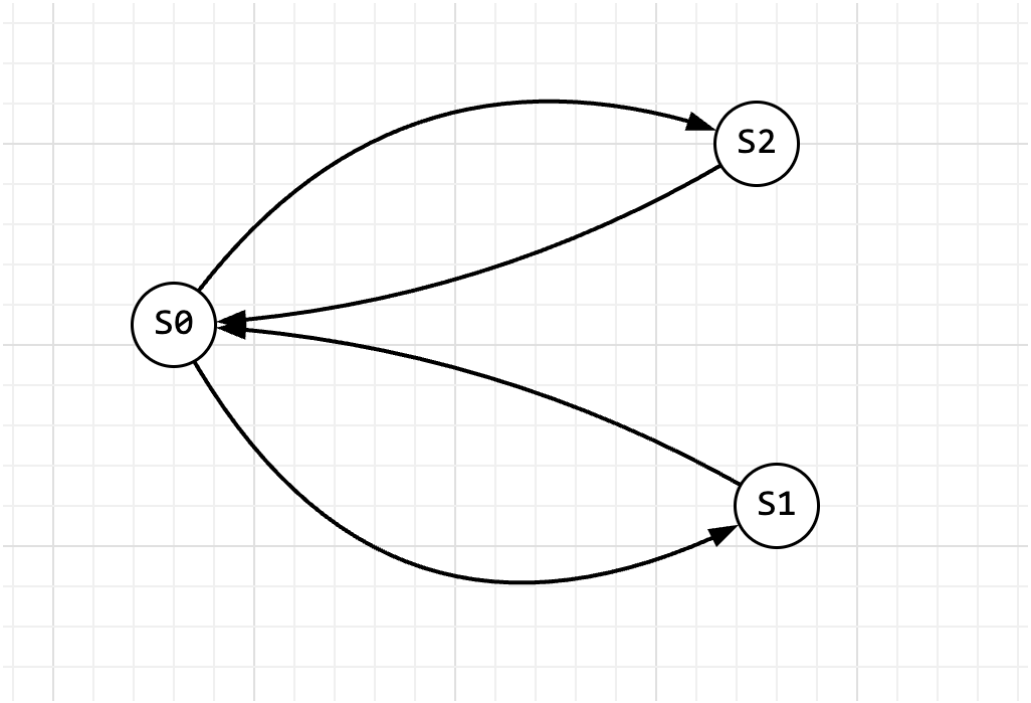

Рис. 7. Граф переходов состояний детерминированного конечного автомата инструмента рисования

Для перехода из состояния IDLE в ADD вызывается функция базового класса всех инструментов рисования $DrawingToolBase.startDrawing()$. В зависимости от типа инструмента рисования в переходе в состояние ADD начинают обрабатываться события мыши такие, как:

1. Движение мыши.
2. Клик.
3. Двойной клик.
4. Зажатие мыши.
5. Сброс зажатия мыши.

Каждый инструмент может обрабатывать вышеперечисленные события различными способами. На практике же многие инструменты имеют фиксированное количество точек для построение геометрического примитива своего типа, а потому механизм добавления нового примитива у них схож.

Рассмотрим его на примере инструмента рисования «Треугольник» (Листинг 10):

Листинг 10. Функция обработки клика мыши.

```
protected override _onClick(param: MouseEventParams) {
  if (!this._drawing || !param.point || !param.time || !this._series) return;
  const price = this._series.coordinateToPrice(param.point.y);
  if (price === null) {
    return;
  }

  const newPoint: Point = { time: param.time, price };
  if (this._previewDrawing == null) {
    this._addPointToCache(newPoint);
    this._addPointToCache(newPoint);
    this._addPreviewDrawing(this._pointsCache);
  } else {
    this._addPointToCache(newPoint);
    this._previewDrawing.addPoint(newPoint);
    if (this._pointsCache.length > 3) {
      this._removePreviewDrawing();
      this._addNewDrawing(this._pointsCache.slice(0, 3));
      this.stopDrawing();
    }
  }
}
```

Как можно заметить, на каждый клик мыши проверяется, находится ли инструмент в состоянии ADD и корректны ли текущие координаты мыши. Далее, точка с координатами мыши добавляется во временное хранилище точек $this.\_addPointToCache(newPoint)$, а также в класс примитива рисования, отвечающий за предварительный просмотр добавляемого примитива. По достижении необходимого количества точек (в данном примере – трех), в массив примитивов добавляется только что построенный, а инструмент рисования переводится в состоянии IDLE вызовом функции $DrawingToolBase.stopDrawing()$.

Для перевода инструмента в состояние MOVE необходимо убедиться, что мышь была зажата и что в этот момент она находится над одним из примитивов. Для этого примитив инструмента рисования реализует функцию

$hitTest$, которая определяет, произошла ли коллизия между точкой под мышью на экране и геометрия примитива. Как только зажатие мыши сбрасывается, инструмент переходит в исходное состояние IDLE.

В целях улучшения архитектуры приложения и упрощения читабельности исходного кода было принято решение вынести функции, которые связаны с нахождением коллизий в класс $CollisionHelper$, а функции, использующиеся, для математических операций и преобразований в класс $MathHelper$. Некоторые из функций, входящих в эти классы, будут рассмотрены в следующих главах в контексте конкретных инструментов.

Инструменты рисования «Прямоугольник» и «Треугольник» имеют существенное концептуальное сходство, а потому будут рассмотрены в одной главе. Ровно по той же причине в следующих главах будут объединены в одни главы другие инструменты рисования.

2.2.1. Реализация инструментов «Прямоугольник», «Треугольник»

Инструменты «Прямоугольник» и «Треугольник» строятся добавлением на график двух и трех точек соответственно. Причём для инструмента «Прямоугольник» первая точка – это левая верхняя точка, а вторая – правая нижняя вершины прямоугольника. Для треугольника геометрический смысл точек и их соответствие вершинам – очевидны.

Каждый геометрический примитив имеет свои параметры визуального стиля. Для простоты было принято решение вынести хранение стилей в класс, определяющий интерфейс $TriangleOptions$ и $RectangleOptions$ (Листинг 11):

Листинг 11. Интерфейс стилей «Треугольника»

```
export interface TriangleOptions {
  fillColor: string;
  previewFillColor: string;
  lineColor: string;
  lineWidth: number,
  labelColor: string;
  labelTextColor: string;
  showLabels: boolean;
  priceLabelFormatter: (price: number) => string;
  timeLabelFormatter: (time: Time) => string;
}
```

Интерфейс стилей «Прямоугольника» практически идентичен, поэтому имеет смысл опустить его.

