ООО «Летари»

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

LeCodes

ОПИСАНИЕ ФУНКЦИОНАЛЬНЫХ ХАРАКТЕРИСТИК ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Версия 1.3.0

Пермь, 2026

Оглавление

1 Назначение и область применения

Настоящий документ содержит описание функциональных характеристик программного обеспечения «LeCodes» (далее — ПО).

LeCodes — среда разработки кроссплатформенных приложений на языке TypeScript. Разработчик описывает интерфейс и логику приложения средствами комплекта разработки (LeCodes SDK: декларативный пользовательский интерфейс, 2D- и 3D-сцены, анимации, физика), компилятор собирает проект в единый переносимый JavaScript-бандл, а среда выполнения (Runtime) исполняет этот бандл нативно на каждой поддерживаемой платформе — без встроенного браузера (WebView) и без установки среды Node.js на устройстве пользователя.

ПО поставляется в составе двух программных компонентов:

- **Компилятор `lecodes-cli`** — консольная утилита разработчика (автономный исполняемый файл), выполняющая сборку проекта в бандл приложения;
- **Среда выполнения LeCodes Runtime** — нативная библиотека, исполняющая скомпилированный бандл на целевой платформе.

Область применения: разработка кроссплатформенных клиентских приложений — игр, интерактивных презентаций, приложений дополненной реальности (AR), обучающих и визуализационных продуктов.

2 Состав и архитектура программного обеспечения

Конвейер обработки: исходный код проекта на языке TypeScript компилируется утилитой `lecodes-cli` в единый бандл приложения (`app.js`), который затем исполняется средой выполнения LeCodes Runtime на целевой платформе.

СХЕМА

```
Исходники проекта (TypeScript)
        |
        v
Компилятор lecodes-cli   -->   app.js  (единый бандл приложения)
        |
        v
Среда выполнения LeCodes Runtime
(Windows / Linux / Astra Linux / macOS / iOS / Android / Web)
```

Один и тот же скомпилированный бандл исполняется на всех платформах: платформенно-зависимая часть полностью инкапсулирована в среде выполнения. Бандл не содержит нативного кода и не изменяется при переносе между платформами.

Архитектура среды выполнения. Среда выполнения LeCodes Runtime — нативная (C++) среда исполнения приложений, состоящая из платформенно-независимого ядра и тонких платформенных «хостов», предоставляющих ядру окно, графический контекст, ввод, звук и сеть. Основные подсистемы:

- **Ядро исполнения** — связывает код приложения с нативными подсистемами; включает встраиваемый интерпретатор ECMAScript **QuickJS** (каждое приложение исполняется в изолированном контексте, доступ к системе — только через явно предоставленные API), подсистему времени и анимаций, загрузчик 3D-ресурсов (glTF/GLB), навигацию и экраны;
- **Подсистема пользовательского интерфейса** — платформенно-независимая раскладка по модели flexbox, стили (цвета, границы, скругления, градиенты, тени, трансформации), анимируемые переходы свойств, виртуализированные списки;
- **Графический 3D-движок** — на основе движка рендеринга **Google Filament** (физически корректный рендеринг, PBR): сцены, камеры, освещение, материалы, скелетная анимация и морфинг, текстуры (PNG/JPEG/KTX2), окружение (IBL); графические интерфейсы OpenGL / OpenGL ES / WebGL / Metal; опционально — физический движок **Jolt Physics**;
- **Графический 2D-движок** — на основе графической библиотеки **sokol** (спрайты, атласы, слои) с 2D-физикой **Box2D**;
- **Платформенные хосты** — Windows, Linux (в том числе Astra Linux), Android, iOS/macOS, Web (WebAssembly): окно, графический контекст, ввод, звук, работа с файлами и сетью.

Поддерживаемые платформы.

Компонент	Платформы
Среда выполнения LeCodes Runtime (настольные ОС)	Windows 10/11 x64; Astra Linux x64; Ubuntu и совместимые дистрибутивы Linux (glibc ≥ 2.24); macOS
Среда выполнения LeCodes Runtime (мобильные ОС)	Android; iOS
Среда выполнения LeCodes Runtime (Web)	Современные браузеры (WebAssembly + WebGL)
Компилятор `lecodes-cli`	Windows x64; Linux x64 (статическая сборка, единый бинарный файл); macOS; любая ОС со средой Node.js ≥ 18 (пакет npm)

Сторонние компоненты. Сторонние компоненты встроены в поставку в виде исходного кода или статических библиотек (внешних загрузок во время работы не происходит) и распространяются под свободными разрешительными лицензиями:

Компонент	Назначение	Лицензия
QuickJS	Движок JavaScript	MIT
Google Filament	Движок 3D-рендеринга	Apache-2.0
Jolt Physics	3D-физика	MIT
tgfx	2D-растеризация интерфейса (настольные ОС)	BSD-3-Clause
sokol	Графическая основа 2D-движка	zlib
Box2D	2D-физика	MIT
miniaudio	Воспроизведение звука	MIT-0 / Public Domain
nanosvg	Разбор SVG	zlib
stb_image	Декодирование изображений	MIT / Public Domain

3 Функциональные характеристики

Функциональные характеристики ПО раскрыты по двум составляющим его компонентам: компилятору `lecodes-cli` (п. 3.1) и среде выполнения LeCodes Runtime (п. 3.2). Полное описание программного интерфейса среды выполнения приведено в разделе «4. Описание программного интерфейса среды выполнения LeCodes Runtime».

3.1 Компилятор lecodes-cli

Компилятор `lecodes-cli` — консольная утилита разработчика для сборки проектов LeCodes. Распространяется как автономный исполняемый файл (сборки windows-x64 и linux-x64 — статическая сборка, единый бинарный файл для всех дистрибутивов Linux), не требует установленной среды Node.js, а также публикуется как пакет npm `lecodes-cli`.

Компиляция. Сборщик **chisel** (собственная разработка, язык Rust) пакетирует проект с гранулярностью до отдельных методов — в итоговый бандл попадает только реально используемый код SDK и проекта (агрессивное удаление неиспользуемого кода). Компилятор SDK транслирует TypeScript-исходники проекта, разрешает модули без внешних импортов и встраивает ресурсы (например, SVG-иконки разворачиваются во встроенные источники на этапе сборки). Результат — файл `app.js`, готовый к исполнению любой средой выполнения LeCodes Runtime.

Основные команды утилиты:

Группа	Команды	Назначение
Создание проекта	`init`	Создание локального проекта с типовой структурой (точка входа, конфигурация, описания типов программного интерфейса)
Сборка и рендеринг	`preview`, `render`	Локальная компиляция проекта в бандл (`app.js`); рендеринг UI-проекта в семантическое JSON-описание или изображение (PNG)
Режим разработки	`dev`	Локальный сервер в сети (LAN): QR-код для подключения устройства, горячая перезагрузка приложения при изменении файлов (по протоколу WebSocket, без перезапуска)
Прототипирование	`design`	LeCodes Design — интерактивная доска экранов для проектирования интерфейсов

3.2 Среда выполнения LeCodes Runtime

Среда выполнения предоставляет прикладному коду единую программную среду со следующими функциональными подсистемами (полное описание программного интерфейса каждой подсистемы приведено в разделе «4. Описание программного интерфейса среды выполнения LeCodes Runtime»):

- **Начало** — Обзор, Соглашения, Глобалы хоста.

- **UI** — Модель элементов, Стилизация, Тема — theme(), Классы стилей, Контейнеры, Текст, изображения, видео, Кнопки и поля ввода, UIScreen · Router, UIPager · UITabs, UIWidget, UIVirtualizedList, NativeView, Шрифты.
- **3D-движок** — Scene, ARScene, Node, Camera, Mesh, Model, Geometry, Material, Texture, Light, Физика 3D, Particles, Ray · Plane · Noise, Файлы сцен — defineScene, Сценарные аспекты, Плагины редактора.
- **2D-движок** — Scene2D, Node2D, Camera2D, Sprite, SpriteAnimation, Физика 2D, Tilemap, Texture2D.
- **Ядро** — signal · computed · effect, Aspect, Canvas, События.
- **Рантайм** — device, app · clipboard · openURL, Input, События касаний, fetch · WebSocket, localStorage, Аудио и видео, Файлы и шеринг, toast · SvgSource, date.
- **Анимация** — animate, Функции сглаживания.
- **Математика** — Mathf, Vec2 и Vec3, Quat, Mat4, Color.
- **Плагины** — CameraView, QRScanner, Geolocation, Push, Service.

Ключевые особенности программного интерфейса:

- единая глобальная программная среда без явного импорта пакетов — реализация внедряется на этапе сборки, неиспользуемый код исключается из продукта;
- согласованные системы координат, единиц измерения и семантики значений для 2D-, 3D- и UI-подсистем (см. раздел «Соглашения»);
- двумерный движок с физикой на базе Box2D, спрайтами, покадровой анимацией и тайловыми картами;
- трёхмерный движок со сценами, мешами и моделями, материалами, освещением, физикой на базе Jolt, системами частиц и поддержкой дополненной реальности (AR);
- декларативная система пользовательского интерфейса: контейнеры, текст и медиа, элементы ввода, экраны и маршрутизация, виджеты, виртуализированные списки, стилизация и шрифты;
- подсистема времени выполнения: доступ к устройству, ввод и жесты касания, работа с сетью (fetch / WebSocket), локальное хранилище, аудио и видео, файлы и системный обмен;
- подсистема анимации со сглаживанием (easing) и подсистема реактивности (signal / computed / effect).

4 Описание программного интерфейса среды выполнения LeCodes Runtime

Ниже приведено полное описание программного интерфейса среды выполнения LeCodes Runtime, сгруппированное по подсистемам. Для каждой подсистемы указаны предоставляемые типы и функции, их сигнатуры, единицы измерения и примеры использования. Материал предназначен для разработчиков прикладного кода.

4.1 Начало

4.1.1 Обзор

Полный справочник API для проектов LeCodes. Всё здесь — **глобальные значения без импортов**: пользовательский код обращается к `UIScreen`, `Scene`, `Sprite`, `fetch`, … напрямую, а сборка подставляет реализацию и вырезает то, что проект не использует.

Начни со страницы Соглашения — единицы измерения, системы координат, семантика значений и правила жизненного цикла, на которые опирается каждая страница. Большинство приложений — UI-приложения: точка входа в UI-слой — Модель элементов (модель элементов плюс устройство типичного приложения: модуль токенов с `theme()`, экраны, каркас вкладок).

> **ЗАМЕТКА**
>
> Это справочник по поверхности API: сигнатуры, единицы, крайние случаи. Учиться удобнее по Гайду — он ведёт через сборку приложения и по дороге ссылается сюда.

Вся поверхность одним взглядом

Одна строка на глобал. Жирная ссылка — страница, которая его документирует.

UI

- модель элементов: фабрики, цепочки, ссылки, условные дети, устройство приложения — **Модель элементов**
- модель `.style()`: свойства, единицы, безопасные зоны, адаптивность, animateTo — **Стилизация**
- `theme` — стилевые переменные на всё приложение (`var()`), системные значения, живая смена темы — **Тема**
- классы стилей: блоки состояний `$name`, `el.class`, каскад, `$pressed`/`$focused` — **Классы стилей**
- `UIRow`, `UIColumn`, `UIBox` (устарел), `UIScrollable`, `UISpacer` — **Контейнеры**
- `UIText`, `UIImage`, `UIVideo`, `assetIcon` — **Текст, изображения, видео**
- `UIButton`, `UIInput`, `UITextArea` — **Кнопки и поля ввода**
- `UIScreen`, `Router` — экраны и навигация — **UIScreen · Router**

- `UIPager`, `UITabs` — свайп-вкладки + свой стек навигации в каждой; `UITabs` — стандартный каркас нижних вкладок — **UIPager · UITabs**
- `UIWidget` — плавающие оверлеи (+ диалоги `UIModal`, привязанные меню `UIPopover`, `UIBottomSheet`) — **UIWidget**
- `UIVirtualizedList` — оконные списки для длинных данных — **UIVirtualizedList**
- `NativeView` — платформенные вью, зарегистрированные хостом (карта, камера, …) — **NativeView**
- `font`, `registerFont` — шрифты — **Шрифты**

3D-движок

- `Scene` — 3D-мир: IBL, bloom, скайбокс, оверлеи — **Scene**
- `ARScene` — AR-сессия: якоря, элементы размещения — **ARScene**
- `Node` — базовый узел: трансформ, иерархия, события — **Node**
- `Camera` — fov, лучи, направление взгляда (через `scene.camera`) — **Camera**
- `Mesh` — фабрики box/sphere/cylinder/plane + своя геометрия — **Mesh**
- `Model`, `ModelAnimation` — загрузка GLB; проигрывание клипов (`model.anim`) — **Model**
- `Geometry` — сырые вершинные данные — **Geometry**
- `Material` — lit/unlit/кастомные шейдерные материалы — **Material**
- `Texture` — картинка 3D-материала (не взаимозаменяема с `Texture2D`) — **Texture**
- `Light` — направленный свет `Light.sun()` + тени (ambient даёт IBL сцены) — **Light**
- `Shape`, `Physics`, `Trigger`, `CharacterController` — тела Jolt, сенсоры, рейкасты — **Физика 3D**
- `Particles`, `dynamic`, `dynamicColor` — GPU-системы частиц — **Particles**
- `Ray`, `Plane`, `Noise` — пикинг и процедурные хелперы — **Ray · Plane · Noise**
- `defineScene`, `use` — сцены как данные (файлы `.scene.ts`, формат редактора сцен) — **Файлы сцен**
- `MoveTo`, `FollowPath`, `Spin`, `LookAt`, `PlayAnimation` — готовые no-code поведения — **Сценарные аспекты**
- `registerEditorWindow`, `registerEditorTool` — плагины редактора сцен (`*.editor.ts`) — **Плагины редактора**

2D-движок

- `Scene2D` — 2D-мир: фон, слои, Y-сортировка, пикинг — **Scene2D**
- `Node2D` — базовый узел: трансформ, иерархия, события — **Node2D**
- `Camera2D` — позиция, зум, экран↔мир (через `scene.camera`) — **Camera2D**
- `Sprite` — узел с текстурой; кадры, tint, flip — **Sprite**
- `SpriteAnimation` — аспект клипов по спрайт-листу (`node.anim`) — **SpriteAnimation**
- `Shape2D`, `Physics2D`, `Trigger2D`, `CharacterController2D` — тела Box2D, сенсоры, запросы — **Физика 2D**

- `Tilemap` — тайловая сетка по атласу, нативный каллинг — **Tilemap**
- `Texture2D` — загрузка 2D-изображений — **Texture2D**

Ядро

- `signal`, `computed`, `effect` — реактивное состояние; биндинги `() => value` в тексте и стилях UI — **signal · computed · effect**
- `Aspect` — подключаемые способности узлов; `node.aspect(Class, opts)`, фазы обновления — **Aspect**
- `Canvas`, `Bitmap` — retained-2D-рисование, запекается в текстуры (2D/3D/UI) — **Canvas**
- паттерн эмиттера событий (`addEventListener`) — общий для сцен/узлов/плееров — **События**

Рантайм

- `device` — платформа, язык, размер, pixelRatio, событие resize, связность, точный touch — **device**
- `app`, `clipboard`, `openURL` — жизненный цикл (pause/resume), диплинки, системный буфер, внешние URL — **app**
- `Input` — опрос клавиатуры (`Input.key(code)`) — **Input**
- `ClickEvent`, `TouchStartEvent` — события указателя + драги `ev.track()` — **События касаний**
- `fetch`, `fetchLocal`, `FormData`, `File` — HTTP через хост; синхронное чтение тела — **fetch · WebSocket**
- `WebSocket` — сокеты как в браузере + свои заголовки — **fetch · WebSocket**
- `localStorage` — персистентные строки ключ/значение — **localStorage**
- `AudioPlayer`, `VideoPlayer` — воспроизведение медиа; видео-как-текстура — **Аудио и видео**
- `openFilePicker`, `share` — выбор файлов, системный шеринг — **Файлы и шеринг**
- `toast`, `SvgSource` — нативный тост; SVG как источник изображения — **toast · SvgSource**
- `date` — даты в стиле dayjs: формат, относительное время, арифметика, сравнение (английский + русский) — **date**

Анимация

- `animate`, `animateMat4`, `stopAnimation`, `pauseAnimation`, `resumeAnimation` — твины значений (длительность в мс) — **animate**
- `easeIn`, `easeOut`, `easeInOut`, `cubicBezier` — функции сглаживания — **Функции сглаживания**

Математика

- `Mathf` — скалярные хелперы: clamp, lerp, damp, remap, random — **Mathf**
- `Vec2`, `Vec3` — флюентные векторы; мутабельные поля, чистые методы, кортежи — **Vec2 и Vec3**

- `Quat` — вращения: fromEuler, fromAxisAngle, slerp, lookRotation — **Quat**
- `Mat4` — матрица 4×4: compose/decompose, lookAt, perspective; открытый `.m` — **Mat4**
- `Color` — парсинг и конвертация любых форм `ColorInput` — **Color**

Плагины

- `CameraView` — превью камеры устройства + фото — **CameraView**
- `QRScanner` — сканирование QR (зависит от хоста) — **QRScanner**
- `Geolocation` — позиция устройства (разово + наблюдение) — **Geolocation**
- `Push` — удалённые push-уведомления (регистрация + события полезной нагрузки; отправку релеит le.codes) — **Push**
- `Service` — сырой канал к headless-сервисам хоста (локальным и сторонним) — **Service**

Глобалы хоста — Глобалы хоста

- `setLoop` / `clearLoop` — покадровый цикл; `dt` в секундах
- `setTimeout` / `setInterval` / `clearTimeout` / `clearInterval` — таймеры (мс)
- `console` — log / warn / info / error
- `asset()` — compile-time-макрос: путь к файлу проекта → URL ресурса в бандле
- `DEG2RAD` / `RAD2DEG` — compile-time-константы углов
- `Style<T>` — вспомогательный тип для переиспользуемых объектов стилей (только для редактора)

Как читать сигнатуры

Сигнатуры записаны как упрощённый TypeScript — без шума перегрузок и вспомогательных дженериков. Значения по умолчанию показаны комментарием (`// по умолчанию 1`), необязательные параметры помечены `?`, а единицы измерения всегда названы там, где число пересекает границу API: секунды или миллисекунды, градусы или радианы, логические пиксели или мировые единицы.

```ts
Mathf.lerp(a, b, t): number            // a + (b − a)·t
sprite.setFramePx(x, y, w, h): this    // под-прямоугольник в пикселях текстуры
animate(target, to, 300)               // длительность в миллисекундах
```

> **СОВЕТ**
>
> Слева — полное дерево навигации по всем разделам. Справочник переведён целиком — все страницы доступны на русском и английском.

