Лабораторная работа 3

Векторный редактор `Zivro`

1. Цель работы

1.1. Дидактическая цель

Овладеть навыками разработки программных модулей, реализующих алгоритмы растровой развёртки (растеризации) контуров фигур и сплошных областей на плоскости, а также получить опыт эмпирического анализа вычислительной трудоёмкости и точности алгоритмов растеризации.

1.2. Практическая цель

Разработать и исследовать программный модуль, выполняющий:

- растеризацию контура фигуры;
- растеризацию (закраску) сплошной области;
- формирование и отображение растрового изображения;
- сбор эмпирических данных по времени выполнения и различию результатов алгоритмов.

В рамках данной работы в качестве программной базы используется проект `Zivro`, разработанный автором специально для выполнения лабораторной работы, в который интегрированы и исследуются соответствующие алгоритмы.

2. Постановка задачи

В рамках лабораторной работы требуется реализовать и проверить функционал, соответствующий методическим указаниям ЛР №3:

1. Реализовать программный модуль растеризации контура и внутренней области фигуры на растровом изображении.
2. Обеспечить вывод результата в окне/холсте приложения с попиксельной визуализацией.
3. Реализовать не менее двух алгоритмов построения контура и не менее двух алгоритмов закраски (в соответствии с вариантом).
4. Добавить эталонный контурный алгоритм (например, Брезенхейм) для сравнения.
5. Реализовать сбор эмпирических данных:
   - время выполнения алгоритмов;
   - различие получаемых растровых развёрток (по отличающимся пикселям).
6. Подготовить и оформить отчёт по выполненной работе.
7. Сформировать приложение с исходными текстами автоматически в отдельный `.md` без изменения исходного файла отчёта.

3. Краткое описание проекта

Для выполнения ЛР в качестве основы используется проект `Zivro` - настольное графическое приложение для работы с векторными объектами (линия, эллипс, ломаная), с собственной моделью документа, иерархией объектов и CPU-рендерингом.

В контексте ЛР проект используется не как объект абстрактного обзора, а как программная платформа, где реализуются и исследуются алгоритмы растеризации контура и закраски.

Приложение использует:

- язык `Zig`;
- UI-библиотеку `dvui` с SDL3-бэкендом;
- модульную архитектуру (UI, модель данных, рендер, сериализация, инструменты).

Точка входа находится в `src/main.zig`: создаётся окно, инициализируется `WindowContext`, далее выполняется event/render loop и отрисовка растрового результата алгоритмов.

4. Структура проекта

Основные каталоги и файлы:

- `build.zig` - сценарий сборки и шагов `run`/`test`;
- `src/main.zig` - запуск приложения и главный цикл;
- `src/WindowContext.zig` - управление открытыми документами и активным документом;
- `src/Canvas.zig` - логика холста, масштабирование, вычисление видимой области, запуск рендера;
- `src/models/*` - модель документа, объектов и их свойств;
- `src/render/*` - CPU-рендер и конвейер отрисовки;
- `src/persistence/json_io.zig` - сохранение/загрузка документов в JSON;
- `src/ui/*` - интерфейсные панели и компоновка экрана;
- `src/tests.zig` - entry-point тестов.

5. Архитектура приложения

5.1. Высокоуровневая схема

Архитектурно проект разделён на три уровня:

1. **UI-уровень** (`src/ui`, `src/Canvas.zig`)
   Обрабатывает пользовательские события, отображает панели и холст, передаёт действия в модель и рендер.
2. **Модель данных** (`src/models`)
   Хранит документ, дерево объектов и свойства (позиция, угол, масштаб, цвет, точки, радиусы и др.).
3. **Рендер-уровень** (`src/render`)
   Преобразует модель объектов в набор пикселей видимой области и формирует текстуру.

5.2. Управление документами

`WindowContext` хранит массив открытых документов (`OpenDocument`) и индекс активного документа.
Каждый `OpenDocument` содержит:

- `Document`;
- экземпляр `CpuRenderEngine`;
- `Canvas`;
- идентификатор выбранного объекта.

Это позволяет одновременно работать с несколькими документами во вкладках.

5.3. Рендер-конвейер

Ключевой путь рендера:

1. `Canvas.redraw()` вычисляет масштаб качества и видимую часть документа;
2. вызывается `RenderEngine.render(...)` (в текущей конфигурации CPU-вариант);
3. `CpuRenderEngine.renderDocument(...)` подготавливает пиксельный буфер;
4. `cpu/draw.zig` рекурсивно обходит объекты документа;
5. для каждого объекта применяется `Transform.compose(parent, local)`;
6. shape-специфичные модули рисуют примитивы в буфер;
7. буфер превращается в текстуру UI.

6. Математические алгоритмы проекта (подробное текстовое описание)

В данном разделе подробно рассмотрены алгоритмы, которые формируют геометрию и цвет в CPU-рендере.
Диаграммы PlantUML будут добавлены отдельным шагом, после фиксации текстовой части.

6.1. Иерархические трансформации объектов

В основе рендера лежит сцена-дерево: каждый объект может иметь дочерние элементы.
Следовательно, координаты дочернего объекта заданы не в мировой системе, а в системе координат родителя.

В `Transform` хранятся:

- `position = (tx, ty)` - перенос;
- `angle = a` - поворот;
- `scale = (sx, sy)` - независимый масштаб по осям;
- `opacity = o` - накопленная непрозрачность.

Для перехода из локальных координат объекта к мировым используется аффинное преобразование:

$$x_w = t_x + (x_l \cdot s_x)\cos a - (y_l \cdot s_y)\sin a,$$
$$y_w = t_y + (x_l \cdot s_x)\sin a + (y_l \cdot s_y)\cos a.$$

При композиции `parent * local` (в `Transform.compose`) выполняются шаги:

1. локальная позиция сначала масштабируется масштабом родителя;
2. результат поворачивается на угол родителя;
3. добавляется перенос родителя;
4. углы складываются: `a_world = a_parent + a_local`;
5. масштабы перемножаются покомпонентно: `sx_world = sx_parent * sx_local`, `sy_world = sy_parent * sy_local`;
6. прозрачности перемножаются: `o_world = o_parent * o_local`.

Почему это важно: такой порядок гарантирует, что при повороте/масштабе группы объектов дочерние элементы ведут себя как единая связанная конструкция.

6.2. Цепочка преобразований координат в рендере

Рендер использует 4 системы координат:

1. **локальная** (внутри shape);
2. **мировая** (внутри документа);
3. **координаты канвы** (после масштабирования документа под текущий размер);
4. **координаты буфера viewport** (видимая часть, начинающаяся с `(0,0)`).

Основные формулы:

- `canvas_x = world_x * scale_x`, `canvas_y = world_y * scale_y`;
- `buffer_x = canvas_x - visible_rect.x`, `buffer_y = canvas_y - visible_rect.y`.

Обратное преобразование также используется (например, в эллипсе):

- `canvas_x = buffer_x + visible_rect.x`, `canvas_y = buffer_y + visible_rect.y`;
- `world_x = canvas_x / scale_x`, `world_y = canvas_y / scale_y`.

Для вычисления локальных координат из мировых применяется обратный поворот и обратный масштаб:

$$dx = x_w - t_x, \quad dy = y_w - t_y,$$
$$x_l = \frac{dx\cos(-a) - dy\sin(-a)}{s_x},$$
$$y_l = \frac{dx\sin(-a) + dy\cos(-a)}{s_y}.$$

Практический смысл: shape можно тестировать аналитически (по формулам) в “своей” удобной локальной системе, независимо от того, как он повернут и где расположен в документе.

6.3. Рисование линии: отсечение + дискретизация + толщина

Алгоритм в `line.zig` состоит из трёх частей.

6.3.1. Отсечение Liang-Barsky Перед рисованием линия отсекается расширенным прямоугольником буфера.
Параметрическая форма отрезка:

$$P(t) = P_0 + t(P_1 - P_0), \quad t \in [0, 1].$$

Для каждого ограничения (`x >= left`, `x <= right`, `y >= top`, `y <= bottom`) обновляется допустимый интервал `[t0, t1]`.
Если после обработки ограничений `t0 > t1`, отрезок полностью вне экрана и пропускается.

Преимущество: вместо “шагать по пикселям и каждый проверять границы” рендер сразу работает только с видимым отрезком.

6.3.2. Дискретизация линии (Bresenham-подобный проход) После отсечения используются целочисленные приращения:

- `dx = abs(x1 - x0)`, `dy = -abs(y1 - y0)`;
- `sx = sign(x1 - x0)`, `sy = sign(y1 - y0)`;
- ошибка `err = dx + dy`.

На каждом шаге:

- вычисляется `e2 = 2*err`;
- если `e2 >= dy`, двигаемся по `x`;
- если `e2 <= dx`, двигаемся по `y`.

Это классическая идея целочисленного интегрирования ошибки для аппроксимации идеального непрерывного отрезка на пиксельной сетке.

6.3.3. Толщина и коррекция по углу Если просто расширять линию равномерно, визуальная толщина может “плыть” при разных углах.
В проекте вычисляется поправка по длине проекций:

- `cos(theta) = |dx| / len`;
- `sin(theta) = |dy| / len`;
- выбирается базис (по X или Y), где ошибка толщины минимальна;
- итоговая толщина пересчитывается через деление на `max(sin, eps)` или `max(cos, eps)`.

Далее вокруг центрального пикселя проводится полоса ширины `thickness_corrected`.

Дополнительно есть режим `draw_when_outside`:

- внутри viewport рисуется полная толщина;
- за пределами viewport — только 1px, чтобы контур не “взрывался” по ширине за экраном.

6.4. Растрирование эллипса и дуги

Алгоритм `ellipse.zig` не использует инкрементальные midpoint-формулы, а работает через аналитическую проверку каждого пикселя в ограничивающем прямоугольнике.

6.4.1. Нормализация координат Для пикселя вычисляются локальные координаты `loc = (x_l, y_l)` и нормализуются:

$$n_x = \frac{x_l}{r_x}, \quad n_y = \frac{y_l}{r_y}, \quad d = n_x^2 + n_y^2.$$

- `d = 1` соответствует идеальному контуру эллипса;
- `d < 1` внутри;
- `d > 1` снаружи.

6.4.2. Полоса обводки заданной толщины Толщина `thickness` переводится в нормированное пространство через меньшую полуось:

- `half_norm = thickness / (2*min(rx, ry));`
- внутренний радиус: `inner = max(0, 1 - half_norm);`
- внешний радиус: `outer = 1 + half_norm`.

Пиксель принадлежит обводке, если:

$$inner^2 \leq d \leq outer^2.$$

Это даёт геометрически корректную полосу вокруг эллипса при произвольном повороте/масштабе объекта.

6.4.3. Дуга через угловой фильтр Если `arc_percent < 100`, из полного эллипса берётся только часть:

- вычисляется длина дуги в радианах: `arc_len = 2*pi*arc_percent/100;`
- для пикселя находится угол через `atan2(ny, nx)` (с поправкой на экранную систему);
- точка принимается только если её угловая позиция не превышает `arc_len`.

Если `closed = true`, концы дуги соединяются с центром двумя радиальными отрезками (используется общий алгоритм линии).

6.5. Ломаная, выделение границы и заливка

В `broken.zig` ломаная строится как цепочка сегментов `P0->P1->...->Pn`, а при `closed` добавляется `Pn->P0`.

Для корректной заливки применяется двухфазный алгоритм.

6.5.1. Фаза 1: сбор пикселей границы Во временном `FillCanvas` при каждом `blendPixelAtBuffer` сохраняется координата пикселя как граничная точка (`border_set`).

Смысл: сначала зафиксировать “жёсткий” контур, затем независимо заполнить внутренность.

6.5.2. Фаза 2: поиск внутренних сегментов по строкам Граничные пиксели сортируются по `(y, x)`. Для каждой строки:

1. выделяются последовательности рёбер;
2. строятся интервалы между соседними рёбрами;
3. из подходящих интервалов выбираются seed-точки (середина интервала).

Это снижает риск старта flood fill с внешней стороны фигуры.

6.5.3. Фаза 3: flood fill (4-связность) От каждого seed выполняется стековый обход соседей:

- влево, вправо, вверх, вниз;
- граничные пиксели не пересекаются;
- уже посещённые пиксели пропускаются.

Каждый найденный внутренний пиксель окрашивается в `fill_color`.

Почему используется отдельный буфер: при полупрозрачности иначе одна и та же область может смешаться несколько раз из-за пересечения сегментов.

6.6. Альфа-смешивание в Premultiplied Alpha (PMA)

Для каждого канала цвета применяется модель:

$$C_{out} = C_{src} + (1 - \alpha_{src})C_{dst}, \quad \alpha_{out} = \alpha_{src} + (1 - \alpha_{src})\alpha_{dst}.$$

В коде `C_src` уже premultiplied (или домножается на opacity трансформа в момент смешивания).

Пошагово:

1. берётся альфа источника `a = src_a/255 * transform.opacity`;
2. вычисляется `inv_a = 1 - a`;
3. каналы `r,g,b` источника масштабируются на `transform.opacity`;
4. формируется новый `dst` по PMA-формуле.

`replace_mode = true` отключает смешивание и просто заменяет пиксель.
Этот режим используется во временных буферах shape-рендера, а затем результат один раз композится в целевой буфер.

6.7. Численная устойчивость и ограничения

В алгоритмах предусмотрены защиты от деградации вычислений:

- защита от деления на ноль в обратных преобразованиях (`scale == 0 -> 1.0`);
- использование `eps` в коррекции толщины линий;
- ограничение минимальных размеров рендер-буфера (`>= 1 px`);
- отсечение слишком больших выходов за viewport до начала растрирования;
- явное округление float->int в точках, где нужна стабильная пиксельная привязка.

Это снижает число визуальных артефактов при малых масштабах, сильных поворотах и частичной видимости объектов.

7. Работа с данными и сериализация

Модуль `src/persistence/json_io.zig` поддерживает:

- `saveToFile(...)` - сериализация в JSON (pretty-print);
- `loadFromFile(...)` - чтение JSON и восстановление структуры.

Для `Document` после парсинга выполняется клонирование, чтобы избежать проблем владения памятью (парсер выделяет память из арены).

8. Сборка, запуск и тестирование

8.1. Сборка и запуск

```
zig build
zig build run
```

8.2. Запуск тестов

```
zig build test
```

Файл `src/tests.zig` подключает модули с `test`-блоками, чтобы они выполнялись в составе общего тестового шага.

9. Автоматическое формирование приложения с исходным кодом

Чтобы не вставлять исходники вручную в конец отчёта, используется скрипт `Report/append_sources_to_report`

Скрипт:

- читает исходный `.md` отчёт;
- добавляет раздел с кодом файлов проекта;
- перед каждым листингом вставляет путь файла;
- сохраняет результат в новый `.md` файл;
- исходный отчёт не изменяет.

Пример запуска:

```
python3 Report/append_sources_to_report.py \
  --input Report/zivro-open-project-report.md \
  --output Report/zivro-open-project-report-with-code.md \
  --base .
```

10. PlantUML-диаграммы

Для отчёта подготовлены диаграммы в формате PlantUML (`.puml`) и сгенерированы их PNG-версии для прямой вставки в документ.

10.1. Архитектура проекта

`Report/uml/zivro-architecture-components.puml` - компонентная архитектура приложения.

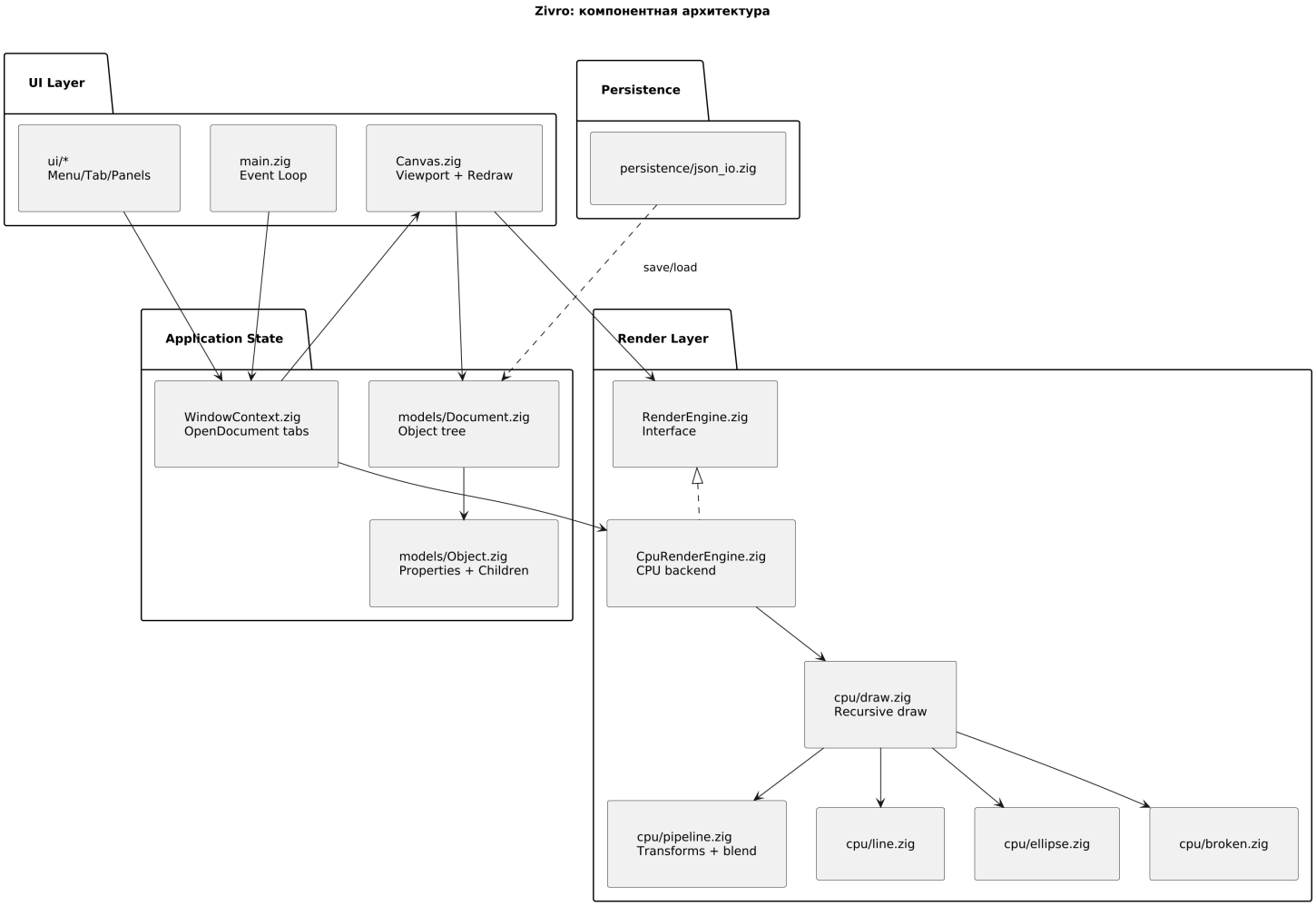

Figure 1: Компонентная архитектура Zivro

`Report/uml/zivro-render-sequence.puml` - последовательность рендера кадра.

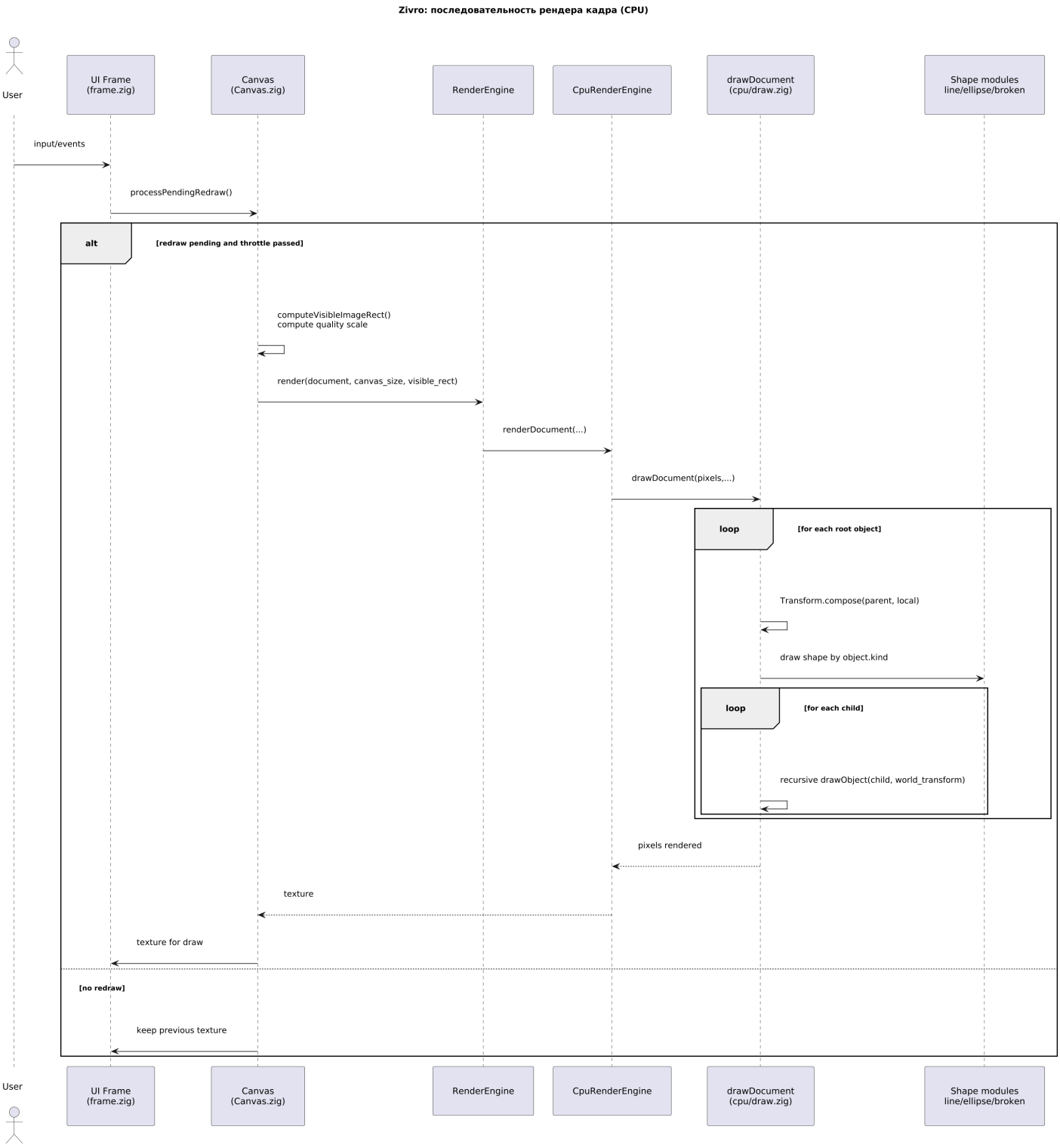

Figure 2: Последовательность рендера кадра

10.2. Управление холстом и viewport

`Report/uml/canvas-viewport-algorithm.puml` - вычисление видимой области, масштабирование по качеству, условия редроу.