Теперь, имея точки для рисования примитива и стили, можно рассмотреть процесс рисования (Листинг 12):

Листинг 12. Отрисовка «Прямоугольника»

```
class RectanglePaneRenderer implements IPrimitivePaneRenderer {
  _points: ViewPoint[];
  _options: RectangleOptions;

  constructor(points: ViewPoint[], options: RectangleOptions) {
    this._points = points;
    this._options = options;
  }

  draw(target: CanvasRenderingTarget2D) {
    target.useBitmapCoordinateSpace((scope) => {
      if (this._points.length < 2) {
        return;
      }

      const ctx = scope.context;

      const calculateDrawingPoint = (point: ViewPoint): ViewPoint => {
        return {
          x: Math.round(point.x * scope.horizontalPixelRatio),
          y: Math.round(point.y * scope.verticalPixelRatio),
        };
      };

      const drawingPoint1: ViewPoint = calculateDrawingPoint(this._points[0]);
      const drawingPoint2: ViewPoint = calculateDrawingPoint(this._points[1]);
      ctx.fillStyle = this._options.fillColor;
      ctx.fillRect(
        Math.min(drawingPoint1.x, drawingPoint2.x),
```

```
        Math.min(drawingPoint1.y, drawingPoint2.y),
        Math.abs(drawingPoint1.x - drawingPoint2.x),
        Math.abs(drawingPoint1.y - drawingPoint2.y));
    });
  }
```

В приведённом коде реализован метод $draw$ класса $RectanglePaneRenderer$, отвечающий за отрисовку «Прямоугольника» на элементе Canvas с учётом коэффициентов масштабирования пикселей. Если передано менее точек для рисования, выполнение метода прекращается, поскольку невозможно определить границы прямоугольника. При наличии двух точек они преобразуются в координаты пиксельного пространства с округлением — с учётом горизонтального и вертикального коэффициентов масштабирования ($horizontalPixelRatio$ и $verticalPixelRatio$). Это обеспечивает точную визуализацию на устройствах с различной плотностью пикселей.

Далее устанавливается цвет заливки согласно параметрам $RectangleOptions$, после чего вызывается метод $fillRect$, отрисовывающий прямоугольник. Его положение и размеры определяются на основе минимальных координат по осям и абсолютных разностей между ними.

Вся отрисовка осуществляется внутри контекста $useBitmapCoordinateSpace$, что обеспечивает согласованность координат с пиксельной сеткой элемента canvas.

Функция отрисовки «Треугольника» похожа, однако она имеет дополнительную третью точку. При наличии двух точек в процессе построения примитива визуализируется прямая линия. При добавлении третьей точки рисуется заполненный треугольник. Для этого строится область заполнения, заполнение которой инициируется вызовом функции $CanvasRenderingContext2D{:}\ beginPath()$. Добавляется несколько линий,

соединяющих точки треугольника, и происходит зарисовка области заполнения выбранным цветом.

Также, определены функции $hitTest$, которые по тем же точкам, с помощью которых рисовались на Canvas примитивы, определяет, попали ли координаты мыши внутрь геометрии примитивов. Для этого используются функции вспомогательного класса:

- $CollisionHelper.IsPointInRectangle(hitTestPoint, rectangleBox)$
- $CollisionHelper.IsPointInTriangle(hitTestPoint, polygonPoints)$

Они не представляют интереса с точки зрения вычислений, поэтому в данной работе подробно не рассматриваются.

2.2.2. Реализация инструментов «Горизонтальная линия», «Вертикальная линия», «Линия тренда»

Описание реализаций инструментов линий приведено в сжатом объёме ввиду их тривиальности. Горизонтальная и вертикальные линии строятся и визуализируются по одной точке на всю длину экрана горизонтально и вертикально соответственно. Реализация «Линии тренда» линий является похожей на реализацию «Прямоугольника» за тем лишь исключением, что вместо прямоугольник по двум точкам рисуется линия.

Функция $hitTest$ также проста и предусматривает совпадение координат на оси абсцисс и ординат для горизонтальной и вертикальных линий с точкой под мышью, а для «Линии тренда» используется функция проверки на коллизию точки и отрезка.

2.2.3. Реализация инструмента рисования «Спираль Фибоначчи»

Инструмент «Спираль Фибоначчи» представляет интерес с точки зрения подхода к композитной визуализации геометрического примитив. Отрисовка спирали Фибоначчи представляет собой визуализацию отрезка, а также последовательности частей окружностей, длины радиусов которых являются последовательностью чисел Фибоначчи. Спираль Фибоначчи является аппроксимацией т.н. Золотой спирали [11]. На рис. 8 представлена аппроксимация Золотой спирали с помощью окружностей радиусов последовательности Фибоначчи:

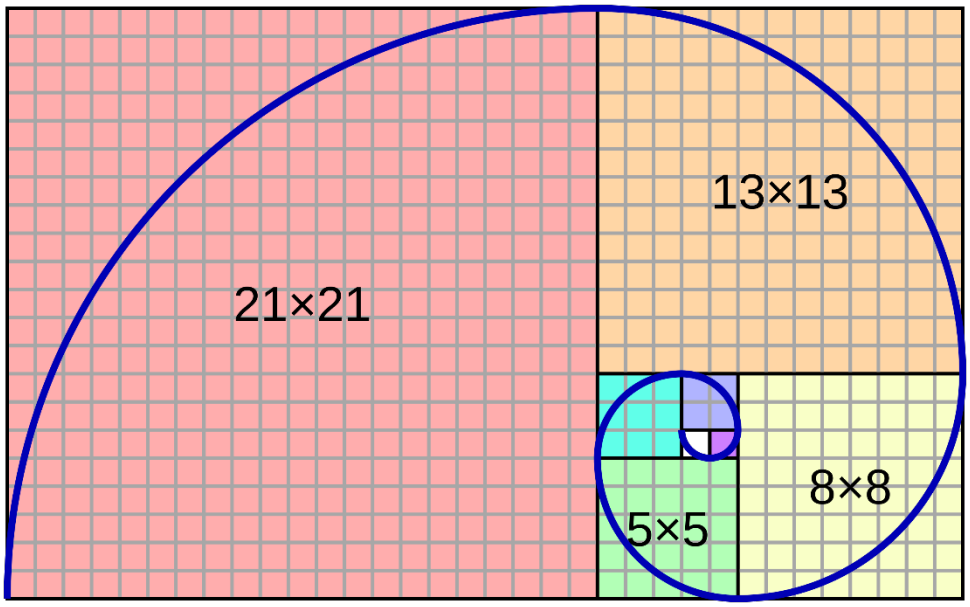

Рис. 8. Спираль Фибоначчи аппроксимирует золотую спираль с использованием четвертинок окружности в квадратах с размерами квадратов, равных числам Фибоначчи. На рисунке показаны квадраты с размерами 1, 1, 2, 3, 5, 8, 13, 21.