4.1.2 Соглашения

Сквозные правила, действующие везде в SDK. Каждая справочная страница их предполагает.

Без импортов — всё глобально

Вся поверхность SDK (`Scene`, `Sprite`, `UIScreen`, `Vec3`, `fetch`, …) внедряется на этапе сборки; пользовательский код её никогда не импортирует. Единственные `import` в проекте — это **относительные пути** к его собственным файлам (`import { hud } from './ui/hud'`) и к его ассетам (`import hero from './hero.png'`). Голый импорт пакета — ошибка сборки.

Точка входа: `main.ts`. В многофайловом проекте он связывает остальные вместе и запускает приложение (`Router.init(home)` или `scene.open()`); каждый другой файл должен быть достижим из него через импорты (модулю с сайд-эффектами всё равно нужен `import './register'`). Без `main.ts` компилятор берёт файл, который никто другой не импортирует (корень графа импортов); если таких несколько, побеждает файл без экспортов, затем самый большой — задай явно через `main.ts` или `--entry`.

Ассеты: ссылайся на файл проекта через import или встроенный макрос `asset('./path')` — это переписывание на этапе компиляции, поэтому путь должен быть строковым литералом. Удалённые `https://…` URL — обычные строки.

Время

- `setLoop(dt => …)` срабатывает каждый кадр; `dt` — это **секунды** с прошлого кадра (~0.016 при 60 fps). Движение, независимое от кадра, — `position += speed * dt`.
- `setTimeout` / `setInterval` принимают **миллисекунды**.
- Длительности анимаций — **миллисекунды**: `animate({ duration })` и UI `.animateTo()` / `.animateFrom()` (`duration`, `delay`).

Пространство, единицы, углы

	2D (`Scene2D`)	**3D (`Scene`)**	**UI / экран**
Оси	X вправо, **Y вверх**	правая система, **Y вверх**, вперёд = **−Z**	X вправо, **Y вниз**
Единица	1 мировая единица = 1 логический px при zoom 1	≈ метры	логические px (dp/pt)
Углы	`rotation` в градусах, CCW	`eulerAngles` в градусах	—

- *Аргументы*-«сырые углы» — **радианы** (`Quat.fromAxisAngle`, `Mat4.rotate*`, `Vec2.rotate`); эйлеровы *свойства и фабрики* — **градусы** (`node.rotation`, `Quat.fromEuler`). `DEG2RAD` / `RAD2DEG` — константы времени компиляции для конвертации.
- Экранные координаты (`clientX/clientY`, `device.width/height`) — **логические px** (как iOS pt / Android dp), никогда не физические пиксели; `device.pixelRatio` — отношение физических к логическим.
- Одна ловушка знака: в 2D-мире Y смотрит вверх, но экранный Y и пространство якоря спрайта смотрят вниз (`anchor: [0.5, 1]` = низ-центр = «ноги»).

Цвета

Везде, где принимается цвет, работает всё это (`ColorInput`): hex-строки `"#e33"`, `"#e33c"`, `"#ff3333"`, `"#ff3333cc"`; упакованный int `0xff3333` (только 24-битный RGB — альфу нельзя передать числом); нормализованные массивы дробных `[r, g, b]` / `[r, g, b, a]` (0…1).

> **NOTE**
>
> hex — единственная *строковая* форма. CSS-стиль `rgba(...)` / именованные цвета не парсятся — они молча становятся непрозрачным чёрным. (UI-стили — исключение: они идут в UI-движок со своим парсером — см. Стилизацию UI.)

Семантика значений в математике

`Vec2` / `Vec3` / `Quat` / `Mat4` — изменяемые структуры с **чистыми методами**: поля можно присваивать (`v.x = 3`), но каждый метод возвращает НОВОЕ значение и никогда не меняет источник. Они итерируемы и взаимозаменяемы с сырыми кортежами — `[0, 1, 0]` работает везде, где принимается `Vec3`.

Геттеры трансформа узла возвращают **свежие копии** (семантика значений, как в Unity):

```ts
node.position.x = 3                                    // ✗ тихий no-op — меняет отброшенную копию
node.x = 3                                             // ✓ сеттер одной оси
const p = node.position; p.y += 1; node.position = p   // ✓ поменять локальную, присвоить обратно
```

События

- Сцены, узлы, `device`, `WebSocket`, медиаплееры: `addEventListener(channel, cb)` / `removeEventListener(channel, cb)`.
- UI-элементы вместо этого используют чейнящиеся методы `on*` (`.onClick(cb)`, `.onChange(cb)`).
- События указателя: `click` срабатывает на отпускании над целью; `touchstart` срабатывает на нажатии, и его объект события может `ev.track({...})`, чтобы захватить остаток жеста. См. События указателя и жесты.

Цепочки

Конструкторы принимают объект опций; настраивающие сеттеры возвращают `this`, поэтому сборка читается одной цепочкой. UI-фабрики чейнят `.style()` и `on*`. Возможности узла прикрепляются как **аспекты** и тоже чейнятся:

```ts
const hero = new Sprite({ texture })
  .aspect(SpriteAnimation, { size: [32, 48], fps: 10, clips: { walk: [1, 2, 3] } })
  .aspect(Shape2D, { box: [16, 8] })
  .aspect(Physics2D, { motion: 'dynamic', fixedRotation: true })
hero.anim.play('walk')          // каждый аспект добавляет именованный аксессор
```

См. Аспекты.

Физика владеет динамическими трансформами

Как только у узла есть `dynamic`-тело (`Physics` / `Physics2D`), шаг физики пишет его трансформ. **Не** задавай `node.position` каждый кадр — управляй телом через `velocity` / `applyImpulse` (кинематические тела: `moveTo`). *Чтение* `node.position` / `node.worldPosition` всегда корректно; SDK автоматически пересинхронизируется с нативным слоем.

Гейтинг хоста — возможности, которых может не быть

Некоторые возможности зависят от сборки хоста. При отсутствии они **молча делают no-op**; проверяй через:

- `Physics.supported` (3D-физика), `Physics2D.supported` (2D-физика)
- `QRScanner.isSupported`
- `device.setPreciseTouch()` — no-op там, где у платформы нет понятия объединённого ввода

Жизненный цикл

- Всё запущенное в `onOpen` экрана должно останавливаться в `onClose` (`clearLoop`, `clearInterval`, `ws.close()`), иначе продолжит работать при навигации.
- Объекты, оборачивающие нативные ресурсы, освобождают их явно там, где есть метод: `node.destroy()` (2D/3D-узлы), `player.dispose()` (медиа — бросает при любом использовании после), `Texture2D.destroy()`, `response.dispose()` (тела fetch). У 3D `Texture` освобождения нет — она живёт до закрытия движка.

4.1.3 Глобалы хоста

Глобалы, предоставляемые самим рантаймом хоста (веб-вьювер, iOS, Android, десктоп), а не внедряемые SDK. Доступны в каждом проекте, в каждом движке.

Кадровый цикл

```ts
setLoop(fn: (dt: number) => void): number
clearLoop(id: number): void
```

Срабатывает каждый отрисованный кадр. `dt` — **секунды** с прошлого кадра (~0.016 при 60 fps) — для движения, независимого от кадра, масштабируй на него: `position += speed * dt`.

```ts
const id = setLoop(dt => {
  if (Input.key('ArrowRight')) player.x += 120 * dt
})
```

> **NOTE**
> цикл работает, пока его не остановят. Всё запущенное в `onOpen` экрана должно останавливаться в его `onClose`, иначе утечёт при навигации.

Таймеры

```ts
setTimeout(fn, ms?): number
setInterval(fn, ms?): number
clearTimeout(id): void
clearInterval(id): void
```

Та же семантика, что в вебе, **миллисекунды**. (Для контраста: `dt` у `setLoop` — секунды.)

Консоль

```ts
console.log(...data) / console.warn(...) / console.info(...) / console.error(...)
```

Единственные существующие методы консоли — нет `console.table`, `console.time` и т. п.

`asset()` — ресурсы проекта

```ts
asset(path: string): string      // макрос времени компиляции — path должен быть строковым ЛИТЕРАЛОМ
```

Переписывается бандлером в импорт ресурса для этого файла; возвращаемая строка — URL собранного ассета. Эквивалентно импорту файла:

```ts
import hero from './hero.png'            // то же самое
const tex = await Texture2D.load(asset('./hero.png'))
```

> **NOTE**
> поскольку это макрос, `asset(someVariable)` — ошибка сборки. Удалённым `https://…` URL он не нужен — передавай их обычными строками.

Константы углов

```ts
DEG2RAD   // π / 180 — подставляется при компиляции
RAD2DEG   // 180 / π
```

Используй их на границе радианы↔градусы (см. Соглашения).

`Style<T>` (тип только для редактора)

```ts
const preset: Style<UIButton> = { bgColor: '#333', borderRadius: 12 }
```

Разрешается в объект стиля, который принимает `el.style(...)` (каждое свойство опционально) — для типизации переиспользуемых пресетов стиля, включая блоки `$`-классов. Чистый тип: в рантайме не существует.

4.2 UI

4.2.1 Модель элементов UI

Как работает UI-слой в целом: элементы — это обычные объекты, создаваемые **глобальными фабричными функциями**, компонуемые передачей **детей как аргументов**, настраиваемые **цепочкой** и обновляемые изменением свойств — без JSX, без virtual DOM, без цикла ре-рендера. Раскладка — флексбокс (Yoga) в логических px, экранное пространство Y вниз. Эта страница про модель; остальная часть раздела:

Система стилей

- Стилизация — `.style()`, словарь свойств, единицы, безопасные зоны, `animateTo`
- Тема — переменные `theme()` на всё приложение; значения `var()`; живая смена темы (тёмный режим — это второй вызов)
- Классы стилей — блоки состояний `$name`, переключаемые через `el.class`; каскадируют на потомков; `$pressed`/`$focused`
- Шрифты — макрос `font()`, `registerFont`

Элементы

- Контейнеры — `UIRow`, `UIColumn`, `UIScrollable`, `UISpacer`
- Элементы контента — `UIText`, `UIImage`, `UIVideo`, `assetIcon`
- Интерактивные элементы — `UIButton`, `UIInput`, `UITextArea`

- Виртуализированные списки — `UIVirtualizedList` для длинных лент и чатов

Навигация и оверлеи

- Экраны и Router — `UIScreen`, `Router`
- UIPager и UITabs — свайпаемые вкладки, стеки на вкладку, стандартная оболочка вкладок
- Виджеты — оверлеи `UIWidget`, `UIModal`, `UIPopover`, `UIBottomSheet`
- NativeView — платформенные вью, зарегистрированные хостом (карта, камера, …)

Форма приложения

Паттерн, который предполагает справочник и которому следуют реальные проекты: **модуль токенов** один раз вызывает `theme()` и экспортирует аксессоры; **экраны** — фабричные функции в своих файлах; `main.ts` собирает оболочку `UITabs` и отдаёт её роутеру.

TS

```
// theme.ts
theme({ color: "#131A17", primaryColor: "#15A34A" })         // текст по умолчанию + акцент
UI-кита
export const colors = theme({ bg: "#F4F6F5", card: "#FFFFFF", muted: "#606B65" })

// screens/home.ts
import { colors } from '../theme'
export const homeScreen = UIScreen(
  UIScrollable(/* контент */).style({ flexGrow: 1, px: "comfort-x" }),
).style({ bgColor: colors.bg, pt: "comfort-top" })

// main.ts
import { homeScreen } from './screens/home'
import { profileScreen } from './screens/profile'
Router.init(UITabs({
  home:    { label: "Home",    icon: assetIcon("lucide:house"), screen: homeScreen },
  profile: { label: "Profile", icon: assetIcon("lucide:user"),  screen: profileScreen },
}))
```

Приложение с одним экраном пропускает оболочку: `screen.open()` — и всё.

Обзор

```ts
let counter = 0
let label: UIText

const screen = UIScreen(
  label = UIText("Taps: 0").style({ color: "white", fontSize: 24, fontWeight: 700 }),
  counter > 0 ? UIText("already tapped") : null,        // null-дети пропускаются
  UIButton(UIText("Tap").style({ color: "white" }))
    .style({ bgColor: "#FF4032", borderRadius: 12, p: 16, alignSelf: "flex-start" })
    .onClick(() => { label.text = `Taps: ${++counter}` }),   // меняем контент напрямую
).style({ bgColor: "black", p: 20, pt: "max(safe-top, 24px)", gap: 16 })

screen.open()
```

Фабрики, не конструкторы

Каждый элемент создаётся вызовом глобальной функции — **никогда не** `new`:

```ts
UIColumn(...children)            // контейнеры: дети — обычные аргументы
UIText(text)                     // элементы контента: сам контент
UIImage(src)
```

Фабрика принимает только **контент** элемента; всё остальное настраивается цепочкой — `.style()`, `.onClick()`, `.append()`, `.animateTo()` — каждый возвращает сам элемент, поэтому конструирование читается одной цепочкой.

> **LEGACY**
> каждая фабрика также принимает объект стиля необязательным первым аргументом (`UIText({ fontSize: 20 }, "Hi")`). Он всё ещё работает и встречается в старых проектах, но новый код задаёт стили через `.style()`.

Элементы несут строку `readonly type` (`"column"`, `"text"`, …), определяющую, что они такое.

Дети и условный рендер

Контейнеры принимают детей обычными аргументами. Аргумент-**массив разворачивается на один уровень** — поэтому размапленный список вставляется напрямую, без spread:

```ts
UIColumn(
  header,
  items.map(Row),                // аргумент-массив — разворачивается в детей
  footer,
)
```

Ребёнок `null`, `undefined` или `false` **пропускается целиком** — ни элемента, ни слота раскладки — это и есть идиома условного рендера (и работает для целых блоков: falsy-аргумент тоже пропускается):

```ts
UIColumn(
  header,
  isLoading ? spinner : null,    // тернарник с null
  showFooter && footer,          // && с коротким замыканием
  showList && items.map(Row),    // условный блок
)
```

Контейнер, чей единственный аргумент — **функция**, трактует её как реактивную привязку детей — см. сигналы.

> **LEGACY**
>
> довариадическая форма — один массив детей, `UIColumn([a, b])` — по-прежнему работает везде (это просто правило разворачивания, применённое к одному аргументу). Новый код пиши с детьми-аргументами.

```ts
// ✗ пустой контейнер как заглушка (веб-привычка) — он всё равно занимает флекс-слот
UIRow(isGroup ? button : UIColumn())
// ✓ null пропускается — без фантомного элемента
UIRow(isGroup ? button : null)
```

Захват ссылок

Чтобы держать хэндл на вложенный элемент, используй **выражение присваивания прямо среди детей** — оно и задаёт переменную, и добавляет элемент:

```ts
let label: UIText
let input: UIInput

UIColumn(
  label = UIText("Hello"),
  input = UIInput(),
)

label.text = "Updated"       // позже
console.log(input.value)
```

`let` — это оператор, а не выражение — объявляй снаружи, присваивай внутри:

```ts
UIRow(let input = UIInput())   // X синтаксическая ошибка
```

Предпочитай захваченные ссылки индексации `container.children` — записи `children` не типизированы (`UINodeChild`), поэтому их обратное чтение требует приведения типа.

Свойства контента против стилей

Изменяемый **контент** живёт прямо на элементе, не в стиле. Его установка сразу перерисовывает, смонтирован он или нет:

```ts
text.text = "Updated"        // UIText
input.value = ""             // UIInput / UITextArea
image.src = newUrl           // UIImage
video.player                 // UIVideo — только чтение, ссылка на его VideoPlayer
```

Стили несут только внешний вид и раскладку. В частности, никогда не клади контент изображения в `bgImage` — это фон контейнера. См. content.md.

Императивное обновление детей

Контейнеры (`UIRow`, `UIColumn`, `UIScrollable`, …) предоставляют прямую манипуляцию детьми — диффинга нет; ты заявляешь изменение:

```ts
list.setContent(items.map(Row))      // заменить всех детей
list.append(row1, row2)              // добавить в конец
list.insert(0, banner)               // добавить по индексу (в .children)
list.remove(row1)                    // удалить по идентичности
list.children                        // текущий массив (только чтение)
```

Все четыре работают и до, и после появления элемента на экране. Для длинных или неограниченных данных используй `UIVirtualizedList` вместо `setContent` по большому массиву.

Чтение измеренного размера — `onLayout` и `getBoundingClientRect`

Размеры существуют только после раскладки. Чтобы *реагировать* на бокс элемента, используй `onLayout`:

```ts
el.onLayout(({ left, top, width, height }) => { ... })   // логические px; top/left относительно родителя
```

Он срабатывает, когда элемент впервые получает раскладку, и снова при каждом изменении его бокса (ресайз, смена контента). Можно зарегистрировать несколько колбэков; каждый вызов чейнится. Его координаты **относительны родителю** — и устаревают, когда предок прокручивается (прокрутка двигает элементы без пересчёта раскладки).

Чтобы прочитать **абсолютную позицию элемента по требованию** — как одноимённый веб-API — используй:

```ts
el.getBoundingClientRect()
// { x, y, left, top, right, bottom, width, height } — пространство устройства, логические px,
// включает смещения прокрутки; читается вживую в момент вызова. null до монтирования элемента
// (и на хостах, где чтение ещё не реализовано).
```

Прямоугольник — в том же пространстве, в котором позиционируется `UIWidget` и в котором события касания сообщают `clientX`/`clientY` — что делает его примитивом якорения для дропдаунов, поповеров и тултипов: прочитай rect триггера на тапе, поставь виджет в `rect.left`/`rect.bottom` (см. UIWidget и UIModal). Якори **позицией один раз при открытии**; не опрашивай каждый кадр.