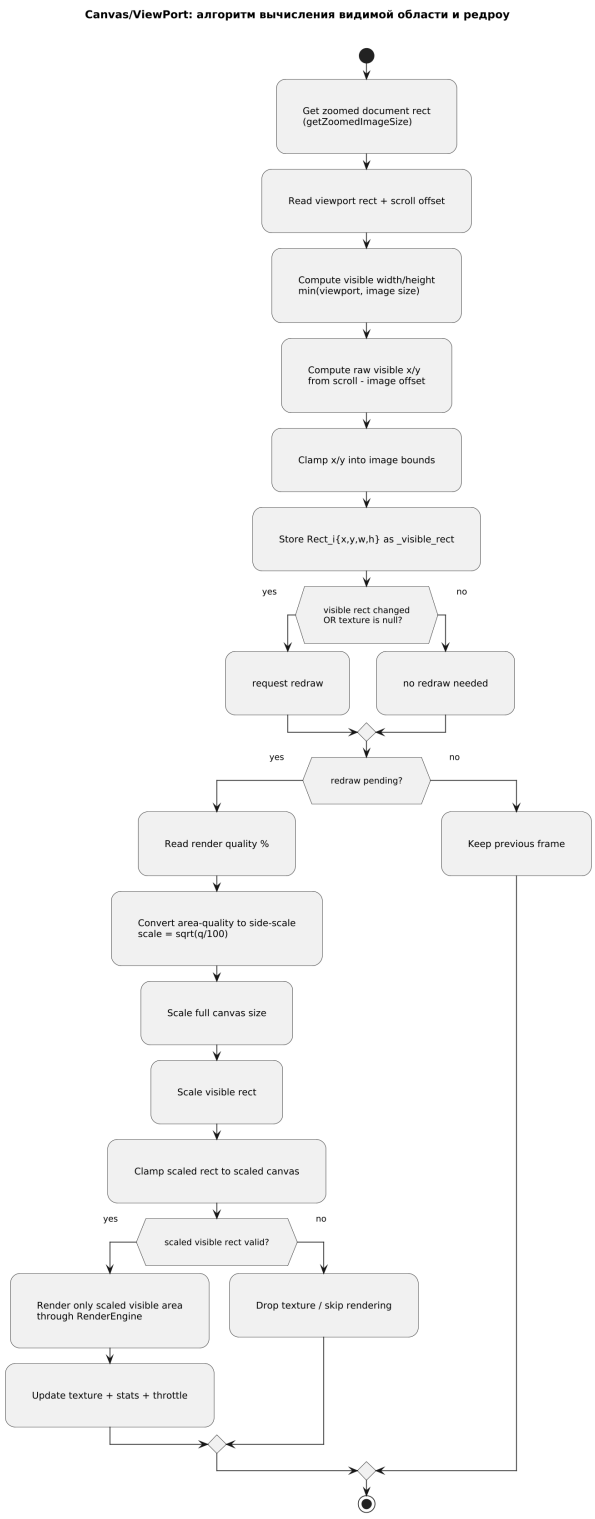

Figure 3: Алгоритм управления viewport и redraw

10.3. Алгоритмы обработки данных и растризации

`Report/uml/transform-compose-algorithm.puml` - композиция трансформаций в иерархии объектов.

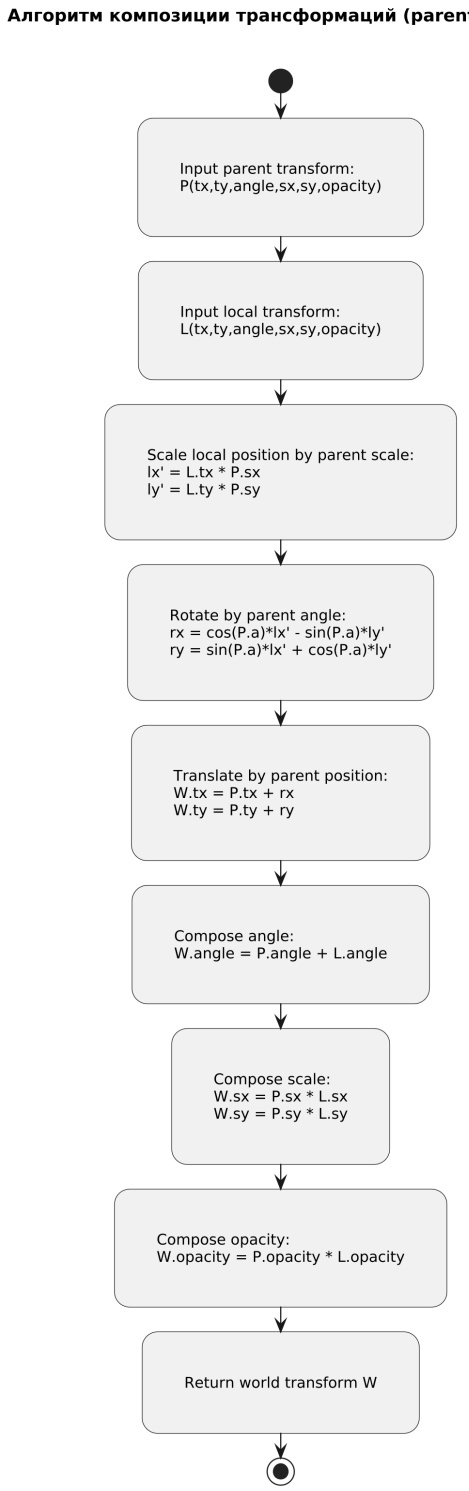

Figure 4: Композиция трансформаций

`Report/uml/coordinate-pipeline.puml` - конвейер преобразований координат.

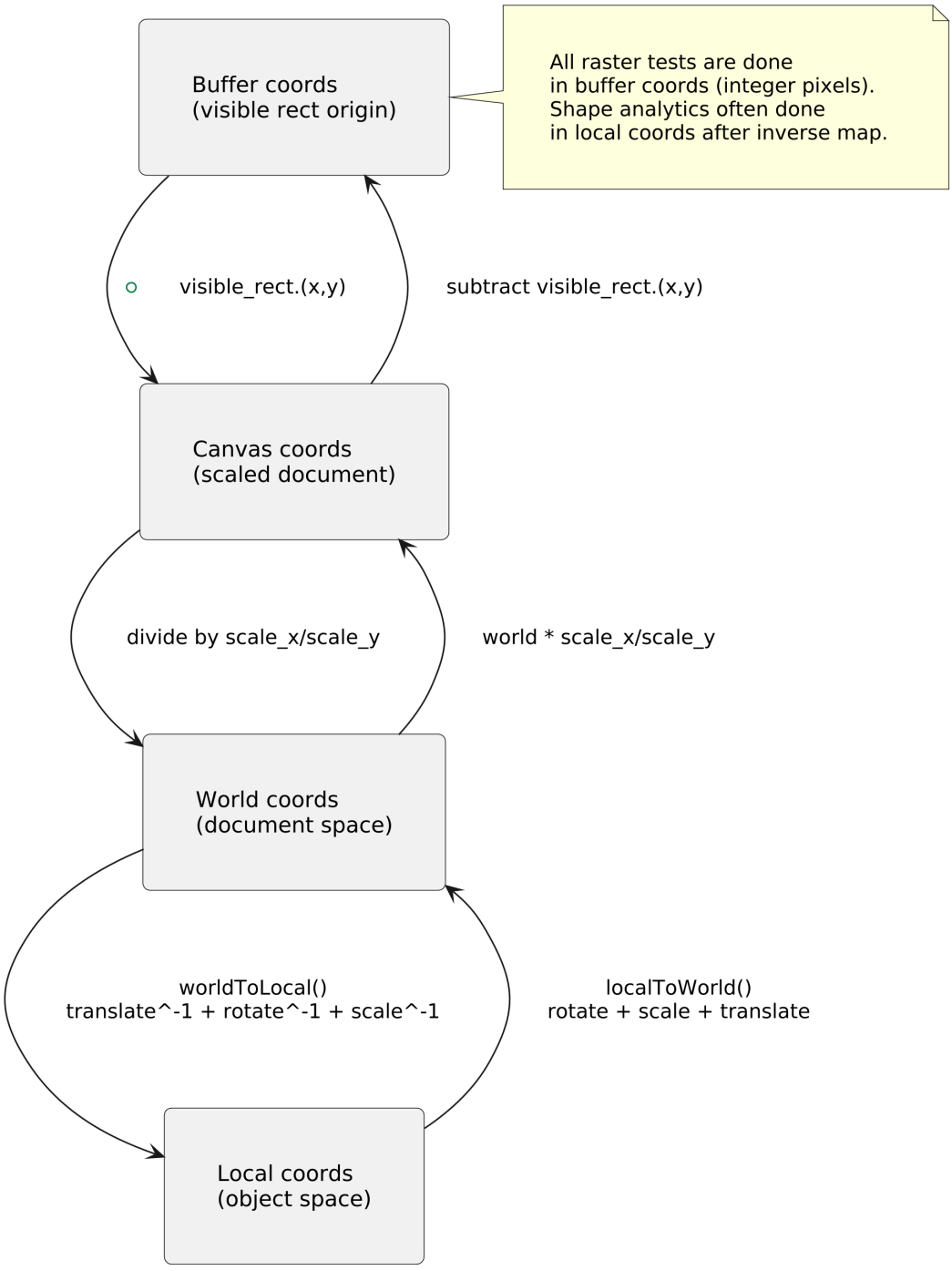

Figure 5: Конвейер преобразований координат

`Report/uml/line-rasterization-flow.puml` - полный алгоритм отрисовки линии.

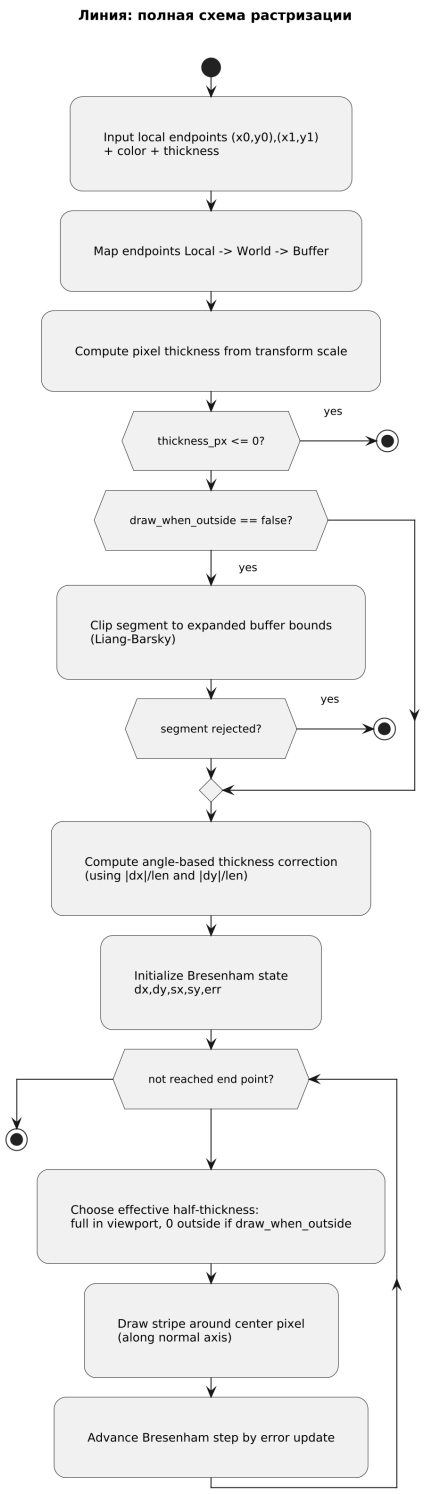

Figure 6: Полный алгоритм растрирования линии

`Report/uml/liang-barsky-clip.puml` - шаги отсечения отрезка методом Liang-Barsky.

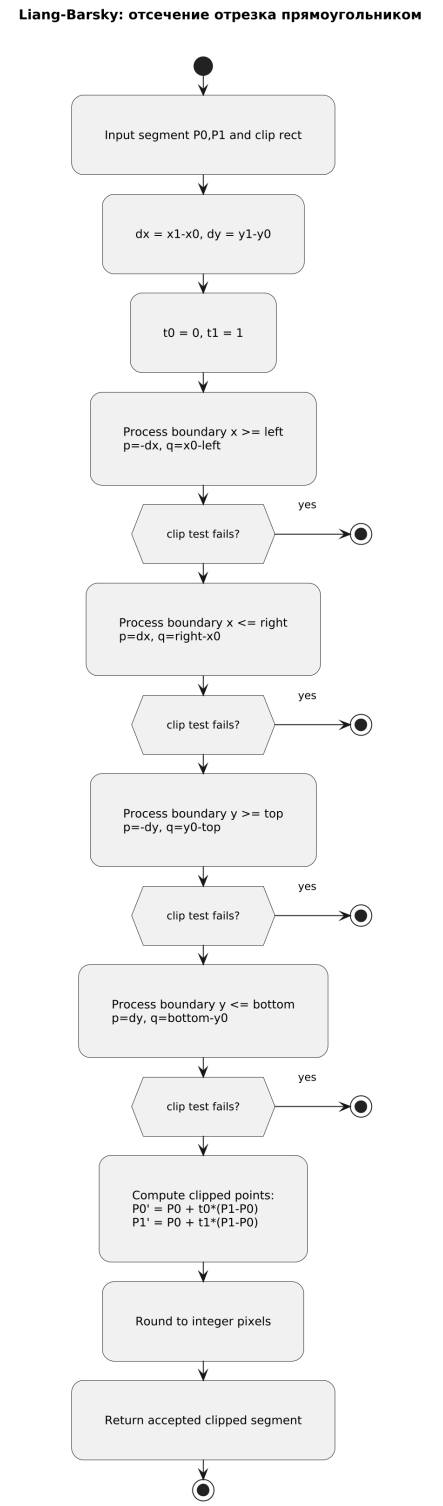

Figure 7: Алгоритм Liang-Barsky

`Report/uml/ellipse-arc-rasterization.puml` - растрирование эллипса/дуги.

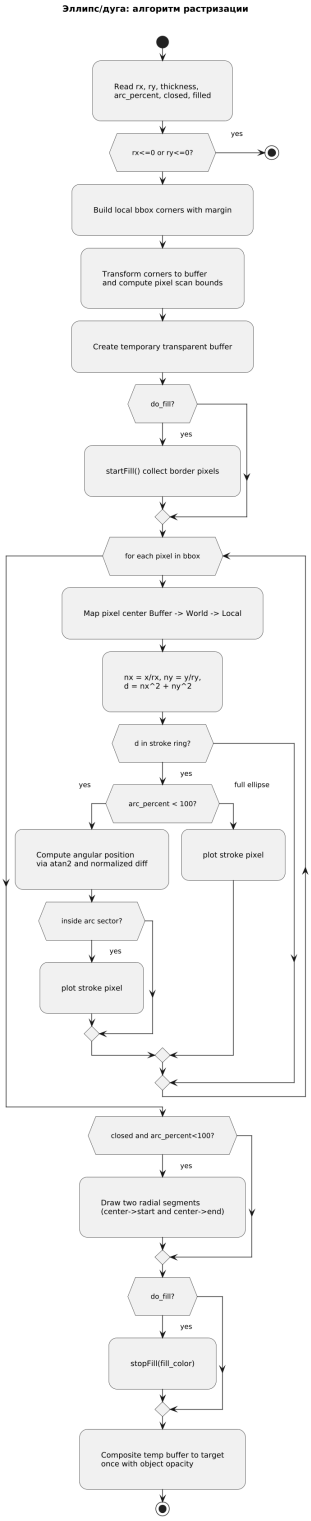

Figure 8: Алгоритм растрирования эллипса и дуги

`Report/uml/polyline-fill-algorithm.puml` - построение и заливка замкнутой ломаной.

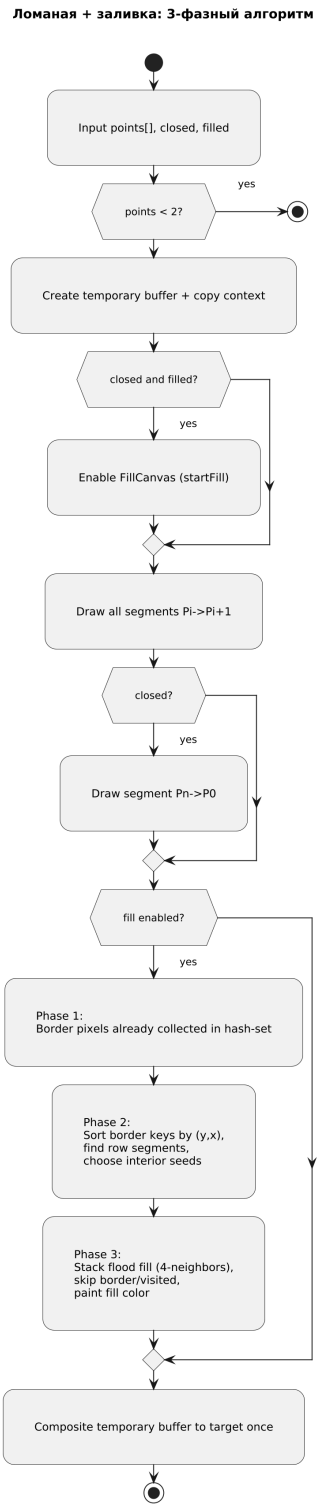

Figure 9: Алгоритм ломаной и заливки

`Report/uml/pma-alpha-blending.puml` - PMA alpha-композиция пикселей.

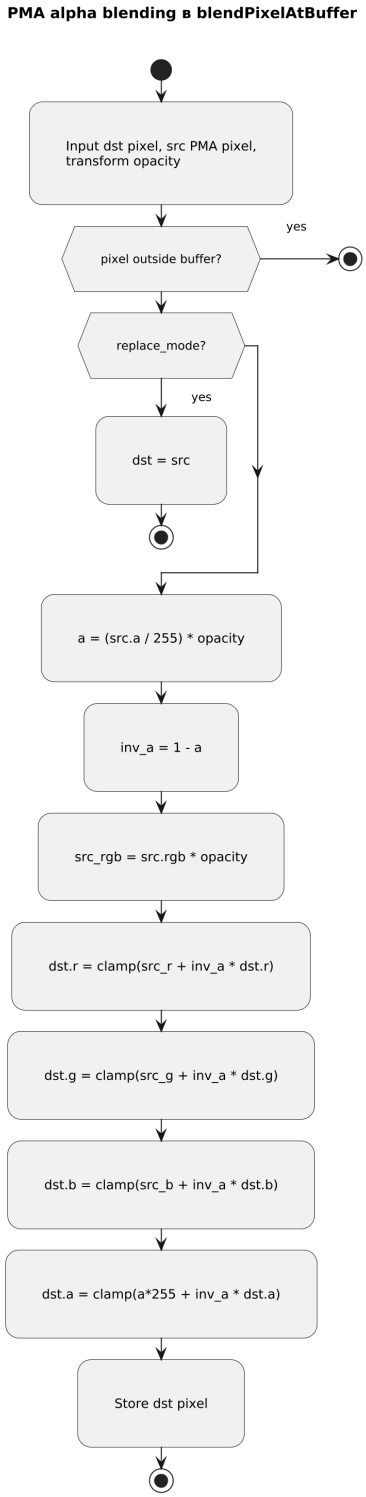

Figure 10: PMA alpha-композиция пикселя

10.4. Скрипт генерации PNG-диаграмм

Для повторной генерации PNG сохранён отдельный скрипт:

- `Report/render_uml_png.py`

Пример запуска в высоком качестве:

```
python3 Report/render_uml_png.py --input-dir Report/uml --dpi 360
```

11. Выводы

В ходе работы разработан и исследован проект Zivro, подготовлено структурированное описание его архитектуры. Проект реализует модульный подход: модель документа, иерархию объектов, CPU-рендер с преобразованиями координат и отдельные алгоритмы растеризации геометрии.

Ключевые математические части - композиция трансформаций, отсечение и растеризация линий, рендер эллипсов/дуг, а также алгоритм заливки замкнутых контуров.
Текстовый отчёт подготовлен без встроенных листингов кода; для генерации версии с приложением исходников создан отдельный автоматизированный скрипт.

Приложение A. Исходные тексты

Сформировано автоматически скриптом Report/append_sources_to_report.py (файлов: 39).