Процесс визуализации реализуется спирали охватывает несколько этапов:

Подготовка входных данных:

На основе двух точек, задаваемых пользователем, вычисляются параметры спирали, описанные в интерфейсе $FibSpiralRenderInfo$ (Листинг 13):

Листинг 13. $FibSpiralRenderInfo$

```
export interface FibSpiralRenderInfo {
  rotationCenter: ViewPoint;
  rayStart: ViewPoint;
  rayEnd: ViewPoint;
  spiralRotationAngle: number;
  numArcs: number;
  arcCenters: ViewPoint[];
  arcRadiuses: number[];
  arcAngles: number[][];
}
```

$rotationCenter$ является центром вращения. Это — исходная точка, от которой начинается построение дуг.

$spiralRotationAngle$ – угол поворота, определяющий ориентацию всей спирали относительно горизонтальной оси.

$numArcs$ – количество частей окружностей, радиусы которых соответствуют числам Фибоначчи.

Центры окружностей, их радиусы, а также сектора, в границах которых они визуализируются $(arcCenters, arcRadiuses, arcAngles)$ – рассчитываются итеративно.

В приложении 1 представлена функция подготовки данных для визуализации Спирали Фибоначчи.

Визуализация данных

При визуализации спирали необходимо сначала изменить параметры координатного пространства графического контекста $CanvasRenderingContext2D$. Это преобразовние необходимо, т.к. данные для отрисовки были построены для спирали, в которой луч для ее построения параллелен горизонтальной оси. Только повернув координатного пространство на заранее посчитанный угол поворот между лучом и

горизонтальной осью, можно достигнуть правильной визуализации геометрического примитива.

Далее последовательно на графическом контексте рисуются окружности, ограниченные секторами для визуализации. В конце рисуется луч, соединяющий центр поворота, являющийся одновременно началом первой окружности, и вторую точку, заданную пользователем на графике.

Реализация функции $hitTest$

Для проверки попадания координат мыши в спираль было решено применить следующее преобразование – вектор, направленный от точки центра вращения спирали до точки нахождения мыши поворачивается на отрицательное значение угла поворота. Так достигается переход в локальные координаты спираль, в которых ось абсцисс параллельно лучу направления спирали.

После вышеописанного преобразование достаточно итеративно проверить на коллизию части окружностей, а также луч. Проверка на коллизию реализована, в частности, с помощью функции класса $CollisionHelper\ HitTestArc$.

2.2.4. Реализация инструмента рисования «Клин Фибоначчи»

Инструмент «Клин Фибоначчи» содержит метод визуализации секторов кольца, ограниченных дугами окружностей с радиусами, пропорциональными выбранным уровням Фибоначчи. Геометрическая структура клина строится на основе заданных пользователем трёх точек (Рис. 9):

- $P_0 = (x_0, y_0)$ – точка центра концентрических колец.
- $P_1 = (x_1, y_1)$ – точка, определяющая радиус колец, а также угол начала сектора.

- $P_2 = (x_2, y_2)$ – точка, определяющая окончательный размер сектора.

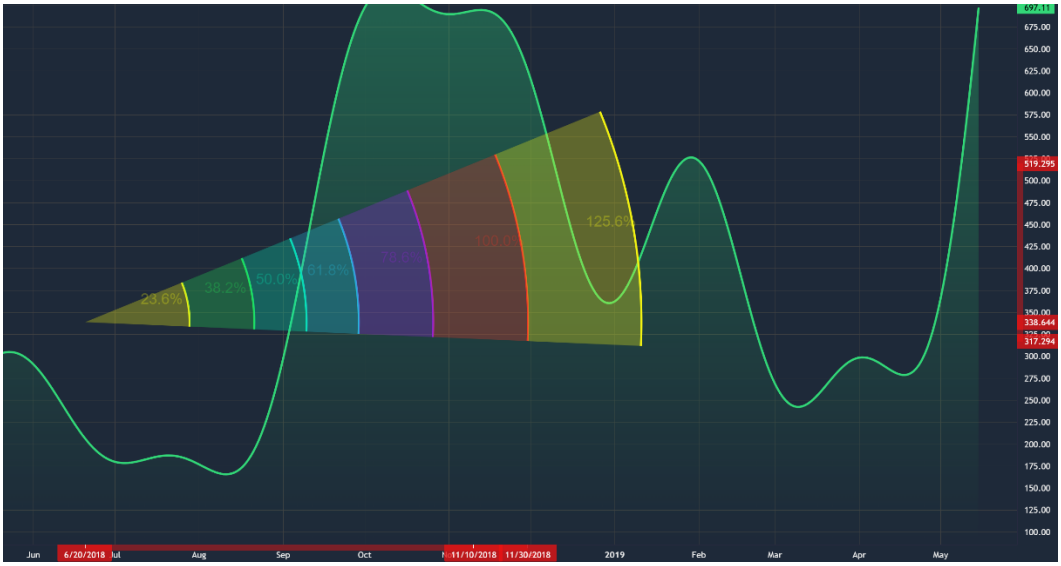

Рис. 9. Пример «Клина Фибоначчи» в разработанной библиотеке

Визуализация данных

По точкам P_0, P_1, P_2 вычисляются:

1. Радиус R как евклидово расстояние между P_0, P_1:

$$R = \sqrt{((x_0 - x_1)^2 + (y_0 - y_1)^2)}$$

2. Вектор направления первого ограничивающего вектора сектора v_1

$$v_1 = P_1 - P_0$$

3. Вектор направления второго ограничивающего вектора сектора v_2:

$$v_2 = P_2 - P_0$$

4. Угол между v_1 и осью абсцисс θ_{start}:

$$\theta_{start} = \angle(\overrightarrow{e_x}, \overrightarrow{v_1})$$

5. Угол θ_{sweep} между v_1 и v_2, определяющий размер сектора:

$$\theta_{sweep} = \angle(\overrightarrow{v_1}, \overrightarrow{v_2})$$

Для отображения клина Фибоначчи используются уровни Фибоначчи:

$$FibLevels = \{f_0, f_1, \dots, f_n\}, f_i \in [0, \infty)$$

Каждый уровень Фибоначчи f_i из массива $FibLevels$ соответствует радиусу:

$$R_i = R \cdot f_i$$

На каждом шаге визуализируется кольцевой сектор между двумя радиусами R_{i-1} и R_i, ограниченный углом $[\theta_{start}, \theta_{start} + \theta_{sweep}]$. Для каждого такого сектора применяется функция $fillAnnulusSector$. В данную функцию передаются параметры:

$$P_0, R_{i-1}, R_i, \theta_{start}, \theta_{sweep}$$

Рисование кольцевого сектора выполняется путём построения дуг большой и малой окружности и соединения их линиями, что даёт замкнутую область кольцевого сектора (Листинг 14):