Переиспользуемые компоненты

«Компонент» — это просто фабричная функция, возвращающая элемент — чейнься по результату как по любому элементу. Общие стили — обычные объекты, типизированные глобальным хелпером `Style<T>` (только тип — нечего импортировать или инстанцировать):

```ts
const heading: Style<UIText> = { color: "white", fontSize: 24, fontWeight: 700 }

const Card = (title: string, subtitle: string) => UIButton(
  UIText(title).style({ fontWeight: 700, color: "white" }),
  UIText(subtitle).style({ color: "#888", fontSize: 13 }),
).style({ px: 16, py: 12, gap: 4, flexDirection: "column", alignItems: "flex-start" })

Card("Title", "Subtitle").style({ bgColor: "#111" }).onClick(() => { ... })
```

В приложении с модулем токенов hex-литералы выше становятся аксессорами темы (`color: colors.muted`) — компоненты тогда автоматически следуют за сменой темы.

> **NOTE**
>
> каждый элемент принимает строку `name` в объекте стиля — семантическую метку, не стиль. Она извлекается при конструировании и выставляется на узле (`el.name`) как стабильный селектор для тестов и инструментов ревью.

Смотрите также

- Стилизация — модель `.style()`, которую предполагает эта страница.
- Тема — переменные на всё приложение и живая смена темы.
- Классы стилей — именованные состояния стилей, каскад.
- События указателя и жесты — объекты событий `onClick` / `onTouchStart`.
- Соглашения — логические px, цепочки, правила жизненного цикла.

4.2.2 Стилизация UI

Система стилей, общая для каждого UI-элемента: один метод `.style()`, один словарь свойств, раскладка только флексбокс (движок Yoga — без CSS-grid, без блочного потока), единицы в логических px. Эта страница — канонический список свойств стиля и форм значений; специфичные для элемента добавки (`objectFit`, `onPressed`, `scrollDirection`, …) живут на странице самого элемента. Два механизма поверх имеют собственные страницы: переменные темы (`theme()`, `var(--x)`) и классы стилей (блоки состояний `$name`).

Обзор

```ts
const card = UIColumn(
  UIText("Title").style({ color: "white", fontSize: 20, fontWeight: 700 }),
)
  .style({ bgColor: "#111", borderRadius: 16, p: 16, gap: 8 })   // слияние — чейнится
  .style({ onLandscape: { flexDirection: "row" } })              // адаптивное
переопределение

card.style.opacity = 0.5                        // мутация одного свойства (горячие пути)
card.animateTo({ bgColor: "#333", duration: 200 })   // твин к новым значениям (мс)
```

Три способа задать стили

```ts
el.style({ ... }): this        // СЛИВАЕТ в текущий стиль (не заменяет); чейнится
el.style.prop = value          // прямая запись одного свойства после создания (горячие пути)
el.style.prop                  // прочитать последнее заданное тобой значение
```

Слияние `.style()` означает, что поздние вызовы переопределяют только упомянутые ключи:

```ts
UIText("Hi").style({ color: "white" }).style({ fontSize: 20 })   // применяются оба
```

Используй прямую мутацию для покадровых обновлений, напр. `el.style.transform =` translateY(${y}px)``.

Любое свойство с примитивным значением также принимает функцию `() => value` — **реактивную привязку**, которая переприменяет себя, когда меняется прочитанный ею сигнал:

```ts
el.style({ bgColor: () => (selected.value ? "#FF4032" : "#333") })
```

Анимация: `animateTo` / `animateFrom`

```ts
el.animateTo({ opacity: 0, bgColor: "#000", duration: 300, delay?: 0, commit?: true,
               loop?: false, loopMode?: "ping-pong" }): this
el.animateFrom({ opacity: 0, duration: 300, delay?: 0 }): this   // от заданных значений → текущие
```

- `duration` / `delay` — **миллисекунды**.

- `animateTo` твинит от текущих значений к заданным и **сразу коммитит** цели в стиль элемента (чтобы чтения и поздние слияния видели финальное состояние). Передай `commit: false`, чтобы анимировать без записи стиля — напр. затухание оверлея прямо перед `.hide()`.
- `animateFrom` — входная анимация: она прыгает к заданным значениям и анимирует обратно к текущему стилю элемента. Она никогда не меняет сохранённый стиль.

Зацикливание

```ts
badge.animateTo({ transform: "scale(1.15)", duration: 600, loop: true })   // пульс навсегда
spin.animateTo({ transform: "rotate(360deg)", duration: 900, loop: true, loopMode: "restart" })
alert.animateTo({ opacity: 0.4, duration: 300, loop: 3 })                  // 3 цикла, затем конец
```

- `loop: true` повторяет бесконечно; `loop: n` прогоняет `n` циклов. `delay` применяется один раз, перед первым циклом.
- `loopMode: "ping-pong"` (по умолчанию) каждый цикл анимирует к целям и обратно — без визуального скачка. `"restart"` прыгает назад и проигрывает вперёд заново — для спиннеров на полный оборот и шиммеров.
- **Зацикленная анимация никогда не коммитит** — это эффект, а не смена состояния. Сохранённый стиль элемента не тронут, и элемент заканчивает в базовом виде, каким бы ни был режим или счётчик.
- Остановка: более поздний `animateTo` / `animateFrom` на элементе заменяет цикл (это и есть идиома — анимируй к состоянию покоя, чтобы остановить пульс). Цикл также останавливается, когда элемент уходит с экрана; он не возобновляется, если экран возвращается из истории роутера.

> **NOTE**
> тип опций также допускает `layer?: number` — лазейку, нацеленную на внутренний слой стиля (механизм за стилями `onPressed`/ориентации). Не часть поддерживаемой поверхности.

Для твинов свободных значений (числа, которые применяешь сам) используй `animate()` — см. animate.

Словарь свойств

Раскладка — **только флексбокс**. Каждый элемент — флекс-контейнер, `position: relative`, `flexDirection: column` по умолчанию. Два исключения: `UIRow` переключает на `row`, а `UIButton` по умолчанию `row` **с центрированием детей по обеим осям** (`justifyContent` + `alignItems` `"center"`) — форма «иконка + подпись»; см. interactive.md.

Раскладка контейнера (`UIRow` / `UIColumn` / `UIScrollable` / экраны / кнопки / виджеты)

```ts
flexDirection:  "row" | "column"
justifyContent: "flex-start" | "center" | "flex-end" | "space-between" | "space-evenly"
alignItems:     "flex-start" | "center" | "flex-end" | "stretch"     // по умолчанию "stretch"
gap:            UIValue
flexWrap:       "nowrap" | "wrap" | "wrap-reverse"
```

Флекс-ребёнок (все элементы)

```ts
flex:        number | string
flexGrow:    number            // по умолчанию 0 — без него ничего не растёт
flexShrink:  number            // по умолчанию 0 — и ничего не сжимается
flexBase:    UIValue | "auto"  // внимание: flexBase, не flexBasis
alignSelf:   "flex-start" | "center" | "flex-end" | "stretch"
aspectRatio: number
```

`flex: 1` разворачивается в `flexGrow: 1, flexShrink: 1, flexBase: 0` — нулевой базис и есть суть. Для **детей равной ширины** (таб-бары, пары кнопок) ставь `flex: 1` на каждого: они делят всю ось поровну. Один `flexGrow: 1` распределяет только *оставшееся* место поверх базисов по контенту, поэтому ребёнок с более длинной подписью остаётся шире.

Размер и позиция (все элементы)

```ts
width, height:            UIValue | "auto"
minWidth, maxWidth,
minHeight, maxHeight:     UIValue | "auto"
position:                 "relative" | "absolute" | "static"    // по умолчанию "relative"
top, left, bottom, right: UIValue | safe-area keyword          // смещения; % разрешён
inset:                    UIValue | "auto"                     // все четыре сразу; % разрешён
boxSizing:                "border-box" | "content-box"         // по умолчанию "border-box"
```

`inset` — сокращение для `top`/`right`/`bottom`/`left` одним значением — обычный способ растянуть абсолютно позиционированного ребёнка на родителя. Полная форма всегда побеждает его, какая бы ни была объявлена последней, так что `{ inset: 0, top: 20 }` значит «заполни, но на 20 вниз от верха»:

```ts
UIView().style({ position: "absolute", inset: 0 })        // оверлей во весь родитель
UIView().style({ position: "absolute", inset: "10%" })    // отступ 10% с каждого края
```

> **NOTE**
>
> в отличие от CSS, `inset` принимает **одно** значение — многозначной формы `inset: "0 10"` нет, в согласии с `padding`/`margin` в этом SDK. Для смещений по сторонам используй полные формы.

Отступы и поля (все элементы)

Сокращения и полные формы взаимозаменяемы (`p` ≡ `padding`, `pt` ≡ `paddingTop`, …):

```ts
p  (padding)                          px (paddingHorizontal)   py (paddingVertical)
pt pb pl pr                           // на сторону
m  (margin)  — also accepts "auto"    mx (marginHorizontal)    my (marginVertical)
mt mb ml mr  — also accept "auto"
```

`mx: "auto"` центрирует элемент фиксированной ширины; одиночное поле `"auto"` толкает его к дальней стороне.

Фон и рамка (рисуемые элементы)

Доступны на контейнерах, `UIScreen`, `UIButton`, `UIInput`, `UIVideo`. У `UIText`, `UIImage` и `UISpacer` есть только `bgColor` (плюс собственный числовой `borderRadius` у `UIImage`).

```ts
bgColor:    Color                                     // ≡ backgroundColor; все элементы
bgImage:    string | FetchResponse | File             // ≡ backgroundImage; только декор
bgSize:     "cover" | "contain" | "tile"              // ≡ backgroundSize
bgGradient: string                                    // один+ слоёв linear-
gradient()/radial-gradient() через запятую

border:       string | number                         // "1px solid #333" или голая ширина
borderWidth:  number
borderColor:  Color
borderTop / borderRight / borderBottom / borderLeft   // сокращения на сторону
borderTopWidth / borderTopColor / …                   // полные формы на сторону

borderRadius: UIValue | string                        // строка = на угол "0 0 20 20"
borderTopLeftRadius / borderTopRightRadius /
borderBottomLeftRadius / borderBottomRightRadius: UIValue
```

Слои фона рисуются снизу вверх: `bgColor` → `bgImage` → `bgGradient` — поэтому градиент поверх изображения делает привычный затемняющий scrim для защиты текста:

```ts
.style({ bgImage: photo, bgSize: "cover",
         bgGradient: "linear-gradient(to top, rgba(0,0,0,0.7), transparent)" })
```

Градиенты (`bgGradient`)

`bgGradient` принимает CSS-строку градиента — `linear-gradient()` или `radial-gradient()`. Цветовые стопы принимают любой `Color` (hex, `rgb()`/`rgba()`, именованный, `transparent`) с необязательными позициями `%`.

Линейный — необязательное направление (`to <side>` / `<angle>deg`, по умолчанию `to bottom` = `180deg`), затем ≥2 стопа:

```ts
.style({ bgGradient: "linear-gradient(135deg, #FF4032, #7B2FF7)" })
.style({ bgGradient: "linear-gradient(to right, #000 0%, #333 60%, #fff 100%)" })
```

Радиальный — необязательный префикс `[<shape> || <extent>]? [at <position>]?`, затем ≥2 стопа:

```ts
.style({ bgGradient: "radial-gradient(#7B2FF7, #0a0a1a)" })                       // по умолчанию: ellipse, farthest-corner, at center
.style({ bgGradient: "radial-gradient(circle at 50% 40%, #7B2FF7, #0a0a1a)" })    // circle, вне центра
.style({ bgGradient: "radial-gradient(ellipse closest-side at top left, #fff, #101014)" })
.style({ bgGradient: "radial-gradient(circle 80px at center, #fff, #101014)" })   // явный радиус
```

Размер радиального — это **ключевое слово extent** или **явные радиусы**. Оба, плюс форма и позиция, учитываются на **каждой** поверхности (браузерный предпросмотр, iOS, headless-рендерер):

Поле	Значения	По умолчанию
shape	`circle` | `ellipse`	`ellipse`
extent	`closest-side` | `closest-corner` | `farthest-side` | `farthest-corner`	`farthest-corner`
явные радиусы	`circle <len>` или `ellipse <len-or-%>{2}` — длины это `px`/`em`/`vw`/`calc()` или `%` от оси бокса	—
`at <position>`	ключевые слова (`center`, `top`, `left`, `bottom right`, …) или `%` (`20% 80%`)	`center`

> **ТОЧНОСТЬ РАДИАЛЬНОГО**
>
> **браузерный предпросмотр** рендерит каждую CSS-радиальную форму точно (это сырой CSS). На **устройстве (iOS)** и в **headless-рендерере** поля выше разрешаются 1:1. Более редкие CSS-формы вне этой таблицы (напр. позиции из 4 значений `at left 10% top 20%`) откатываются там к центрированному дефолту, тогда как предпросмотр остаётся точным. `conic-gradient()` не поддерживается.

Наслоение нескольких градиентов. Раздели несколько градиентов запятыми (как CSS `background-image`), чтобы сложить их — первый в списке рисуется сверху, поэтому веди теми, у кого есть `transparent`-области:

```ts
.style({ bgColor: "#0a0a1a", bgGradient:
  "radial-gradient(circle at 15% 20%, #7B2FF7, transparent 60%), " +
  "radial-gradient(circle at 85% 80%, #2FB0F7, transparent 55%)" })
```

Линейные и радиальные слои можно смешивать, каждый учитывает полный синтаксис выше, а стек композится на **каждой** поверхности (браузерный предпросмотр, iOS, headless-рендерер) — обычный способ строить мягкие мульти-свечения или «авроровые» фоны.

> **NOTE**
>
> `bgImage` — декор за детьми. Изображение, которое *является* контентом, — это `UIImage` — см. content.md.

Всё остальное (все элементы)

```ts
opacity:       number | string          // 0..1
transform:     string                   // строка в стиле CSS, напр. `translateY(12px)` — отлично для жестов
display:       "none" | "flex"          // "none" убирает из раскладки
overflow:      "visible" | "hidden"     // по умолчанию "hidden" — дети обрезаются
pointerEvents: "all" | "none"           // рисуемые элементы; "none" пропускает тапы насквозь
```

Значения и единицы (`UIValue`)

```ts
16                  // голое число = логические px (единица по умолчанию везде)
"50%"               // процент от родителя
"50vw" "50vh"       // ширина/высота вьюпорта
"50vmin" "50vmax"
"1.5em"             // × СОБСТВЕННЫЙ fontSize элемента (по умолчанию 14) — наследования стиля НЕТ
"calc(100vw - 32px)"
"min(...)" "max(...)" "clamp(min, val, max)"
"var(--name)"       // переменная темы, необязательный фолбэк "var(--name, 16px)" — см. theme.md
```

- `em` разрешается относительно собственного `fontSize` элемента, никогда родительского — задай `fontSize` на том же элементе, иначе `em` значит «относительно 14px».
- `calc()` / `min()` / `max()` / `clamp()` комбинируют px, единицы вьюпорта, `em` и ключевые слова safe-area и вкладываются свободно — но **не поддерживают `%`**.

```ts
width: "calc(100% - 20px)"   // ✗ % не может быть внутри calc
width: "calc(100vw - 20px)"  // ✓ используй единицы вьюпорта
```

Цвета (только UI-стили)

Строки цвета UI парсятся UI-движком и принимают **больше**, чем `ColorInput` со стороны движка (соглашения):

```ts
"#f33" "#f33c" "#ff3333" "#ff3333cc"      // hex, 3/4/6/8 цифр
"rgb(255, 51, 51)" "rgba(255, 51, 51, 0.8)"
0xff3333                                   // упакованный int
// именованные — ровно этот набор, ничего больше:
"white" "black" "red" "green" "blue" "yellow" "orange" "purple"
"gray" "cyan" "magenta" "brown" "transparent" "clear"
```

> **NOTE**
> этот парсер существует только для UI-стилей. API 2D/3D (`Sprite.color`, `Material`, фон `Scene2D`) принимают только hex-строки и упакованные int — `rgba(...)` и именованные цвета там молча становятся непрозрачным чёрным.

Безопасные зоны и комфорт

Два семейства ключевых слов краёв, разделённые по владельцу:

- `safe-top` / `safe-bottom` / `safe-left` / `safe-right` — **сырые отступы** устройства (вырез, home-индикатор). Факты о железе; 0 на устройствах без них.
- `comfort-top` / `comfort-bottom` / `comfort-left` / `comfort-right` — **где контенту комфортно начинаться**. На отступе-*зазоре* (вырез/home-индикатор iOS, жестовая навигация — запас уже встроен) это отступ с нижней границей из настраиваемой в приложении ручки, `max(safe-edge, knob)`. На системной *панели* точной высоты (статус-бар Android; трёхкнопочная панель навигации — контент на границе отступа касается хрома) ручка **добавляется** сверх него, `safe-edge + knob`. Хост объявляет, какие края — панели, поэтому один и тот же экран везде разрешается в правильное для платформы значение: таб-бар с `pb: "comfort-bottom"` сидит вплотную над home-индикатором iOS (34), но держит зазор над кнопочной панелью Android (48 + 4). Дефолты ручек: 12 логических px сверху, 4 снизу, 16 по горизонтали — так что `comfort-left`/`comfort-right` заодно служат **гаттером страницы**, а в ландшафте вырез автоматически побеждает.

Парные и всесторонние формы (каждая сторона разрешается независимо — вырез в ландшафте различает лево и право): `comfort-x` на `px`/`mx`, `comfort-y` на `py`/`my`, `comfort-all` и `safe-all` — только на сокращениях `p`/`m`.