A.1. build.zig

```
const std = @import("std");

pub fn build(b: *std.Build) void {
    const target = b.standardTargetOptions(.{});
    const optimize = b.standardOptimizeOption(.{});

    const dvui_dep = b.dependency("dvui", .{ .target = target, .optimize = optimize, .backend = .sdl3 });

    const exe = b.addExecutable(.{
        .name = "Zivro",
        // .use_llvm = true,
        // .use_lld = true,
        .root_module = b.createModule(.{
            .root_source_file = b.path("src/main.zig"),
            .target = target,
            .optimize = optimize,
            .imports = &.{
                .{ .name = "dvui", .module = dvui_dep.module("dvui_sdl3") },
                .{ .name = "sdl-backend", .module = dvui_dep.module("sdl3") },
            },
        }),
    });
    exe.bundle_compiler_rt = true;
    b.installArtifact(exe);

    const run_step = b.step("run", "Run the app");

    const run_cmd = b.addRunArtifact(exe);
    run_step.dependOn(&run_cmd.step);

    run_cmd.step.dependOn(b.getInstallStep());

    if (b.args) |args| {
        run_cmd.addArgs(args);
    }

    const exe_tests = b.addTest(.{
        .root_module = b.createModule(.{
            .root_source_file = b.path("src/tests.zig"),
            .target = target,
            .optimize = optimize,

            .imports = &.{
                .{ .name = "dvui", .module = dvui_dep.module("dvui_sdl3") },
                .{ .name = "sdl-backend", .module = dvui_dep.module("sdl3") },
            },
        }),
```

```
    });

    const run_exe_tests = b.addRunArtifact(exe_tests);

    const test_step = b.step("test", "Run tests");
    test_step.dependOn(&run_exe_tests.step);
}
```

A.2. build.zig.zon

```
.{
    // This is the default name used by packages depending on this one. For
    // example, when a user runs `zig fetch --save <url>`, this field is used
    // as the key in the `dependencies` table. Although the user can choose a
    // different name, most users will stick with this provided value.
    //
    // It is redundant to include "zig" in this name because it is already
    // within the Zig package namespace.
    .name = .Zivro,
    // This is a [Semantic Version](https://semver.org/).
    // In a future version of Zig it will be used for package deduplication.
    .version = "0.0.0",
    // Together with name, this represents a globally unique package
    // identifier. This field is generated by the Zig toolchain when the
    // package is first created, and then *never changes*. This allows
    // unambiguous detection of one package being an updated version of
    // another.
    //
    // When forking a Zig project, this id should be regenerated (delete the
    // field and run `zig build`) if the upstream project is still maintained.
    // Otherwise, the fork is *hostile*, attempting to take control over the
    // original project's identity. Thus it is recommended to leave the comment
    // on the following line intact, so that it shows up in code reviews that
    // modify the field.
    .fingerprint = 0x62d7eba14686811e, // Changing this has security and trust implications.
    // Tracks the earliest Zig version that the package considers to be a
    // supported use case.
    .minimum_zig_version = "0.15.2",
    // This field is optional.
    // Each dependency must either provide a `url` and `hash`, or a `path`.
    // `zig build --fetch` can be used to fetch all dependencies of a package, recursively.
    // Once all dependencies are fetched, `zig build` no longer requires
    // internet connectivity.
    .dependencies = .{
        .dvui = .{
        .url = "git+https://github.com/david-vanderson/dvui#edb2d5a4cd2981fca74ee5f096277b91333c1316",
            .hash = "dvui-0.4.0-dev-AQFJmeGB3QAwun9qF76CDE5IopA4nUVRgD-IwwTsOo4H",
        },

        // See `zig fetch --save <url>` for a command-line interface for adding dependencies.
        //.example = .{
        //    // When updating this field to a new URL, be sure to delete the corresponding
        //    // `hash`, otherwise you are communicating that you expect to find the old hash at
        //    // the new URL. If the contents of a URL change this will result in a hash mismatch
        //    // which will prevent zig from using it.
        //    .url = "https://example.com/foo.tar.gz",
        //
        //    // This is computed from the file contents of the directory of files that is
        //    // obtained after fetching `url` and applying the inclusion rules given by
        //    // `paths`.
        //    //
```

```
        //    // This field is the source of truth; packages do not come from a `url`; they
        //    // come from a `hash`. `url` is just one of many possible mirrors for how to
        //    // obtain a package matching this `hash`.
        //    //
        //    // Uses the [multihash](https://multiformats.io/multihash/) format.
        //    .hash = "...",
        //
        //    // When this is provided, the package is found in a directory relative to the
        //    // build root. In this case the package's hash is irrelevant and therefore not
        //    // computed. This field and `url` are mutually exclusive.
        //    .path = "foo",
        //
        //    // When this is set to `true`, a package is declared to be lazily
        //    // fetched. This makes the dependency only get fetched if it is
        //    // actually used.
        //    .lazy = false,
        //},
    },
    // Specifies the set of files and directories that are included in this package.
    // Only files and directories listed here are included in the `hash` that
    // is computed for this package. Only files listed here will remain on disk
    // when using the zig package manager. As a rule of thumb, one should list
    // files required for compilation plus any license(s).
    // Paths are relative to the build root. Use the empty string (`""`) to refer to
    // the build root itself.
    // A directory listed here means that all files within, recursively, are included.
    .paths = .{
        "build.zig",
        "build.zig.zon",
        "src",
        // For example...
        //"LICENSE",
        //"README.md",
    },
}
```

A.3. src/Canvas.zig

```
const std = @import("std");
const builtin = @import("builtin");
const dvui = @import("dvui");
const Document = @import("models/Document.zig");
const RenderEngine = @import("render/RenderEngine.zig").RenderEngine;
const basic_models = @import("models/basic_models.zig");
const Rect_i = basic_models.Rect_i;
const Size_i = basic_models.Size_i;
const Point2_f = @import("models/basic_models.zig").Point2_f;
const Color = dvui.Color;
const tools = @import("toolbar/tools.zig");
const Toolbar = @import("toolbar/Toolbar.zig");
const random_document = @import("random_document.zig");
const Canvas = @This();

allocator: std.mem.Allocator,
document: *Document,
render_engine: RenderEngine,
toolbar: Toolbar,
texture: ?dvui.Texture = null,
pos: dvui.Point = dvui.Point{ .x = 400, .y = 400 },
scroll: dvui.ScrollInfo = .{
    .vertical = .auto,
    .horizontal = .auto,
},
```

```
native_scaling: bool = true,
cursor_document_point: ?Point2_f = null,
draw_document: bool = true,
show_render_stats: bool = true,
/// Rect тулбара (из предыдущего кадра) для исключения кликов по нему из handleCanvasMouse.
toolbar_rect_scale: ?dvui.RectScale = null,
/// Rect панели свойств (из предыдущего кадра) для исключения кликов по нему из handleCanvasMouse.
properties_rect_scale: ?dvui.RectScale = null,
redraw_throttle_ms: u32 = 50,
frame_index: u64 = 0,

_zoom: f32 = 1,
_rendering_quality: f32 = 100.0,
_last_redraw_time_ms: i64 = 0, // Метка последней перерисовки чтобы ограничить частоту
_visible_rect: ?Rect_i = null,
_redraw_pending: bool = false,

pub fn init(allocator: std.mem.Allocator, document: *Document, engine: RenderEngine) Canvas {
    return .{
        .allocator = allocator,
        .document = document,
        .render_engine = engine,
        .toolbar = Toolbar.init(&tools.default_tools),
    };
}

pub fn deinit(self: *Canvas) void {
    self.toolbar.deinit();
    if (self.texture) |texture| {
        dvui.Texture.destroyLater(texture);
        self.texture = null;
    }
}

fn redraw(self: *Canvas) !void {
    const full = self.getZoomedImageSize();

    const vis_full: Rect_i = self._visible_rect orelse Rect_i{ .x = 0, .y = 0, .w = 0, .h = 0 };

    if (vis_full.w == 0 or vis_full.h == 0) {
        if (self.texture) |tex| {
            dvui.Texture.destroyLater(tex);
            self.texture = null;
        }
        return;
    }

    // Качество рендеринга задаётся в процентах площади (1–100),
    // при этом фактически уменьшаем ширину/высоту холста на корень из этой доли.
    const quality_percent: f32 = self.getRenderingQuality();
    const quality_area: f32 = quality_percent / 100.0;
    const quality_side: f32 = std.math.sqrt(quality_area);
    const scale: f32 = std.math.clamp(quality_side, 0.01, 1.0);

    const canvas_size: Size_i = .{
        .w = @max(@as(u32, 1), @as(u32, @intFromFloat(@as(f32, @floatFromInt(full.w)) * scale))),
        .h = @max(@as(u32, 1), @as(u32, @intFromFloat(@as(f32, @floatFromInt(full.h)) * scale))),
    };

    var vis_scaled = Rect_i{
        .x = @as(u32, @intFromFloat(@as(f32, @floatFromInt(vis_full.x)) * scale)),
        .y = @as(u32, @intFromFloat(@as(f32, @floatFromInt(vis_full.y)) * scale)),
        .w = @max(@as(u32, 1), @as(u32, @intFromFloat(@as(f32, @floatFromInt(vis_full.w)) * scale))),
        .h = @max(@as(u32, 1), @as(u32, @intFromFloat(@as(f32, @floatFromInt(vis_full.h)) * scale))),
    };

    if (vis_scaled.x >= canvas_size.w or vis_scaled.y >= canvas_size.h) {
```

```
        if (self.texture) |tex| {
            dvui.Texture.destroyLater(tex);
            self.texture = null;
        }
        return;
    }

    const max_vis_w: u32 = canvas_size.w - vis_scaled.x;
    const max_vis_h: u32 = canvas_size.h - vis_scaled.y;
    if (vis_scaled.w > max_vis_w) vis_scaled.w = max_vis_w;
    if (vis_scaled.h > max_vis_h) vis_scaled.h = max_vis_h;

    const new_texture = if (self.draw_document)
        self.render_engine.render(self.document, canvas_size, vis_scaled) catch null
    else
        self.render_engine.example(canvas_size, vis_scaled) catch null;

    if (new_texture) |tex| {
        if (self.texture) |old_tex| {
            dvui.Texture.destroyLater(old_tex);
        }

        self.texture = tex;
    }
    self._last_redraw_time_ms = std.time.milliTimestamp();
    self.frame_index += 1;
    self.redraw_throttle_ms = @max(1, @as(u32, @intCast(self.render_engine.getStats().render_time_ns /
     ↪  std.time.ns_per_ms / 3)));
}

pub fn exampleReset(self: *Canvas) !void {
    self.render_engine.exampleReset();
    try self.redraw();
}

pub fn addRandomShapes(self: *Canvas) !void {
    try random_document.addRandomShapes(self.document, self.allocator, std.crypto.random);
    self.requestRedraw();
}

pub fn setZoom(self: *Canvas, value: f32) void {
    self._zoom = @max(value, 0.01);
}

pub fn addZoom(self: *Canvas, value: f32) void {
    self._zoom += value;
    self._zoom = @max(self._zoom, 0.01);
}

pub fn multZoom(self: *Canvas, value: f32) void {
    self._zoom *= value;
    self._zoom = @max(self._zoom, 0.01);
}

pub fn getZoom(self: Canvas) f32 {
    return self._zoom;
}

pub fn setRenderingQuality(self: *Canvas, value: f32) void {
    self._rendering_quality = std.math.clamp(value, 1.0, 100.0);
    self.requestRedraw();
}

pub fn getRenderingQuality(self: Canvas) f32 {
    return self._rendering_quality;
}
```

```
pub fn requestRedraw(self: *Canvas) void {
    self._redraw_pending = true;
}

pub fn processPendingRedraw(self: *Canvas) !void {
    if (!self._redraw_pending) return;
    if (self.redraw_throttle_ms == 0) {
        self._redraw_pending = false;
        try self.redraw();
        return;
    }
    const now_ms = std.time.milliTimestamp();
    const elapsed: i64 = if (self._last_redraw_time_ms == 0) self.redraw_throttle_ms else now_ms -
    ↪  self._last_redraw_time_ms;
    if (elapsed < @as(i64, @intCast(self.redraw_throttle_ms))) return;
    self._redraw_pending = false;
    try self.redraw();
}

pub fn getZoomedImageSize(self: Canvas) Rect_i {
    const doc = self.document;
    return .{
        .x = @intFromFloat(self.pos.x),
        .y = @intFromFloat(self.pos.y),
        .w = @intFromFloat(doc.size.w * self._zoom),
        .h = @intFromFloat(doc.size.h * self._zoom),
    };
}

/// Точка контента -> координаты документа.
pub fn contentPointToDocument(self: Canvas, content_point: dvui.Point, natural_scale: f32) Point2_f {
    const img = self.getZoomedImageSize();
    const px_x = content_point.x * natural_scale - @as(f32, @floatFromInt(img.x));
    const px_y = content_point.y * natural_scale - @as(f32, @floatFromInt(img.y));
    return .{
        .x = px_x / self._zoom,
        .y = px_y / self._zoom,
    };
}

/// Точка контента внутри холста.
pub fn isContentPointOnDocument(self: Canvas, content_point: dvui.Point, natural_scale: f32) bool {
    const img = self.getZoomedImageSize();
    const left_n = @as(f32, @floatFromInt(img.x)) / natural_scale;
    const top_n = @as(f32, @floatFromInt(img.y)) / natural_scale;
    const right_n = @as(f32, @floatFromInt(img.x + img.w)) / natural_scale;
    const bottom_n = @as(f32, @floatFromInt(img.y + img.h)) / natural_scale;
    return content_point.x >= left_n and content_point.x < right_n and
        content_point.y >= top_n and content_point.y < bottom_n;
}

pub fn updateVisibleImageRect(self: *Canvas, viewport: dvui.Rect, scroll_offset: dvui.Point) bool {
    const next = computeVisibleImageRect(self.*, viewport, scroll_offset);
    var changed = false;
    if (self._visible_rect) |vis| {
        changed |= next.x != vis.x or next.y != vis.y or next.w != vis.w or next.h != vis.h;
    }
    self._visible_rect = next;
    if (changed or self.texture == null) {
        return true;
    }
    return false;
}

fn computeVisibleImageRect(self: Canvas, viewport: dvui.Rect, scroll_offset: dvui.Point) Rect_i {
    const image_rect = self.getZoomedImageSize();
```

```
    const img_w: u32 = image_rect.w;
    const img_h: u32 = image_rect.h;

    const vis_w: u32 = @min(@as(u32, @intFromFloat(viewport.w)), img_w);
    const vis_h: u32 = @min(@as(u32, @intFromFloat(viewport.h)), img_h);

    const raw_x: i64 = @intFromFloat(scroll_offset.x - @as(f32, @floatFromInt(image_rect.x)));
    const raw_y: i64 = @intFromFloat(scroll_offset.y - @as(f32, @floatFromInt(image_rect.y)));

    const vis_x: u32 = @intCast(std.math.clamp(raw_x, 0, @as(i64, img_w) - @as(i64, vis_w)));
    const vis_y: u32 = @intCast(std.math.clamp(raw_y, 0, @as(i64, img_h) - @as(i64, vis_h)));

    return Rect_i{
        .x = vis_x,
        .y = vis_y,
        .w = vis_w,
        .h = vis_h,
    };
}
```

A.4. src/WindowContext.zig

```
const std = @import("std");
const Canvas = @import("Canvas.zig");
const CpuRenderEngine = @import("render/CpuRenderEngine.zig");
const RenderEngine = @import("render/RenderEngine.zig").RenderEngine;
const Document = @import("models/Document.zig");
const random_document = @import("random_document.zig");
const basic_models = @import("models/basic_models.zig");

const WindowContext = @This();

pub const OpenDocument = struct {
    document: Document,
    cpu_render: CpuRenderEngine,
    canvas: Canvas,
    /// Выбранный объект в дереве (id объекта).
    selected_object_id: ?u64 = null,

    pub fn init(allocator: std.mem.Allocator, self: *OpenDocument) void {
        initWithDocument(allocator, self, .init(.{
            .w = 800,
            .h = 600,
        }));
    }

    pub fn initWithDocument(allocator: std.mem.Allocator, self: *OpenDocument, doc: Document) void {
        self.document = doc;
        self.cpu_render = CpuRenderEngine.init(allocator, .Squares);
        self.canvas = Canvas.init(
            allocator,
            &self.document,
            (&self.cpu_render).renderEngine(),
        );
        self.selected_object_id = null;
    }

    pub fn deinit(self: *OpenDocument, allocator: std.mem.Allocator) void {
        self.document.deinit(allocator);
        self.canvas.deinit();
    }
};

allocator: std.mem.Allocator,
documents: std.ArrayList(*OpenDocument),
active_document_index: ?usize,
```

```zig
pub fn init(allocator: std.mem.Allocator) !WindowContext {
    const documents = std.ArrayList(*OpenDocument).empty;
    const active_document_index: ?usize = null;
    return .{
        .allocator = allocator,
        .documents = documents,
        .active_document_index = active_document_index,
    };
}

pub fn deinit(self: *WindowContext) void {
    for (self.documents.items) |ptr| {
        ptr.deinit(self.allocator);
        self.allocator.destroy(ptr);
    }
    self.documents.deinit(self.allocator);
}

pub fn activeDocument(self: *WindowContext) ?*OpenDocument {
    const i = self.active_document_index orelse return null;
    if (i >= self.documents.items.len) return null;
    return self.documents.items[i];
}

pub fn addNewDocument(self: *WindowContext) !void {
    const ptr = try self.allocator.create(OpenDocument);
    errdefer self.allocator.destroy(ptr);
    OpenDocument.init(self.allocator, ptr);
    //try random_document.addRandomShapes(&ptr.document, std.crypto.random);
    try self.documents.append(self.allocator, ptr);
    self.active_document_index = self.documents.items.len - 1;
}

pub fn addDocument(self: *WindowContext, doc: Document) !void {
    const ptr = try self.allocator.create(OpenDocument);
    errdefer self.allocator.destroy(ptr);
    var doc_mut = doc;
    errdefer doc_mut.deinit(self.allocator);
    OpenDocument.initWithDocument(self.allocator, ptr, doc_mut);
    try self.documents.append(self.allocator, ptr);
    self.active_document_index = self.documents.items.len - 1;
}

pub fn setActiveDocument(self: *WindowContext, index: usize) void {
    if (index < self.documents.items.len) {
        self.active_document_index = index;
    }
}

pub fn closeDocument(self: *WindowContext, index: usize) void {
    if (index >= self.documents.items.len) return;
    const open_doc = self.documents.items[index];
    open_doc.deinit(self.allocator);
    self.allocator.destroy(open_doc);
    _ = self.documents.orderedRemove(index);

    if (self.active_document_index) |*active| {
        if (index < active.*) {
            active.* -= 1;
        } else if (index == active.*) {
            if (self.documents.items.len > 0) {
                active.* = @min(index, self.documents.items.len - 1);
            } else {
                self.active_document_index = null;
            }
        }
    }
```

```
}
```

A.5. src/icons.zig

```zig
const dvui = @import("dvui");

pub const line = dvui.entypo.flow_line;
pub const ellipse = dvui.entypo.circle;
pub const broken = dvui.entypo.line_graph;
pub const trash = dvui.entypo.trash;
pub const cross = dvui.entypo.cross;
pub const plus = dvui.entypo.plus;
```

A.6. src/main.zig

```zig
const std = @import("std");
const dvui = @import("dvui");
const SDLBackend = @import("sdl-backend");
const WindowContext = @import("WindowContext.zig");
const ui = @import("ui/frame.zig");

pub fn main() !void {
    // std.heap.GeneralPurposeAllocator was renamed to DebugAllocator recently.
    var gpa: std.heap.DebugAllocator(.{}) = .init;
    defer _ = gpa.deinit();

    const allocator = gpa.allocator();

    var backend = try SDLBackend.initWindow(.{
        .allocator = allocator,
        .size = .{ .w = 800.0, .h = 600.0 },
        .title = "My DVUI App",
        .vsync = true,
    });
    defer backend.deinit();

    var win = try dvui.Window.init(@src(), allocator, backend.backend(), .{
        .theme = switch (backend.preferredColorScheme() orelse .light) {
            .light => dvui.Theme.builtin.adwaita_light,
            .dark => dvui.Theme.builtin.adwaita_dark,
        },
    });
    defer win.deinit();

    var ctx = try WindowContext.init(allocator);
    defer ctx.deinit();

    var interrupted = false;
    main_loop: while (true) {
        const nstime = win.beginWait(interrupted);
        try win.begin(nstime);
        try backend.addAllEvents(&win);

        _ = SDLBackend.c.SDL_SetRenderDrawColor(backend.renderer, 0, 0, 0, 255);
        _ = SDLBackend.c.SDL_RenderClear(backend.renderer);

        if (!ui.guiFrame(&ctx)) break :main_loop;

        const end_micros = try win.end(.{});
        try backend.setCursor(win.cursorRequested());
        try backend.textInputRect(win.textInputRequested());
        try backend.renderPresent();

        const wait_event_micros = win.waitTime(end_micros);
        interrupted = try backend.waitEventTimeout(wait_event_micros);
    }
}
```

A.7. src/models/Document.zig

```zig
const std = @import("std");
const basic_models = @import("basic_models.zig");
const Size_f = basic_models.Size_f;
const Document = @This();

pub const Object = @import("Object.zig");
const shape = @import("shape/shape.zig");

size: Size_f,
objects: std.ArrayList(Object),
next_object_id: u64,

pub fn init(size: Size_f) Document {
    return .{
        .size = size,
        .objects = std.ArrayList(Object).empty,
        .next_object_id = 1,
    };
}

pub fn deinit(self: *Document, allocator: std.mem.Allocator) void {
    for (self.objects.items) |*obj| obj.deinit(allocator);
    self.objects.deinit(allocator);
}

pub fn clone(self: *const Document, allocator: std.mem.Allocator) !Document {
    var objects_list = std.ArrayList(Object).empty;
    errdefer {
        for (objects_list.items) |*obj| obj.deinit(allocator);
        objects_list.deinit(allocator);
    }
    var next_id = self.next_object_id;
    for (self.objects.items) |obj| {
        try objects_list.append(allocator, try obj.clone(allocator, &next_id));
    }
    return .{
        .size = self.size,
        .objects = objects_list,
        .next_object_id = next_id,
    };
}

pub fn addObject(self: *Document, allocator: std.mem.Allocator, template: Object) !void {
    const obj = try template.clone(allocator, &self.next_object_id);
    try self.objects.append(allocator, obj);
}

/// Добавляет объект в документ: как ребёнка родителя (если id найден), иначе в корень.
pub fn addObjectUnderParentId(self: *Document, allocator: std.mem.Allocator, parent_id: ?u64, template:
↪ Object) !void {
    if (parent_id) |id| {
        if (self.findObjectById(id)) |parent| {
            try parent.addChild(allocator, template, &self.next_object_id);
            return;
        }
    }
    try self.addObject(allocator, template);
}

pub fn addShape(self: *Document, allocator: std.mem.Allocator, parent: ?*Object, shape_kind:
↪ Object.ShapeKind) !void {
    const obj = try shape.createObject(allocator, shape_kind);
    if (parent) |p| {
        try p.addChild(allocator, obj, &self.next_object_id);
    } else {
```

```
        try self.addObject(allocator, obj);
    }
}

/// Удаляет объект из документа (из корня или из детей родителя). Возвращает true, если объект был найден и
 ↪ удалён.
pub fn removeObject(self: *Document, allocator: std.mem.Allocator, obj: *Object) bool {
    for (self.objects.items, 0..) |*item, i| {
        if (item == obj) {
            var removed = self.objects.orderedRemove(i);
            removed.deinit(allocator);
            return true;
        }
        if (removeFromChildren(allocator, &item.children, obj)) return true;
    }
    return false;
}

/// Удаляет объект по id. Возвращает true, если объект был найден и удалён.
pub fn removeObjectById(self: *Document, allocator: std.mem.Allocator, obj_id: u64) bool {
    for (self.objects.items, 0..) |*item, i| {
        if (item.id == obj_id) {
            var removed = self.objects.orderedRemove(i);
            removed.deinit(allocator);
            return true;
        }
        if (removeFromChildrenById(allocator, &item.children, obj_id)) return true;
    }
    return false;
}

pub fn findObjectById(self: *Document, obj_id: u64) ?*Object {
    for (self.objects.items) |*item| {
        if (item.id == obj_id) return item;
        if (findInChildrenById(&item.children, obj_id)) |found| return found;
    }
    return null;
}

fn removeFromChildren(allocator: std.mem.Allocator, children: *std.ArrayList(Object), obj: *Object) bool {
    for (children.items, 0..) |*item, i| {
        if (item == obj) {
            var removed = children.orderedRemove(i);
            removed.deinit(allocator);
            return true;
        }
        if (removeFromChildren(allocator, &item.children, obj)) return true;
    }
    return false;
}

fn removeFromChildrenById(allocator: std.mem.Allocator, children: *std.ArrayList(Object), obj_id: u64)
 ↪ bool {
    for (children.items, 0..) |*item, i| {
        if (item.id == obj_id) {
            var removed = children.orderedRemove(i);
            removed.deinit(allocator);
            return true;
        }
        if (removeFromChildrenById(allocator, &item.children, obj_id)) return true;
    }
    return false;
}

fn findInChildrenById(children: *std.ArrayList(Object), obj_id: u64) ?*Object {
    for (children.items) |*item| {
        if (item.id == obj_id) return item;
```

```
        if (findInChildrenById(&item.children, obj_id)) |found| return found;
    }
    return null;
}
```