Листинг 14. Функция визуализации кольцевого сектора

```
export function fillAnnulusSector(renderingScope: BitmapCoordinatesRenderingScope,
annulusRenderInfo: AnnulusSectorRenderInfo, fillColor: string, lineColor: string) {
  const ctx = renderingScope.context;

  ctx.fillStyle = fillColor;

  const center = annulusRenderInfo.annulusCenter;
  const radiusSmall = annulusRenderInfo.radiusSmall;
  const radiusBig = annulusRenderInfo.radiusBig;
  const angleStart = annulusRenderInfo.startAngle;
  const angleEnd = annulusRenderInfo.startAngle + annulusRenderInfo.sweepAngle;

  const centerVec = new Vector2D(center.x, center.y);

  let r2 = new Vector2D(radiusSmall, 0);
  r2 = centerVec.add(MathHelper.RotateVector(r2, angleEnd));

  const isClockwise: boolean = annulusRenderInfo.sweepAngle < 0;

  ctx.lineWidth = 5;
  ctx.strokeStyle = lineColor;

  ctx.beginPath();
  ctx.arc(center.x, center.y, radiusBig, angleStart, angleEnd, isClockwise);
  ctx.lineTo(r2.x, r2.y);
  ctx.arc(center.x, center.y, radiusSmall, angleEnd, angleStart, !isClockwise);
  ctx.closePath();
  ctx.fill();

  ctx.beginPath();
  ctx.arc(center.x, center.y, radiusBig, angleStart, angleEnd, isClockwise);
  ctx.stroke();
}
```

После отрисовки последовательности концентрических кольцевых секторов выполняется отрисовка меток с уровнем Фибоначчи f_i. Для каждого f_i вычисляется вектор биссектрисы сектора b:

$$b = v_1 \text{ повернутый на } \frac{\theta_{sweep}}{2}$$

С помощью b вычисляется текущая координата метки с уровнем Фибоначчи в точке T:

$$T = (t_x, t_y) = P_0 + b \cdot f_i$$

Реализация функции *hitTest*

Функция ***hitTest*** данного примитива для каждого концентрического кольцевого сектора проверяет коллизию дуги этого сектор, используя уже приведённую выше функцию $HitTestArc$ (Листинг 15).

2.2.5. Реализация инструмента рисования «Коррекция Фибоначчи»

Инструмент «Коррекция Фибоначчи» представляет визуализацию последовательности параллельных отрезков, расстояния между которыми вычисляются по уровням Фибоначчи. Области между отрезками заполняется заданными цветами (Рис. 10).

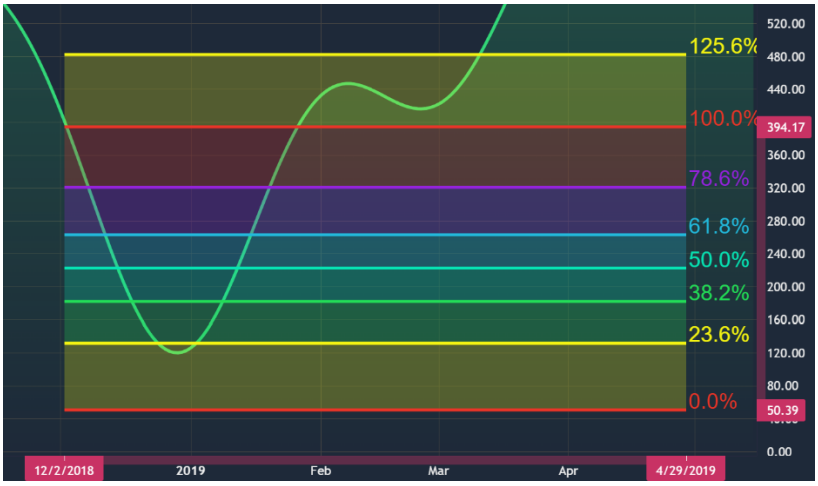

Рис. 10. «Коррекция Фибоначчи»

Геометрическая структура «Коррекции Фибоначчи» строится на основе заданных пользователем двух точек:

- $P_0 = (x_0, y_0)$ – точка, определяющая начало линий коррекции.
- $P_1 = (x_1, y_1)$ – точка, определяющая конец линий коррекции.

Визуализация данных

По точкам P_0, P_1 вычисляются:

1. Вектор направления dir от P_0 к P_1:

$$dir = P_1 - P_0$$

2. Проекция вектора $ProjDir_y$ на ось ординат:

$$ProjDir_y = (0, dir_y)$$

3. Для каждого уровня Фибоначчи f_i из $FibLevels$:

$$FibLevels = \{f_0, f_1, \dots, f_n\}, f_i \in [0, \infty)$$

вычисляются начало S и конец отрезка E линия по формуле:

$$S_i = \left(P_{0_x}, P_{0_y} + ProjDir_y \cdot f_i\right)$$

$$E_i = \left(P_{1_x}, P_{0_y} + ProjDir_y \cdot f_i\right)$$

Далее с помощью $CanvasRenderingContext2D$ визуализируются отрезки от S_i до E_i для каждого f_i, а также заполняются прямоугольные области заданными цветами между этими отрезками (Приложение 2).

Реализация функции $hitTest$

Для выявления факта коллизии функция $hitTest$ данного примитива проверяет коллизии со всеми построенными отрезками (S_i, E_i) точки координата мыши на графике.

2.2.6. Реализация инструмента рисования «Кривая»

Инструмент «Кривая» представляет визуализацию кривых Безье [12], заполнение области между этими кривыми и ограничивающим отрезком, а также проверку на коллизию точки мыши на графике с визуализированным геометрическим примитивом с использованием аппроксимации области под кривой через многоугольник. Результат визуализации представлен на Рис. 11:

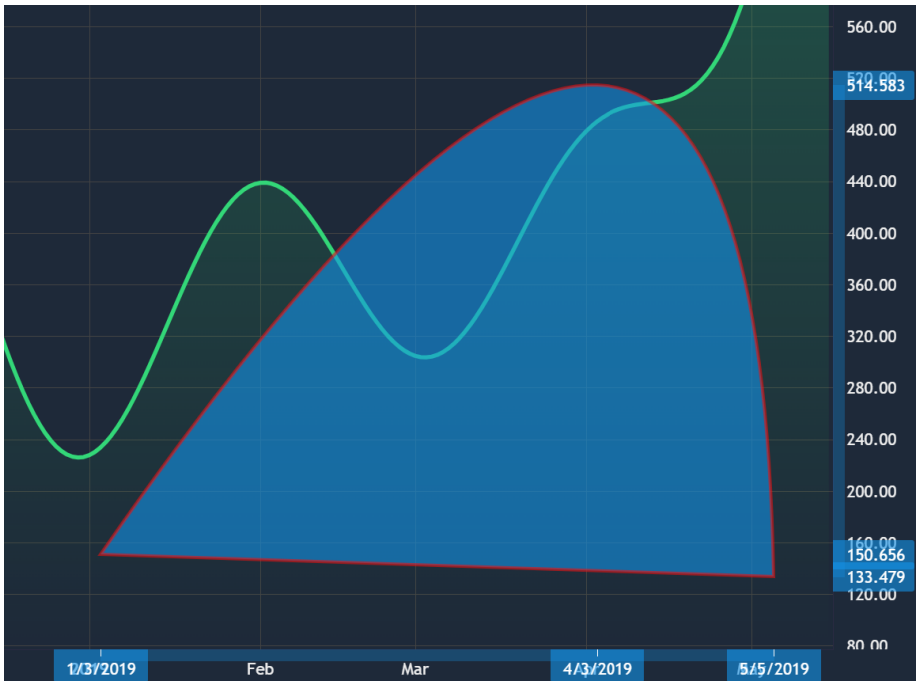

Рис. 11. «Кривая»

Геометрическая структура «Кривой» строится на основе заданных пользователем трех точек:

- $P_0 = (x_0, y_0)$ – первая точка основания.
- $P_1 = (x_1, y_1)$ – вторая точка основания.
- $P_2 = (x_2, y_2)$ – верхняя точка.

Визуализация данных

По точкам P_0, P_1, P_2 происходят следующие вычисления и преобразования:

1. Вычисляются параметры для построения двух смежных квадратичных кривых l и m. Эти параметры включают в себя контрольные точки первой и второй кривых соответственно $\{l_1, l_2, l_3\}$, $\{m_1, m_2, m_3\}$:

$$dir = P_0 - P_2$$

$$l_1 = P_0, l_2 = P_1 + (dir \cdot 0.25), l_3 = P_1$$

$$m_1 = P_1, m_2 = P_1 - (dir \cdot 0.25), m_3 = P_2$$

2. Далее выполняется преобразование контрольных точек данных квадратичных кривых l и m - $\{l_1, l_2, l_3\}$, $\{m_1, m_2, m_3\}$ к контрольным точкам кубических кривых $\tilde{l}$ и $\tilde{m}$ - $\{\tilde{l_1}, \tilde{l_2}, \tilde{l_3}, \tilde{l_4}\}$, $\{\tilde{m_1}, \tilde{m_2}, \tilde{m_3}, \tilde{m_4}\}$:
    a. Вычисление контрольных точек кубических кривых $\tilde{l}$ и $\tilde{m}$:

$$\tilde{l_1} = l_1, \tilde{l_4} = l_3, \tilde{m_1} = m_1, \tilde{m_4} = m_3$$

    b. Вычисление опорных точек кубических кривых $\tilde{l}$ и $\tilde{m}$:

$$\tilde{l_2} = l_1 + (l_2 - l_1) \cdot \frac{2}{3}, \tilde{l_3} = l_3 + (l_2 - l_3) \cdot \frac{2}{3},$$

$$\tilde{m_2} = m_1 + (m_2 - m_1) \cdot \frac{2}{3}, \tilde{m_3} = m_3 + (m_2 - m_3) \cdot \frac{2}{3}$$

Далее с помощью $CanvasRenderingContext2D$ визуализируются кривые Безье $\tilde{l}$, $\tilde{m}$ с помощью последовательного вызова функций (Приложение 3):

1. $CanvasRenderingContext2D.moveTo\ (\tilde{l_1})$
2. $CanvasRenderingContext2D.bezierCurveTo\ (\tilde{l_2}, \tilde{l_3}, \tilde{l_4})$
3. $CanvasRenderingContext2D.moveTo\ (\tilde{m_1})$
4. $CanvasRenderingContext2D.bezierCurveTo\ (\tilde{m_2}, \tilde{m_3}, \tilde{m_4})$

Реализация функции *hitTest*

Для выявления факта коллизии функция ***hitTest*** кривой проверяет коллизию точки мыши на графике с внутренностью многоугольника, который является аппроксимацией кривой. Для построения данной аппроксимации используется следующий подход:

Так как квадратичная кривая представлена уравнением (1):

$$f(t) = (1-t)k_1 + 2t(1-t)k_2 + t^2k_3 \ (1),$$

где $\{k_1, k_2, k_3\} -$ контрольные точки квадратичной кривой k

то можно совершить итерации по переменной t в промежутке $t \in [0, 1]$ с некоторым шагом s по уже построенным квадратичным кривым l и m. Таким образом, каждая точка $f_i = f(t_i)$ будет является точкой на кривой. Теперь, соединив последовательно соседние точки на кривой отрезками, можно построить многоугольник.

Теперь можно выполнить проверку на вхождение точки в многоугольник с помощью известного алгоритма трассировки луча [13], реализованного в рамках библиотеки в вспомогательном классе для проверки коллизий $CollisionHelper$ (Приложение 4).

2.2.7.Реализация инструмента рисования «Ломаная»

Инструмент «Ломанная» представляет визуализацию ломанной линии с произвольным количеством точек для построения, а также проверку на коллизию точки мыши на графике с визуализированным геометрическим примитивом Результат визуализации представлен на Рис. 12:

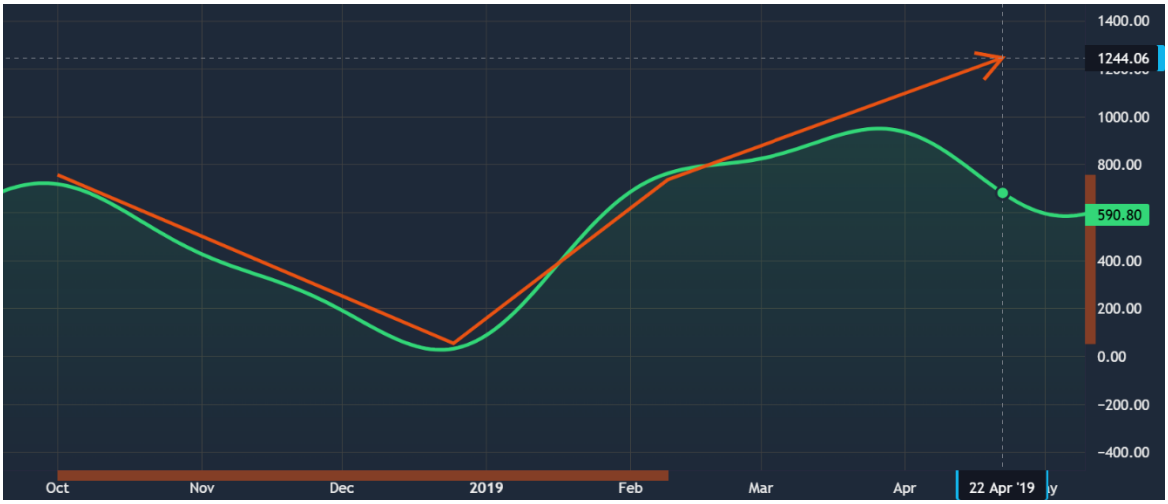

Рис. 12. «Ломанная»