Где они принимаются: padding и margin на сторону, смещения позиции (`top`/`left`/`bottom`/`right`) и внутри `calc()`/`min()`/`max()`:

```ts
bar.style({ pt: 8, pb: "comfort-bottom" })        // таб-бар: 34 над home-индикатором, 52 над кнопочной панелью Android, 4 на устройствах без отступов
page.style({ px: "comfort-x" })                   // гаттер страницы, который в ландшафте ещё и обходит вырез
list.style({ pb: "calc(comfort-bottom + 56px)" }) // прокручиваемый контент, освобождающий место под плавающую панель 56px
```

Настраивай ручки (или любой край) через тему:

```ts
theme({ "comfort-left": 20, "comfort-right": 20 })   // гаттер этого приложения — 20
```

Переменные темы — `theme()`

Любое значение стиля может быть **переменной темы** — `"var(--accent)"` — разрешаемой вживую из единой таблицы приложения, которую ведёт `theme()`; повторный вызов `theme()` перестилизует работающий UI на месте (тёмная тема — это второй вызов). Каноническая страница — theme.md: определение переменных, паттерн аксессоров, системные ключи (`color`, `fontFamily`, `primaryColor`, панель `UITabs`, ручки комфорта), живая смена темы и `replaceTransition`.

```ts
const T = theme({ accent: "#15A34A" })     // T.accent === "var(--accent)"
label.style({ color: T.accent })
```

Адаптивность: `onLandscape` / `onPortrait`

Любой объект стиля может вкладывать переопределения ориентации; они сливаются сверху, когда устройство в этой ориентации, и снимаются, когда она меняется:

```ts
screen.style({ flexDirection: "column", p: 16, onLandscape: { flexDirection: "row", p: 32 }
})
```

Классы стилей — `$name`

Ключ с префиксом `$` внутри `.style()` объявляет **именованное состояние стиля** — как `onPressed`, но с любым именем и управляемое тобой через прокси `el.class`. Классы **каскадируют на потомков** (один переключатель перестилизует целый составной контрол), а два имени переключает система: `$pressed` и `$focused`. Каноническая страница — classes.md: объявление, контракт `el.class`, правила каскада, зарезервированные классы и приоритет.

```ts
const toggle = UIButton(UIText("Dark mode")).style({
  bgColor: "#222",
  $checked: { bgColor: "#FF4032", duration: 150 },
})
toggle.onClick(() => toggle.class.checked = !toggle.class.checked)
```

Дефолты, которые удивляют

- `flexShrink: 0` — элементы не сжимаются под размер. Исключения — `UIScrollable`, `UIVirtualizedList` и `UIPager` (у них по умолчанию `flexShrink: 1`, поэтому они сжимаются/прокручиваются вместо переполнения) — но оборачивающим контейнерам между ними и экраном всё ещё нужен `flexShrink: 1` руками.
- `flexGrow: 0` — ничего не растёт вдоль главной оси без спроса; неявных мин-размеров тоже нет.
- `alignItems: "stretch"` — дети **заполняют поперечную ось** по умолчанию; задай размер или `alignSelf`, чтобы отказаться.
- `overflow: "hidden"` — дети обрезаются по боксу родителя.
- `position: "relative"`, `boxSizing: "border-box"` (width/height включают отступ + рамку).
- Фон экрана **чёрный**, а `fontSize` по умолчанию 14 — всегда задавай `bgColor` и `color` текста явно.

Логические px и дизайн-канвас

Все единицы — логические px (iOS pt / Android dp), никогда физические — `fontSize: 16` выглядит одинаково на каждом устройстве. Проектируй под канвас телефона шириной ~360–430 (390 — хороший дефолт) и высотой ~670–930; вертикальное пространство скудно на экранах класса SE, поэтому длинный контент — в теле `UIScrollable` (сами экраны никогда не прокручиваются).

Подводные камни

```ts
display: "grid"                    // ✗ только флексбокс (Yoga)
lineHeight: 1.5                    // ✗ число это px (= 1.5px); множитель это "1.5em"
node.style({ width: "calc(100% - 20px)" })   // ✗ без % внутри calc — используй 100vw
```

Смотрите также

- Модель элементов UI — фабрики, дети, ссылки, `onLayout`.
- Тема — переменные на всё приложение, `var()`, живая смена темы.
- Классы стилей — состояния `$name`, каскад, `$pressed`/`$focused`.
- Контейнеры — где применяются свойства раскладки контейнера.
- Элементы контента — стили текста, `objectFit`, `tintColor`.
- animate — твины свободных значений и функции сглаживания.

4.2.3 Тема

Одна таблица переменных на всё приложение, разрешаемая вживую ядром UI. `theme(values)` **сливает** переменные в неё и возвращает типизированные аксессоры, чьи значения *и есть* строки `var()`; любое значение стиля может ссылаться на переменную, а повторный вызов `theme()` перестилизует **работающий UI на месте** — тёмная тема, смена бренда или персональный акцент — это ещё один вызов, а не пересборка. Эта страница — канонический дом `theme()`; словарь свойств, в которые переменные подставляются, — в styling.md.

Обзор

Стандартная форма — маленький модуль токенов, который импортирует каждый экран: палитра в `theme()`, шкалы — обычные константы:

TS

```
// theme.ts — определи один раз, экспортируй аксессоры
const palette = {
  bg: "#F4F6F5", card: "#FFFFFF", border: "#E4E8E6",
  text: "#131A17", muted: "#606B65",
  accent: "#15A34A", accentSoft: "#E7F6ED", onAccent: "#FFFFFF",
}

// Системные ключи: цвет текста по умолчанию + чем красят себя кит-компоненты (панель
UITabs).
theme({ color: palette.text, primaryColor: palette.accent, mutedColor: palette.muted })

export const colors: { [K in keyof typeof palette]: string } = theme(palette)
export const font = {    // шкала типографики — распыляй в стили: .style({ ...font.h2 })
  h2: { fontSize: 22, fontWeight: 700 }, body: { fontSize: 16 },
  small: { fontSize: 14 }, tiny: { fontSize: 12, fontWeight: 500 },
}
```

TS

```
// любой экран
import { colors, font } from './theme'

const card = UIColumn(
  UIText("Total").style({ ...font.small, color: colors.muted }),
  UIText("$1,240").style({ ...font.h2 }),
).style({ bgColor: colors.card, border: `1px solid ${colors.border}`, borderRadius: 16, p: 16
})
```

TS

```
// позже, откуда угодно — каждый стиль, ссылающийся на токен, перекрашивается на месте
theme({ bg: "#0C0E13", card: "#161A23", text: "#EEF1F8", border: "#2A3040" })   // тёмная
тема
```

`colors.card` — это буквально строка `"var(--card)"` — хост разрешает её в момент применения стиля. Шкалы отступов и типографики остаются обычными TS-константами: переменная темы — для того, что должно уметь *меняться, пока приложение работает*.

> **NOTE**
>
> экспорт аксессоров аннотирован `{ [K in keyof typeof palette]: string }` не случайно. `theme()` выводит литеральные типы (`"var(--accent)"`), и хелпер вида `(color = colors.accent)` иначе сузил бы свой параметр до этого одного токена.

Определение переменных

```ts
theme(values): accessors      // СЛИВАЕТ в живую таблицу; неупомянутые ключи сохраняют значения
```

- **Строки проходят как есть** — цвета, имена шрифтов, целые выражения (`"max(safe-top, 24px)"`).
- **Числа — длины** (логические px).
- `null` **удаляет ключ** (ручки комфорта сбрасываются к встроенным дефолтам).
- Env-имена (`safe-top`…, `vw`/`vh`/`vmin`/`vmax`) — **факты хоста, никогда не ключи темы** — записи пропускаются с предупреждением в консоли.
- Один нестилевой ключ: `replaceTransition` ниже.

Каждый вызов возвращает аксессоры для определённых им ключей; возвращённый объект — всё, что нужно экспортировать из модуля токенов.

Чтение переменных

```ts
label.style({ color: T.accent })                    // аксессор — T.accent === "var(--accent)"
label.style({ color: "var(--accent)" })             // сырая строковая форма — то же самое
icon.style({ tintColor: "var(--accent, #0A84FF)" })  // фолбэк действует, пока ключ не задан
list.style({ pl: "calc(var(--comfort-left) * 2)" })  // компонуется в calc()/min()/max()
```

`var()` работает в **любом значении UI-стиля** и больше нигде — что это исключает, см. Подводные камни. `var()` для ключа, который никогда не определялся, разрешается в свой фолбэк или в ничто.

Системные переменные

Некоторые ключи читает сам SDK, а не только твои стили. Они заранее типизированы как статические свойства глобала — `theme.color`, `theme.primaryColor`, … — так что модуль токенов им не нужен.

Текст по умолчанию — два ключа управляют каждым `UIText`, который не задал собственное значение, так что светлая тема — это одна строка вместо `color: "black"` на каждой подписи:

```ts
theme({ color: "#1C1C1E", fontFamily: font("manrope") })   // весь нестилизованный текст следует
```

Не заданные, они равны дефолтам хоста (белый текст, шрифт хоста). Собственные `color` / `fontFamily` элемента всегда побеждают. Фон экрана остаётся у каждого экрана свой (`bgColor`)

— стилизуй его явно.

Кит-компоненты — `UITabs` стилизует свою панель целиком через переменные темы (с тёмными фолбэками, так что приложение без темы всё равно выглядит правильно):

Ключ	Управляет
`primaryColor`	иконка + подпись активного таба (и общий «акцент приложения» для кит-компонентов)
`mutedColor`	иконка + подпись неактивного таба / вторичный контент
`tabbarBg`	поверхность таб-бара
`tabbarBorder`	верхняя волосяная линия таб-бара
`screenBg`	фон по умолчанию за экранами табов
`badgeColor`	точка бейджа / пилюля счётчика

Ручки комфорта — `"comfort-top"` / `"comfort-bottom"` / `"comfort-left"` / `"comfort-right"` настраивают комфортные токены безопасных зон (только простые длины):

```ts
theme({ "comfort-left": 20, "comfort-right": 20 })   // гаттер страниц этого приложения — 20
```

> **NOTE**
> голый токен стиля (`pt: "comfort-top"`) применяет *формулу* безопасной зоны; `var(--comfort-top)` читает **сырое значение ручки** (по умолчанию 12) для ручной композиции.

Живая смена темы

Поскольку каждый `var()` переразрешается на месте, вызов `theme()` перестилизует то, что сейчас на экране, — без пересборки экрана, без протаскивания пропсов, без точки вызова, которую можно забыть. Поэтому таблица — правильный дом для всего, что решается *в рантайме*:

```ts
// Тёмная тема: второй вызов.
const setDark = (dark: boolean) => theme(dark ? darkPalette : lightPalette)

// Акцент с сервера: выведи несколько переменных из одного hex и перезапиши их.
// Каждый стиль, где написано var(--accent) / var(--accentSoft), следует — включая экраны,
// которые уже смонтированы.
const applyBrand = (hex: string | null) =>
  theme(hex ? { accent: hex, accentSoft: soften(hex) } : { accent: "#15A34A", accentSoft:
"#E7F6ED" })
```

Записи применяются сразу, когда бы ни случились — между вызовом `theme()` и экранами, построенными до или после него, нет опасности порядка.

`replaceTransition`

Один ключ потребляется роутером SDK и не попадает в таблицу переменных: общий для приложения переход по умолчанию для `Router.replace` (из коробки `"none"`).

```ts
theme({ replaceTransition: "fade" })   // любое имя перехода или свой спек; null сбрасывает
```

Явный `Router.replace(screen, { transition })` на вызов всё равно побеждает. Аксессор не возвращается — `var(--replaceTransition)` не значение стиля. См. screen-router.md.

Подводные камни

```ts
// ✗ var() вне UI-стилей — SVG-XML, цвета движка, параметры нативных вью получают буквальную строку
SvgSource(`<circle fill="${colors.accent}" ...>`)      // не рендерит ничего полезного
sprite.color = colors.accent                           // API движка парсят только hex/упакованный int
// ✓ держи сырой объект палитры экспортированным рядом с аксессорами для этих потребителей
SvgSource(`<circle fill="${palette.accent}" ...>`)

// ✗ темизация env-имени — принадлежит хосту, пропускается с предупреждением
theme({ "safe-top": 0 })

// ✗ формула в ручке комфорта — ручки принимают только простые длины
theme({ "comfort-bottom": "max(safe-bottom, 16px)" })
// ✓ пиши формулу инлайн в стиле или как собственную переменную
theme({ tabInset: "max(safe-bottom, 16px)" }); bar.style({ pb: "var(--tabInset)" })
```

Смотрите также

- Стилизация — словарь свойств и формы значений, куда подставляются переменные.
- Классы стилей — именованные состояния на элемент (тема = на всё приложение, классы = на элемент).
- Шрифты — `theme({ fontFamily: font(...) })`, механизм шрифта по умолчанию.
- UIPager и UITabs — таб-бар, который стилизуют кит-переменные.

4.2.4 Классы стилей

Класс стиля — это именованное состояние стиля, которое ты объявляешь инлайн и переключаешь из кода — как `onPressed`, но с любым именем и управляемое тобой: выбрано,

отмечено, активно, развёрнуто. Объявляется блоком с префиксом `$` внутри `.style()`, переключается через прокси `el.class`, и — то, что делает составные контролы дешёвыми, — класс, установленный на элементе, **каскадирует на всех его потомков**, зажигая и их одноимённые `$`-блоки.

Обзор

Кнопка «избранное»: один переключатель перестилизует контейнер, иконку *и* подпись — каждая часть объявляет свою реакцию, ничего не перерендеривается:

```ts
const fav = UIButton(
  UIImage(assetIcon("lucide:heart")).style({ width: 18, height: 18,
    tintColor: "#8a919e", $fav: { tintColor: "#ff453a" } }),
  UIText("Favorite").style({ color: "#8a919e", $fav: { color: "#ff453a" } }),
).style({ height: 36, px: 12, gap: 6, borderRadius: 18,
          bgColor: "#17181c", $fav: { bgColor: "#2a181a", duration: 150 } })

fav.onClick(ev => ev.target.class.fav = !ev.target.class.fav)
```

Объявление: блоки `$name`

Ключ с префиксом `$` внутри любого вызова `.style()` объявляет класс; блок держит переопределения плюс необязательный переход:

```ts
el.style({
  bgColor: "#151515",
  $selected: { bgColor: "#1d2b45", duration: 150 },   // duration/delay (мс) анимируют смену
})
```

Каждый элемент в каскаде анимируется по `duration` **собственного** блока — переключение это одна запись, переходы локальны.

Управление: `el.class`

`el.class` — прокси с тем же контрактом, что и `el.style`:

```ts
el.class.selected                    // чтение: активен ли на ЭТОМ элементе? → boolean
el.class.selected = true             // активировать (false деактивирует)
el.class.open = !el.class.open      // переключить
el.class.done = () => sig.value      // реактивная привязка — перепримeняется при смене сигнала
el.class({ checked: true, done: () => sig.value })   // пакетная форма: возвращает элемент, чейнится
Object.keys(el.class)                // активные имена классов элемента
```

- Имя работает **с ведущим `$` или без** — `el.class.checked` и `el.class.$checked` эквивалентны (удобно при копировании ключа из объявления).
- Активный набор **запоминается на элементе** — класс остаётся активным при закрытии и повторном открытии экрана.
- Реактивная форма привязывает класс к сигналу; ручное переключение и привязка — один механизм.

Каскад

Класс, установленный на элементе, активен и на **всех его потомках** — их одноимённые `$`-блоки тоже зажигаются, как класс `.dark` на `<body>` в CSS. Это паттерн для любого составного контрола: части объявляют свои реакции, контейнер несёт один флаг.

Кнопка таб-бара с подсветкой, привязанной к активному индексу пейджера, — иконка и подпись следуют одной привязке на кнопке:

```ts
const activeTab = signal(0)
pager.onSelect(i => activeTab.value = i)      // тапы И свайпы двигают подсветку

const TabButton = (label: string, icon: string, i: number) => UIButton(
  UIImage(assetIcon(icon)).style({ width: 24, height: 24,
    tintColor: "#98A29C", $active: { tintColor: "#15A34A" } }),
  UIText(label).style({ fontSize: 12, mt: 4, color: "#98A29C", $active: { color: "#15A34A" } }),
)
  .style({ flex: 1, flexDirection: "column", py: 6 })
  .class({ active: () => activeTab.value === i })     // ОДНА привязка ведёт каждую часть
  .onClick(() => pager.select(i))
```

Правила каскада:

- Чтения отражают только **собственные** классы элемента — `el.class.active` на иконке выше равен `false`, даже пока каскад кнопки её стилизует.

- **Опции отказа нет**: пока класс есть у предка, `el.class.name = false` у потомка сбрасывает только его собственный флаг.
- Каскад **останавливается на presentable-корнях** — размещённые экраны (страницы `UIPager`) и виджеты не наследуют от того, что их содержит.

`$pressed` и `$focused`

Два имени классов **зарезервированы и переключаются системой**: `$pressed`, пока элемент удерживается, `$focused`, пока инпут в фокусе. В отличие от `onPressed`/`onFocused` они каскадируют — дети кнопки могут перестилизоваться во время её нажатия:

```ts
UIButton(
  UIImage(icon).style({ $pressed: { opacity: 0.5 } }),
  UIText("Buy").style({ $pressed: { color: "#999" } }),
).style({ $pressed: { transform: "scale(0.97)" } })
```

Предпочитай `$pressed`/`$focused` в новом коде — один словарь для элемента и его детей. `onPressed`/`onFocused` остаются формами на элемент, **без каскада**; они никогда не зажигаются от предка — то, что нужно на вложенных интерактивах (кнопка внутри кликабельной карточки не должна «нажиматься» вместе с карточкой).

Приоритет

Когда два состояния задают одно свойство, побеждает более позднее и более интерактивное, от низшего к высшему:

```
base style  <  $classes (later-declared beats earlier)  <  $pressed / $focused  <  onPressed / onFocused
```

Отклик нажатия/фокуса всегда бьёт остальные классы, пока элемент нажат или в фокусе, независимо от порядка объявления — поэтому отмеченная кнопка всё ещё показывает нажатие:

```ts
UIButton(UIText("Save")).style({
  bgColor: "#222",
  $checked: { bgColor: "#2a7" },        // показывается, пока отмечено
  $pressed: { bgColor: "#195" },        // побеждает, пока палец внизу, даже когда отмечено
})
```

> **РЕАКТИВНЫЙ ЯРЛЫК**
> одному вычисляемому значению класс не нужен — любое примитивное свойство стиля принимает привязку `() => value` напрямую (`bgColor: () => sel.value ? "#FF4032" : "#333"`). Тянись к классу, когда хочешь *именованный, переиспользуемый* блок переопределений — особенно такой, на который через каскад реагируют несколько элементов.

Смотрите также

- Стилизация — словарь свойств, из которого сделаны блоки классов.
- Тема — переменные на всё приложение (тема = на всё приложение, классы = состояния на элемент).
- Сигналы — реактивность за привязками `() => value`.
- Интерактивные элементы — `onPressed`/`onFocused`, ripple, механика нажатия.

4.2.5 Контейнеры UI

Структурные элементы: `UIColumn` стекает детей вертикально, `UIRow` — горизонтально, `UIScrollable` делает область переполнения панируемой, `UISpacer` съедает свободное место. Экраны сами никогда не прокручиваются — экран-один-длинный-поток это фиксированный хром плюс одно тело `UIScrollable` с `flexGrow: 1`; см. screen-router.md.