A.8. src/models/Object.zig

```
const std = @import("std");
const Property = @import("Property.zig").Property;
const PropertyData = @import("Property.zig").Data;
const Object = @This();

pub const ShapeKind = enum {
    line,
    ellipse,
    broken,
};

const default_common_data = [_]PropertyData{
    .{ .position = .{ .x = 0, .y = 0 } },
    .{ .angle = 0 },
    .{ .scale = .{ .scale_x = 1, .scale_y = 1 } },
    .{ .visible = true },
    .{ .opacity = 1.0 },
    .{ .locked = false },
    .{ .stroke_rgba = 0x000000FF }, // чёрный, полная непрозрачность
    .{ .thickness = 2.0 },
};

pub const defaultCommonProperties: [default_common_data.len]Property = blk: {
    var result: [default_common_data.len]Property = undefined;
    for (default_common_data, &result) |d, *p| {
        p.* = .{ .data = d };
    }
    break :blk result;
};

id: u64,
shape: ShapeKind,
properties: std.ArrayList(Property),
children: std.ArrayList(Object),

pub fn getProperty(self: Object, comptime tag: std.meta.Tag(PropertyData)) ?@FieldType(PropertyData,
↪  @tagName(tag)) {
    for (self.properties.items) |*prop| {
        if (std.meta.activeTag(prop.data) == tag) return @field(prop.data, @tagName(tag));
    }
    return null;
}

/// Забирает владение Property
pub fn setProperty(self: *Object, allocator: std.mem.Allocator, prop: Property) !void {
    for (self.properties.items, 0..) |*p, i| {
        if (std.meta.activeTag(p.data) == std.meta.activeTag(prop.data)) {
            if (p.data == .points) allocator.free(p.data.points);
            self.properties.items[i] = prop;
            return;
        }
    }
    return error.PropertyNotFound;
}

pub fn addChild(self: *Object, allocator: std.mem.Allocator, template: Object, next_id: *u64) !void {
    const obj = try template.clone(allocator, next_id);
    try self.children.append(allocator, obj);
}
```

```zig
pub fn clone(self: Object, allocator: std.mem.Allocator, next_id: *u64) !Object {
    var properties_list = std.ArrayList(Property).empty;
    errdefer properties_list.deinit(allocator);
    for (self.properties.items) |prop| {
        try properties_list.append(allocator, try prop.clone(allocator));
    }

    var children_list = std.ArrayList(Object).empty;
    errdefer children_list.deinit(allocator);
    for (self.children.items) |child| {
        try children_list.append(allocator, try child.clone(allocator, next_id));
    }

    return .{
        .id = allocId(next_id),
        .shape = self.shape,
        .properties = properties_list,
        .children = children_list,
    };
}

fn allocId(next_id: *u64) u64 {
    const id = next_id.*;
    next_id.* += 1;
    return id;
}

pub fn deinit(self: *Object, allocator: std.mem.Allocator) void {
    for (self.children.items) |*child| child.deinit(allocator);
    self.children.deinit(allocator);
    for (self.properties.items) |*prop| prop.deinit(allocator);
    self.properties.deinit(allocator);
    self.* = undefined;
}
```

A.9. src/models/Property.zig

```zig
const std = @import("std");
const basic_models = @import("basic_models.zig");
const Point2_f = basic_models.Point2_f;
const Scale2_f = basic_models.Scale2_f;
const Size_f = basic_models.Size_f;
const Radii_f = basic_models.Radii_f;

pub const Data = union(enum) {
    position: Point2_f,
    angle: f32,
    scale: Scale2_f,
    visible: bool,
    opacity: f32,
    locked: bool,

    size: Size_f,
    radii: Radii_f,
    /// Процент дуги эллипса: 100 — полный эллипс, 50 — полуэллипс (0..100).
    arc_percent: f32,
    end_point: Point2_f,

    /// Владеет памятью; при deinit/clone — free/duplicate.
    points: []const Point2_f,
    /// Замкнутый контур (для ломаной: отрезок последняя–первая точка + заливка).
    closed: bool,

    /// Включена ли заливка.
    filled: bool,

    /// Цвет заливки, 0xRRGGBBAA.
```

```
    fill_rgba: u32,
    /// Цвет обводки, 0xRRGGBBAA.
    stroke_rgba: u32,

    thickness: f32,
};

pub const Property = struct {
    data: Data,

    pub fn deinit(self: *Property, allocator: std.mem.Allocator) void {
        switch (self.data) {
            .points => |slice| allocator.free(slice),
            else => {},
        }
        self.* = undefined;
    }

    pub fn clone(self: Property, allocator: std.mem.Allocator) !Property {
        return switch (self.data) {
            .points => |slice| .{
                .data = .{
                    .points = blk: {
                        const copy = try allocator.dupe(Point2_f, slice);
                        break :blk copy;
                    },
                },
            },
            else => .{ .data = self.data },
        };
    }
};
```

A.10. src/models/basic_models.zig

```
pub const Rect_i = struct {
    x: u32,
    y: u32,
    w: u32,
    h: u32,
};

pub const Size_i = struct {
    w: u32,
    h: u32,
};

pub const Size_f = struct {
    w: f32,
    h: f32,
};

pub const Point2_f = struct {
    x: f32 = 0,
    y: f32 = 0,
};

pub const Point2_i = struct {
    x: i32 = 0,
    y: i32 = 0,
};

pub const Radii_f = struct {
    x: f32,
    y: f32,
};
```

```
pub const Scale2_f = struct {
    scale_x: f32 = 1,
    scale_y: f32 = 1,
};

pub const Rect_f = struct {
    x: f32 = 0,
    y: f32 = 0,
    w: f32 = 0,
    h: f32 = 0,
};
```

A.11. src/models/shape/broken.zig

```
const std = @import("std");
const Object = @import("../Object.zig");
const Property = @import("../Property.zig").Property;
const PropertyData = @import("../Property.zig").Data;
const Point2_f = @import("../basic_models.zig").Point2_f;
const Rect_f = @import("../basic_models.zig").Rect_f;
const shape_mod = @import("shape.zig");

/// Свойства фигуры по умолчанию (добавляются к общим). points — слайс на статический массив.
pub const default_shape_properties_points = [_]Point2_f{
    .{ .x = 0, .y = 0 },
    .{ .x = 80, .y = 0 },
    .{ .x = 80, .y = 60 },
};
pub const default_shape_properties = [_]Property{
    .{ .data = .{ .points = &default_shape_properties_points } },
    .{ .data = .{ .closed = false } },
    .{ .data = .{ .filled = true } },
    .{ .data = .{ .fill_rgba = 0x000000FF } },
};
```

A.12. src/models/shape/ellipse.zig

```
const std = @import("std");
const Object = @import("../Object.zig");
const Property = @import("../Property.zig").Property;
const Rect_f = @import("../basic_models.zig").Rect_f;
const shape_mod = @import("shape.zig");

/// Свойства фигуры по умолчанию (добавляются к общим).
pub const default_shape_properties = [_]Property{
    .{ .data = .{ .radii = .{ .x = 50, .y = 50 } } },
    .{ .data = .{ .arc_percent = 100.0 } },
    .{ .data = .{ .closed = true } },
    .{ .data = .{ .filled = false } },
    .{ .data = .{ .fill_rgba = 0x000000FF } },
};
```

A.13. src/models/shape/line.zig

```
const std = @import("std");
const Object = @import("../Object.zig");
const Property = @import("../Property.zig").Property;
const Rect_f = @import("../basic_models.zig").Rect_f;
const shape_mod = @import("shape.zig");

/// Свойства фигуры по умолчанию (добавляются к общим).
pub const default_shape_properties = [_]Property{
    .{ .data = .{ .end_point = .{ .x = 100, .y = 200 } } },
};
```

A.14. src/models/shape/shape.zig

```zig
const std = @import("std");
const Object = @import("../Object.zig");
const Property = @import("../Property.zig").Property;
const PropertyData = @import("../Property.zig").Data;
const defaultCommonProperties = Object.defaultCommonProperties;
const basic_models = @import("../basic_models.zig");
const Point2_f = basic_models.Point2_f;
const line = @import("line.zig");
const ellipse = @import("ellipse.zig");
const broken = @import("broken.zig");
pub const Rect = basic_models.Rect_f;

/// Добавляет к объекту список свойств фигуры. Для .points дублирует слайс (объект владеет).
fn appendShapeProperties(allocator: std.mem.Allocator, obj: *Object, props: []const Property) !void {
    for (props) |prop| {
        if (prop.data == .points) {
            const pts = prop.data.points;
            const copy = try allocator.dupe(Point2_f, pts);
            try obj.properties.append(allocator, .{ .data = .{ .points = copy } });
        } else {
            try obj.properties.append(allocator, prop);
        }
    }
}

/// Создаёт объект с дефолтными общими и фигурными свойствами.
pub fn createObject(allocator: std.mem.Allocator, shape_kind: Object.ShapeKind) !Object {
    var obj = try createWithCommonProperties(allocator, shape_kind);
    errdefer obj.deinit(allocator);
    switch (shape_kind) {
        .line => try appendShapeProperties(allocator, &obj, &line.default_shape_properties),
        .ellipse => try appendShapeProperties(allocator, &obj, &ellipse.default_shape_properties),
        .broken => {
            try appendShapeProperties(allocator, &obj, &broken.default_shape_properties);
           try obj.setProperty(allocator, .{ .data = .{ .fill_rgba = obj.getProperty(.stroke_rgba).? } });
        },
    }
    return obj;
}

fn createWithCommonProperties(allocator: std.mem.Allocator, shape_kind: Object.ShapeKind) !Object {
    var properties_list = std.ArrayList(Property).empty;
    errdefer properties_list.deinit(allocator);
    for (defaultCommonProperties) |prop| try properties_list.append(allocator, prop);
    return .{
        .id = 0,
        .shape = shape_kind,
        .properties = properties_list,
        .children = std.ArrayList(Object).empty,
    };
}
```

A.15. src/persistence/json_io.zig

```zig
const std = @import("std");
const Document = @import("../models/Document.zig");

/// Сохраняет значение произвольного типа T в JSON-файл.
pub fn saveToFile(comptime T: type, value: *const T, path: []const u8) !void {
    var file = try std.fs.cwd().createFile(path, .{ .truncate = true });
    defer file.close();

    var buffer: [4096]u8 = undefined;
    var writer = file.writer(&buffer);
```

```
    try std.json.Stringify.value(value, .{ .whitespace = .indent_2 }, &writer.interface);

    try writer.interface.flush();
}

/// Загружает значение типа T из JSON-файла.
/// Для Document после разбора делается клон через allocator, т.к. парсер выделяет память
/// из арены — при закрытии документа её нельзя освобождать нашим аллокатором.
pub fn loadFromFile(comptime T: type, allocator: std.mem.Allocator, path: []const u8) !T {
    var file = try std.fs.cwd().openFile(path, .{});
    defer file.close();

    const data = try file.readToEndAlloc(allocator, std.math.maxInt(usize));
    defer allocator.free(data);

    var parsed = try std.json.parseFromSlice(T, allocator, data, .{ .ignore_unknown_fields = true });
    if (T == Document) {
        defer parsed.deinit();
        return try parsed.value.clone(allocator);
    }
    return parsed.value;
}
```

A.16. src/random_document.zig

```
const std = @import("std");
const Document = @import("models/Document.zig");
const Object = Document.Object;
const shape = @import("models/shape/shape.zig");
const basic_models = @import("models/basic_models.zig");
const Size_f = basic_models.Size_f;
const Point2_f = basic_models.Point2_f;
const Scale2_f = basic_models.Scale2_f;
const Radii_f = basic_models.Radii_f;

fn randFloat(rng: std.Random, min: f32, max: f32) f32 {
    return min + (max - min) * rng.float(f32);
}

fn randRgba(rng: std.Random) u32 {
    const r = rng.int(u8);
    const g = rng.int(u8);
    const b = rng.int(u8);
    const a: u8 = @intCast(rng.intRangeLessThan(usize, 128, 256));
    return (@as(u32, r) << 24) | (@as(u32, g) << 16) | (@as(u32, b) << 8) | a;
}

fn randomShapeKind(rng: std.Random) Object.ShapeKind {
    const shapes_implemented = [_]Object.ShapeKind{ .line, .ellipse, .broken };
    return shapes_implemented[rng.intRangeLessThan(usize, 0, shapes_implemented.len)];
}

fn randomizeObjectProperties(allocator: std.mem.Allocator, doc_size: *const Size_f, obj: *Object, rng:
 ↪  std.Random) !void {
    const margin: f32 = 8;
    const max_x = @max(0, doc_size.w - margin);
    const max_y = @max(0, doc_size.h - margin);

    try obj.setProperty(allocator, .{ .data = .{
        .position = .{
            .x = randFloat(rng, margin, if (max_x > margin) max_x else margin),
            .y = randFloat(rng, margin, if (max_y > margin) max_y else margin),
        },
    } });
    try obj.setProperty(allocator, .{ .data = .{ .angle = randFloat(rng, 0, 2 * std.math.pi) } });
    try obj.setProperty(allocator, .{ .data = .{
        .scale = .{
```

```
            .scale_x = randFloat(rng, 0.25, 2.0),
            .scale_y = randFloat(rng, 0.25, 2.0),
        },
    } });
    try obj.setProperty(allocator, .{ .data = .{ .visible = true } });
    try obj.setProperty(allocator, .{ .data = .{ .opacity = randFloat(rng, 0.3, 1.0) } });
    try obj.setProperty(allocator, .{ .data = .{ .locked = rng.boolean() } });

    const stroke = randRgba(rng);
    try obj.setProperty(allocator, .{ .data = .{ .stroke_rgba = stroke } });
    obj.setProperty(allocator, .{ .data = .{ .fill_rgba = randRgba(rng) } }) catch {};
    const thickness = randFloat(rng, max_x * 0.001, max_x * 0.01);
    try obj.setProperty(allocator, .{ .data = .{ .thickness = thickness } });

    switch (obj.shape) {
        .line => {
            const len = randFloat(rng, 20, @min(doc_size.w, doc_size.h) * 0.5);
            const angle = randFloat(rng, 0, 2 * std.math.pi);
            try obj.setProperty(allocator, .{ .data = .{
                .end_point = .{
                    .x = std.math.cos(angle) * len,
                    .y = std.math.sin(angle) * len,
                },
            } });
        },
        .ellipse => {
            const max_r = @min(120, @min(doc_size.w / 4, doc_size.h / 4));
            try obj.setProperty(allocator, .{ .data = .{
                .radii = .{
                    .x = randFloat(rng, 8, @max(8, max_r)),
                    .y = randFloat(rng, 8, @max(8, max_r)),
                },
            } });
        },
        .broken => {
            var list = std.ArrayList(Point2_f).empty;
            defer list.deinit(allocator);
            const n = rng.intRangeLessThan(usize, 2, 9);
            var x: f32 = 0;
            var y: f32 = 0;
            for (0..n) |_| {
                try list.append(allocator, .{ .x = x, .y = y });
                x += randFloat(rng, -40, 80);
                y += randFloat(rng, -30, 60);
            }
            const slice = try allocator.dupe(Point2_f, list.items);
            errdefer allocator.free(slice);
            try obj.setProperty(allocator, .{ .data = .{ .points = slice } });
        },
    }
}

/// Создаёт в документе случайные фигуры (line, ellipse, broken).
pub fn addRandomShapes(doc: *Document, allocator: std.mem.Allocator, rng: std.Random) !void {
    const max_total: usize = 80;
    var total_count: usize = 0;

    const n_root = rng.intRangeLessThan(usize, 6, 15);
    for (0..n_root) |_| {
        if (total_count >= max_total) break;
        var obj = try shape.createObject(allocator, randomShapeKind(rng));
        defer obj.deinit(allocator);
        try randomizeObjectProperties(allocator, &doc.size, &obj, rng);
        try doc.addObject(allocator, obj);
        total_count += 1;
    }
```

```
    var stack = std.ArrayList(*Object).empty;
    defer stack.deinit(allocator);
    for (doc.objects.items) |*obj| {
        try stack.append(allocator, obj);
    }
    while (stack.pop()) |obj| {
        if (total_count >= max_total) continue;
        const n_children = rng.intRangeLessThan(usize, 0, 2);
        const base_len = obj.children.items.len;
        for (0..n_children) |_| {
            if (total_count >= max_total) break;
            var child = try shape.createObject(allocator, randomShapeKind(rng));
            defer child.deinit(allocator);
            try randomizeObjectProperties(allocator, &doc.size, &child, rng);
            try obj.addChild(allocator, child, &doc.next_object_id);
            total_count += 1;
        }
        for (obj.children.items[base_len..]) |*child| {
            try stack.append(allocator, child);
        }
    }
}
```

A.17. src/render/CpuRenderEngine.zig

```
const std = @import("std");
const builtin = @import("builtin");
const dvui = @import("dvui");
const RenderEngine = @import("RenderEngine.zig").RenderEngine;
const Document = @import("../models/Document.zig");
const basic_models = @import("../models/basic_models.zig");
const cpu_draw = @import("cpu/draw.zig");
const RenderStats = @import("RenderStats.zig");
const Size_i = basic_models.Size_i;
const Rect_i = basic_models.Rect_i;
const Allocator = std.mem.Allocator;
const Color = dvui.Color;

const CpuRenderEngine = @This();
const Type = enum {
    Gradient,
    Squares,
};

type: Type,
gradient_start: Color.PMA = .{ .r = 0, .g = 0, .b = 0, .a = 255 },
gradient_end: Color.PMA = .{ .r = 255, .g = 255, .b = 255, .a = 255 },
_allocator: Allocator,
_renderStats: RenderStats,

pub fn init(allocator: Allocator, render_type: Type) CpuRenderEngine {
    return .{
        ._allocator = allocator,
        .type = render_type,
        ._renderStats = .{
            .render_time_ns = 0,
        },
    };
}

pub fn exampleReset(self: *CpuRenderEngine) void {
    var prng = std.Random.DefaultPrng.init(@intCast(std.time.microTimestamp()));
    const random = prng.random();
    self.gradient_start = Color.PMA{ .r = random.int(u8), .g = random.int(u8), .b = random.int(u8), .a =
    ↪  255 };
    self.gradient_end = Color.PMA{ .r = random.int(u8), .g = random.int(u8), .b = random.int(u8), .a = 255
    ↪  };
```

```
}

fn renderGradient(self: CpuRenderEngine, pixels: []Color.PMA, width: u32, height: u32, full_w: u32,
 ↪  full_h: u32, visible_rect: Rect_i) void {
    var y: u32 = 0;
    while (y < height) : (y += 1) {
        var x: u32 = 0;
        while (x < width) : (x += 1) {
            const gx: u32 = visible_rect.x + x;
            const gy: u32 = visible_rect.y + y;

            const denom_x: f32 = if (full_w > 1) @as(f32, @floatFromInt(full_w - 1)) else 1;
            const denom_y: f32 = if (full_h > 1) @as(f32, @floatFromInt(full_h - 1)) else 1;
            const fx: f32 = @as(f32, @floatFromInt(gx)) / denom_x;
            const fy: f32 = @as(f32, @floatFromInt(gy)) / denom_y;
            const factor: f32 = std.math.clamp((fx + fy) / 2, 0, 1);

            const r_f: f32 = @as(f32, @floatFromInt(self.gradient_start.r)) + factor * (@as(f32,
             ↪  @floatFromInt(self.gradient_end.r)) - @as(f32, @floatFromInt(self.gradient_start.r)));
            const g_f: f32 = @as(f32, @floatFromInt(self.gradient_start.g)) + factor * (@as(f32,
             ↪  @floatFromInt(self.gradient_end.g)) - @as(f32, @floatFromInt(self.gradient_start.g)));
            const b_f: f32 = @as(f32, @floatFromInt(self.gradient_start.b)) + factor * (@as(f32,
             ↪  @floatFromInt(self.gradient_end.b)) - @as(f32, @floatFromInt(self.gradient_start.b)));

            const r: u8 = @intFromFloat(std.math.clamp(r_f, 0, 255));
            const g: u8 = @intFromFloat(std.math.clamp(g_f, 0, 255));
            const b: u8 = @intFromFloat(std.math.clamp(b_f, 0, 255));
            pixels[y * width + x] = .{ .r = r, .g = g, .b = b, .a = 255 };
        }
    }
}

fn renderSquares(pixels: []Color.PMA, canvas_size: Size_i, visible_rect: Rect_i) void {
    const colors = [_]Color.PMA{
        .{ .r = 255, .g = 0, .b = 0, .a = 255 },
        .{ .r = 255, .g = 165, .b = 0, .a = 255 },
        .{ .r = 255, .g = 255, .b = 0, .a = 255 },
        .{ .r = 0, .g = 255, .b = 0, .a = 255 },
        .{ .r = 0, .g = 255, .b = 255, .a = 255 },
        .{ .r = 0, .g = 0, .b = 255, .a = 255 },
    };

    const squares_num = 5;
    var thikness: u32 = @intFromFloat(@as(f32, @floatFromInt(canvas_size.w + canvas_size.h)) / 2 * 0.03);
    if (thikness == 0) thikness = 1;

    const squares_sum_w = canvas_size.w - thikness * (squares_num + 1);
    const base_w = squares_sum_w / squares_num;
    const extra_w = squares_sum_w % squares_num;
    const squares_sum_h = canvas_size.h - thikness * (squares_num + 1);
    const base_h = squares_sum_h / squares_num;
    const extra_h = squares_sum_h % squares_num;

    var x_pos: [6]u32 = undefined;
    x_pos[0] = 0;
    for (1..squares_num + 1) |i| {
        const w = base_w + if (i - 1 < extra_w) @as(u32, 1) else 0;
        x_pos[i] = x_pos[i - 1] + thikness + w;
    }

    var y_pos: [6]u32 = undefined;
    y_pos[0] = 0;
    for (1..squares_num + 1) |i| {
        const h = base_h + if (i - 1 < extra_h) @as(u32, 1) else 0;
        y_pos[i] = y_pos[i - 1] + thikness + h;
    }
```

```
    var y: u32 = 0;
    while (y < visible_rect.h) : (y += 1) {
        const canvas_y = y + visible_rect.y;
        if (canvas_y >= canvas_size.h) continue;
        var x: u32 = 0;
        while (x < visible_rect.w) : (x += 1) {
            const canvas_x = x + visible_rect.x;
            if (canvas_x >= canvas_size.w) continue;

            var vertical_index: ?u32 = null;
            for (0..x_pos.len) |i| {
                if (canvas_x >= x_pos[i] and canvas_x < x_pos[i] + thikness) {
                    vertical_index = @intCast(i);
                    break;
                }
            }

            var horizontal_index: ?u32 = null;
            for (0..y_pos.len) |i| {
                if (canvas_y >= y_pos[i] and canvas_y < y_pos[i] + thikness) {
                    horizontal_index = @intCast(i);
                    break;
                }
            }

            if (vertical_index) |idx| {
                pixels[y * visible_rect.w + x] = colors[idx];
            } else if (horizontal_index) |idx| {
                pixels[y * visible_rect.w + x] = colors[idx];
            } else {
                var square_x: u32 = 0;
                for (0..squares_num) |i| {
                    if (canvas_x >= x_pos[i] + thikness and canvas_x < x_pos[i + 1]) {
                        square_x = @intCast(i);
                        break;
                    }
                }
                var square_y: u32 = 0;
                for (0..squares_num) |i| {
                    if (canvas_y >= y_pos[i] + thikness and canvas_y < y_pos[i + 1]) {
                        square_y = @intCast(i);
                        break;
                    }
                }
                if (square_x % 2 == square_y % 2) {
                    pixels[y * visible_rect.w + x] = .{ .r = 255, .g = 255, .b = 255, .a = 255 };
                } else {
                    pixels[y * visible_rect.w + x] = .{ .r = 0, .g = 0, .b = 0, .a = 255 };
                }
            }
        }
    }
}

pub fn example(self: CpuRenderEngine, canvas_size: Size_i, visible_rect: Rect_i) !?dvui.Texture {
    const full_w = canvas_size.w;
    const full_h = canvas_size.h;

    const width = visible_rect.w;
    const height = visible_rect.h;

    const pixels = try self._allocator.alloc(Color.PMA, @as(usize, width) * height);
    defer self._allocator.free(pixels);

    switch (self.type) {
        .Gradient => self.renderGradient(pixels, width, height, full_w, full_h, visible_rect),
        .Squares => renderSquares(pixels, canvas_size, visible_rect),
```

```
    }

    return try dvui.textureCreate(pixels, width, height, .nearest, .rgba_8_8_8_8);
}

pub fn renderEngine(self: *CpuRenderEngine) RenderEngine {
    return .{ .cpu = self };
}

/// Растеризует документ в текстуру (фон + фигуры через конвейер).
pub fn renderDocument(self: *CpuRenderEngine, document: *const Document, canvas_size: Size_i,
↪  visible_rect: Rect_i) !?dvui.Texture {
    const width = visible_rect.w;
    const height = visible_rect.h;
    const pixels = try self._allocator.alloc(Color.PMA, @as(usize, width) * height);
    defer self._allocator.free(pixels);

    for (pixels) |*p| p.* = .{ .r = 255, .g = 255, .b = 255, .a = 255 };
    var t = try std.time.Timer.start();
    try cpu_draw.drawDocument(pixels, width, height, visible_rect, document, canvas_size, self._allocator);
    self._renderStats.render_time_ns = t.read();

    return try dvui.textureCreate(pixels, width, height, .nearest, .rgba_8_8_8_8);
}

pub fn getStats(self: CpuRenderEngine) RenderStats {
    return self._renderStats;
}
```