Геометрическая структура «Ломаной» строится на основе заданных пользователем точек:

- $\{P_0, P_1, \dots, P_i, \dots, P_n\}$

Визуализация данных

По точкам $P_0, P_1, \dots, P_i, \dots, P_n$ происходят следующие вычисления и преобразования:

a. Вычисляются отрезки, соединяющие соседние точки $\{P_i, P_{i-1}\}$ для всех $i \in [1, n]$.

b. Далее выполняется построение точек для геометрического примитива стрелки $\{Arrow_0, Arrow_1, Arrow_2\}$, которая рисуется на последней точке P_n ломаной и указывает на направление, определяемое последними двумя точками ломаной P_n, P_{n-1}:

$$v_1 = P_n - P_{n-1}$$

Вычисляется нормализованный вектор v_2, перпендикулярный к v_1:

$$v_2 = \frac{\left(-v_{1_y}, v_{1_x}\right)}{\left|\left(-v_{1_y}, v_{1_x}\right)\right|}$$

Впоследствии вектор v_2 умножается на скалярное значение базовой длины l стрелки и на отношение стороны катета в прямоугольном треугольнике со с углами в 30 и 60 градусов, соответствующему отступу, применив который, можно вычислить точки $Arrow_0$, $Arrow_2$:

$$v_2 = v_2 \cdot l \cdot \frac{\sqrt{3}}{3}$$

$$Arrow_0 = P_n - v_1 \cdot l + v_2$$

$$Arrow_1 = P_n$$

$$Arrow_2 = P_n - v_1 \cdot l - v_2$$

Далее с помощью $CanvasRenderingContext2D$ визуализируются построенные отрезки и построенный примитив стрелки с помощью последовательного вызова функций (Приложение 5):

1. $CanvasRenderingContext2D.moveTo(P_0)$
2. $CanvasRenderingContext2D.lineTo(P_i),\ \forall\, i \in [1, n]$
3. $CanvasRenderingContext2D.moveTo(Arrow_0)$
4. $CanvasRenderingContext2D.moveTo(Arrow_1)$
5. $CanvasRenderingContext2D.moveTo(Arrow_2)$

Реализация функции $\boldsymbol{hitTest}$

Для выявления факта коллизии, функция $\boldsymbol{hitTest}$ кривой проверяет коллизию точки мыши на графике с построенными отрезками $\{P_i, P_{i-1}\}$ для всех $i \in [1, n]$, а также с отрезками $\{Arrow_0, Arrow_1\}$, $\{Arrow_1, Arrow_2\}$.

2.3. Реализация технических индикаторов

Тут будет текст про реализацию технических индикаторов.

2.4. Создание и публикация npm библиотеки

Тут будет текст про npm библиотеку.

2.5. Создание и публикация тестового приложения

Тут будет текст про публикацию тестового приложения.

Данные для построения графика получены по протоколу REST API из общедоступного API Binance. Данное API позволяет совершить бесплатно до 1200 запросов в минуту, что вполне отвечает задачам приложения.

Заключение

В рамках работы были исследованы основные инструменты анализа финансовых данных, произведён сравнительный анализ существующих решений. Была разработана и опубликована библиотека с визуальными инструментами для анализа финансовых данных на языке TypeScript с использованием Lightweight Charts и Canvas API.

Реализованный набор инструментов рисования включает ключевые средства, применяемые в техническом анализе: инструменты линий, инструменты на основе уровней Фибоначчи, кривая, ломанная линия. Библиотека также включает поддержку базовых технических индикаторов, таких как линии Боллинджера и скользящие средние. Предусмотрена возможность масштабирования объектов на графике.

Также было разработано тестовое веб-приложение с применением фреймворка Vue для демонстрации возможности библиотеки.

Список литературы

1. Платформа TradingView – URL: https://tradingview.com/ (дата обращения 15.05.2025)
2. Murphy J. J. Technical analysis of the financial markets: A comprehensive guide to trading methods and applications. – Penguin, 1999.
3. Куликов Л. А. Форекс для начинающих. Справочник биржевого спекулянта //СПб.: Питер–2006.–384 с. – 2006.
4. Документация языка PineScript – URL: https://www.tradingview.com/pine-script-docs/ (дата обращения 15.05.2025)
5. Документация библиотеки Lightweight Charts – URL: https://tradingview.github.io/lightweight-charts/ (дата обращения 15.05.2025)
6. Библиотека Plotty – URL: https://plotly.com/graphing-libraries/ (дата обращения 15.05.2025)
7. Библиотека Go-Chart – URL: https://github.com/wcharczuk/go-chart (дата обращения 15.05.2025)
8. Библиотека TA-Lib – URL: https://github.com/TA-Lib/ta-lib (дата обращения 15.05.2025)
9. Документация Canvas API – URL: https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API (дата обращения 15.05.2025)
10. Z-Order – URL: https://en.wikipedia.org/wiki/Z-order API (дата обращения 15.05.2025)

11. Золотая спираль – URL: https://en.wikipedia.org/wiki/Golden_spiral (дата обращения 15.05.2025)
12. Кривые Безье – URL: https://en.wikipedia.org/wiki/B%C3%A9zier_curve (дата обращения 15.05.2025)
13. Задача на проверку принадложености точки к многоугольнику – URL: https://en.wikipedia.org/wiki/Point_in_polygon (дата обращения 15.05.2025)