Обзор

TS

```
const screen = UIScreen(
  UIRow(
    UIText("Library").style({ fontSize: 24, fontWeight: 700, color: "white" }),
    UISpacer(),                                          // толкает счётчик к дальнему краю
    UIText("12 items").style({ color: "#888" }),
  ).style({ px: 16, py: 12, alignItems: "center" }),

  UIScrollable(
    UIColumn(UIText("First").style({ color: "white" })).style({ p: 16 }),
    UIColumn(UIText("Second").style({ color: "white" })).style({ p: 16 }),
  ).style({ flexGrow: 1, gap: 8 })                      // flexGrow: 1 = занять место под
шапкой
    .onScroll(pos => console.log("scrolled to", pos)),
).style({ bgColor: "black", pt: "safe-top" })

screen.open()
```

UIRow / UIColumn

```ts
UIColumn(...children: (UINodeChild | UINodeChild[])[]): UIColumn
UIRow(...children: (UINodeChild | UINodeChild[])[]): UIRow
```

Дети вариадические; аргумент-**массив разворачивается на один уровень**, так что `UIColumn(header, items.map(Row), footer)` не нуждается в spread. Одинокий аргумент-функция — реактивная привязка детей — см. сигналы. Единственная разница между двумя контейнерами — дефолтный `flexDirection` (`column` против `row`) — оба принимают полную поверхность стиля контейнера (`UIContainerStyle` = свойства элемента + рисуемого + раскладки контейнера, см. styling.md). Дети `null` / `undefined` / `false` пропускаются — см. overview.md.

`UIButton` — тоже полноценный контейнер, но с **другими дефолтами**: `row` плюс центрирование детей по обеим осям — см. interactive.md.

Избегай контейнеров-обёрток, существующих только для выравнивания чего-то — выравнивание это свойство контейнера:

```ts
UIRow(UIColumn().style({ flexGrow: 1 }), label)      // ✗ фантомный элемент-распорка
UIRow(label).style({ justifyContent: "flex-end" })     // ✓
```

Управление детьми

Все контейнеры на этой странице делят один императивный API детей (работает до и после появления элемента на экране; без диффинга):

```ts
c.append(...nodes: UINodeChild[]): this
c.insert(index: number, ...nodes: UINodeChild[]): this   // индекс в c.children
c.remove(...nodes: UINode[]): this                        // удаляет по идентичности
c.setContent(children: UINodeChild[]): this               // заменить всё
c.children                                                 // readonly UINodeChild[]
```

`setContent` — примитив «ре-рендера» — построй свежий массив (напр. `items.map(Row)`) и подмени его. Для длинных или неограниченных данных используй `UIVirtualizedList`.

UIBox — устарел

```ts
UIBox(...children): UIBox   // легаси — используй UIRow / UIColumn
```

Легаси-контейнер, чьё единственное отличие — дефолт `justifyContent` и `alignItems` в `"center"` (дети центрированы по обеим осям). Оставлен для старых проектов; новый код

пиши с `UIRow`/`UIColumn` плюс явное выравнивание.

UIScrollable

Тот самый скролл-контейнер — каждая прокручиваемая область это он: тело длинного экрана (экраны сами никогда не прокручиваются — см. screen-router.md), список под закреплённой шапкой, горизонтальная карусель.

TS

```
UIScrollable(...children: (UINodeChild | UINodeChild[])[]): UIScrollable
```

Дополнительные стили поверх поверхности контейнера:

TS

```
scrollDirection:     "horizontal" | "vertical"      // по умолчанию vertical
showScrollbar:       boolean
overscrollMode:      "none" | "absorb" | "default"  // поведение края при протягивании за
контент
refreshControlColor: Color                          // подкрашивает спиннер pull-to-refresh
keyboardDismissMode: "interactive" | "scroll" | "none"
// как жест прокрутки в ЭТОМ scrollable прячет клавиатуру. "interactive" (по умолчанию):
// протягивание вниз над клавиатурой уводит её вслед за пальцем (ощущение чата). "scroll":
любой
// драг прячет её в момент начала (ощущение результатов поиска). "none": прокрутка никогда не
// прячет. См. доктрину скрытия клавиатуры в interactive.md.
snap:                "none" | "start" | "center" | "end"   // по умолчанию none
// пейджинг: когда драг заканчивается, остановиться на границе прямого ребёнка вдоль оси
прокрутки.
// Значение выбирает, где этот ребёнок встаёт во вьюпорте — "start" выравнивает по переднему
краю,
// "center" центрирует, "end" — по заднему. Цели снапа — сами дети, поэтому их размеры могут
// различаться (карусель карточек с выглядывающими соседями — это `snap: "center"`).
// См. «Карусель / пейджер» ниже.
```

События прокрутки (чейнятся, как все `on*`):

TS

```
s.onScroll(cb: (scrollPosition: number) => void): UIScrollable   // логические px от
стартового края
                                                                 // (горизонтальный сообщает
смещение x)
s.onScrollRelease(cb: () => void): UIScrollable                  // палец поднят
s.onOverscroll(cb: (delta: number) => void): UIScrollable        // протянут за край (px)
```

Pull-to-refresh — onRefresh

```ts
s.onRefresh(cb: () => void | Promise<void>): UIScrollable   // спиннер висит, пока промис не уляжется
```

```ts
UIScrollable(rows)
  .style({ flexGrow: 1, refreshControlColor: "#FF4032" })
  .onRefresh(async () => { await load() })
```

> **NOTE**
>
> подключай `onRefresh` **до** монтирования элемента — хосты читают колбэк при создании узла. Только вертикальные scrollable и только нативные хосты (веб-вьюеры — no-op). У `UIVirtualizedList` те же `onRefresh` + `refreshControlColor`.

> **NOTE**
>
> `scrollTo` **нет** — позицию `UIScrollable` нельзя задать программно. Это текущее ограничение. Если нужна программная прокрутка (`scrollTo` / `scrollToEnd` / `scrollToKey`), используй `UIVirtualizedList`.

> **NOTE**
>
> `UIScrollable` по умолчанию имеет `flexShrink: 1` (единственное исключение из глобального дефолта `flexShrink: 0`), поэтому вертикальный scrollable в колонке сжимается и прокручивается вместо переполнения. Оборачивающие контейнеры между ним и экраном по-прежнему дефолтятся в 0 и требуют `flexShrink: 1` вручную; верни scrollable `flexShrink: 0`, чтобы отказаться. `flexGrow: 1` для «занять оставшееся место» всё ещё задаёшь ты сам.

Горизонтальная карусель:

```ts
UIScrollable(items.map(Card))
  .style({ scrollDirection: "horizontal", showScrollbar: false, gap: 12, px: 16 })
```

Карусель / пейджер — snap

`snap` заставляет scrollable останавливаться на границах детей. Обёртки `UISlider`/`UICarousel` нет — snap и есть примитив, а пейджер — это горизонтальный scrollable с детьми во всю ширину плюс `onScroll` для отслеживания активной страницы:

```ts
UIScrollable(slides.map(s => Slide(s).style({ width: "100%" })))
  .style({ scrollDirection: "horizontal", showScrollbar: false, snap: "start" })
```

Текущая страница. События страницы нет — индекс это `onScroll`, делённый на ширину страницы. Меряй шаг через `onLayout` (никогда не предполагай ширину экрана: паддинги, инсеты и split view её меняют) и веди сигнал, чтобы точки/подписи к нему привязались:

```ts
const page = signal(0)
let step = 1

const track = UIScrollable(slides.map(Slide))
  .style({ scrollDirection: "horizontal", snap: "start" })
  .onLayout(({ width }) => { if (width > 0) step = width })
  .onScroll(x => { page.value = Math.max(0, Math.min(slides.length - 1, Math.round(x / step))) })
```

`page` обновляется непрерывно во время драга — он переключается, когда слайд проходит середину, что и должен делать индикатор-точки. Колбэка «устаканился на странице n» нет; `onScrollRelease` срабатывает при отрыве пальца, до конца снап-анимации.

Карусель карточек с выглядывающими соседями использует более узких детей и `snap: "center"`. Цели снапа — прямые дети, поэтому карточки могут быть разной ширины (счётчик по неровным детям требует их смещений, а не одного `step`).

`start` и `end` учитывают собственный паддинг scrollable: при `paddingHorizontal: 20` ребёнок останавливается у 20px-жёлоба, а не вплотную к краю экрана — так позиции снапа согласуются с естественной позицией покоя контейнера. `center` не зависит от паддинга по построению.

> **NOTE**
>
> snap доводит только жесты; хосты без поддержки деградируют до свободной прокрутки (web-lite — полностью, wasm-вьюер — пока никак). Комбинируй с `onScroll` для индекса — программного `scrollTo` на `UIScrollable` по-прежнему нет (см. заметку выше), поэтому навигация тапом по точкам требует `UIVirtualizedList` или ждёт закрытия этого пробела.

Перетаскиваемым детям внутри скроллера нужно `claim` их направление жеста, иначе прокрутка украдёт указатель — см. touch.md.

UISpacer

```ts
UISpacer(): UISpacer      // без детей; его поверхность стиля — только ElementStyle — без фона
```

Гибкое пустое место: по умолчанию `flexGrow: 1`, съедает свободное место вдоль главной оси родителя. Тянись к нему, только когда обычное выравнивание не выражает раскладку — один элемент, толкнутый к дальнему краю, пока остальные стоят:

```ts
UIRow(title, UISpacer(), closeButton)
```

Если *все* дети двигаются вместе, `justifyContent` (`"space-between"`, `"flex-end"`, …) делает ту же работу без лишнего элемента.

Фаст-пасы компилятора — `__UIColumn` и компания

`__UIColumn`, `__UIRow`, `__UIBox`, `__UIButton`, `__UIScreen`, `__UIScrollable`, `__UIWidget` — пути конструирования без диспетчеризации, в которые проход chisel `flatten_ui` опускает вызовы фабрик, когда доказал, что аргументы — обычные дети (напр. `UIColumn(a, b)` → `__UIColumn([a, b])`). Это вывод компилятора — **никогда не вызывай их руками**; API — публичные фабрики.

Смотрите также

- Модель элементов UI — дети, условный рендер, ссылки.
- Стилизация — словарь стиля контейнера (`gap`, `alignItems`, …).
- Экраны и навигация — экраны никогда не прокручиваются; фиксированный хром + прокручиваемое тело.
- Виртуализированные списки — оконный рендер + программная прокрутка.

4.2.6 Элементы контента UI

Листовые элементы, которые что-то отображают: `UIText`, `UIImage`, `UIVideo`. У каждого одно изменяемое свойство контента (`.text`, `.src`, `.player`), живущее на элементе — контент никогда не стиль. Для редактируемого текста см. `UIInput` в interactive.md.

Обзор

```ts
import photo from './photo.jpg'

const caption = UIText("Loading…").style({ color: "white", fontSize: 16 })

const screen = UIScreen(
  UIImage(photo).style({ width: "100%", height: 220, borderRadius: 16, objectFit: "cover" }),
  caption,
).style({ bgColor: "black", p: 16, pt: "safe-top", gap: 12 })

screen.open()
caption.text = "Sunset over the harbor"     // перерисовывается сразу
```

UIText

```ts
UIText(text: string): UIText

text.text = "Updated"       // get/set отображаемой строки
```

Специфичные для текста стили (поверх общих стилей элемента):

```ts
textAlign:      "start" | "center" | "end" | "left" | "right"   // по умолчанию left
fontFamily:     string            // системные: "serif", "sans-serif", "monospaced";
кастомные через font() (см. fonts.md)
fontSize:       UIValue           // по умолчанию 14
lineHeight:     UIValue | "normal"
fontWeight:     number | "normal" | "bold"     // 400, 700, …
fontStyle:      "normal" | "italic"
color:          Color
textDecoration: "underline" | "line-through" | "none"
letterSpacing:  UIValue
lineClamp:      number            // обрезать до N строк; 0 / не задано = без ограничения
textOverflow:   "ellipsis" | "clip"            // по умолчанию ellipsis
```

> **NOTE**
>
> `lineHeight: 1.5` значит 1.5 **px** — голое число это px везде. Множитель это `"1.5em"` (разрешается относительно собственного `fontSize` этого элемента; наследования нет).

Обрезка текста до N строк

`lineClamp` ограничивает текст N строками, и элемент **меряется** в эту высоту — окружающая раскладка перетекает вокруг обрезанного бокса, а не вокруг полной строки:

```ts
UIText(article.summary).style({ lineClamp: 3 })                       // 3 строки, затем "…"
UIText(article.summary).style({ lineClamp: 3, textOverflow: "clip" })  // 3 строки, жёсткий срез
```

`textOverflow` действует только вместе с `lineClamp` — без ограничения бокс размеряется под всю строку, так что ничего не переполняется. `"clip"` на вебе приблизителен: он режет по боксу line-height, а не по глифу, поэтому нижний выносной элемент последней строки может срезаться не так, как на iOS. `"ellipsis"` точен на обоих.

> **NOTE**
> фон экрана по умолчанию чёрный — всегда задавай `color` явно.

`.style()` у `UIText` не принимает свойств рамки/фонового-изображения — `bgColor`, отступы и поля работают, но декорированный текстовый чип — это `UIRow`/`UIColumn` вокруг `UIText`. Кастомные шрифты нужно зарегистрировать до открытия экрана — см. fonts.md.

UIImage

```ts
UIImage(src: ImageSource): UIImage

// ImageSource:
//   string          — удалённый URL, или asset() / импортированный файл проекта
//   FetchResponse   — скачанное тело, используемое напрямую как пиксели
//   File            — из openFilePicker()
//   SvgSource       — обёрнутый сырой SVG XML
//   Canvas          — 2D Canvas, запекается в текстуру при присвоении

img.src            // получить (разрешённый) источник / задать новый — обновляет отображаемое изображение
```

Специфичные для изображения стили:

```ts
objectFit:    "cover" | "contain" | "fill"   // как источник ложится в бокс
tintColor:    Color                          // только SVG-источники
borderRadius: number                         // px (только число на UIImage)
```

> **NOTE**
>
> `tintColor` применяется **только к SVG-источникам** и заменяет **все** цвета `fill` и `stroke` в SVG — это для монохромных иконок, не для многоцветной графики.

> **NOTE**
>
> голая строка относительного пути (`UIImage("./photo.png")`) не разрешается в собранный ассет — обычные строки работают только для `https://…` URL. Импортируй файл или используй `asset('./photo.png')`.

Иконки: ***`assetIcon("pack:name")`***

Для иконок предпочитай компайл-макрос `assetIcon()` ручному написанию SVG: он разрешает одну иконку из реестра иконок во время **сборки** и инлайнит её как источник изображения (та же форма, что возвращает `SvgSource`), поэтому она рендерится офлайн и одинаково на каждом хосте — без сети в момент рендера.

```ts
UIImage(assetIcon("lucide:bell")).style({ width: 24, height: 24, tintColor: "#8a8f98" })
UIImage(assetIcon("lucide:check", { color: colors.accent })).style({ width: 20, height: 20 })
```

Id `"pack:name"` должен быть **строковым литералом** (неизвестный id — ошибка компиляции; паки смотри на https://icon-registry.jt3.ru). Перекрашивай опцией `{ color }` — hex-**литерал** запекается в SVG при компиляции, токен/выражение применяется как tint — или стилевым свойством `tintColor`.

Обрезка атласа: ***`setSourceRect`***

```ts
img.setSourceRect(x: number, y: number, w: number, h: number): this   // пиксели текстуры
```

Рендерит только под-прямоугольник источника — примитив спрайт-листа. `w`/`h` прямоугольника становятся собственным размером элемента (один кадр, не весь атлас), и обрезанный кадр всегда заполняет бокс, переопределяя `objectFit`. Чейнится и безопасно вызывается до появления элемента на экране (начальная обрезка применяется при монтировании). Анимация в стиле спрайта — это смена прямоугольника на кадр:

```ts
const icon = UIImage(atlasUrl).style({ width: 64, height: 64 }).setSourceRect(0, 0, 128, 128)
let frame = 0
setInterval(() => { icon.setSourceRect((++frame % 8) * 128, 0, 128, 128) }, 100)
```

Ловушка: `bgImage` — не контент изображения

```ts
const img = UIImage("")            // ✗ пустой источник как заглушка
img.style.bgImage = url            // ✗ bgImage это декор контейнера, не контент
const img = UIImage(url)           // ✓ источник идёт в конструктор…
img.src = newUrl                   // ✓ …и меняется через .src
```

UIVideo

```ts
UIVideo(player: VideoPlayer): UIVideo


video.player       // только чтение — VideoPlayer, переданный при конструировании
```

Элемент — это лишь экранная поверхность; воспроизведение живёт целиком на `VideoPlayer` — создай его первым, управляй напрямую:

```ts
const player = new VideoPlayer(asset('./intro.mp4'))
const video = UIVideo(player).style({ width: "100%", height: 220, borderRadius: 12,
objectFit: "cover" })
player.play()
```

Специфичный для видео стиль:

```ts
objectFit: "cover" | "contain" | "fill"
```

`UIVideo` также принимает рисуемые стили (`border*`, `bgColor`, …) для обрамления. Плеер — нативный ресурс — вызови `dispose()`, когда видео ушло навсегда, и останови воспроизведение в `onClose` экрана (см. Соглашения — Жизненный цикл).

Смотрите также

- Модель элементов UI — свойства контента против стилей, захват ссылок.
- Стилизация — общие стили, единицы, разрешение `em`.
- Шрифты — `registerFont` для кастомного `fontFamily`.
- Медиаплееры — поверхность управления `VideoPlayer`.
- Интерактивные элементы — `UIInput` / `UITextArea` для редактируемого текста.

4.2.7 Кнопки и поля ввода

Три элемента, принимающие ввод пользователя. `UIButton` — единственный *тапабельный* контейнер — обработчики касаний существуют только на `UIButton`, `UIScreen` и `UIWidget` (см.

События указателя); чтобы сделать что-либо кликабельным (карточку, строку списка, иконку), оберни это в `UIButton`. `UIInput` — однострочное текстовое поле, `UITextArea` — его многострочный собрат.