A.18. src/render/RenderEngine.zig

```
const dvui = @import("dvui");
const CpuRenderEngine = @import("CpuRenderEngine.zig");
const Document = @import("../models/Document.zig");
const basic_models = @import("../models/basic_models.zig");
const RenderStats = @import("RenderStats.zig");

pub const RenderEngine = union(enum) {
    cpu: *CpuRenderEngine,

    pub fn exampleReset(self: RenderEngine) void {
        switch (self) {
            .cpu => |cpu_r| cpu_r.exampleReset(),
        }
    }

    pub fn example(self: RenderEngine, canvas_size: basic_models.Size_i, visible_rect:
    ↪  basic_models.Rect_i) !?dvui.Texture {
        return switch (self) {
            .cpu => |cpu_r| cpu_r.example(canvas_size, visible_rect),
        };
    }

    pub fn getStats(self: RenderEngine) RenderStats {
        return switch (self) {
            .cpu => |cpu_r| cpu_r.getStats(),
        };
    }

    /// Растеризует документ в текстуру.
    pub fn render(self: RenderEngine, document: *const Document, canvas_size: basic_models.Size_i,
    ↪  visible_rect: basic_models.Rect_i) !?dvui.Texture {
        return switch (self) {
            .cpu => |cpu_r| cpu_r.renderDocument(document, canvas_size, visible_rect),
        };
    }
};
```

A.19. src/render/RenderStats.zig

```
const RenderStats = @This();

render_time_ns: u64, // Время рендера кадра в микросекундах
```

A.20. src/render/cpu/broken.zig

```
const std = @import("std");
const Document = @import("../../models/Document.zig");
const pipeline = @import("pipeline.zig");
const line = @import("line.zig");
const DrawContext = pipeline.DrawContext;
const Color = @import("dvui").Color;

const Object = Document.Object;
const default_stroke: Color.PMA = .{ .r = 0, .g = 0, .b = 0, .a = 255 };
const default_fill: Color.PMA = .{ .r = 0, .g = 0, .b = 0, .a = 0 };
const default_thickness: f32 = 2.0;

/// Ломаная по точкам, обводка stroke_rgba. При closed — отрезок последняя–первая точка и заливка
 ↪  fill_rgba.
pub fn draw(
    ctx: *DrawContext,
    obj: *const Object,
    allocator: std.mem.Allocator,
) !void {
    const pts = obj.getProperty(.points) orelse return;
    if (pts.len < 2) return;
    const stroke = if (obj.getProperty(.stroke_rgba)) |stroke_rgba| pipeline.rgbaToPma(stroke_rgba) else
     ↪  default_stroke;
    const thickness = obj.getProperty(.thickness) orelse default_thickness;
    const closed = obj.getProperty(.closed) orelse false;
    const filled = obj.getProperty(.filled) orelse true;
    const fill_color = if (obj.getProperty(.fill_rgba)) |fill_rgba| pipeline.rgbaToPma(fill_rgba) else
     ↪  default_fill;

    const buffer = try allocator.alloc(Color.PMA, ctx.buf_width * ctx.buf_height);
    @memset(buffer, .{ .r = 0, .g = 0, .b = 0, .a = 0 });
    defer allocator.free(buffer);

    var copy_ctx = ctx.*;
    copy_ctx.pixels = buffer;
    copy_ctx.replace_mode = true;

    const do_fill = closed and filled;

    if (do_fill) {
        copy_ctx.startFill(allocator) catch return;
    }

    var i: usize = 0;
    while (i + 1 < pts.len) : (i += 1) {
        line.drawLine(&copy_ctx, pts[i].x, pts[i].y, pts[i + 1].x, pts[i + 1].y, stroke, thickness, true);
    }
    if (closed and pts.len >= 2) {
        const last = pts.len - 1;
        line.drawLine(&copy_ctx, pts[last].x, pts[last].y, pts[0].x, pts[0].y, stroke, thickness, true);
    }

    if (do_fill) {
        copy_ctx.stopFill(allocator, fill_color);
    }

    ctx.compositeDrawerContext(&copy_ctx, copy_ctx.transform.opacity);
}
```

A.21. src/render/cpu/draw.zig

```zig
const std = @import("std");
const Document = @import("../../models/Document.zig");
const pipeline = @import("pipeline.zig");
const line = @import("line.zig");
const ellipse = @import("ellipse.zig");
const broken = @import("broken.zig");
const basic_models = @import("../../models/basic_models.zig");
const Rect_i = basic_models.Rect_i;
const Size_i = basic_models.Size_i;

const Object = Document.Object;
const DrawContext = pipeline.DrawContext;
const Transform = pipeline.Transform;

fn isVisible(obj: *const Object) bool {
    return obj.getProperty(.visible) orelse true;
}

fn drawObject(
    ctx: *DrawContext,
    obj: *const Object,
    parent_transform: Transform,
    allocator: std.mem.Allocator,
) !void {
    if (!isVisible(obj)) return;
    const local = Transform.init(obj);
    const world = Transform.compose(parent_transform, local);
    ctx.setTransform(world);

    switch (obj.shape) {
        .line => line.draw(ctx, obj),
        .ellipse => try ellipse.draw(ctx, obj, allocator),
        .broken => try broken.draw(ctx, obj, allocator),
    }

    for (obj.children.items) |*child| {
        try drawObject(ctx, child, world, allocator);
    }
}

/// Рекурсивно рисует документ в буфер (объекты и потомки по порядку).
/// allocator опционален; если передан, ломаные рисуются через слой (без двойного наложения при alpha < 1).
pub fn drawDocument(
    pixels: []@import("dvui").Color.PMA,
    buf_width: u32,
    buf_height: u32,
    visible_rect: Rect_i,
    document: *const Document,
    canvas_size: Size_i,
    allocator: std.mem.Allocator,
) !void {
    const scale_x: f32 = if (document.size.w > 0) @as(f32, @floatFromInt(canvas_size.w)) / document.size.w
    ↪  else 0;
    const scale_y: f32 = if (document.size.h > 0) @as(f32, @floatFromInt(canvas_size.h)) / document.size.h
    ↪  else 0;

    var ctx = DrawContext{
        .pixels = pixels,
        .buf_width = buf_width,
        .buf_height = buf_height,
        .visible_rect = visible_rect,
        .scale_x = scale_x,
        .scale_y = scale_y,
    };
    const identity = Transform{};
```

```
    for (document.objects.items) |*obj| {
        try drawObject(&ctx, obj, identity, allocator);
    }
}
```

A.22. src/render/cpu/ellipse.zig

```
const std = @import("std");
const std_math = std.math;
const Document = @import("../../models/Document.zig");
const pipeline = @import("pipeline.zig");
const line = @import("line.zig");
const DrawContext = pipeline.DrawContext;
const Color = @import("dvui").Color;
const basic_models = @import("../../models/basic_models.zig");
const Point2_f = basic_models.Point2_f;

const Object = Document.Object;
const default_stroke: Color.PMA = .{ .r = 0, .g = 0, .b = 0, .a = 255 };
const default_fill: Color.PMA = .{ .r = 0, .g = 0, .b = 0, .a = 0 };
const default_thickness: f32 = 2.0;

/// Эллипс с центром (0,0) и полуосями radii. Обводка — полоса расстояния до контура (чёткая линия, не
 ↪  круги).
/// arc_percent: 100 — полный эллипс, иначе одна дуга; обход в коде от (0,ry) по квадрантам (визуально
 ↪  может казаться от низа против часовой из-за экранной Y).
/// Отрисовка в отдельный буфер и один composite, чтобы при alpha<255 пиксели не накладывались несколько
 ↪  раз.
pub fn draw(ctx: *DrawContext, obj: *const Object, allocator: std.mem.Allocator) !void {
    const radii = obj.getProperty(.radii) orelse return;
    const rx = radii.x;
    const ry = radii.y;
    if (rx <= 0 or ry <= 0) return;

    const stroke = if (obj.getProperty(.stroke_rgba)) |stroke_rgba| pipeline.rgbaToPma(stroke_rgba) else
     ↪  default_stroke;
    const thickness = if (obj.getProperty(.thickness)) |thickness| thickness else default_thickness;
    const arc_percent = std_math.clamp(if (obj.getProperty(.arc_percent)) |arc_percent| arc_percent else
     ↪  100.0, 0.0, 100.0);
    const closed = obj.getProperty(.closed) orelse true;
    const filled = obj.getProperty(.filled) orelse false;
    const fill_color = if (obj.getProperty(.fill_rgba)) |fill_rgba| pipeline.rgbaToPma(fill_rgba) else
     ↪  default_fill;
    const do_fill = filled and (closed or arc_percent >= 100.0);

    const t = &ctx.transform;
    const min_r = @min(rx, ry);
    const half_norm = thickness / (2.0 * min_r);
    const inner = @max(0.0, 1.0 - half_norm);
    const outer = 1.0 + half_norm;
    const d_inner_sq = inner * inner;
    const d_outer_sq = outer * outer;

    const margin = 1.0 + half_norm;
    const corners = [_]Point2_f{
        .{ .x = -rx * margin, .y = -ry * margin },
        .{ .x = rx * margin, .y = -ry * margin },
        .{ .x = rx * margin, .y = ry * margin },
        .{ .x = -rx * margin, .y = ry * margin },
    };
    var min_bx: f32 = std_math.inf(f32);
    var min_by: f32 = std_math.inf(f32);
    var max_bx: f32 = -std_math.inf(f32);
    var max_by: f32 = -std_math.inf(f32);
    for (corners) |c| {
        const w = ctx.localToWorld(c.x, c.y);
        const b = ctx.worldToBufferF(w.x, w.y);
```

```
            min_bx = @min(min_bx, b.x);
            min_by = @min(min_by, b.y);
            max_bx = @max(max_bx, b.x);
            max_by = @max(max_by, b.y);
        }
        const buf_w: i32 = @intCast(ctx.buf_width);
        const buf_h: i32 = @intCast(ctx.buf_height);
        const x0: i32 = @max(0, @as(i32, @intFromFloat(std_math.floor(min_bx))));
        const y0: i32 = @max(0, @as(i32, @intFromFloat(std_math.floor(min_by))));
        const x1: i32 = @min(buf_w, @as(i32, @intFromFloat(std_math.ceil(max_bx))) + 1);
        const y1: i32 = @min(buf_h, @as(i32, @intFromFloat(std_math.ceil(max_by))) + 1);

        const buffer = try allocator.alloc(Color.PMA, ctx.buf_width * ctx.buf_height);
        @memset(buffer, .{ .r = 0, .g = 0, .b = 0, .a = 0 });
        defer allocator.free(buffer);

        var stroke_ctx = ctx.*;
        stroke_ctx.pixels = buffer;
        stroke_ctx.replace_mode = true;

        if (do_fill) {
            stroke_ctx.startFill(allocator) catch return;
        }

        const inv_rx = 1.0 / rx;
        const inv_ry = 1.0 / ry;
        const arc_len = 2.0 * std_math.pi * arc_percent / 100.0;

        var by: i32 = y0;
        while (by < y1) : (by += 1) {
            const buf_y = @as(f32, @floatFromInt(by)) + 0.5;
            var bx: i32 = x0;
            while (bx < x1) : (bx += 1) {
                const buf_x = @as(f32, @floatFromInt(bx)) + 0.5;
                const w = stroke_ctx.bufferToWorld(buf_x, buf_y);
                const loc = stroke_ctx.worldToLocal(w.x, w.y);
                const nx = loc.x * inv_rx;
                const ny = loc.y * inv_ry;
                const d = nx * nx + ny * ny;
                if (d < d_inner_sq or d > d_outer_sq) continue;
                if (arc_percent < 100.0) {
                    var diff = std_math.pi / 2.0 - std_math.atan2(ny, nx);
                    if (diff < 0) diff += 2.0 * std_math.pi;
                    if (diff > arc_len) continue;
                }
                stroke_ctx.blendPixelAtBuffer(bx, by, stroke);
            }
        }

        if (closed and arc_percent < 100.0) {
            const end_x = rx * std_math.sin(arc_len);
            const end_y = ry * std_math.cos(arc_len);
            line.drawLine(&stroke_ctx, 0, 0, 0, ry, stroke, thickness, false);
            line.drawLine(&stroke_ctx, 0, 0, end_x, end_y, stroke, thickness, false);
        }

        if (do_fill) {
            stroke_ctx.stopFill(allocator, fill_color);
        }

        ctx.compositeDrawerContext(&stroke_ctx, t.opacity);
    }
```

A.23. src/render/cpu/line.zig

```
const std = @import("std");
const Document = @import("../../models/Document.zig");
```

```zig
const pipeline = @import("pipeline.zig");
const DrawContext = pipeline.DrawContext;
const Color = @import("dvui").Color;
const base_models = @import("../../models/basic_models.zig");
const Point2_i = base_models.Point2_i;

const Object = Document.Object;
const default_stroke: Color.PMA = .{ .r = 0, .g = 0, .b = 0, .a = 255 };
const default_thickness: f32 = 2.0;

/// Линия от (0,0) до end_point.
pub fn draw(ctx: *DrawContext, obj: *const Object) void {
    const end_point = obj.getProperty(.end_point) orelse return;
    const end_x = end_point.x;
    const end_y = end_point.y;
    const stroke = if (obj.getProperty(.stroke_rgba)) |stroke_rgba| pipeline.rgbaToPma(stroke_rgba) else
    ↪  default_stroke;
    const thickness = obj.getProperty(.thickness) orelse default_thickness;
    drawLine(ctx, 0, 0, end_x, end_y, stroke, thickness, false);
}

/// Рисует отрезок по локальным концам (перевод в буфер внутри).
/// draw_when_outside: если true, участки линии за экраном тоже рисуются (толщиной 1 px); в буфере —
↪  обычная толщина.
pub fn drawLine(ctx: *DrawContext, x0: f32, y0: f32, x1: f32, y1: f32, color: Color.PMA, thickness: f32,
↪  draw_when_outside: bool) void {
    const w0 = ctx.localToWorld(x0, y0);
    const w1 = ctx.localToWorld(x1, y1);
    const b0 = ctx.worldToBuffer(w0.x, w0.y);
    const b1 = ctx.worldToBuffer(w1.x, w1.y);
    const t = &ctx.transform;
    const scale = @sqrt(t.scale.scale_x * ctx.scale_x * t.scale.scale_y * ctx.scale_y);
    const thickness_px: u32 = @as(u32, @intFromFloat(std.math.round(thickness * scale)));
    if (thickness_px > 0)
        drawLineInBuffer(ctx, b0.x, b0.y, b1.x, b1.y, color, thickness_px, draw_when_outside);
}

inline fn clip(p: f32, q: f32, t0: *f32, t1: *f32) bool {
    if (p == 0) {
        return q >= 0;
    }
    const r = q / p;
    if (p < 0) {
        if (r > t1.*) return false;
        if (r > t0.*) t0.* = r;
    } else {
        if (r < t0.*) return false;
        if (r < t1.*) t1.* = r;
    }
    return true;
}

/// Liang–Barsky отсечение отрезка (x0,y0)-(x1,y1) прямоугольником [left,right]x[top,bottom].
/// Координаты концов модифицируются по месту. Возвращает false, если отрезок целиком вне прямоугольника.
fn liangBarskyClip(
    x0: *i32,
    y0: *i32,
    x1: *i32,
    y1: *i32,
    left: i32,
    top: i32,
    right: i32,
    bottom: i32,
) bool {
    const fx0: f32 = @floatFromInt(x0.*);
    const fy0: f32 = @floatFromInt(y0.*);
    const fx1: f32 = @floatFromInt(x1.*);
```

```
    const fy1: f32 = @floatFromInt(y1.*);

    const dx = fx1 - fx0;
    const dy = fy1 - fy0;

    var t0: f32 = 0.0;
    var t1: f32 = 1.0;

    const fl: f32 = @floatFromInt(left);
    const ft: f32 = @floatFromInt(top);
    const fr: f32 = @floatFromInt(right);
    const fb: f32 = @floatFromInt(bottom);

    if (!clip(-dx, fx0 - fl, &t0, &t1)) return false; // x >= left
    if (!clip(dx, fr - fx0, &t0, &t1)) return false; // x <= right
    if (!clip(-dy, fy0 - ft, &t0, &t1)) return false; // y >= top
    if (!clip(dy, fb - fy0, &t0, &t1)) return false; // y <= bottom

    const nx0 = fx0 + dx * t0;
    const ny0 = fy0 + dy * t0;
    const nx1 = fx0 + dx * t1;
    const ny1 = fy0 + dy * t1;

    x0.* = @intFromFloat(std.math.round(nx0));
    y0.* = @intFromFloat(std.math.round(ny0));
    x1.* = @intFromFloat(std.math.round(nx1));
    y1.* = @intFromFloat(std.math.round(ny1));

    return true;
}

/// Отсекает отрезок буфером ctx (0..buf_width-1, 0..buf_height-1).
fn clipLineToBuffer(ctx: *DrawContext, a: *Point2_i, b: *Point2_i, thickness: i32) bool {
    var x0 = a.x;
    var y0 = a.y;
    var x1 = b.x;
    var y1 = b.y;

    const left: i32 = -thickness;
    const top: i32 = -thickness;
    const right: i32 = @as(i32, @intCast(ctx.buf_width - 1)) + thickness;
    const bottom: i32 = @as(i32, @intCast(ctx.buf_height - 1)) + thickness;

    if (!liangBarskyClip(&x0, &y0, &x1, &y1, left, top, right, bottom)) {
        return false;
    }

    a.* = .{ .x = x0, .y = y0 };
    b.* = .{ .x = x1, .y = y1 };
    return true;
}

fn drawLineInBuffer(ctx: *DrawContext, bx0: i32, by0: i32, bx1: i32, by1: i32, color: Color.PMA,
 ↪  thickness_px: u32, draw_when_outside: bool) void {
    // Коррекция толщины в зависимости от угла линии.
    var thickness_corrected: u32 = thickness_px;
    var use_vertical: bool = undefined;
    const dx_f: f32 = @floatFromInt(bx1 - bx0);
    const dy_f: f32 = @floatFromInt(by1 - by0);
    const len: f32 = @sqrt(dx_f * dx_f + dy_f * dy_f);
    if (len > 0) {
        const cos_theta = @abs(dx_f) / len;
        const sin_theta = @abs(dy_f) / len;
        const desired: f32 = @floatFromInt(thickness_px);
        const eps: f32 = 1e-3;

        const vertical_based = desired / @max(sin_theta, eps);
```

```
        const horizontal_based = desired / @max(cos_theta, eps);

        use_vertical = sin_theta >= cos_theta;
        const corrected_f = if (use_vertical) vertical_based else horizontal_based;

        thickness_corrected = @max(@as(u32, 1), @as(u32, @intFromFloat(std.math.round(corrected_f))));
    }
    const half_thickness: i32 = @intCast(thickness_corrected / 2);
    const thickness_corrected_i: i32 = @as(i32, @intCast(thickness_corrected));

    var p0 = Point2_i{ .x = bx0, .y = by0 };
    var p1 = Point2_i{ .x = bx1, .y = by1 };

    // Отсечение только когда не рисуем вне viewport: иначе линия идёт целиком, толщина 1 px снаружи.
    if (!draw_when_outside) {
        if (!clipLineToBuffer(ctx, &p0, &p1, @as(i32, @intCast(thickness_corrected)))) return;
    }

    var x0 = p0.x;
    var y0 = p0.y;
    const ex = p1.x;
    const ey = p1.y;

    const dx: i32 = @intCast(@abs(ex - x0));
    const sx: i32 = if (x0 < ex) 1 else -1;
    const dy_abs: i32 = @intCast(@abs(ey - y0));
    const dy: i32 = -dy_abs;
    const sy: i32 = if (y0 < ey) 1 else -1;

    var err: i32 = dx + dy;
    const buf_w_i: i32 = @intCast(ctx.buf_width);
    const buf_h_i: i32 = @intCast(ctx.buf_height);

    while (true) {
        // Внутри viewport — полная толщина; снаружи при draw_when_outside — только 1 пиксель.
        const in_viewport = x0 >= -thickness_corrected_i and x0 < buf_w_i + thickness_corrected_i and y0 >=
         ↪  -thickness_corrected_i and y0 < buf_h_i + thickness_corrected_i;
        const effective_half: i32 = if (draw_when_outside and !in_viewport) 0 else half_thickness;

        var thick: i32 = -effective_half;
        while (thick <= effective_half) {
            const x = if (use_vertical) x0 + thick else x0;
            const y = if (use_vertical) y0 else y0 + thick;
            ctx.blendPixelAtBuffer(x, y, color);

            thick += 1;
        }

        if (x0 == ex and y0 == ey) break;

        const e2: i32 = 2 * err;
        if (e2 >= dy) {
            err += dy;
            x0 += sx;
        }
        if (e2 <= dx) {
            err += dx;
            y0 += sy;
        }
    }
}
```