Приложение 1. Функция подготовки данных «Спирали Фибоначчи»

```
    updateRenderInfo(drawingPoints: ViewPoint[]): 
FibSpiralRenderInfo {
        let spiralRotationCenter = { x: 0, y: 0 };
        let spiralRotationAngle: number = 0;
        let numArcs: number = 0;
        let arcCenters: ViewPoint[] = [];
        let arcRadiuses: number[] = [];
        let arcAngles: number[][] = [];
        let rayStart = { x: 0, y: 0 };
        let rayEnd = { x: 0, y: 0 };;

        const pointsAreValid = drawingPoints.length == 
2 && drawingPoints[0].x != null && drawingPoints[1].x 
!= null && drawingPoints[0].y != null && 
drawingPoints[1].y != null;
        const pointAreEqual = pointsAreValid && 
drawingPoints[0].x == drawingPoints[1].x && 
drawingPoints[0].y == drawingPoints[1].y;

        if (pointsAreValid && !pointAreEqual) {
            const p1: Vector2D = new 
Vector2D(drawingPoints[0].x, drawingPoints[0].y);
            const p2: Vector2D = new 
Vector2D(drawingPoints[1].x, drawingPoints[1].y);

            let initDir = p2.subtract(p1);

            spiralRotationAngle = -1 * 
MathHelper.AngleBetweenVectors(initDir, new 
Vector2D(1, 0));
            initDir = MathHelper.RotateVector(initDir, 
-spiralRotationAngle);

            const rotationCenter: Vector2D = p1;
            const directionPoint: Vector2D = 
p1.add(initDir);

            let a: number;
            {
```

```
                const p1 = new
Point2D(rotationCenter.x, rotationCenter.y);
                const p2 = new
Point2D(directionPoint.x, directionPoint.y);
                a = p1.distanceTo(p2)[0] /
(Math.sqrt(55.0) + 1);
            }

            numArcs = 11;

            const bCounterClockwise: boolean = true;

            const clockwiseCoef: number =
bCounterClockwise ? 1.0 : -1.0;
            arcCenters = new
Array<ViewPoint>(numArcs);

            arcCenters[0] = { x: 0.0, y: clockwiseCoef
* a };
            arcCenters[1] = { x: -a, y: clockwiseCoef
* a };
            arcCenters[2] = { x: -a, y: clockwiseCoef
* 0.0 };
            arcCenters[3] = { x: a, y: clockwiseCoef *
0.0 };
            arcCenters[4] = { x: a, y: clockwiseCoef *
3.0 * a };
            arcCenters[5] = { x: -4.0 * a, y:
clockwiseCoef * 3.0 * a };
            arcCenters[6] = { x: -4.0 * a, y:
clockwiseCoef * -5.0 * a };
            arcCenters[7] = { x: 9.0 * a, y:
clockwiseCoef * -5.0 * a };
            arcCenters[8] = { x: 9.0 * a, y:
clockwiseCoef * 16.0 * a };
            arcCenters[9] = { x: -25.0 * a, y:
clockwiseCoef * 16.0 * a };
            arcCenters[10] = { x: -25.0 * a, y:
clockwiseCoef * -39.0 * a };

            arcRadiuses = new Array<number>(numArcs);
            arcRadiuses[0] = a;
            arcRadiuses[1] = 2.0 * a;
            for (let i = 2; i < 11; i++) {
```

```
                arcRadiuses[i] = arcRadiuses[i - 1] +
arcRadiuses[i - 2];
            }

            arcAngles = new Array<number[]>(numArcs);
            if (bCounterClockwise) {
                arcAngles[0] = [270.0, 90.0];
                arcAngles[1] = [0.0, 90.0];
                arcAngles[2] = [90.0, 90.0];
                arcAngles[3] = [180.0, 90.0];
                arcAngles[4] = [270.0, 90.0];
                arcAngles[5] = [0.0, 90.0];
                arcAngles[6] = [90.0, 90.0];
                arcAngles[7] = [180.0, 90.0];
                arcAngles[8] = [270.0, 90.0];
                arcAngles[9] = [0.0, 90.0];
                arcAngles[10] = [90.0, 90.0];
            } else {
                arcAngles[0] = [90.0, -90.0];
                arcAngles[1] = [360.0, -90.0];
                arcAngles[2] = [270.0, -90.0];
                arcAngles[3] = [180.0, -90.0];
                arcAngles[4] = [90.0, -90.0];
                arcAngles[5] = [360.0, -90.0];
                arcAngles[6] = [270.0, -90.0];
                arcAngles[7] = [180.0, -90.0];
                arcAngles[8] = [90.0, -90.0];
                arcAngles[9] = [360.0, -90.0];
                arcAngles[10] = [270.0, -70.0];
            }

            spiralRotationCenter = { x: p1.x, y: p1.y
};

            rayStart = spiralRotationCenter;

            const bigNumber: number = 3000; // Hack:
extending ray beyond the end of the screen
            rayEnd = { x: directionPoint.x +
bigNumber, y: directionPoint.y };
        }

        return {
            rotationCenter: spiralRotationCenter,
```

```
            spiralRotationAngle: spiralRotationAngle,
            numArcs,
            arcCenters,
            arcRadiuses,
            arcAngles,
            rayStart,
            rayEnd,
        }
    }
```

Приложение 2. Функция отрисовки «Коррекции Фибоначчи»

```
draw(target: CanvasRenderingTarget2D) {
        target.useBitmapCoordinateSpace(scope => {
            if (this._points.length < 2) {
                return;
            }

            const ctx = scope.context;

            const calculateDrawingPoint = (point:
ViewPoint): ViewPoint => {
                return {
                    x: Math.round(point.x *
scope.horizontalPixelRatio),
                    y: Math.round(point.y *
scope.verticalPixelRatio)
                };
            };

            for (let i = 0; i < this._points.length;
i++) {
                this._points[i] =
calculateDrawingPoint(this._points[i]);
            }

            const dir: Vector2D = new Vector2D(0,
this._points[1].y - this._points[0].y);

            ctx.font = '36px Arial';

            const fibonacciLevels =
this._options.fibonacciLevels;
            const fibonacciLineColors =
this._options.fibonacciLineColors;

            const oldGlobalAlpha = ctx.globalAlpha;
            ctx.globalAlpha = 0.25;

            //  filling background first
            for (let i: number = 0; i <
fibonacciLevels.length; i++) {
                const currIndex = i;
```

```
                const nextIndex = currIndex + 1 <
fibonacciLevels.length ? currIndex + 1 : currIndex;
                const curLevel: number =
fibonacciLevels[currIndex];
                const nextLevel: number =
fibonacciLevels[nextIndex];
                if (currIndex != nextIndex) {
                    ctx.fillStyle =
fibonacciLineColors[nextIndex %
fibonacciLineColors.length];
                    const currY = new Vector2D(0,
this._points[0].y).add(dir.scale(0, curLevel)).y;
                    const nextY = new Vector2D(0,
this._points[0].y).add(dir.scale(0, nextLevel)).y;

                    ctx.beginPath();
                    ctx.moveTo(this._points[0].x,
currY);
                    ctx.lineTo(this._points[1].x,
currY);
                    ctx.lineTo(this._points[1].x,
nextY);
                    ctx.lineTo(this._points[0].x,
nextY);
                    ctx.fill();
                }
            }

            ctx.globalAlpha = oldGlobalAlpha;

            for (let i: number = 0; i <
fibonacciLevels.length; i++) {
                ctx.strokeStyle =
fibonacciLineColors[i % fibonacciLineColors.length];
                ctx.fillStyle = fibonacciLineColors[i
% fibonacciLineColors.length];
                ctx.lineWidth = 5;

                const level = fibonacciLevels[i];
                const y = new Vector2D(0,
this._points[0].y).add(dir.scale(0, level)).y;
                ctx.beginPath();
                ctx.moveTo(this._points[0].x, y);
                ctx.lineTo(this._points[1].x, y);
```

```
                ctx.stroke();
                ctx.fillText(`${(level *
100).toFixed(1)}%`, (this._points[1].x + 4), (y - 2));
            }
        });
    }
```