Обзор

```ts
let input: UIInput

const form = UIColumn(
  input = UIInput()
    .style({ height: 40, px: 12, borderRadius: 8, bgColor: "white", color: "black",
             placeholder: "Your name", placeholderColor: "#999" })
    .onChange(v => console.log("typing:", v)),
  UIButton(UIText("Submit").style({ color: "white" }))
    .style({ height: 40, justifyContent: "center", bgColor: "#FF4032", borderRadius: 8,
             onPressed: { opacity: 0.7 }, rippleColor: "default" })
    .onClick(() => console.log("submitted:", input.value)),
).style({ gap: 8 })
```

UIButton

```ts
UIButton(...children: (UINodeChild | UINodeChild[])[]): UIButton
```

Контейнер с дефолтом `flexDirection: "row"` и детьми, центрированными по обеим осям (`justifyContent` + `alignItems` `"center"`) — кнопка «иконка+подпись» раскладывается правильно без стилизации. Полная поверхность контейнера: `.append` / `.insert` / `.remove` / `.setContent`, `.style`, `.animateTo` / `.animateFrom`, `.onLayout`. Используй `flexDirection: "column"` для кнопки в форме карточки.

```ts
button.onClick(ev => …)        // отпускание над кнопкой; ev: ClickEvent
button.onTouchStart(ev => …)   // нажатие; ev.track({...}) начинает жест перетаскивания
button.onLongPress(ev => …)    // палец удержан ~0.5 с; ev.track({...}) для перетаскивания;
ev: LongPressEvent
button.isPressed(): boolean    // true, пока палец сейчас на кнопке
```

События `onClick` / `onTouchStart` / `onLongPress` несут `clientX` / `clientY` / `pointerId` (логические px). Перетаскивания — `ev.track()`, `claim` — работают ровно как описано в События указателя.

`onLongPress` срабатывает, когда палец удерживается на кнопке дольше порога долгого нажатия (~0.5 с), не соскальзывая. Обработанное долгое нажатие **проглатывает клик**, который иначе последовал бы за отпусканием, — так что `onClick` и `onLongPress` чисто

сосуществуют. Как и у `onTouchStart`, событие может `ev.track({...})` — так что «зажми, удерживай, затем тащи» — это один обработчик:

```ts
button.onLongPress(ev => {
  device.vibrate("medium")                // подтверждаем, что удержание засчитано
  ev.track({
    claim: true,
    onMove: ({ deltaX, deltaY }) => moveElement(deltaX, deltaY),
    onEnd:  () => commit(),
  })
})
```

> **NOTE**
> кнопки не рисуют собственного оформления — стилизуй их как любой контейнер (`bgColor`, `borderRadius`, отступы). Дай кнопке явный `height`: сама по себе она высотой только со свой текст.

Отклик на нажатие — по желанию

```ts
button.style({
  onPressed: { opacity: 0.7, bgColor: "#c22", duration: 150 },  // стиль во время нажатия
  rippleColor: "default",                                       // Android-ripple; или любой Color
})
```

Визуально при нажатии ничего не происходит, пока не попросишь. `onPressed` принимает рисуемые/базовые стили (`bgColor`, `opacity`, `border*`, `transform`, …), применяемые, пока палец внизу, плюс необязательный переход `duration` (мс). `rippleColor` — **только Android** и *заменяет* там визуал `onPressed` — так что установка обоих даёт ripple на Android и стиль `onPressed` на iOS.

В новом коде предпочитай зарезервированный класс **`$pressed`**: тот же отклик на самой кнопке, плюс он *каскадирует* — дети (иконка, подпись) могут объявить собственные блоки `$pressed` и реагировать на нажатие кнопки. `onPressed` остаётся строго поэлементной формой (он никогда не срабатывает от нажатия предка — то, что нужно для вложенных интерактивов) и побеждает `$pressed`, когда объявлены оба. См. Классы стилей.

Для стойкого визуального состояния, которое переключаешь сам (выбрано, отмечено, активно), а не управляемого системой, объяви **класс стиля** (`$name`) и переключай его через `el.class.name = …` — см. Классы стилей. Отклик на нажатие всегда бьёт класс, пока нажато.

UIInput и UITextArea

```ts
UIInput(): UIInput          // однострочное поле
UITextArea(): UITextArea    // многострочное
```

```ts
input.value                          // get/set текущего текста (свойство контента, не стиль)
input.onChange(cb: (value: string) => void): this   // каждое изменение
input.onFocus(cb: () => void): this
input.onBlur(cb: () => void): this
input.onSubmit(cb: (value: string) => void): this   // нажата клавиша return (только UIInput)
input.focus()                        // программный фокус (открывает клавиатуру); no-op до монтирования
input.blur()                         // снять фокус (прячет клавиатуру)
```

Предзаполнение поля

`value` можно писать **до открытия экрана** — так строится экран настроек или форма редактирования: заполни поля, пока собираешь дерево, затем открой его.

```ts
const name = UIInput().style({ placeholder: "Name" })
const bio = UITextArea().style({ maxHeight: 120 })

name.value = user.name           // задано до открытия — поле поднимется заполненным
bio.value = user.bio             // textarea поднимется уже выросшей под контент

UIScreen(UIColumn(name, bio, UIButton("Save").onClick(() => save(name.value, bio.value)))).open()
```

Чтение возвращает живой текст, пока экран открыт, и последнее заданное тобой значение, пока нет. Набранное **пользователем** не переносится через закрытие/открытие — переоткрытый экран строится заново и поднимается со значениями, заданными твоим кодом. Используй `screen.keepAlive()`, когда полузаполненная форма должна пережить уход навигацией.

Специфичные для инпута стили, поверх обычных стилей элемента + текста (`fontSize`, `color`, `fontFamily`, …):

TS

```
input.style({
  placeholder: "Search…",
  placeholderColor: "#999",
  type: "search",         // "text" (по умолчанию) | "password" | "search" | "phone" |
"email"
                          //   | "number" | "decimal" | "url" | "date" | "time"
  enterKey: "search",     // подпись клавиши return: "done" | "go" | "next" | "search" |
"send" (только UIInput)
  maxLength: 32,          // жёсткий предел, хост навязывает его при наборе/вставке
  autocapitalize: "none", // "none" | "words" | "sentences" | "characters"; не задано =
дефолт платформы
  autocorrect: false,     // платформенная автокоррекция/подсказки; не задано = дефолт
платформы
  onFocused: { borderColor: "#FF4032", duration: 150 },   // стиль во время фокуса (как
onPressed;
                                                          // каскадная форма: $focused)
})
```

> **WARNING**
>
> значения `type` — **подсказки клавиатуры, а не валидаторы** — `number` показывает цифровую клавиатуру, но не блокирует вставленные буквы; валидируй в `onChange` сам. Значение телефонной клавиатуры — **`phone`** (в LeCodes нет HTML-подобного `tel`).

> **NOTE**
>
> дай инпутам явный `height` и `flexGrow: 1` (в строке) или `width` — инпут схлопывается до ширины своего плейсхолдера и высоты своего текста, он не растягивается как веб-`<input>`.

Пикеры даты и времени (`type: "date" | "time"`)

`date` и `time` — **виды пикеров, а не клавиатуры**: фокус на поле открывает нативный пикер платформы в слоте клавиатуры (колёса iOS, календарь браузера), а свободный набор отключён — значение приходит только из пикера или из кода. Само поле остаётся обычным, полностью стилизуемым инпутом.

- `input.value` всегда **каноничен**: `"YYYY-MM-DD"` для `date`, `"HH:MM"` (24-часовой формат) для `time` — это получает `onChange` и это же задаёшь программно. То, что *пользователь* видит в поле, — локализованная строка, которую форматирует хост; никогда не парси отображение.
- Пустое значение показывает `placeholder`. На iOS подтверждение панелью **Done** принимает текущую позицию колёс, даже если пользователь их не крутил (открыл → Done = сегодня); тап по пустому месту просто закрывает без принятия.

- Вся обычная механика работает без изменений: стилизация `onFocused`, `focus()`/`blur()`, `keyboardShrink`, доктрина скрытия клавиатуры.

```TS
const birthday = UIInput().style({ type: "date", placeholder: "Date of birth" })
birthday.onChange((v) => console.log(v))   // "1990-04-27"
birthday.value = "1990-04-27"              // канонично на входе, локализованное отображение на выходе
```

Авторост (UITextArea)

`UITextArea` **без явного** `height` **измеряет собственный контент** и растёт построчно по мере набора. Ограничь диапазон через `minHeight` / `maxHeight` — за `maxHeight` текст прокручивается внутри поля. Фиксированный `height` отключает авторост (внутренняя прокрутка с самого начала). Это рецепт композера чата:

```TS
const composer = UITextArea().style({
  placeholder: "Message", fontSize: 16,
  maxHeight: 108,          // ~5 строк, дальше прокрутка внутри
  keyboardDismiss: false,  // чат: тап по переписке не прячет клавиатуру
  // без height — растёт с контентом; ширина приходит от растяжения обёртки
})
UIRow(
  UIColumn(composer).style({ flexGrow: 1, flexShrink: 1, flexBase: 0, px: 14, py: 11, borderRadius: 22 }),
  sendButton,
).style({ alignItems: "flex-end", gap: 10 })   // кнопка отправки остаётся на якоре, пока поле растёт
```

Не давай самой textarea `flexGrow`/`flexBase` — flex-базис переопределяет внутреннюю высоту контента, и поле перестаёт расти.

Отправка и поток формы

`onSubmit` срабатывает, когда пользователь нажимает клавишу return на клавиатуре (**только** `UIInput` — в `UITextArea` Enter — это перевод строки, и точка). При отправке клавиатура прячется, **кроме** `enterKey: "next"` — тогда фокус должен перенести ты сам, и клавиатура остаётся:

```TS
const name = UIInput().style({ enterKey: "next" }).onSubmit(() => email.focus())
const email = UIInput().style({ type: "email", enterKey: "done" })
  .onSubmit(() => submitForm())
```

`focus()` также покрывает экраны поиска с фокусом при открытии и валидацию в духе «сфокусируй невалидное поле».

Политика клавиатуры: ***`keyboardShrink`***

```ts
input.style({ keyboardShrink: false })   // клавиатура накрывает UI — без релэйаута
```

Что делает раскладка, пока клавиатура поднята, решает **сфокусированный инпут**. `true` (по умолчанию): вьюпорт раскладки сжимается до области над клавиатурой (один релэйаут). `false`: клавиатура накрывает UI без пересчёта раскладки — для композеров чата, управляющих своим отступом самостоятельно, и полноэкранных канвасов, где рефлоу хуже перекрытия. В режиме оверлея читай `app.keyboardHeight` / событие `"keyboard"`, чтобы освободить место самому:

```ts
// композер чата: режим оверлея + свой отступ — pb едет на клавиатуре
input.style({ keyboardShrink: false })
app.addEventListener("keyboard", (height) => {
  composerRow.style.pb = Math.max(24, height)    // 24 ≈ нижняя безопасная зона
})
```

Всё остальное про клавиатуру — **работа хоста**: он прокручивает сфокусированный инпут в видимую область внутри ближайшего скролл-контейнера, а скрытие следует одной универсальной доктрине:

- **Тап по контролу никогда не прячет.** Тап по `UIButton` срабатывает с поднятой клавиатурой; тап по другому инпуту просто переносит фокус. (Кнопка отправки рядом с композером работает без мигания клавиатуры — вызови `input.blur()` из обработчика, когда прятать *нужно*.)
- **Прокрутка никогда не прячет** — если скроллируемый сам не попросил. Так что выпадающие подсказки или список автокомплита под сфокусированным инпутом можно прокручивать и тапать его пункты с поднятой клавиатурой — без спецобработки. На iOS перетаскивание вниз, дошедшее до самой клавиатуры, прячет её интерактивно (жест iMessage); обычная прокрутка — нет. Скроллируемый может сменить собственную политику стилем `keyboardDismissMode` (containers.md): `"scroll"` заставляет любой драг в нём прятать сразу при старте — правильное ощущение для списка результатов поиска — а `"none"` выключает даже интерактивный драг.
- **Тап по неинтерактивной области прячет** — фон, текст, картинки. Это единственная часть, от которой инпут может отказаться через `keyboardDismiss: false` (композеры чата — тогда тапы не прячут никогда, и остаются только `blur()`, клавиша return и жест перетаскивания вниз на iOS).
- Клавиша return прячет согласно `enterKey` (кроме `"next"`), а iOS показывает панель Done над клавиатурами без клавиши return (`number` / `decimal` / `phone`). `input.blur()` остаётся

программным путём.

Обычно приложение не содержит **никакого кода про клавиатуру**, кроме цепочки `enterKey: "next"` → `next.focus()` — плюс `keyboardDismiss: false` на композере чата.

Подводные камни

```ts
// ✗ обработчик касания на обычном контейнере
UIColumn(...).onTouchStart(cb)              // нет такого метода — только
UIButton/UIScreen/UIWidget
// ✓ оберни в UIButton
UIButton(...).onTouchStart(cb)

// ✗ кнопка рядом с инпутом 40px сжимается до высоты своего текста
UIRow(input.style({ height: 40 }), UIButton(icon))
// ✓ дай контролам одинаковый явный height
UIRow(input.style({ height: 40 }), UIButton(icon).style({ height: 40 }))
```

Смотрите также

- События указателя и жесты — объекты событий, `ev.track()`, `claim`.
- Стилизация — `.style()`, состояния стиля, `animateTo`.
- Контейнеры — правила раскладки `UIRow` / `UIColumn`, которые наследует кнопка.

4.2.8 Экраны и Router

`UIScreen` — корень каждого UI-дерева — он всегда заполняет устройство и ведёт себя как `UIColumn`. Экран никогда не прокручивается; прокрутка живёт в ребёнке (см. ниже). Одноэкранное приложение вызывает `screen.open()`; многостраничное отдаёт свои экраны `Router` и навигирует через `push` / `pop` / `replace`.

Обзор

```ts
const detail = UIScreen(
  UIText("Detail").style({ fontSize: 24, fontWeight: 700, color: "white" }),
).style({ bgColor: "black", p: 20, pt: "max(safe-top, 24px)" })
  .onBackPressed(() => Router.pop())

const home = UIScreen(
  UIText("Home").style({ fontSize: 24, fontWeight: 700, color: "white" }),
  UIButton(UIText("Open detail").style({ color: "white" }))
    .style({ bgColor: "#FF4032", borderRadius: 12, p: 16 })
    .onClick(() => Router.push(detail)),
).style({ bgColor: "black", p: 20, pt: "max(safe-top, 24px)", gap: 16 })

Router.init(home)
```

UIScreen

```ts
UIScreen(...children: (UINodeChild | UINodeChild[])[]): UIScreen

screen.open(): void     // показать этот экран напрямую (одноэкранные приложения; прячет активный Router)
screen.close(): void    // закрыть напрямую открытый экран
```

Полная поверхность контейнера: `.append` / `.insert` / `.remove` / `.setContent`, `.style`, `.animateTo` / `.animateFrom`, `.onLayout`.

> **NOTE**
> экран всегда заполняет устройство — стили размера на нём (`width`, `height`, `flexGrow`, `flexShrink`, `position`) — no-op. Стилизуй его отступы, фон и раскладку детей. Фон по умолчанию чёрный; всегда задавай `bgColor` и цвета текста явно.

Жизненный цикл — onOpen / onClose

```ts
screen.onOpen(cb: () => void): this    // экран стал активным
screen.onClose(cb: () => void): this   // экран перестал быть активным
```

Они срабатывают на *каждую* активацию, не только на первую: уход с экрана вызывает его `onClose`, возврат к нему — снова `onOpen`. Всё запущенное в `onOpen` должно останавливаться в `onClose` — циклы, интервалы, сокеты — иначе продолжит работать за другими экранами. См. Жизненный цикл.

```ts
let loopId: number | null = null
screen
  .onOpen(() => { loopId = setLoop(dt => tick(dt)) })
  .onClose(() => { if (loopId !== null) { clearLoop(loopId); loopId = null } })
```

Касания и кнопка «назад»

```ts
screen.onTouchStart(ev => …)          // срабатывает на касания где угодно на экране (полноэкранные жесты)
screen.onBackPressed(cb: () => void)  // аппаратный/жестовый «назад» Android — обычно Router.pop()
```

`onTouchStart` поддерживает `ev.track()` как кнопка — см. События указателя.

Экраны никогда не прокручиваются

Экран — **фиксированный корень раскладки**: у него нет позиции прокрутки, нет скролл-событий, нет pull-to-refresh. Прокрутка всегда живёт в ребёнке. Канонический экран — фиксированный хром (шапка, таб-бар) плюс **одно прокручиваемое тело** с `flexGrow: 1`:

```ts
UIScreen(
  Header("Profile", BackButton()),                // фиксированный хром — стоит на месте
  UIScrollable(content)                        // ОДНО прокручиваемое тело
    .style({ flexGrow: 1, px: 24, gap: 16 })
    .onRefresh(async () => { await load() }),     // pull-to-refresh живёт на скроллируемом
).style({ bgColor: "black", pt: "safe-top" })
```

Скролл-события (`onScroll` / `onScrollRelease` / `onOverscroll`), pull-to-refresh (`onRefresh`) и стиль `refreshControlColor` принадлежат `UIScrollable` — и `UIVirtualizedList` для длинных данных.

Router

```ts
Router.init(homePage, opts?: { showDefaultBackButton?: boolean })  // вызови один раз в файле входа
Router.push(screen)                    // положить в стек, экран становится активным
Router.pop(to?: number)                // по умолчанию -1 = на экран назад
Router.replace(screen, opts?: { transition })   // подменить верхний экран
Router.current                         // вершина стека роутера (геттер)
Router.hide()                          // спрятать весь роутер (стек остаётся живым)
Router.restore()                       // вернуть его, реактивировав верхний экран

Presentable.current                    // то, что реально на экране прямо сейчас (или null)
```

`Router.current` — вершина стека — он остаётся осмысленным, даже пока роутер приостановлен прямым `open()`. `Presentable.current` — реально видимый сейчас пункт назначения (`UIScreen`, `Scene`, …, или `null` до первого открытия) — к нему прикрепляется `UIWidget.show()`.