A.24. src/render/cpu/pipeline.zig

```
const std = @import("std");
const dvui = @import("dvui");
const basic_models = @import("../../models/basic_models.zig");
```

```zig
const Document = @import("../../models/Document.zig");
const Point2_f = basic_models.Point2_f;
const Point2_i = basic_models.Point2_i;
const Scale2_f = basic_models.Scale2_f;
const Rect_i = basic_models.Rect_i;
const Color = dvui.Color;

/// Трансформ объекта: позиция, угол, масштаб, непрозрачность.
pub const Transform = struct {
    position: Point2_f = .{},
    angle: f32 = 0,
    scale: Scale2_f = .{},
    opacity: f32 = 1.0,

    pub fn init(obj: *const Document.Object) Transform {
        const pos = obj.getProperty(.position) orelse Point2_f{ .x = 0, .y = 0 };
        const angle = obj.getProperty(.angle) orelse 0;
        const scale = obj.getProperty(.scale) orelse Scale2_f{ .scale_x = 1, .scale_y = 1 };
        const opacity = obj.getProperty(.opacity) orelse 1.0;
        return .{
            .position = pos,
            .angle = angle,
            .scale = scale,
            .opacity = opacity,
        };
    }

    /// Композиция: world = parent * local.
    pub fn compose(parent: Transform, local: Transform) Transform {
        const cos_a = std.math.cos(parent.angle);
        const sin_a = std.math.sin(parent.angle);
        const sx = parent.scale.scale_x * local.scale.scale_x;
        const sy = parent.scale.scale_y * local.scale.scale_y;
        const local_px = local.position.x * parent.scale.scale_x;
        const local_py = local.position.y * parent.scale.scale_y;
        const rx = cos_a * local_px - sin_a * local_py;
        const ry = sin_a * local_px + cos_a * local_py;
        return .{
            .position = .{
                .x = parent.position.x + rx,
                .y = parent.position.y + ry,
            },
            .angle = parent.angle + local.angle,
            .scale = .{ .scale_x = sx, .scale_y = sy },
            .opacity = parent.opacity * local.opacity,
        };
    }
};

/// Мировые -> локальные для заданного трансформа.
pub fn worldToLocalTransform(t: Transform, wx: f32, wy: f32) Point2_f {
    const dx = wx - t.position.x;
    const dy = wy - t.position.y;
    const cos_a = std.math.cos(-t.angle);
    const sin_a = std.math.sin(-t.angle);
    const sx = if (t.scale.scale_x != 0) t.scale.scale_x else 1.0;
    const sy = if (t.scale.scale_y != 0) t.scale.scale_y else 1.0;
    return .{
        .x = (dx * cos_a - dy * sin_a) / sx,
        .y = (dx * sin_a + dy * cos_a) / sy,
    };
}

/// Конвейер отрисовки: локальные координаты -> трансформ -> буфер.
pub const DrawContext = struct {
    pixels: []Color.PMA,
    buf_width: u32,
```

```
buf_height: u32,
visible_rect: Rect_i,
scale_x: f32,
scale_y: f32,
transform: Transform = .{},
/// Если true, blendPixelAtBuffer перезаписывает пиксель без бленда
replace_mode: bool = false,
_fill_canvas: ?*FillCanvas = null,

pub fn setTransform(self: *DrawContext, t: Transform) void {
    self.transform = t;
}

/// Локальные -> мировые.
pub fn localToWorld(self: *const DrawContext, local_x: f32, local_y: f32) Point2_f {
    const t = &self.transform;
    const cos_a = std.math.cos(t.angle);
    const sin_a = std.math.sin(t.angle);
    return .{
        .x = t.position.x + (local_x * t.scale.scale_x) * cos_a - (local_y * t.scale.scale_y) * sin_a,
        .y = t.position.y + (local_x * t.scale.scale_x) * sin_a + (local_y * t.scale.scale_y) * cos_a,
    };
}

/// Мировые -> буфер (float).
pub fn worldToBufferF(self: *const DrawContext, wx: f32, wy: f32) Point2_f {
    const canvas_x = wx * self.scale_x;
    const canvas_y = wy * self.scale_y;
    const vx = @as(f32, @floatFromInt(self.visible_rect.x));
    const vy = @as(f32, @floatFromInt(self.visible_rect.y));
    return .{
        .x = canvas_x - vx,
        .y = canvas_y - vy,
    };
}

/// Мировые -> буфер (целые).
pub fn worldToBuffer(self: *const DrawContext, wx: f32, wy: f32) Point2_i {
    const b = self.worldToBufferF(wx, wy);
    return .{
        .x = @intFromFloat(std.math.round(b.x)),
        .y = @intFromFloat(std.math.round(b.y)),
    };
}

/// Буфер -> мировые.
pub fn bufferToWorld(self: *const DrawContext, buf_x: f32, buf_y: f32) Point2_f {
    const vx = @as(f32, @floatFromInt(self.visible_rect.x));
    const vy = @as(f32, @floatFromInt(self.visible_rect.y));
    const canvas_x = buf_x + vx;
    const canvas_y = buf_y + vy;
    const sx = if (self.scale_x != 0) self.scale_x else 1.0;
    const sy = if (self.scale_y != 0) self.scale_y else 1.0;
    return .{
        .x = canvas_x / sx,
        .y = canvas_y / sy,
    };
}

/// Мировые -> локальные.
pub fn worldToLocal(self: *const DrawContext, wx: f32, wy: f32) Point2_f {
    return worldToLocalTransform(self.transform, wx, wy);
}

/// Смешивает цвет в пикселе буфера с учётом opacity трансформа. В replace_mode просто перезаписывает
 ↪  пиксель.
/// Если активен fill canvas, каждый записанный пиксель помечается как граница для заливки.
```

```
pub fn blendPixelAtBuffer(self: *DrawContext, bx_i32: i32, by_i32: i32, color: Color.PMA) void {
    if (self._fill_canvas) |fc| fc.setBorder(bx_i32, by_i32);

    if (bx_i32 < 0 or by_i32 < 0 or bx_i32 >= self.buf_width or by_i32 >= self.buf_height) return;
    const bx: u32 = @intCast(bx_i32);
    const by: u32 = @intCast(by_i32);
    const idx = by * self.buf_width + bx;
    const dst = &self.pixels[idx];
    if (self.replace_mode) {
        dst.* = color;
        return;
    }
    const t = &self.transform;
    const a = @as(f32, @floatFromInt(color.a)) / 255.0 * t.opacity;
    const src_r = @as(f32, @floatFromInt(color.r)) * t.opacity;
    const src_g = @as(f32, @floatFromInt(color.g)) * t.opacity;
    const src_b = @as(f32, @floatFromInt(color.b)) * t.opacity;
    const inv_a = 1.0 - a;
    dst.r = @intFromFloat(std.math.clamp(src_r + inv_a * @as(f32, @floatFromInt(dst.r)), 0, 255));
    dst.g = @intFromFloat(std.math.clamp(src_g + inv_a * @as(f32, @floatFromInt(dst.g)), 0, 255));
    dst.b = @intFromFloat(std.math.clamp(src_b + inv_a * @as(f32, @floatFromInt(dst.b)), 0, 255));
    dst.a = @intFromFloat(std.math.clamp(a * 255 + inv_a * @as(f32, @floatFromInt(dst.a)), 0, 255));
}

/// Накладывает буфер другого контекста на этот с заданной прозрачностью (один бленд на пиксель).
 ↪  Размеры буферов должны совпадать.
pub fn compositeDrawerContext(self: *DrawContext, other: *const DrawContext, opacity: f32) void {
    if (self.buf_width != other.buf_width or self.buf_height != other.buf_height) return;
    const n = self.buf_width * self.buf_height;
    for (0..n) |i| {
        const src = other.pixels[i];
        if (src.a == 0) continue;
        const dst = &self.pixels[i];
        const a = @as(f32, @floatFromInt(src.a)) / 255.0 * opacity;
        const src_r = @as(f32, @floatFromInt(src.r)) * opacity;
        const src_g = @as(f32, @floatFromInt(src.g)) * opacity;
        const src_b = @as(f32, @floatFromInt(src.b)) * opacity;
        const inv_a = 1.0 - a;
        dst.r = @intFromFloat(std.math.clamp(src_r + inv_a * @as(f32, @floatFromInt(dst.r)), 0, 255));
        dst.g = @intFromFloat(std.math.clamp(src_g + inv_a * @as(f32, @floatFromInt(dst.g)), 0, 255));
        dst.b = @intFromFloat(std.math.clamp(src_b + inv_a * @as(f32, @floatFromInt(dst.b)), 0, 255));
       dst.a = @intFromFloat(std.math.clamp(a * 255 + inv_a * @as(f32, @floatFromInt(dst.a)), 0, 255));
    }
}

/// Пиксель в локальных координатах (трансформ + PMA).
pub fn blendPixelLocal(self: *DrawContext, local_x: f32, local_y: f32, color: Color.PMA) void {
    const w = self.localToWorld(local_x, local_y);
    const b = self.worldToBufferF(w.x, w.y);
    const bx: i32 = @intFromFloat(b.x);
    const by: i32 = @intFromFloat(b.y);
    const vw = @as(i32, @intCast(self.visible_rect.w));
    const vh = @as(i32, @intCast(self.visible_rect.h));
    if (bx < 0 or bx >= vw or by < 0 or by >= vh) return;
    self.blendPixelAtBuffer(bx, by, color);
}

/// Начинает сбор границ для заливки: создаёт FillCanvas и при последующих вызовах blendPixelAtBuffer
 ↪  помечает пиксели как границу.
pub fn startFill(self: *DrawContext, allocator: std.mem.Allocator) !void {
    const fc = try FillCanvas.init(allocator, self.buf_width, self.buf_height);
    const ptr = try allocator.create(FillCanvas);
    ptr.* = fc;
    self._fill_canvas = ptr;
}

/// Рисует заливку по собранным границам цветом color, освобождает FillCanvas и сбрасывает режим.
```

```
    pub fn stopFill(self: *DrawContext, allocator: std.mem.Allocator, color: Color.PMA) void {
        const fc = self._fill_canvas orelse return;
        self._fill_canvas = null;
        fc.fillColor(self, allocator, color);
        fc.deinit();
        allocator.destroy(fc);
    }
};

/// Конвертирует u32 0xRRGGBBAA в Color.PMA.
pub fn rgbaToPma(rgba: u32) Color.PMA {
    const r: u8 = @intCast((rgba >> 24) & 0xFF);
    const g: u8 = @intCast((rgba >> 16) & 0xFF);
    const b: u8 = @intCast((rgba >> 8) & 0xFF);
    const a: u8 = @intCast((rgba >> 0) & 0xFF);
    if (a == 0) return .{ .r = 0, .g = 0, .b = 0, .a = 0 };
    const af: f32 = @as(f32, @floatFromInt(a)) / 255.0;
    return .{
        .r = @intFromFloat(@as(f32, @floatFromInt(r)) * af),
        .g = @intFromFloat(@as(f32, @floatFromInt(g)) * af),
        .b = @intFromFloat(@as(f32, @floatFromInt(b)) * af),
        .a = a,
    };
}

/// Контекст для заполнения фигур цветом. Границы хранятся в set — по x и y можно добавлять произвольные
 ↪ точки.
const FillCanvas = struct {
    /// Множество пикселей границы (x, y) — без ограничения по размеру буфера.
    border_set: std.AutoHashMap(Point2_i, void),
    buf_width: u32,
    buf_height: u32,

    pub fn init(allocator: std.mem.Allocator, width: u32, height: u32) !FillCanvas {
        const border_set = std.AutoHashMap(Point2_i, void).init(allocator);
        return .{
            .border_set = border_set,
            .buf_width = width,
            .buf_height = height,
        };
    }

    pub fn deinit(self: *FillCanvas) void {
        self.border_set.deinit();
    }

    /// Добавляет точку границы; координаты x, y могут быть любыми (условно бесконечное поле).
    pub fn setBorder(self: *FillCanvas, x: i32, y: i32) void {
        self.border_set.put(.{ .x = x, .y = y }, {}) catch {};
    }

    /// Заливка четырёхсвязным стековым алгоритмом от первой найденной внутренней точки.
    pub fn fillColor(self: *FillCanvas, draw_ctx: *DrawContext, allocator: std.mem.Allocator, color:
     ↪ Color.PMA) void {
        const n = self.border_set.count();
        if (n == 0) return;

        const buf_w_i: i32 = @intCast(self.buf_width);
        const buf_h_i: i32 = @intCast(self.buf_height);

        // Ключи один раз по (y, x) — по строкам x уже будут отсортированы.
        var keys_buf = std.ArrayList(Point2_i).empty;
        defer keys_buf.deinit(allocator);
        keys_buf.ensureTotalCapacity(allocator, n) catch return;
        var iter = self.border_set.keyIterator();
        while (iter.next()) |k| {
            keys_buf.appendAssumeCapacity(k.*);
```

```
    }
    std.mem.sort(Point2_i, keys_buf.items, {}, struct {
        fn lessThan(_: void, a: Point2_i, b: Point2_i) bool {
            if (a.y != b.y) return a.y < b.y;
            return a.x < b.x;
        }
    }.lessThan);

    // Семена: по строкам находим сегменты (пары x), пересекаем с окном буфера, берём середину сегмента.
    var seeds = findFillSeeds(keys_buf.items, buf_w_i, buf_h_i, allocator) catch return;
    defer seeds.deinit(allocator);

    var stack = std.ArrayList(Point2_i).empty;
    defer stack.deinit(allocator);
    var filled = std.AutoHashMap(Point2_i, void).init(allocator);
    defer filled.deinit();

    for (seeds.items) |s| {
        if (self.border_set.contains(s)) continue;
        if (filled.contains(s)) continue;
        stack.clearRetainingCapacity();
        stack.append(allocator, s) catch return;
        while (stack.pop()) |cell| {
            if (self.border_set.contains(cell)) continue;
            const gop = filled.getOrPut(cell) catch return;
            if (gop.found_existing) continue;

            if (cell.x >= 0 and cell.x < buf_w_i and cell.y >= 0 and cell.y < buf_h_i) {
                draw_ctx.blendPixelAtBuffer(cell.x, cell.y, color);
            }
            if (cell.x > 0) stack.append(allocator, .{ .x = cell.x - 1, .y = cell.y }) catch return;
            if (cell.x < buf_w_i - 1) stack.append(allocator, .{ .x = cell.x + 1, .y = cell.y }) catch
             ↪  return;
            if (cell.y > 0) stack.append(allocator, .{ .x = cell.x, .y = cell.y - 1 }) catch return;
            if (cell.y < buf_h_i - 1) stack.append(allocator, .{ .x = cell.x, .y = cell.y + 1 }) catch
             ↪  return;
        }
    }
}

/// По строкам: рёбра (подряд идущие x) → сегменты между ними. Семена — середины чётных сегментов (при
 ↪  чётном числе границ).
fn findFillSeeds(
    keys: []const Point2_i,
    buf_w_i: i32,
    buf_h_i: i32,
    allocator: std.mem.Allocator,
) !std.ArrayList(Point2_i) {
    var list = std.ArrayList(Point2_i).empty;
    errdefer list.deinit(allocator);
    var segments = std.ArrayList(struct { left: i32, right: i32 }).empty;
    defer segments.deinit(allocator);

    var i: usize = 0;
    while (i < keys.len) {
        const y = keys[i].y;
        const row_start = i;
        while (i < keys.len and keys[i].y == y) : (i += 1) {}
        const row = keys[row_start..i];
        if (row.len < 2 or y < 0 or y >= buf_h_i) continue;

        segments.clearRetainingCapacity();
        var run_end_x: i32 = row[0].x;
        for (row[1..]) |p| {
            if (p.x != run_end_x + 1) {
                try segments.append(allocator, .{ .left = run_end_x + 1, .right = p.x - 1 });
                run_end_x = p.x;
```

```
                } else {
                    run_end_x = p.x;
                }
            }
            // Семена только при чётном числе границ
            if ((segments.items.len + 1) % 2 != 0) continue;

            for (segments.items, 0..) |seg, gi| {
                if (gi % 2 != 0 or seg.left > seg.right) continue;
                const left = @max(seg.left, 0);
                const right = @min(seg.right, buf_w_i - 1);
                if (left <= right) {
                    try list.append(allocator, .{ .x = left + @divTrunc(right - left, 2), .y = y });
                }
            }
        }
        return list;
    }
};
```

A.25. src/tests.zig

```
// Корень для `zig build test`. Тесты из импортированных здесь модулей выполняются (в Zig не подтягиваются
 ↪  из транзитивных импортов).
// Добавляй сюда _ = @import("path/to/module.zig"); для каждого модуля с test-блоками.
// Чтобы увидеть список всех тестов: после `zig build test` выполни `./zig-out/bin/test`.
test "discover tests" {
    _ = @import("main.zig");
    _ = @import("models/Property.zig");
    _ = @import("models/shape/shape.zig");
}

// Убедиться, что выполнились все ожидаемые тесты: этот тест пройдёт только если до него дошли (т.е. все
 ↪  предыдущие прошли).
test "all module tests completed" {
    const std = @import("std");
    std.debug.print("\n  (все тесты модулей выполнены)\n", .{});
}
```

A.26. src/toolbar/Tool.zig

```
const basic_models = @import("../models/basic_models.zig");
const Point2_f = basic_models.Point2_f;
const Canvas = @import("../Canvas.zig");
const Document = @import("../models/Document.zig");
const pipeline = @import("../render/cpu/pipeline.zig");
const Transform = pipeline.Transform;

pub const ToolContext = struct {
    canvas: *Canvas,
    document_point: Point2_f,
    selected_object_id: ?u64,

    pub fn addObject(self: *const ToolContext, template: Document.Object) !void {
        var obj = template;
        const local_pos = self.computeLocalPosition();
        try obj.setProperty(self.canvas.allocator, .{ .data = .{ .position = local_pos } });
        try self.canvas.document.addObjectUnderParentId(self.canvas.allocator, self.selected_object_id,
         ↪  obj);
        self.canvas.requestRedraw();
    }

    fn computeLocalPosition(self: *const ToolContext) Point2_f {
        if (self.selected_object_id) |parent_id| {
            if (findWorldTransformById(self.canvas.document, parent_id)) |parent_world| {
                return pipeline.worldToLocalTransform(parent_world, self.document_point.x,
                 ↪  self.document_point.y);
```

```
            }
        }
        return self.document_point;
    }
};

fn findWorldTransformById(doc: *Document, target_id: u64) ?Transform {
    const identity = Transform{};
    for (doc.objects.items) |*obj| {
        if (findWorldTransformInTree(obj, identity, target_id)) |t| return t;
    }
    return null;
}

fn findWorldTransformInTree(obj: *const Document.Object, parent: Transform, target_id: u64) ?Transform {
    const local = Transform.init(obj);
    const world = Transform.compose(parent, local);
    if (obj.id == target_id) return world;
    for (obj.children.items) |*child| {
        if (findWorldTransformInTree(child, world, target_id)) |t| return t;
    }
    return null;
}

pub const Tool = struct {
    onCanvasClick: *const fn (*const ToolContext) anyerror!void,
};
```

A.27. src/toolbar/Toolbar.zig

```
const Tool = @import("Tool.zig");

const Toolbar = @This();

pub const ToolDescriptor = struct {
    name: []const u8,
    icon_tvg: []const u8,
    implementation: *const Tool.Tool,
};

tools: []const ToolDescriptor,
selected_index: ?usize,

pub fn init(tools_list: []const ToolDescriptor) Toolbar {
    return .{
        .tools = tools_list,
        .selected_index = null,
    };
}

pub fn deinit(_: *Toolbar) void {}

pub fn currentDescriptor(self: *const Toolbar) ?*const ToolDescriptor {
    if (self.tools.len == 0) return null;
    if (self.selected_index) |index| {
        return &self.tools[index];
    }
    return null;
}

pub fn select(self: *Toolbar, index: ?usize) void {
    if (index == self.selected_index) {
        self.selected_index = null;
        return;
    }
    if (index) |i| {
        if (i < self.tools.len) {
```

```
            self.selected_index = i;
        }
    } else {
        self.selected_index = null;
    }
}
```

A.28. src/toolbar/tools.zig

```
const Toolbar = @import("Toolbar.zig");
const line = @import("tools/line.zig");
const ellipse = @import("tools/ellipse.zig");
const broken = @import("tools/broken.zig");
const icons = @import("../icons.zig");

pub const default_tools = [_]Toolbar.ToolDescriptor{
    .{
        .name = "Line",
        .icon_tvg = icons.line,
        .implementation = &line.tool,
    },
    .{
        .name = "Ellipse",
        .icon_tvg = icons.ellipse,
        .implementation = &ellipse.tool,
    },
    .{
        .name = "Broken line",
        .icon_tvg = icons.broken,
        .implementation = &broken.tool,
    },
};
```

A.29. src/toolbar/tools/broken.zig

```
const Tool = @import("../Tool.zig");
const shape = @import("../../models/shape/shape.zig");

fn onCanvasClick(ctx: *const Tool.ToolContext) !void {
    const canvas = ctx.canvas;
    var obj = shape.createObject(canvas.allocator, .broken) catch return;
    defer obj.deinit(canvas.allocator);
    try ctx.addObject(obj);
}
pub const tool = Tool.Tool{ .onCanvasClick = onCanvasClick };
```

A.30. src/toolbar/tools/ellipse.zig

```
const Tool = @import("../Tool.zig");
const shape = @import("../../models/shape/shape.zig");

fn onCanvasClick(ctx: *const Tool.ToolContext) !void {
    const canvas = ctx.canvas;
    var obj = shape.createObject(canvas.allocator, .ellipse) catch return;
    defer obj.deinit(canvas.allocator);
    try ctx.addObject(obj);
}
pub const tool = Tool.Tool{ .onCanvasClick = onCanvasClick };
```

A.31. src/toolbar/tools/line.zig

```
const std = @import("std");
const Canvas = @import("../../Canvas.zig");
const Tool = @import("../Tool.zig");
const shape = @import("../../models/shape/shape.zig");
```

```
fn onCanvasClick(ctx: *const Tool.ToolContext) !void {
    const canvas = ctx.canvas;
    var obj = shape.createObject(canvas.allocator, .line) catch return;
    defer obj.deinit(canvas.allocator);
    try ctx.addObject(obj);
}
pub const tool = Tool.Tool{ .onCanvasClick = onCanvasClick };
```