Приложение 3. Функция отрисовки «Кривой»

```
export function fillBezierPath(renderingScope:
BitmapCoordinatesRenderingScope, bezierCurveInfo:
BezierCurvesPointsInfo, fillColor: string, lineColor:
string) {
    const ctx = renderingScope.context;

    const bezierSplines: Point2D[] =
Array<Point2D>(bezierCurveInfo.endPoints.length +
bezierCurveInfo.controlPoints1.length +
bezierCurveInfo.controlPoints2.length);
    /// ----------------------------------------
--------------------
    /// bezierSplines array structure:
    ///
    /// [point_0, control_point_0_0,
control_point_1_0, point_1,
    ///           control_point_0_1,
control_point_1_1, point_2,
    ///           ....
    ///           control_point_0_i,
control_point_1_i, point_i,
    ///           ....
    ///           control_point_0_n,
control_point_1_n, point_n]
    /// ----------------------------------------
--------------------

    let i = 0;
    bezierCurveInfo.endPoints.forEach((point) => {
        bezierSplines[i] = point;
        i += 3;
    });

    i = 1;
    for (let j = 0; j <
bezierCurveInfo.controlPoints1.length; ++j) {
        bezierSplines[i] =
bezierCurveInfo.controlPoints1[j];
        bezierSplines[i + 1] =
bezierCurveInfo.controlPoints2[j];
        i += 3;
```

```
    }
    ctx.beginPath();
    ctx.fillStyle = fillColor;

    ctx.moveTo(bezierSplines[0].x,
bezierSplines[0].y);
    for (let i = 0; i + 3 < bezierSplines.length; i +=
3) {
        ctx.bezierCurveTo(bezierSplines[i + 1].x,
bezierSplines[i + 1].y,
            bezierSplines[i + 2].x, bezierSplines[i +
2].y,
            bezierSplines[i + 3].x, bezierSplines[i +
3].y);
    }
    ctx.closePath();
    ctx.fill();

    ctx.strokeStyle = lineColor;
    ctx.lineWidth = 4;
    ctx.stroke();
}
```

Приложение 4. Функция проверки на вхождение точки в многоугольник

```
    public static IsPointInPolygon(point: Point2D,
polygonPoints: Point2D[]): boolean {
        const x: number = point.x;
        const y: number = point.y;

        let i: number;
        let j: number = polygonPoints.length - 1;
        let oddNodes: boolean = false;

        for (i = 0; i < polygonPoints.length; i++) {
            if ((polygonPoints[i].y < y &&
polygonPoints[j].y >= y
                || polygonPoints[j].y < y &&
polygonPoints[i].y >= y)
                && (polygonPoints[i].x <= x ||
polygonPoints[j].x <= x)) {
                const tmp: boolean =
polygonPoints[i].x + (y - polygonPoints[i].y) /
(polygonPoints[j].y - polygonPoints[i].y) *
(polygonPoints[j].x - polygonPoints[i].x) < x;
                oddNodes = oddNodes !== tmp;
            }
            j = i;
        }

        return oddNodes;
    }
```

Приложение 5. Функции визуализации «Ломаной»

```
export function
CalculateClassicArrowRenderInfo(lineStart: Point2D,
lineEnd: Point2D): ClassicArrowRenderInfo {
    const p0: Vector2D = new Vector2D(lineStart.x,
lineStart.y);
    const p1: Vector2D = new Vector2D(lineEnd.x,
lineEnd.y);

    let lineDirectionVec = p1.subtract(p0);
    let linePerpendicularVec = new Vector2D(-
lineDirectionVec.y, lineDirectionVec.x);

    linePerpendicularVec =
linePerpendicularVec.normalize();
    lineDirectionVec = lineDirectionVec.normalize();

    const defaultArrowLength: number = 35.0;

    linePerpendicularVec.x *= defaultArrowLength /
Math.sqrt(3.0);
    linePerpendicularVec.y *= defaultArrowLength /
Math.sqrt(3.0);

    lineDirectionVec.x *= -defaultArrowLength;
    lineDirectionVec.y *= -defaultArrowLength;

    const arrowWing1 =
p1.add(lineDirectionVec.add(linePerpendicularVec));
    const arrowBase = p1;
    const arrowWing2 =
p1.add(lineDirectionVec.subtract(linePerpendicularVec)
);

    return {
        arrowWing1: new Point2D(arrowWing1.x,
arrowWing1.y),
        arrowBase: new Point2D(arrowBase.x,
arrowBase.y),
        arrowWing2: new Point2D(arrowWing2.x,
arrowWing2.y),
    };
```

```
}

    draw(target: CanvasRenderingTarget2D) {
        target.useBitmapCoordinateSpace(scope => {
            if (this._points.length < 2) {
                return;
            }

            const ctx = scope.context;
            const calculateDrawingPoint = (point:
ViewPoint): ViewPoint => {
                return {
                    x: Math.round(point.x *
scope.horizontalPixelRatio),
                    y: Math.round(point.y *
scope.verticalPixelRatio)
                };
            };

            for (let i = 0; i < this._points.length;
i++) {
                this._points[i] =
calculateDrawingPoint(this._points[i]);
            }

            ctx.beginPath();
            ctx.lineWidth = 5;
            ctx.strokeStyle = this._fillColor;
            ctx.moveTo(this._points[0].x,
this._points[0].y);
            for (let i = 1; i < this._points.length;
i++) {
                ctx.lineTo(this._points[i].x,
this._points[i].y);
            }
            ctx.stroke();

            // drawing arrow
            const n = this._points.length;
            const p0: Point2D = new
Point2D(this._points[n - 2].x, this._points[n - 2].y);
            const p1: Point2D = new
Point2D(this._points[n - 1].x, this._points[n - 1].y);
```

```
            const arrowRenderInfo =
CalculateClassicArrowRenderInfo(p0, p1);

            ctx.beginPath();
            ctx.moveTo(arrowRenderInfo.arrowWing2.x,
arrowRenderInfo.arrowWing2.y);
            ctx.lineTo(arrowRenderInfo.arrowBase.x,
arrowRenderInfo.arrowBase.y);
            ctx.lineTo(arrowRenderInfo.arrowWing1.x,
arrowRenderInfo.arrowWing1.y);
            ctx.stroke();
        });
```