- `pop(to)` — отрицательные значения относительны (`-2` = на два экрана назад); `0` и положительные значения — абсолютный индекс в стеке (`0` = домашний экран). Экраны, покидающие стек, уничтожаются.
- Переходы `replace`: `"slide-from-left" | "slide-from-right" | "slide-from-top" | "slide-from-bottom" | "zoom" | "zoom-out" | "zoom-in" | "fade" | "none"` — по умолчанию `"none"`, темируется на всё приложение: `theme({ replaceTransition: "fade" })` (любое имя или кастомная спецификация; явный `transition` в вызове всё равно побеждает, `null` сбрасывает — см. theme.md).
- `showDefaultBackButton` у `init` (по умолчанию `false`) показывает предоставленную хостом кнопку «назад».

Как экраны живут в стеке

Экраны *ниже* верхнего остаются смонтированными — их деревья элементов и состояние выживают, они просто не рендерятся. Навигация запускает колбэки жизненного цикла: `push` вызывает `onClose` уходящего экрана и `onOpen` входящего; `pop` вызывает их в обратном порядке, уничтожает нативные деревья снятых экранов и снова вызывает `onOpen` на экране, куда ты приземляешься. Снятый объект `UIScreen` всё ещё пригоден — повторный `push` его перемонтирует. `Router.hide()` вызывает `onClose` верхнего экрана; `restore()` снова вызывает `onOpen`. Вызов `screen.open()`, пока роутер активен, прячет роутер (его стек остаётся восстановимым).

События смены маршрута

```ts
Router.addEventListener("change", (screen: UIScreen) => …)   // срабатывает после каждой навигации
Router.removeEventListener("change", cb)
```

Колбэк получает вновь активный экран — полезно для общего таб-бара или аналитики.

Подводные камни

```ts
// ✗ скролл-методы на экране — экраны никогда не прокручиваются
UIScreen(...).onRefresh(load)       // метода нет на экране
// ✓ прокрутка и pull-to-refresh живут на теле UIScrollable
UIScreen(UIScrollable(...).style({ flexGrow: 1 }).onRefresh(load))

// ✗ цикл, запущенный в onOpen, никогда не остановлен — продолжает работать после Router.push()
screen.onOpen(() => setLoop(update))
// ✓ спаруй его в onClose (срабатывает на каждый уход навигацией)
```

Смотрите также

- UIWidget — глобальные оверлеи над всем (`show()`) или прикреплённые к странице (`attachTo(owner)`).
- Контейнеры — правила раскладки `UIColumn`, `UIScrollable`.
- Соглашения § Жизненный цикл.
- События указателя и жесты.

4.2.9 UIPager

`UIPager` — единственный элемент навигации по экранам: **вкладки-сиблинги, нативно свайпающиеся в стороны, каждая со своим стеком навигации**. Передай массив экранов — они встанут рядом друг с другом; `push` открывает экран поверх *текущей* вкладки (внутри пейджера), а нативный краевой свайп назад (или `pop`) его разматывает. Он мапится на нативные контейнеры каждой платформы (iOS `UIPageViewController` + `UINavigationController` на вкладку), так что свайпы, пуши и жест «назад» — настоящие, а не переизобретённые.

```ts
const pager = UIPager(HomeScreen(), SearchScreen(), ProfileScreen())

// позже, откуда угодно внутри хостимого экрана — без проброса ссылки:
UIPager.push(PostScreen(id))   // въезжает поверх текущей вкладки
UIPager.pop()
```

Пейджер с **одной** вкладкой — это просто стек навигации: отдельного элемента-стека нет. Хосты понимают это буквально: с одной вкладкой iOS вовсе пропускает пейджинговый контейнер и встраивает только `UINavigationController`, так что накладных расходов против выделенного компонента-стека — ноль.

UITabs — стандартная оболочка с нижними вкладками

Для типовой формы — пейджер + нижняя панель — используй `UITabs` вместо ручной сборки. Это чистая SDK-композиция (`UIScreen` из `[ UIPager (flexGrow: 1), темированная панель ]`, ноль нового ABI) со встроенными синхронизацией выбора, подсветкой активной вкладки, отступом под безопасную зону и бейджами. Ключи — id вкладок, в порядке вкладок; кнопки называются `tab-<id>` (flow-тесты могут тапать их по имени):

```ts
const tabs = UITabs({
  home:    { label: "Home",    icon: assetIcon("lucide:house"), screen: homeScreen },
  profile: { label: "Profile", icon: assetIcon("lucide:user"),  screen: profileScreen },
})
Router.init(tabs)                 // UITabs И ЕСТЬ UIScreen / Presentable

tabs.select("profile")            // переключить вкладку (мгновенно; `true` — слайд)
tabs.tab                          // id активной вкладки (геттер)
tabs.onSelect((id, i) => ...)     // тап по панели, свайп или select()
tabs.badge("home", 3)             // пилюля со счётчиком; `true` = точка; false/null/0
очищает
tabs.pager                        // пейджер под панелью (push/pop/depth)
```

Панель стилизует себя через переменные темы, с фолбэками, чтобы нетемированное приложение всё равно выглядело правильно: активное `var(--primaryColor)`, неактивное `var(--mutedColor)`, поверхность `var(--tabbarBg)`, волосяная линия `var(--tabbarBorder)`, бейдж `var(--badgeColor)` — один вызов `theme({ primaryColor, tabbarBg })` перестилизует её на всё приложение (те же переменные читает панель `defineTabs` дизайн-скаффолда). Активная кнопка также несёт класс стиля `active` (каскадирует на её контент). Для более тонкого контроля перестилизуй `tabs.bar`; для полностью кастомной панели спустись к `UIPager` + собственной строке.

Навигация внутри вкладки не меняется: `UIPager.push(detail)` с любого экрана сохраняет панель; `Router.push(...)` открывается над всей оболочкой (панель закрыта).

Конструирование

```ts
UIPager(root)                          // одна вкладка = простой стек с корнем `root`
UIPager(a, b, c)                       // три вкладки рядом (выбрана a)
// Вкладки вариадичны, как дети контейнера: аргумент-массив расплющивается, falsy-элементы
// пропускаются — UIPager(home, isAdmin && adminTab) работает. Стили — через чейнящийся
.style({...}).
```

Дети — **только `UIScreen`ы** — каждая страница несёт полный контракт экрана (`onOpen`/`onClose`, `onBackPressed`) и раскладывается в бокс пейджера. Чтобы хостить простой элемент, оберни его: `UIPager(UIScreen(el))`. Вкладки **фиксируются при конструировании** — динамические части — это выбор вкладки и стек каждой из них.

Пейджер заполняет свой слот как любой элемент — дай ему `flexGrow: 1` (или размер), чтобы он не схлопнулся (`flexShrink: 1` уже по умолчанию, как у `UIScrollable`, так что он оставляет место таб-бару):

```ts
UIScreen(
  UIPager(HomeTab(), ProfileTab()).style({ flexGrow: 1 }),
  tabBar,
)
```

Вкладки

```ts
pager.select(2)                  // мгновенное переключение (no-op вне диапазона)
pager.select(2, true)            // анимированный слайд с учётом направления
pager.index                      // выбранная вкладка (геттер)
pager.onSelect(i => { ... })     // срабатывает на устоявшийся свайп или select()
```

`select` **мгновенен по умолчанию** — платформенная конвенция для таб-баров, где вкладки — параллельные режимы, а не последовательность слева направо. Передай `animated: true` для скользящего переключения top-tab-интерфейсов (ощущение Material). Свайп всегда анимирован — это физическое перетаскивание.

Вкладки хранят свои стеки: свайпни прочь с вкладки глубиной в три экрана, свайпни назад — и она ровно там, где ты её оставил. Корни вкладок живут постоянно — переключение вкладок никогда их не пересобирает.

Свайп между вкладками работает только на корне вкладки. Как только текущая вкладка углублена (`depth > 1`), горизонтальный жест принадлежит краевому свайпу назад; пейджинг

снова включается, когда вкладка вернулась к корню. Это нативный арбитраж — настраивать нечего.

Стек (на вкладку)

```ts
pager.push(screen)     // въезжает новой вершиной на ТЕКУЩУЮ вкладку; свайп назад её снимает
pager.pop()            // снять вершину текущей вкладки (no-op на корне вкладки)
pager.popToRoot()      // размотать текущую вкладку до корня
pager.replace(screen)  // подменить вершину текущей вкладки (на глубине 1 подменяет корень вкладки)

pager.stack            // UIScreen[] текущей вкладки, корень → вершина (геттер)
pager.depth            // 1 = только корень (геттер)
pager.onChange(depth => { ... })   // срабатывает на push, pop, replace или свайп назад
```

Опции перехода на каждый push нет — это фиксированный нативный push/pop, ровно то, что позволяет жесту «назад» обратимо тянуть его.

Доступ откуда угодно — `UIPager.current`

`UIPager` — глобал, так что ссылку в дочерние экраны не пробрасываешь. Амбиентные вызовы действуют на **пейджер, владеющий видимым сейчас экраном**:

```ts
UIPager.current        // пейджер, чей экран видим, или null
UIPager.push(screen)   // → current.push(screen)
UIPager.pop()          // → current.pop()
UIPager.popToRoot()
```

```ts
const HomeTab = () => UIScreen(
  row("Open the first post", () => UIPager.push(PostScreen(posts[0]))),
)
```

С пейджерами, вложенными в страницы, `UIPager.current` разрешается во **внутреннейший** — `push` изнутри страницы попадает в пейджер, на который пользователь реально смотрит. Если у видимого экрана нет объемлющего пейджера, `current` — `null`, и амбиентные вызовы — no-op с предупреждением.

Таб-бары

`UITabs` даёт стандартную нижнюю панель. Для кастомной панели поверх сырого пейджера — собери свою и веди её в обе стороны:

```ts
const pager = UIPager(HomeTab(), ProfileTab())
const tabBar = UIRow(
  UIButton(UIText("Home")).onClick(() => pager.select(0)),
  UIButton(UIText("Profile")).onClick(() => pager.select(1)),
)
pager.onSelect(i => highlight(tabBar, i))   // свайпы тоже двигают подсветку

Router.init(UIScreen(pager.style({ flexGrow: 1 }), tabBar))
```

Кнопка «назад» (Android) и свайп назад

Интерактивный краевой свайп назад снимает верх текущей вкладки автоматически. Для аппаратной/программной **кнопки «назад»** сначала сними верх текущей вкладки, затем провались в собственную обработку:

```ts
screen.onBackPressed(() => {
  if (UIPager.current && UIPager.current.depth > 1) {
    UIPager.current.pop()
    return
  }
  // ...твоё собственное поведение «назад» (выход, подтверждение и т.д.)
})
```

Связь с `Router`

`Router` навигирует между цельноэкранными пунктами назначения и кросс-типовыми переходами (экран ↔ сцена ↔ нативная вью ↔ видео) на корне приложения. `UIPager` — навигационный *регион внутри* экрана. Они сосуществуют, и выбор из любой области — это просто глагол: `UIPager.push` открывает **внутри** региона (панели остаются), `Router.push` открывает **поверх** всего (накрывает весь экран, в котором живёт пейджер). См. screen-router.md. `UIPager` сам не пункт назначения — ты не вызываешь на нём `open()` и не передаёшь его в `Router.push`; встраивай его в экран.

4.2.10 UIWidget

Плавающий оверлей над текущей страницей: всегда position-fixed относительно устройства, рисуется над контентом destination. Используй для шторок (bottom sheet), диалогов, тостов-с-действиями, плавающих плееров — и это санкционированный способ положить UI поверх `Scene`. Создай виджет **один раз** на уровне модуля и переиспользуй — видимость императивна. Для стандартного диалога бери готовый `UIModal` ниже, а не собирай scrim + анимации + обработчики закрытия вручную; для перетаскиваемой шторки с позициями примагничивания — `UIBottomSheet`.

`show()` открывает виджет как **глобальный оверлей** над всеми destination: он остаётся поднятым через навигацию, пока ты не вызовешь `hide()`. Навигация никогда не трогает неприкреплённый виджет — модалка с `overlayColor` всё равно блокирует взаимодействие под собой, так что закрывай её из её собственных кнопок / `onOverlayTap` / `onBackPressed`.

Чтобы виджет стал частью *страницы* — HUD поверх `Scene` — сначала прикрепи его: `attachTo(owner)` кладёт виджет на слой этой страницы, так что он показывается и скрывается вместе со страницей, едет в её переходе, а экран, запушенный сверху, его накрывает.

Обзор

TS

```
const dialog = UIWidget(
  UIText("Delete item?").style({ fontWeight: 700, fontSize: 18, color: "black" }),
  UIButton(UIText("Delete").style({ color: "white" }))
    .style({ bgColor: "#FF4032", borderRadius: 8, height: 40, justifyContent: "center" })
    .onClick(() => { doDelete(); dialog.hide() }),
)
.style({ left: 24, right: 24, top: "40%", p: 16, gap: 12, borderRadius: 16,
         bgColor: "white", overlayColor: "rgba(0,0,0,0.5)" })
.onOverlayTap(() => dialog.hide())
.onBackPressed(() => dialog.hide())

// где угодно, на любом экране:
dialog.show()
```

Создание и видимость

TS

```
UIWidget(...children: (UINodeChild | UINodeChild[])[]): UIWidget

widget.show(): void          // смонтировать как глобальный оверлей (до этого скрыт)
widget.hide(): void          // размонтировать
widget.isShow: boolean       // геттер — показан ли сейчас?

widget.attachTo(owner: Presentable | null): this   // положить виджет на слой страницы
владельца
```

Полная поверхность контейнера: `.append` / `.insert` / `.remove` / `.setContent`, `.style`, `.onLayout`. Позиционируй его через `top` / `left` / `right` / `bottom` / `width` / `height` — координаты в пространстве экрана устройства (значения safe-area вроде `"safe-bottom"` работают).

`attachTo(owner)` делает виджет принадлежащим этому destination (`Scene`, `UIScreen`, …): он виден, только пока владелец презентован — скрывается, когда запушенный экран накрывает владельца, и возвращается, когда pop его открывает — и анимируется вместе с владельцем во

время переходов. Прикрепляй до `show()` (пока виджет скрыт); привязка липкая до `attachTo(null)`, который снова делает виджет глобальным оверлеем. HUD для `ARScene` можно прикрепить до `open()`, чтобы он был на месте с первого кадра.

UIModal — диалоги со встроенным бойлерплейтом

Для стандартного диалога используй `UIModal`, а не собирай паттерн вручную. Это **и есть** `UIWidget` (та же стилизация, дети, касания и поверхность `attachTo`), который дополнительно:

- имеет scrim по умолчанию (`overlayColor: "rgba(0, 0, 0, 0.5)"` — переопредели его или передай `overlayColor: null`, чтобы убрать),
- анимируется на `show()`/`hide()` — фейд 200 мс, если не заменишь позу через `transition()` (выход играется с `commit: false` и размонтирует по завершении, так что стиль нетронут к следующему показу),
- закрывается сам по тапу на scrim и по кнопке «назад» Android — `dismissible(false)` выключает и то, и другое для диалогов с принудительным выбором.

TS

```
UIModal(style?, children?): UIModal   // те же перегрузки, что у UIWidget

modal.show(): void                    // смонтировать + входной переход (no-op, пока открыт)
modal.hide(): void                    // переход выхода, затем размонтирование (no-op, пока
закрывается)
modal.isOpen: boolean                 // true от show() до начала hide()

modal.transition(hidden): this        // заменить анимацию показа/скрытия — см. ниже
modal.onOpen(cb: () => void): this    // после того как show() его смонтирует
modal.onClose(cb: () => void): this   // когда начинается закрытие — тап по scrim, «назад»
или hide()
modal.dismissible(enabled: boolean): this
```

TS

```
const confirm = UIModal(
  UIText("Delete item?").style({ fontWeight: 700, fontSize: 18, color: "black" }),
  UIButton(UIText("Delete").style({ color: "white" }))
    .style({ bgColor: "#FF4032", borderRadius: 8, height: 40 })
    .onClick(() => { doDelete(); confirm.hide() }),
).style({ left: 24, right: 24, top: "40%", p: 16, gap: 12, borderRadius: 16, bgColor: "white"
})

confirm.show()   // где угодно — scrim, фейд-ин, кнопка «назад» и тап по scrim уже подключены
```

`transition(hidden)` — замена анимации показа/скрытия

`hidden` — это поза «за экраном»: `show()` анимирует *из* неё, `hide()` — *в* неё, а её `duration` (мс, по умолчанию 200) задаёт время обоих направлений плюс отложенного размонтирования. Поза заменяет дефолтный `{ opacity: 0 }` целиком — слайд без фейда это просто слайд. Scrim сохраняет собственный фейд, если поза сама не управляет `overlayColor`. Длины в `transform` — px (проценты не поддерживаются) — сдвигай минимум на высоту самой модалки.

```ts
// статичная нижняя панель — одна строка отличия от диалога:
const panel = UIModal(…)
  .style({ left: 0, right: 0, bottom: 0, height: 400, borderRadius: 20, bgColor: "white" })
  .transition({ transform: "translateY(480px)", duration: 250 })
panel.show()
```

Перетаскиваемую шторку (палец примагничивается между позициями) не собирай руками — используй `UIBottomSheet` ниже.

UIPopover — привязанные меню

Для дропдауна, контекстного меню или тултипа используй `UIPopover` — `UIModal`, чей scrim по умолчанию `"transparent"` (невидим, но всё равно перехватывает: тап снаружи закрывает, и ничто под ним не может скроллиться, пока он открыт, так что якорь не уедет из-под него), а позиция берётся из якоря, переданного в `show()`:

```ts
UIPopover(style?, children?): UIPopover   // те же перегрузки, что у UIWidget/UIModal

popover.show(anchor?): void   // anchor: любой элемент или сырая точка { x, y } (меню по
долгому нажатию)
popover.hide(): void          // переход выхода, затем размонтирование — плюс всё, что есть у
UIModal
                              // (isOpen, onOpen/onClose, dismissible, transition — по
умолчанию
                              // фейд 120 мс)
```

```ts
const item = (label: string, action: () => void) =>
  UIButton(UIText(label).style({ color: "white" }))
    .style({ height: 40, px: 12, justifyContent: "flex-start" })
    .onClick(action)

const menu = UIPopover(
  item("Rename", () => { menu.hide(); rename() }),
  item("Delete", () => { menu.hide(); remove() }),
).style({ width: 220, borderRadius: 12, bgColor: "#222", py: 4 })

moreButton.onClick(() => menu.show(moreButton))              // привязано к кнопке
rowButton.onLongPress(ev => menu.show({ x: ev.clientX, y: ev.clientY }))   // у пальца
```

Размещение автоматическое и вычисляется **один раз на показ**: под левым краем якоря (зазор 4 px), с переворотом наверх, когда нет места, с прижатием на 8 px внутрь вьюпорта (флип/прижатие уточняются на первой раскладке поповера, внутри входного фейда — коррекцию ты не увидишь). Поповер сам прикрепляется к `Presentable.current`, так что скрывается вместе со страницей, на которой открылся, и едет в её переходе; явный `attachTo(owner)` до `show()` уважается. Привязанные поповеры не следуют за якорем — позиция-один-раз это платформенная конвенция.