A.32. src/ui/canvas_view.zig

```
const std = @import("std");
const dvui = @import("dvui");
const dvui_ext = @import("dvui_ext.zig");
const Canvas = @import("../Canvas.zig");
const Document = @import("../models/Document.zig");
const Property = @import("../models/Property.zig").Property;
const PropertyData = @import("../models/Property.zig").Data;
const Rect_i = @import("../models/basic_models.zig").Rect_i;
const Point2_f = @import("../models/basic_models.zig").Point2_f;
const Tool = @import("../toolbar/Tool.zig");
const RenderStats = @import("../render/RenderStats.zig");
const icons = @import("../icons.zig");

pub fn canvasView(canvas: *Canvas, selected_object_id: ?u64, content_rect_scale: dvui.RectScale) void {
    var textured = dvui_ext.texturedBox(content_rect_scale, dvui.Rect.all(20));
    {
        var overlay = dvui.overlay(@src(), .{ .expand = .both });
        {
            const overlay_parent = dvui.parentGet();
            const init_options: dvui.ScrollAreaWidget.InitOpts = .{
                .scroll_info = &canvas.scroll,
                .vertical_bar = .auto,
                .horizontal_bar = .auto,
                .process_events_after = false,
            };
            var scroll = dvui.scrollArea(
                @src(),
                init_options,
                .{
                    .expand = .both,
                    .background = false,
                },
            );
            {
                drawCanvasContent(canvas, scroll);
                handleCanvasZoom(canvas, scroll);
                handleCanvasMouse(canvas, scroll, selected_object_id);
            }

            const scroll_parent = dvui.parentGet();
            dvui.parentSet(overlay_parent);

            const vbar = scroll.vbar;
            const hbar = scroll.hbar;
            if (vbar != null) {

                // std.debug.print("{any}", .{vbar.?.data()});
            }
            if (hbar != null) {
                // std.debug.print("{any}", .{hbar.?.data()});
            }

            // Тулбар поверх scroll
            var toolbar_box = dvui.box(
                @src(),
                .{ .dir = .horizontal },
                .{},
```

```
            );
            {
                drawToolbar(canvas);
               // Сохраняем rect тулбара для следующего кадра — в handleCanvasMouse исключаем из него клики
                canvas.toolbar_rect_scale = toolbar_box.data().contentRectScale();
            }
            toolbar_box.deinit();

            // Панель свойств поверх scroll (правый верхний угол)
            if (selected_object_id) |obj_id| {
                if (canvas.document.findObjectById(obj_id)) |obj| {
                    var properties_box = dvui.box(
                        @src(),
                        .{ .dir = .horizontal },
                        .{
                            .gravity_x = 1.0,
                            .gravity_y = 0.0,
                        },
                    );
                    {
                        drawPropertiesPanel(canvas, obj);
                        // Сохраняем rect панели свойств для следующего кадра — в handleCanvasMouse
                        ↪  исключаем из него клики
                        canvas.properties_rect_scale = properties_box.data().contentRectScale();
                    }
                    properties_box.deinit();
                }
            }

            drawCanvasLabelPanel();
            if (canvas.show_render_stats)
                drawStatsPanel(canvas.render_engine.getStats(), canvas.frame_index);

            if (canvas.properties_rect_scale) |prs| {
                for (dvui.events()) |*e| {
                    if (e.handled) continue;
                    if (e.evt != .mouse) continue;
                    const mouse = &e.evt.mouse;
                    if (mouse.action != .wheel_x and mouse.action != .wheel_y) continue;
                    const pt = prs.pointFromPhysical(mouse.p);
                    const r = prs.r;
                    if (pt.x >= 0 and pt.x * prs.s < r.w and pt.y >= 0 and pt.y * prs.s < r.h) {
                        e.handled = true;
                    }
                }
            }

            if (!init_options.process_events_after) {
                if (scroll.scroll) |*sc| {
                    dvui.clipSet(sc.prevClip);
                    sc.processEventsAfter();
                }
            }

            dvui.parentSet(scroll_parent);
            scroll.deinit();
        }
        overlay.deinit();
    }
    textured.deinit();
}

fn drawCanvasContent(canvas: *Canvas, scroll: anytype) void {
    const natural_scale = if (canvas.native_scaling) 1 else dvui.windowNaturalScale();
    const img_size = canvas.getZoomedImageSize();
    const viewport_rect = scroll.data().contentRect();
    const scroll_current = dvui.Point{ .x = canvas.scroll.viewport.x, .y = canvas.scroll.viewport.y };
```

```
    const viewport_px = dvui.Rect{
        .x = viewport_rect.x * natural_scale,
        .y = viewport_rect.y * natural_scale,
        .w = viewport_rect.w * natural_scale,
        .h = viewport_rect.h * natural_scale,
    };
    const scroll_px = dvui.Point{
        .x = scroll_current.x * natural_scale,
        .y = scroll_current.y * natural_scale,
    };

    const changed = canvas.updateVisibleImageRect(viewport_px, scroll_px);
    if (changed)
        canvas.requestRedraw();
    canvas.processPendingRedraw() catch |err| {
        std.debug.print("processPendingRedraw error: {}\n", .{err});
    };

    const content_w_px: u32 = img_size.x + img_size.w;
    const content_h_px: u32 = img_size.y + img_size.h;
    const content_w = @as(f32, @floatFromInt(content_w_px)) / natural_scale;
    const content_h = @as(f32, @floatFromInt(content_h_px)) / natural_scale;

    var canvas_layer = dvui.overlay(
        @src(),
        .{ .min_size_content = .{ .w = content_w, .h = content_h }, .background = false },
    );
    {
        if (canvas.texture) |tex| {
            const vis = canvas._visible_rect orelse Rect_i{ .x = 0, .y = 0, .w = 0, .h = 0 };
            const left = @as(f32, @floatFromInt(img_size.x + vis.x)) / natural_scale;
            const top = @as(f32, @floatFromInt(img_size.y + vis.y)) / natural_scale;

            _ = dvui.image(
                @src(),
                .{ .source = .{ .texture = tex } },
                .{
                    .background = false,
                    .expand = .none,
                    .gravity_x = 0.0,
                    .gravity_y = 0.0,
                    .margin = .{ .x = left, .y = top, .w = canvas.pos.x, .h = canvas.pos.y },
                    .min_size_content = .{
                        .w = @as(f32, @floatFromInt(vis.w)) / natural_scale,
                        .h = @as(f32, @floatFromInt(vis.h)) / natural_scale,
                    },
                    .max_size_content = .{
                        .w = @as(f32, @floatFromInt(vis.w)) / natural_scale,
                        .h = @as(f32, @floatFromInt(vis.h)) / natural_scale,
                    },
                },
            );
        }
    }
    canvas_layer.deinit();
}

fn handleCanvasZoom(canvas: *Canvas, scroll: anytype) void {
    const ctrl = dvui.currentWindow().modifiers.control();
    if (!ctrl) return;

    const natural_scale = if (canvas.native_scaling) 1 else dvui.windowNaturalScale();

    for (dvui.events()) |*e| {
        switch (e.evt) {
            .mouse => |*mouse| {
```

```
                const action = mouse.action;
                if (dvui.eventMatchSimple(e, scroll.data()) and (action == .wheel_x or action == .wheel_y)) {
                    switch (action) {
                        .wheel_y => |y| {
                            const viewport_pt = scroll.data().contentRectScale().pointFromPhysical(mouse.p);
                            const content_pt = dvui.Point{
                                .x = viewport_pt.x + canvas.scroll.viewport.x,
                                .y = viewport_pt.y + canvas.scroll.viewport.y,
                            };
                            const doc_pt = canvas.contentPointToDocument(content_pt, natural_scale);

                            // canvas.addZoom(y / 1000);
                            canvas.multZoom(1 + y / 2000);
                            canvas.requestRedraw();

                            const doc_pt_after = canvas.contentPointToDocument(content_pt, natural_scale);

                            const zoom = canvas.getZoom();
                            const dx = (doc_pt_after.x - doc_pt.x) * zoom / natural_scale;
                            const dy = (doc_pt_after.y - doc_pt.y) * zoom / natural_scale;

                            canvas.scroll.viewport.x -= dx;
                            canvas.scroll.viewport.y -= dy;
                        },
                        else => {},
                    }
                    e.handled = true;
                }
            },
            else => {},
        }
    }
}

fn handleCanvasMouse(canvas: *Canvas, scroll: *dvui.ScrollAreaWidget, selected_object_id: ?u64) void {
    const natural_scale = if (canvas.native_scaling) 1 else dvui.windowNaturalScale();
    const scroll_data = scroll.data();

    for (dvui.events()) |*e| {
        switch (e.evt) {
            .mouse => |*mouse| {
                if (mouse.action != .press or mouse.button != .left) continue;
                if (e.handled) continue;
                if (!dvui.eventMatchSimple(e, scroll_data)) continue;

                // Не обрабатывать клик, если он попал в область тулбара (rect с предыдущего кадра).
                if (canvas.toolbar_rect_scale) |trs| {
                    const pt = trs.pointFromPhysical(mouse.p);
                    const r = trs.r;
                    if (pt.x >= 0 and pt.x * trs.s < r.w and pt.y >= 0 and pt.y * trs.s < r.h) continue;
                }
                // Не обрабатывать клик, если он попал в область панели свойств (rect с предыдущего кадра).
                if (canvas.properties_rect_scale) |prs| {
                    const pt = prs.pointFromPhysical(mouse.p);
                    const r = prs.r;
                    if (pt.x >= 0 and pt.x * prs.s < r.w and pt.y >= 0 and pt.y * prs.s < r.h) continue;
                }

                const viewport_pt = scroll_data.contentRectScale().pointFromPhysical(mouse.p);
                const content_pt = dvui.Point{
                    .x = viewport_pt.x + canvas.scroll.viewport.x,
                    .y = viewport_pt.y + canvas.scroll.viewport.y,
                };
                const doc_pt = canvas.contentPointToDocument(content_pt, natural_scale);
                canvas.cursor_document_point = if (canvas.isContentPointOnDocument(content_pt,
↪  natural_scale)) doc_pt else null;
                if (canvas.cursor_document_point) |point| {
```

```
                    if (canvas.toolbar.currentDescriptor()) |desc| {
                        var ctx = Tool.ToolContext{
                            .canvas = canvas,
                            .document_point = point,
                            .selected_object_id = selected_object_id,
                        };
                        desc.implementation.onCanvasClick(&ctx) catch |err| {
                            std.debug.print("onCanvasClick error: {}\n", .{err});
                        };
                    }
                }
            },
            else => {},
        }
    }
}

fn drawToolbar(canvas: *Canvas) void {
    const tools_list = canvas.toolbar.tools;
    if (tools_list.len == 0) return;

    var bar = dvui.box(
        @src(),
        .{ .dir = .vertical },
        .{
            .gravity_x = 0.0,
            .gravity_y = 0.0,
            .padding = dvui.Rect.all(6),
            .corner_radius = dvui.Rect.all(8),
            .background = true,
            .color_fill = dvui.Color.black.opacity(0.2),
            .margin = dvui.Rect{ .x = 16, .y = 16 },
        },
    );
    {
        var to_select: ?usize = null;
        for (tools_list, 0..) |*tool_desc, i| {
            const is_selected = canvas.toolbar.selected_index == i;
            const selected_fill = dvui.themeGet().focus;
            const opts: dvui.Options = .{
                .id_extra = i,
                .color_fill = if (is_selected) selected_fill else null,
            };
            if (dvui.buttonIcon(@src(), tool_desc.name, tool_desc.icon_tvg, .{}, .{}, opts)) {
                to_select = i;
            }
        }
        if (to_select) |index| {
            canvas.toolbar.select(index);
        }
    }
    bar.deinit();
}

fn drawPropertiesPanel(canvas: *Canvas, selected_object: *Document.Object) void {
    var panel = dvui.box(
        @src(),
        .{ .dir = .vertical },
        .{
            .padding = dvui.Rect.all(8),
            .corner_radius = dvui.Rect.all(8),
            .background = true,
            .color_fill = dvui.Color.black.opacity(0.2),
            .min_size_content = .width(300),
            .max_size_content = .width(300),
            .margin = dvui.Rect{ .w = 32, .y = 16, .h = 100 },
        },
```

```
    );
    {
        dvui.label(@src(), "Properties", .{}, .{});
        var scroll = dvui.scrollArea(@src(), .{
            .horizontal = .none,
            .vertical = .auto,
        }, .{
            .expand = .both,
        });
        {
            for (selected_object.properties.items, 0..) |*prop, i| {
                drawPropertyEditor(canvas, selected_object, prop, i);
            }
        }
        scroll.deinit();
    }
    panel.deinit();
}

fn drawCanvasLabelPanel() void {
    var panel = dvui.box(
        @src(),
        .{ .dir = .vertical },
        .{
            .gravity_x = 0.5,
            .gravity_y = 0.0,
            .padding = dvui.Rect.all(8),
            .corner_radius = dvui.Rect.all(8),
            .background = true,
            .color_fill = dvui.Color.black.opacity(0.2),
            .margin = dvui.Rect{ .x = 16, .y = 16 },
        },
    );
    {
        dvui.label(@src(), "Canvas", .{}, .{});
    }
    panel.deinit();
}

fn drawStatsPanel(stats: RenderStats, frame_index: u64) void {
    var panel = dvui.box(
        @src(),
        .{ .dir = .vertical },
        .{
            .gravity_x = 0.0,
            .gravity_y = 1.0,
            .padding = dvui.Rect.all(8),
            .corner_radius = dvui.Rect.all(8),
            .background = true,
            .color_fill = dvui.Color.black.opacity(0.2),
            .margin = dvui.Rect{ .x = 16, .h = 16 },
        },
    );
    {
        dvui.label(@src(), "Frame time: {d:.2}ms", .{@as(f32, @floatFromInt(stats.render_time_ns)) /
↪  std.time.ns_per_ms}, .{});
        dvui.label(@src(), "Frame index: {}", .{frame_index}, .{});
    }
    panel.deinit();
}

fn applyPropertyPatch(canvas: *Canvas, obj: *Document.Object, patch: Property) void {
    obj.setProperty(canvas.allocator, patch) catch {};
    canvas.requestRedraw();
}
```

```
fn drawPropertyEditor(canvas: *Canvas, obj: *Document.Object, prop: *const Property, row_index: usize)
 ↪  void {
    const row_id: usize = row_index * 16;
    const is_even = row_index % 2 == 0;
    var row = dvui.box(
        @src(),
        .{ .dir = .vertical },
        .{
            .id_extra = row_id,
            .expand = .horizontal,
            .padding = dvui.Rect{ .y = 2, .x = 4 },
            .corner_radius = dvui.Rect.all(4),
            .background = is_even,
            .color_fill = if (is_even) dvui.Color.black.opacity(0.4) else .{},
        },
    );
    {
        const tag = std.meta.activeTag(prop.data);

        dvui.labelNoFmt(@src(), propertyLabel(tag), .{}, .{});

        switch (prop.data) {
            .position => |pos| {
                var next = pos;
                var changed = false;
                {
                    var subrow = dvui.box(@src(), .{ .dir = .horizontal }, .{ .expand = .horizontal });
                    dvui.labelNoFmt(@src(), "x:", .{}, .{});
                    const T = @TypeOf(next.x);
                    const res = dvui.textEntryNumber(@src(), T, .{ .value = &next.x }, .{ .expand =
                     ↪  .horizontal });
                    subrow.deinit();
                    changed = res.changed or changed;
                }
                {
                    var subrow = dvui.box(@src(), .{ .dir = .horizontal }, .{ .expand = .horizontal });
                    dvui.labelNoFmt(@src(), "y:", .{}, .{});
                    const T = @TypeOf(next.y);
                    const res = dvui.textEntryNumber(@src(), T, .{ .value = &next.y }, .{ .expand =
                     ↪  .horizontal });
                    subrow.deinit();
                    changed = res.changed or changed;
                }
                if (changed) {
                    applyPropertyPatch(canvas, obj, .{ .data = .{ .position = next } });
                }
            },
            .angle => |angle| {
                var next = angle;
                {
                    var subrow = dvui.box(@src(), .{ .dir = .horizontal }, .{ .expand = .horizontal });
                    dvui.labelNoFmt(@src(), "deg:", .{}, .{});
                    var degrees: f32 = next * 180.0 / std.math.pi;
                    const res = dvui.textEntryNumber(@src(), f32, .{ .value = &degrees }, .{ .expand =
                     ↪  .horizontal });
                    subrow.deinit();
                    if (res.changed) {
                        next = degrees * std.math.pi / 180.0;
                        applyPropertyPatch(canvas, obj, .{ .data = .{ .angle = next } });
                    }
                }
            },
            .scale => |scale| {
                var next = scale;
                var changed = false;
                {
                    var subrow = dvui.box(@src(), .{ .dir = .horizontal }, .{ .expand = .horizontal });
```

```
            dvui.labelNoFmt(@src(), "x:", .{}, .{});
            const T = @TypeOf(next.scale_x);
            const res = dvui.textEntryNumber(@src(), T, .{ .value = &next.scale_x, .min = @as(T,
             ↪  0.0), .max = @as(T, 10.0) }, .{ .expand = .horizontal });
            subrow.deinit();
            changed = res.changed or changed;
        }
        {
            var subrow = dvui.box(@src(), .{ .dir = .horizontal }, .{ .expand = .horizontal });
            dvui.labelNoFmt(@src(), "y:", .{}, .{});
            const T = @TypeOf(next.scale_y);
            const res = dvui.textEntryNumber(@src(), T, .{ .value = &next.scale_y, .min = @as(T,
             ↪  0.0), .max = @as(T, 10.0) }, .{ .expand = .horizontal });
            subrow.deinit();
            changed = res.changed or changed;
        }
        if (changed) {
            applyPropertyPatch(canvas, obj, .{ .data = .{ .scale = next } });
        }
    },
    .visible => |v| {
        var next = v;
        if (dvui.checkbox(@src(), &next, "Visible", .{})) {
            applyPropertyPatch(canvas, obj, .{ .data = .{ .visible = next } });
        }
    },
    .opacity => |opacity| {
        var next = opacity;
        if (dvui.sliderEntry(@src(), "{d:0.2}", .{ .value = &next, .min = 0.0, .max = 1.0,
         ↪  .interval = 0.01 }, .{ .expand = .horizontal })) {
            applyPropertyPatch(canvas, obj, .{ .data = .{ .opacity = next } });
        }
    },
    .locked => |v| {
        var next = v;
        if (dvui.checkbox(@src(), &next, "Locked", .{})) {
            applyPropertyPatch(canvas, obj, .{ .data = .{ .locked = next } });
        }
    },
    .size => |size| {
        var next = size;
        var changed = false;
        {
            var subrow = dvui.box(@src(), .{ .dir = .horizontal }, .{ .expand = .horizontal });
            dvui.labelNoFmt(@src(), "w:", .{}, .{});
            const T = @TypeOf(next.w);
            const res = dvui.textEntryNumber(@src(), T, .{ .value = &next.w, .min = @as(T, 0.0) },
             ↪  .{ .expand = .horizontal });
            subrow.deinit();
            changed = res.changed or changed;
        }
        {
            var subrow = dvui.box(@src(), .{ .dir = .horizontal }, .{ .expand = .horizontal });
            dvui.labelNoFmt(@src(), "h:", .{}, .{});
            const T = @TypeOf(next.h);
            const res = dvui.textEntryNumber(@src(), T, .{ .value = &next.h, .min = @as(T, 0.0) },
             ↪  .{ .expand = .horizontal });
            subrow.deinit();
            changed = res.changed or changed;
        }
        if (changed) {
            applyPropertyPatch(canvas, obj, .{ .data = .{ .size = next } });
        }
    },
    .radii => |radii| {
        var next = radii;
        var changed = false;
```

```
        {
            var subrow = dvui.box(@src(), .{ .dir = .horizontal }, .{ .expand = .horizontal });
            dvui.labelNoFmt(@src(), "x:", .{}, .{});
            const T = @TypeOf(next.x);
            const res = dvui.textEntryNumber(@src(), T, .{ .value = &next.x, .min = @as(T, 0.0) },
             ↪  .{ .expand = .horizontal });
            subrow.deinit();
            changed = res.changed or changed;
        }
        {
            var subrow = dvui.box(@src(), .{ .dir = .horizontal }, .{ .expand = .horizontal });
            dvui.labelNoFmt(@src(), "y:", .{}, .{});
            const T = @TypeOf(next.y);
            const res = dvui.textEntryNumber(@src(), T, .{ .value = &next.y, .min = @as(T, 0.0) },
             ↪  .{ .expand = .horizontal });
            subrow.deinit();
            changed = res.changed or changed;
        }
        if (changed) {
            applyPropertyPatch(canvas, obj, .{ .data = .{ .radii = next } });
        }
    },
    .arc_percent => |pct| {
        var next = pct;
        if (dvui.sliderEntry(@src(), "{d:0.0}%", .{ .value = &next, .min = 0.0, .max = 100.0,
         ↪  .interval = 1.0 }, .{ .expand = .horizontal })) {
            applyPropertyPatch(canvas, obj, .{ .data = .{ .arc_percent = next } });
        }
    },
    .end_point => |pt| {
        var next = pt;
        var changed = false;
        {
            var subrow = dvui.box(@src(), .{ .dir = .horizontal }, .{ .expand = .horizontal });
            dvui.labelNoFmt(@src(), "x:", .{}, .{});
            const T = @TypeOf(next.x);
            const res = dvui.textEntryNumber(@src(), T, .{ .value = &next.x }, .{ .expand =
             ↪  .horizontal });
            subrow.deinit();
            changed = res.changed or changed;
        }
        {
            var subrow = dvui.box(@src(), .{ .dir = .horizontal }, .{ .expand = .horizontal });
            dvui.labelNoFmt(@src(), "y:", .{}, .{});
            const T = @TypeOf(next.y);
            const res = dvui.textEntryNumber(@src(), T, .{ .value = &next.y }, .{ .expand =
             ↪  .horizontal });
            subrow.deinit();
            changed = res.changed or changed;
        }
        if (changed) {
            applyPropertyPatch(canvas, obj, .{ .data = .{ .end_point = next } });
        }
    },
    .points => |points| {
        var list = std.ArrayList(Point2_f).empty;
        list.appendSlice(canvas.allocator, points) catch {
            dvui.label(@src(), "Points: {d}", .{points.len}, .{});
            return;
        };
        defer list.deinit(canvas.allocator);
        dvui.label(@src(), "Points: {d}", .{list.items.len}, .{});

        var changed = false;
        var to_delete: ?usize = null;

        for (list.items, 0..) |*pt, i| {
```

```
// Одна строка: крестик удаления + paned с X/Y пополам
var subrow = dvui.box(
    @src(),
    .{ .dir = .horizontal },
    .{
        .expand = .horizontal,
        .id_extra = i,
    },
);
{
    // Крестик удаления
    if (dvui.buttonIcon(@src(), "Delete", icons.cross, .{}, .{}, .{
        .id_extra = i,
        .gravity_y = 0.5,
        .margin = .{
            .x = 8,
        },
    })) {
        to_delete = i;
    }

    // Панель с X и Y, разделёнными пополам
    var split_ratio: f32 = 0.5;
    var paned = dvui.paned(
        @src(),
        .{
            .direction = .horizontal,
            .collapsed_size = 0.0,
            .split_ratio = &split_ratio,
            .handle_size = 0,
        },
        .{
            .expand = .horizontal,
        },
    );
    {
        if (paned.showFirst()) {
            var x_box = dvui.box(
                @src(),
                .{ .dir = .horizontal },
                .{ .expand = .both },
            );
            {
                dvui.labelNoFmt(@src(), "x:", .{}, .{
                    .gravity_y = 0.5,
                });
                const Tx = @TypeOf(pt.x);
                const res_x = dvui.textEntryNumber(
                    @src(),
                    Tx,
                    .{ .value = &pt.x },
                    .{ .expand = .horizontal },
                );
                changed = res_x.changed or changed;
            }
            x_box.deinit();
        }

        if (paned.showSecond()) {
            var y_box = dvui.box(
                @src(),
                .{ .dir = .horizontal },
                .{ .expand = .both },
            );
            {
                dvui.labelNoFmt(@src(), "y:", .{}, .{
                    .gravity_y = 0.5,
```

```
                        });
                        const Ty = @TypeOf(pt.y);
                        const res_y = dvui.textEntryNumber(
                            @src(),
                            Ty,
                            .{ .value = &pt.y },
                            .{ .expand = .horizontal },
                        );
                        changed = res_y.changed or changed;
                    }
                    y_box.deinit();
                }
            }
            paned.deinit();
        }
        subrow.deinit();
    }

    // Удаление выбранной точки
    if (to_delete) |idx| {
        _ = list.orderedRemove(idx);
        changed = true;
    }

    // Кнопка добавления новой точки (одна на весь список)
    if (dvui.button(@src(), "Add point", .{}, .{})) {
        const T = @TypeOf(list.items[0]);
        const new_point: T = if (list.items.len > 0)
            list.items[list.items.len - 1]
        else
            .{ .x = 0, .y = 0 };
        list.append(canvas.allocator, new_point) catch {};
        changed = true;
    }

    if (changed) {
        const slice = canvas.allocator.dupe(Point2_f, list.items) catch return;
        obj.setProperty(canvas.allocator, .{ .data = .{ .points = slice } }) catch {
            canvas.allocator.free(slice);
            return;
        };
        canvas.requestRedraw();
    }
},
.fill_rgba => |rgba| {
    drawColorEditor(canvas, obj, rgba, true);
},
.stroke_rgba => |rgba| {
    drawColorEditor(canvas, obj, rgba, false);
},
.thickness => |t| {
    var next = t;
    {
        var subrow = dvui.box(@src(), .{ .dir = .horizontal }, .{ .expand = .horizontal });
        dvui.labelNoFmt(@src(), "thickness:", .{}, .{});
        const T = @TypeOf(next);
        const res = dvui.textEntryNumber(@src(), T, .{ .value = &next, .min = @as(T, 0.0), .max
         ↪  = @as(T, 100.0) }, .{ .expand = .horizontal });
        subrow.deinit();
        if (res.changed) {
            applyPropertyPatch(canvas, obj, .{ .data = .{ .thickness = next } });
        }
    }
},
.closed => |v| {
    var next = v;
    if (dvui.checkbox(@src(), &next, "Closed", .{})) {
```

```
                    applyPropertyPatch(canvas, obj, .{ .data = .{ .closed = next } });
                }
            },
            .filled => |v| {
                var next = v;
                if (dvui.checkbox(@src(), &next, "Filled", .{})) {
                    applyPropertyPatch(canvas, obj, .{ .data = .{ .filled = next } });
                }
            },
        }
    }
    row.deinit();
}

fn drawColorEditor(canvas: *Canvas, obj: *Document.Object, rgba: u32, is_fill: bool) void {
    var hsv = dvui.Color.HSV.fromColor(rgbaToColor(rgba));
    if (dvui.colorPicker(
        @src(),
        .{ .hsv = &hsv, .dir = .horizontal, .sliders = .rgb, .alpha = true, .hex_text_entry = true },
        .{ .expand = .horizontal },
    )) {
        const next = colorToRgba(hsv.toColor());
        const patch: Property = if (is_fill)
            .{ .data = .{ .fill_rgba = next } }
        else
            .{ .data = .{ .stroke_rgba = next } };
        applyPropertyPatch(canvas, obj, patch);
    }
}

fn propertyLabel(tag: std.meta.Tag(PropertyData)) []const u8 {
    return switch (tag) {
        .position => "Position",
        .angle => "Angle",
        .scale => "Scale",
        .visible => "Visible",
        .opacity => "Opacity",
        .locked => "Locked",
        .size => "Size",
        .radii => "Radii",
        .arc_percent => "Arc %",
        .end_point => "End point",
        .points => "Points",
        .fill_rgba => "Fill color",
        .stroke_rgba => "Stroke color",
        .thickness => "Thickness",
        .closed => "Closed",
        .filled => "Filled",
    };
}

fn rgbaToColor(rgba: u32) dvui.Color {
    return .{
        .r = @intCast((rgba >> 24) & 0xFF),
        .g = @intCast((rgba >> 16) & 0xFF),
        .b = @intCast((rgba >> 8) & 0xFF),
        .a = @intCast((rgba >> 0) & 0xFF),
    };
}

fn colorToRgba(color: dvui.Color) u32 {
    return (@as(u32, color.r) << 24) |
        (@as(u32, color.g) << 16) |
        (@as(u32, color.b) << 8) |
        (@as(u32, color.a) << 0);
}
```

A.33. src/ui/dvui_ext.zig

```
const std = @import("std");
const dvui = @import("dvui");
const TexturedBox = @import("./types/TexturedBox.zig");

pub fn texturedBox(rs: dvui.RectScale, corner_radius: dvui.Rect) TexturedBox {
    return TexturedBox.init(rs, corner_radius);
}
```

A.34. src/ui/frame.zig

```
const dvui = @import("dvui");
const WindowContext = @import("../WindowContext.zig");
const menu_bar = @import("menu_bar.zig");
const tab_bar = @import("tab_bar.zig");
const left_panel = @import("left_panel.zig");
const right_panel = @import("right_panel.zig");

pub fn guiFrame(ctx: *WindowContext) bool {
    for (dvui.events()) |*e| {
        if (e.evt == .window and e.evt.window.action == .close) return false;
        if (e.evt == .app and e.evt.app.action == .quit) return false;
    }

    var root = dvui.box(
        @src(),
        .{ .dir = .vertical },
        .{ .expand = .both, .background = true, .style = .window },
    );
    {
        menu_bar.menuBar(ctx);
        tab_bar.tabBar(ctx);

        var content_row = dvui.box(@src(), .{ .dir = .horizontal }, .{ .expand = .both });
        {
            left_panel.leftPanel(ctx);

            right_panel.rightPanel(ctx);
        }
        content_row.deinit();
    }
    root.deinit();

    return true;
}
```

A.35. src/ui/left_panel.zig

```
const std = @import("std");
const dvui = @import("dvui");
const WindowContext = @import("../WindowContext.zig");
const Document = @import("../models/Document.zig");
const icons = @import("../icons.zig");
const Object = Document.Object;

const panel_gap: f32 = 12;
const panel_padding: f32 = 5;
const panel_radius: f32 = 24;
const fill_color = dvui.Color.black.opacity(0.2);

const ObjectTreeCallback = union(enum) {
    select: u64,
    delete: u64,
};

fn shapeLabel(shape: Object.ShapeKind) []const u8 {
```

```
    return switch (shape) {
        .line => "Line",
        .ellipse => "Ellipse",
        .broken => "Broken line",
    };
}

fn objectTreeRow(open_doc: *WindowContext.OpenDocument, obj: *Object, depth: u32, object_callback:
↪   *?ObjectTreeCallback) void {
    const indent_px = depth * 18;
    const is_selected: bool = open_doc.selected_object_id == obj.id;
    const row_id: usize = @intCast(obj.id);

    const focus_color = dvui.themeGet().focus;

    var row = dvui.box(
        @src(),
        .{ .dir = .horizontal },
        .{
            .id_extra = row_id,
            .expand = .horizontal,
        },
    );
    {
        var hovered: bool = false;
        const row_data = row.data();
        var select_row: bool = false;

        // Ручная обработка hover/click по строке без пометки события как handled,
        // чтобы кнопка удаления могла нормально получать свои события.
        for (dvui.events()) |*e| {
            switch (e.evt) {
                .mouse => |*mouse| {
                    if (!dvui.eventMatchSimple(e, row_data)) continue;
                    hovered = true;
                    if (mouse.action == .release and mouse.button == .left) {
                        select_row = true;
                    }
                },
                else => {},
            }
        }

        const background = is_selected or hovered;
        var content = dvui.box(@src(), .{ .dir = .horizontal }, .{
            .expand = .horizontal,
            .margin = dvui.Rect{ .x = @floatFromInt(indent_px) },
            .background = background,
            .color_fill = if (is_selected) focus_color.opacity(0.35) else if (hovered)
            ↪   focus_color.opacity(0.18) else null,
        });
        {
            dvui.labelNoFmt(
                @src(),
                shapeLabel(obj.shape),
                .{},
                .{ .id_extra = row_id, .expand = .horizontal },
            );

            if (hovered) {
                const delete_opts: dvui.Options = .{
                    .id_extra = row_id +% 1,
                    .margin = dvui.Rect{ .x = 4, .w = 6 },
                    .padding = dvui.Rect.all(2),
                    .gravity_y = 0.5,
                    .gravity_x = 1.0,
                };
```

```
                if (dvui.buttonIcon(@src(), "Delete object", icons.trash, .{}, .{}, delete_opts)) {
                    object_callback.* = .{ .delete = obj.id };
                    select_row = false;
                }
            }
        }
        content.deinit();

        if (select_row) {
            object_callback.* = .{ .select = obj.id };
        }
    }
    row.deinit();
    for (obj.children.items) |*child| {
        objectTreeRow(open_doc, child, depth + 1, object_callback);
    }
}

fn objectTree(ctx: *WindowContext) void {
    const active_doc = ctx.activeDocument();
    var object_callback: ?ObjectTreeCallback = null;
    if (active_doc) |open_doc| {
        const doc = &open_doc.document;
        if (doc.objects.items.len == 0) {
            dvui.label(@src(), "No objects", .{}, .{});
        } else {
            for (doc.objects.items) |*obj| {
                objectTreeRow(open_doc, obj, 0, &object_callback);
            }
        }
        if (object_callback) |callback| {
            switch (callback) {
                .select => |obj_id| {
                    if (open_doc.selected_object_id == obj_id) {
                        open_doc.selected_object_id = null;
                    } else {
                        open_doc.selected_object_id = obj_id;
                    }
                },
                .delete => |obj_id| {
                    _ = doc.removeObjectById(ctx.allocator, obj_id);
                    if (open_doc.selected_object_id == obj_id)
                        open_doc.selected_object_id = null;
                    open_doc.canvas.requestRedraw();
                },
            }
        }
    } else {
        dvui.label(@src(), "No document", .{}, .{});
    }
}

pub fn leftPanel(ctx: *WindowContext) void {
    var padding = dvui.Rect.all(panel_gap);
    padding.w = 0;
    var panel = dvui.box(
        @src(),
        .{ .dir = .vertical },
        .{
            .expand = .vertical,
            // Фиксированная ширина левой панели
            .min_size_content = .{ .w = 220 },
            .max_size_content = .{ .h = undefined, .w = 220 },
            .background = true,
            .padding = padding,
        },
    );
```

```
{
    // Нижняя часть: настройки
    var settings_section = dvui.box(
        @src(),
        .{ .dir = .vertical },
        .{
            .expand = .horizontal,
            .gravity_y = 1.0,
            .margin = .{ .y = 5 },
            .padding = dvui.Rect.all(panel_padding),
            .corner_radius = dvui.Rect.all(panel_radius),
            .color_fill = fill_color,
            .background = true,
        },
    );
    {
        dvui.label(@src(), "Settings", .{}, .{});

        const active_doc = ctx.activeDocument();
        if (active_doc) |doc| {
            const canvas = &doc.canvas;
            if (dvui.checkbox(@src(), &canvas.native_scaling, "Scaling", .{})) {}
            if (dvui.checkbox(@src(), &canvas.draw_document, "Draw document", .{})) {
                canvas.requestRedraw();
            }
            if (dvui.checkbox(@src(), &canvas.show_render_stats, "Show stats", .{})) {}
            {
                dvui.label(@src(), "Rendering quality", .{}, .{});
                var quality = canvas.getRenderingQuality();
                if (dvui.sliderEntry(
                    @src(),
                    "{d:0.0}%",
                    .{ .value = &quality, .min = 1.0, .max = 100.0, .interval = 1.0 },
                    .{ .expand = .horizontal },
                )) {
                    canvas.setRenderingQuality(quality);
                }
            }
            if (!canvas.draw_document) {
                if (dvui.button(@src(), if (doc.cpu_render.type == .Gradient) "Gradient" else
                ↪  "Squares", .{}, .{})) {
                    if (doc.cpu_render.type == .Gradient) {
                        doc.cpu_render.type = .Squares;
                    } else {
                        doc.cpu_render.type = .Gradient;
                    }
                    canvas.requestRedraw();
                }
            }
            if (dvui.button(@src(), "Add random shapes", .{}, .{})) {
                canvas.addRandomShapes() catch {};
            }
            if (dvui.button(@src(), "Request redraw", .{}, .{})) {
                canvas.requestRedraw();
            }
        } else {
            dvui.label(@src(), "No document", .{}, .{});
        }
    }
    settings_section.deinit();

    // Верхняя часть: дерево объектов
    var tree_section = dvui.box(
        @src(),
        .{ .dir = .vertical },
        .{
```

```
                .expand = .both,
                .padding = dvui.Rect.all(panel_padding),
                .corner_radius = dvui.Rect.all(panel_radius),
                .color_fill = fill_color,
                .background = true,
            },
        );
        {
            dvui.label(@src(), "Objects", .{}, .{
                .font = .{
                    .line_height_factor = dvui.themeGet().font_heading.line_height_factor,
                    .size = dvui.themeGet().font_heading.size + 8,
                },
                .gravity_x = 0.5,
            });
            var scroll = dvui.scrollArea(
                @src(),
                .{ .vertical = .auto, .horizontal = .auto },
                .{ .expand = .both, .background = false },
            );
            {
                objectTree(ctx);
            }
            scroll.deinit();
        }
        tree_section.deinit();
    }
    panel.deinit();
}
```

A.36. src/ui/menu_bar.zig

```
const std = @import("std");
const dvui = @import("dvui");
const WindowContext = @import("../WindowContext.zig");
const Document = @import("../models/Document.zig");
const json_io = @import("../persistence/json_io.zig");

pub fn menuBar(ctx: *WindowContext) void {
    var m = dvui.menu(@src(), .horizontal, .{ .background = true, .expand = .horizontal });
    defer m.deinit();

    if (dvui.menuItemLabel(@src(), "File", .{ .submenu = true }, .{ .expand = .none })) |r| {
        var fm = dvui.floatingMenu(@src(), .{ .from = r }, .{});
        defer fm.deinit();

        if (dvui.menuItemLabel(@src(), "Open", .{}, .{ .expand = .horizontal }) != null) {
            m.close();
            openDocumentDialog(ctx);
        }

        if (dvui.menuItemLabel(@src(), "Save As", .{}, .{ .expand = .horizontal }) != null) {
            m.close();
            saveAsDialog(ctx);
        }
    }
}

fn openDocumentDialog(ctx: *WindowContext) void {
    const path_z = dvui.dialogNativeFileOpen(ctx.allocator, .{
        .title = "Open",
        .filters = &.{ "*.json" },
        .filter_description = "JSON files",
    }) catch |err| {
        std.debug.print("Open dialog error: {}\n", .{err});
        return;
    } orelse return;
```

```
    defer ctx.allocator.free(path_z);

    const path = path_z[0..path_z.len];
    const doc = json_io.loadFromFile(Document, ctx.allocator, path) catch |err| {
        std.debug.print("Open file error: {}\n", .{err});
        return;
    };
    ctx.addDocument(doc) catch |err| {
        std.debug.print("Add document error: {}\n", .{err});
        return;
    };
}

fn saveAsDialog(ctx: *WindowContext) void {
    const open_doc = ctx.activeDocument() orelse return;
    const path_z = dvui.dialogNativeFileSave(ctx.allocator, .{
        .title = "Save As",
        .filters = &.{ "*.json" },
        .filter_description = "JSON files",
    }) catch |err| {
        std.debug.print("Save dialog error: {}\n", .{err});
        return;
    } orelse return;
    defer ctx.allocator.free(path_z);

    const path_raw = path_z[0..path_z.len];
    json_io.saveToFile(Document, &open_doc.document, path_raw) catch |err| {
        std.debug.print("Save file error: {}\n", .{err});
        return;
    };
}
```

A.37. src/ui/right_panel.zig

```
const std = @import("std");
const dvui = @import("dvui");
const WindowContext = @import("../WindowContext.zig");
const canvas_view = @import("canvas_view.zig");

pub fn rightPanel(ctx: *WindowContext) void {
    const fill_color = dvui.Color.black.opacity(0.25);
    var back = dvui.box(
        @src(),
        .{ .dir = .horizontal },
        .{ .expand = .both, .padding = dvui.Rect.all(12), .background = true },
    );
    {
        var panel = dvui.box(
            @src(),
            .{ .dir = .vertical },
            .{
                .expand = .both,
                .background = true,
                .padding = dvui.Rect.all(5),
                .corner_radius = dvui.Rect.all(24),
                .color_fill = fill_color,
            },
        );
        {
            const active_doc = ctx.activeDocument();
            if (active_doc) |doc| {
                const content_rect_scale = panel.data().contentRectScale();
                canvas_view.canvasView(&doc.canvas, doc.selected_object_id, content_rect_scale);
            } else {
                noDocView(ctx);
            }
        }
```

```
        panel.deinit();
    }
    back.deinit();
}

fn noDocView(ctx: *WindowContext) void {
    var center = dvui.box(
        @src(),
        .{ .dir = .vertical },
        .{
            .expand = .both,
            .padding = dvui.Rect.all(20),
        },
    );
    {
        var box = dvui.box(
            @src(),
            .{ .dir = .vertical },
            .{
                .gravity_x = 0.5,
                .gravity_y = 0.5,
            },
        );
        {
            dvui.label(@src(), "No document open", .{}, .{});
            if (dvui.button(@src(), "New document", .{}, .{})) {
                ctx.addNewDocument() catch |err| {
                    std.debug.print("addNewDocument error: {}\n", .{err});
                };
            }
        }
        box.deinit();
    }
    center.deinit();
}
```

A.38. src/ui/tab_bar.zig

```
const std = @import("std");
const dvui = @import("dvui");
const icons = @import("../icons.zig");
const WindowContext = @import("../WindowContext.zig");

const DocCallback = union(enum) {
    select: usize,
    close: usize,
};

fn documentTab(ctx: *WindowContext, index: usize, callback: *?DocCallback) void {
    const row_id: usize = index;
    const is_selected = if (ctx.active_document_index) |active| active == index else false;
    const focus_color = dvui.themeGet().focus;

    var row = dvui.box(
        @src(),
        .{ .dir = .horizontal },
        .{
            .id_extra = row_id,
            .expand = .none,
            .gravity_y = 0.5,
        },
    );
    {
        var hovered: bool = false;
        var select_row: bool = false;
        const row_data = row.data();
```

```
        // Ручная обработка hover/click по строке без пометки события как handled,
        // чтобы кнопка закрытия могла нормально получать свои события.
        for (dvui.events()) |*e| {
            switch (e.evt) {
                .mouse => |*mouse| {
                    if (!dvui.eventMatchSimple(e, row_data)) continue;
                    hovered = true;
                    if (mouse.action == .release and mouse.button == .left) {
                        select_row = true;
                    }
                },
                else => {},
            }
        }

        var overlay = dvui.overlay(@src(), .{
            .margin = dvui.Rect{ .x = 4, .w = 4 },
            .padding = dvui.Rect{ .x = 12, .y = 0, .w = 0, .h = 0 },
            .background = is_selected or hovered,
            .color_fill = if (is_selected) focus_color.opacity(0.35) else if (hovered)
             ↪  focus_color.opacity(0.18) else null,
            .corner_radius = dvui.Rect.all(4),
        });
        {
            var buf: [32]u8 = undefined;
            const label = std.fmt.bufPrint(&buf, "Doc {d}", .{index + 1}) catch "Doc";
            dvui.labelNoFmt(@src(), label, .{}, .{
                .gravity_x = 0.5,
                .gravity_y = 0.5,
                .margin = .{ .w = 24 },
            });

            if (hovered) {
                if (dvui.buttonIcon(@src(), "Close", icons.cross, .{}, .{}, .{
                    .id_extra = row_id +% 1,
                    .margin = dvui.Rect{ .x = 8, .w = 6 },
                    .padding = dvui.Rect.all(2),
                    .gravity_x = 1,
                    .gravity_y = 0.5,
                })) {
                    callback.* = .{ .close = index };
                    select_row = false;
                }
            }
        }
        overlay.deinit();

        if (select_row) {
            callback.* = .{ .select = index };
        }
    }
    row.deinit();
}

pub fn tabBar(ctx: *WindowContext) void {
    var bar = dvui.box(
        @src(),
        .{ .dir = .horizontal },
        .{
            .expand = .horizontal,
            .background = true,
            .padding = .{
                .x = 12,
                .w = 12,
            },
        },
    );
```

```
    {
        var callback: ?DocCallback = null;
        for (ctx.documents.items, 0..) |_, i| {
            documentTab(ctx, i, &callback);
        }
        if (callback) |action| {
            switch (action) {
                .select => |index| ctx.setActiveDocument(index),
                .close => |index| ctx.closeDocument(index),
            }
        }

        if (dvui.buttonIcon(@src(), "Create", icons.plus, .{}, .{}, .{
            .gravity_y = 0,
        })) {
            ctx.addNewDocument() catch |err| {
                std.debug.print("addNewDocument error: {}\n", .{err});
            };
        }
    }
    bar.deinit();
}
```

A.39. src/ui/types/TexturedBox.zig

```
const std = @import("std");
const dvui = @import("dvui");
const TexturedBox = @This();

parent: dvui.Widget,
rs: dvui.RectScale,
pic: ?dvui.Picture,
corner_radius: dvui.Rect,

pub fn init(rs: dvui.RectScale, corner_radius: dvui.Rect) TexturedBox {
    const parent = dvui.parentGet();
    const pic = dvui.Picture.start(rs.r);
    return .{
        .parent = parent,
        .corner_radius = corner_radius,
        .rs = rs,
        .pic = pic,
    };
}

pub fn deinit(self: *TexturedBox) void {
    if (self.pic) |*picture| {
        picture.stop();

        const tex = dvui.textureFromTarget(picture.texture) catch null;
        if (tex) |t| {
            dvui.Texture.destroyLater(t);
            dvui.renderTexture(t, self.rs, .{
                .corner_radius = self.corner_radius,
            }) catch {};
        }
    }
}
```