Под капотом это ровно рецепт `getBoundingClientRect()` + `attachTo` (см. overview.md) — собирай вручную, только когда нужна логика размещения, которую компонент не предлагает.

Модальное поведение — overlayColor

```ts
widget.style({ overlayColor: "rgba(0, 0, 0, 0.5)" })   // Color | null
widget.onOverlayTap(cb: () => void): this
```

`overlayColor` добавляет полноэкранный scrim *за* виджетом, блокирующий все тапы под ним — это и превращает виджет в модалку (диалог / шторку). `"transparent"` невидим, но всё равно перехватывает; `null` (по умолчанию) убирает слой целиком. Тап по scrim’у вызывает `onOverlayTap` — обычно `() => widget.hide()`.

Касания и кнопка «назад»

```ts
widget.onTouchStart(ev => …)          // ev.track({...}) для жестов перетаскивания (перетаскивание шторки)
widget.onBackPressed(cb: () => void)   // Android «назад», пока виджет поднят — обычно hide()
```

Виджеты — один из трёх элементов, принимающих касания (с `UIButton` и `UIScreen`) — см. События указателя.

Анимации выхода — animateTo с commit: false

```ts
widget.animateTo({ overlayColor, opacity, transform, …, duration?, delay?, commit?: boolean
})
widget.animateFrom({ … })              // анимировать от заданных значений к текущему стилю
```

`animateTo` обычно пишет целевые значения в стиль виджета при старте. Для анимации *выхода* это неверно — следующий `show()` стартовал бы с затухших значений. Передай `commit: false`, чтобы проиграть анимацию без сохранения, и `hide()` по завершении:

```ts
const close = () => {
  widget.animateTo({ opacity: 0, commit: false, duration: 250 })
  setTimeout(() => widget.hide(), 250)   // стиль всё ещё с opacity 1 для следующего show()
}
```

UIBottomSheet — перетаскиваемая шторка с несколькими детентами

Виджет, прижатый к нижнему краю. **По умолчанию размером с контент** — высотой в своих детей (с потолком в экран), одна позиция, перетаскивание вниз закрывает: форма action sheet, ноль конфигурации. Добавь **детенты** для модели картографического приложения — позиции примагничивания, между которыми пользователь перетаскивает на нативных хостах. Всё модальное (scrim, `onOpen`/`onClose`, закрытие по тапу на scrim и кнопке «назад», `dismissible(false)`) работает ровно как в `UIModal`.

TS

```
// размер по контенту: action sheet — это просто дети
const actions = UIBottomSheet(rows).style({ bgColor: "white", borderRadius: 20 })
actions.show()

// детенты: модель картографического приложения
const sheet = UIBottomSheet(
  UIColumn().style({ width: 50, height: 6, borderRadius: 3, bgColor: "#D9D9D9", mx: "auto",
my: 12 }),
  UIScrollable(results),
)
  .style({ bgColor: "white", borderRadius: 20 })
  .detents([0.3, 0.6, 1])              // доли высоты экрана, по возрастанию
  .onDetentChange(i => { ... })         // каждая остановка: примагничивание пальцем или
setDetent()

sheet.show()                            // въезжает к текущему детенту (изначально индекс 0)
sheet.setDetent(2)                      // программно, с анимацией
sheet.hide()                            // выезжает, scrim гаснет, размонтируется
```

- **Размер**: без `detents()` бокс шторки — её контент (`maxHeight: "100%"`; для длинного контента положи внутрь `UIScrollable`). С `detents()` бокс — это *самый высокий* детент (`height` + `bottom: 0` выставляются за тебя); нижние детенты показывают верхний срез. Смены детента и перетаскивание — чистая трансляция, контент никогда не перекладывается.
- **Перетаскивание** (нативные хосты): примагничивание пальцем с учётом скорости, rubber-band выше верхнего детента и передача скролла — `UIScrollable` внутри скроллится нормально на верхнем детенте, а тяга вниз с его верха передаёт жест шторке. На вебе шторка статична; смены детента анимируются.
- **Закрытие**: перетаскивание ниже нижнего детента закрывает шторку (вызывает `onClose`); `dismissible(false)` вместо этого сворачивает её к нижнему детенту — персистентная шторка в стиле карт.
- Создай её **один раз** на уровне модуля, как каждый виджет.

(Полностью кастомный жест всё ещё возможен с обычным `UIWidget` + трекингом `onTouchStart` `claim: "pan-y"` — см. touch — но сначала бери `UIBottomSheet`: нативную физику перетаскивания из JS не повторить.)

Подводные камни

```ts
// ✗ пересоздание виджета на каждый экран / на каждый показ
const openSheet = () => UIWidget(...).show()  // течёт новый виджет на каждый вызов
// ✓ создай один раз на уровне модуля, show()/hide() тот же экземпляр

// ✗ ждать, что он появится при создании
const w = UIWidget(...)                       // скрыт до w.show()
```

Смотрите также

- UIScreen и Router — стек экранов, над которым плавают виджеты.
- События указателя и жесты — `ev.track()`, `claim`.
- Стилизация — `animate`, `animateTo`, трансформы.

4.2.11 UIVirtualizedList

Оконный список для длинных или неограниченных данных — ленты, чаты, результаты поиска. В любой момент смонтированы только строки рядом с вьюпортом (плюс буфер overscan). Используй его вместо `UIScrollable` + `items.map(Row)`, когда число элементов велико, растёт со временем или неизвестно; для дюжины статичных строк проще обычный `UIScrollable`.

Данные управляются **императивно** — диффинга нет. Создай список один раз, затем вызывай `setData` / `append` / `update` на нём; никогда не пересобирай список, чтобы сменить его содержимое.

Обзор

```ts
type Post = { id: number; title: string; body: string }
let page = 1

const list = UIVirtualizedList<Post>({
  keyOf: p => String(p.id),
  estimatedHeight: p => 72 + Math.ceil(p.body.length / 38) * 20,
  render: p => UIColumn(
    UIText(p.title).style({ fontWeight: 700, color: "white", fontSize: 16 }),
    UIText(p.body).style({ color: "#bbb", fontSize: 14 }),
  ).style({ gap: 4, px: 16, py: 14 }),
}).style({ flexGrow: 1 })
  .onEndReached(600, async () => {
    const res = await fetch(`https://example.com/posts?page=${page++}`)
    list.append(...res.json<Post[]>())
  })
```

Конфигурация

```ts
UIVirtualizedList<T>(config: {
  keyOf: (item: T) => string                        // СТАБИЛЬНЫЙ уникальный id (никогда не индекс массива)
  render: (item: T) => UINodeChild                  // строит одну строку, вызывается при входе в окно
  estimatedHeight: number | ((item: T) => number)   // догадка в px до измерения строки
  overscan?: number                                 // px, удерживаемые смонтированными вокруг вьюпорта; по умолчанию: высота одного вьюпорта
  inverted?: boolean                                // режим чата, по умолчанию false
})
```

- `keyOf` — ключи идентифицируют строки через `setData` / `update` / `removeByKey`. Индекс не стабилен, как только элементы вставляются или удаляются.
- `render` **должен быть чистым над элементом**: строка может быть размонтирована и перерендерена в *любой* момент, когда покидает и снова входит в окно, поэтому её вывод может зависеть только от переданного элемента — не от внешнего изменяемого состояния и не от захваченного элемента, который ты меняешь позже. Чтобы изменить смонтированную строку, поменяй элемент и вызови `.update(item)`.
- `estimatedHeight` задаёт размер строки до её первого реального измерения (вычисляется один раз на элемент, при добавлении). Достаточно, чтобы было близко — лучшие

догадки лишь снижают дрожание прокрутки.

- `inverted: true` — режим чата: первая раскладка стартует прокрученной к концу, а `append`, пока внизу, авто-скроллит к новой строке.

Стилизация: те же стили элемента, что у других контейнеров. У списка **нет собственной высоты** — дай ему `flexGrow: 1` (или явную `height`), чтобы он занял доступное место; без размера он схлопывается в ноль и ничего не рендерит. `flexShrink: 1` уже по умолчанию (как у `UIScrollable`), так что явная высота, которая не влезает, сжимается вместо переполнения.

Операции с данными

```ts
list.setData(items: T[]): this        // заменить всё (диффится по ключу)
list.append(...items: T[]): this      // добавить в конец (чат: новейшее сообщение)
list.prepend(...items: T[]): this     // добавить в начало БЕЗ скачка прокрутки
(компенсируется)
list.update(...items: T[]): this      // перерендерить строки с тем же keyOf(); неизвестные
ключи игнорируются
list.removeByKey(...keys: string[]): this

list.itemCount: number                // элементов сейчас в наличии (геттер)
list.getItem(key: string): T | undefined
```

`update` перерендеривает строку, только если она сейчас смонтирована; строки вне окна подхватывают новый элемент естественно, когда снова входят. `prepend` — примитив загрузки истории — старые сообщения появляются сверху, не сдвигая то, что читает пользователь.

Прокрутка

```ts
list.scrollTo(offset: number, animated?: boolean): void    // px-смещение; animated по
умолчанию true
list.scrollToKey(key: string, animated?: boolean): void
list.scrollToEnd(animated?: boolean): void

list.onScroll(cb: (scrollPosition: number) => void): this
```

Pull-to-refresh

```ts
list.onRefresh(cb: () => void | Promise<void>): this   // спиннер остаётся, пока промис не
завершится
```

Стиль `refreshControlColor` тонирует спиннер. Вешай `onRefresh` **до** монтирования списка — хосты читают колбэк при создании узла. Только нативные хосты (веб-вьюеры — no-op). Тот

же контракт, что у `UIScrollable`.

Краевые колбэки — пагинация

```ts
list.onEndReached(thresholdPx: number, callback: () => void): this     // рядом с НИЗОМ контента
list.onStartReached(thresholdPx: number, callback: () => void): this   // рядом с ВЕРХОМ (история чата)
```

Порог (px от края) идёт **первым** и обязателен. Колбэк срабатывает один раз, когда прокрутка входит в зону порога, и защёлкивается — он не сработает снова, пока пользователь не выкрутится из зоны. Держи свои флаги `busy` / `end` для запросов в полёте и исчерпанной пагинации.

```ts
list.onEndReached(600, loadNextPage)      // ✓
list.onEndReached(loadNextPage)           // ✗ неверный порядок — порог это первый аргумент
```

Подводные камни

```ts
// ✗ пересоздание списка (или вызов setData с готовыми элементами) для «ре-рендера»
screenBody.setContent([ UIVirtualizedList({...}) ])   // теряет прокрутку, перемонтирует всё
// ✓ держи один экземпляр, меняй через setData/append/update

// ✗ строка читает внешнее изменяемое состояние — устаревает после цикла размонтирования/перерендера
render: m => UIText(selectedId === m.id ? "✓ " + m.text : m.text)
// ✓ положи состояние в элемент и вызови update()
list.update({ ...m, selected: true })

// ✗ нет размера — у списка нет собственной высоты, он схлопывается в 0 и ничего не показывает
list.style({ flexGrow: 1 })   // ✓ (или явная height)
```

Смотрите также

- Контейнеры — `UIScrollable` для короткого статичного контента.
- UIScreen и Router — экраны никогда не скроллятся; список — прокручиваемое тело экрана.
- Кнопки и поля ввода — строки `UIButton` внутри списка.

4.2.12 NativeView

Зарегистрированная хостом платформенная вью — карта, QR-сканер, превью камеры, … Хост регистрирует *возможность* (`registerView("map", factory)` в своём коде); **когда** и **с чем** её показать — это код приложения. `QRScanner` и `CameraView` — типизированные обёртки ровно над этим каналом.

Опциональна для хоста — всегда проверяй `NativeView.isSupported(name)`, прежде чем предлагать фичу.

Обзор

```ts
if (NativeView.isSupported('map')) {
  const map = NativeView('map', { zoom: 12 })
    .style({ flexGrow: 1, borderRadius: 12 })
    .on('markerTap', data => toast(data.title))

  screen.append(map)                          // встроена в раскладку…
  await map.call('setCenter', 37.62, 55.75)   // …и управляется по JSON-каналу
}
```

Destination И элемент

Двойственная природа, как у каждого Presentable:

- **Встроенная**: помести её среди детей экрана — она раскладывается своими стилями как любой элемент (`ElementStyle` + drawable-стили, `animateTo`/`animateFrom`, классы, `onLayout`).
- **Полноэкранная**: `map.open({ transition })` / `map.close()` — или `Router.push(map)`; она становится текущим destination с `onOpen`/`onClose`/`onBackPressed`, как экран.

Один экземпляр живёт в одном месте за раз. Пуш уже встроенного экземпляра в полный экран — это *промоушен*: нативная вью перепривязывается к новому родителю с сохранением состояния (работающая камера продолжает работать). `open()` бросает исключение на хостах без навигационного канала Presentable.

Канал call/event

```ts
const result = await view.call(method, ...args)   // JSON-сериализация в обе стороны; reject при
                                                   // неизвестной вью/методе или без поддержки хоста
view.on(event, data => ...)                        // события, которые эмитит нативная сторона
view.off(event, callback)
```

`params` (второй аргумент) уходит в фабрику хоста при создании — используй его для опций момента конструирования; всё, что после, — через `call`.

Пишем типизированную обёртку

Плагины поставляют типизированный TS-класс над сырым каналом, а не выставляют строки коду приложения — конвенцию смотри в `QRScanner`/`CameraView` (src/plugins/): зафиксируй `viewName`/`params`, наслои типизированные методы поверх `call`/`on` и выстави геттер `isSupported`.

Для возможностей **без визуальной поверхности** (GPS, bluetooth, …) используй сервисный канал — тот же протокол без вью; `Geolocation` — его первый плагин.

Смотрите также

- QRScanner / CameraView — поставляемые обёртки.
- Geolocation — headless-собрат (сервисный канал).
- Экраны и Router — destination и переходы.

4.2.13 Шрифты

Шрифты декларативны: **макрос** `font()` времени компиляции устанавливает шрифт и возвращает имя его семейства — использовать шрифт *и значит* объявить его. Нечего загружать, ждать или гейтить; начертания едут вместе с приложением (бандлятся локально или префетчатся хостом до первой отрисовки), и текст просто рендерится ими. Системным семействам установка не нужна: `"serif"`, `"sans-serif"`, `"monospaced"` работают как значения `fontFamily` из коробки.

Обзор

```ts
theme({ fontFamily: font("manrope") })          // дефолтный шрифт текста на всё приложение

UIScreen(
  UIText("Display").style({ fontFamily: font("unbounded"), fontSize: 32 }),
  UIText("Body text"),                           // дефолт темы — Manrope
).open()                                        // никакого Promise.all, никакого гейтинга — никогда
```

font(id) — реестр

`font("manrope")` резолвится по курируемому реестру шрифтов (~26 OFL-семейств, с поддержкой кириллицы, если не отмечено иное) и компилируется в строку имени семейства `"Manrope"`; подразумеваемые им начертания регистрируются автоматически до запуска твоего кода.

```ts
font(id: string, opts?: { weights?: number[], italic?: boolean }): string
```

- По умолчанию устанавливает стандартные веса семейства (обычно 400/500/700). `{ weights: [400, 600] }` сужает или расширяет набор — ставь веса, которые реально используешь (`fontWeight: 600` хочет начертание 600; иначе хосты подставляют ближайшее).
- `{ italic: true }` добавляет курсивные начертания рядом с прямыми (ошибка компиляции называет семейства/веса, у которых курсива нет).
- Аргумент должен быть **строковым литералом** — неизвестный id это ошибка компиляции со списком валидных.

Id реестра по категориям:

категория	id
sans-serif	`inter` `manrope` `golos-text` `onest` `rubik` `nunito` `montserrat` `ibm-plex-sans` `pt-sans` `jost` `exo-2`
display	`unbounded` `russo-one` `oswald` `comfortaa` `space-grotesk` (только латиница)
serif	`playfair-display` `lora` `pt-serif` `cormorant` `literata`
monospace	`jetbrains-mono` `fira-code` `ibm-plex-mono`
handwriting	`caveat` `pacifico`

font(“./path.ttf”) — свои файлы шрифтов

Аргумент в форме пути устанавливает файл шрифта проекта. Компилятор сам читает файл — настоящее имя семейства из его таблицы `name`, вес и курсив из `OS/2` — так что нет никаких конвенций именования и никаких опций:

```ts
theme({ fontFamily: font("./fonts/Brand.ttf") })   // → "Brand" (из файла)
font("./fonts/Brand-Bold.ttf")                     // то же семейство, добавляет жирное начертание
```

Только `.ttf`/`.otf` (конвертируй экспорты WOFF/WOFF2); путь относителен вызывающему файлу, как у `asset()`.

Системные шрифты

Три обобщённых семейства — просто строки `fontFamily`: без `font()`, ничего не устанавливается и не скачивается (вызов `font("serif")` — ошибка компиляции, указывающая сюда):

```ts
UIText("Quote").style({ fontFamily: "serif" })
UIText("SKU-10442").style({ fontFamily: "monospaced" })   // CSS-овский "monospace" тоже принимается
```

Каждая платформа резолвит их в своё нативное начертание (serif на iOS — New York, на Windows — Times New Roman, и так далее) — используй их, когда хочешь платформенный вид, и шрифт из реестра, когда нужно одно и то же начертание везде. Снапшоты дизайн-борда и `lecodes render` пришпиливают их к бандленным конкретным начертаниям, чтобы headless-вывод был детерминированным: `serif` → PT Serif, `monospaced` → JetBrains Mono, `sans-serif` → Roboto (дефолтный шрифт UI).

Как это загружается (и почему ты никогда не ждёшь)

Объявленные начертания известны на компиляции: локальные/standalone-сборки бандлят их рядом с приложением (они регистрируются синхронно на старте — вообще без мигания), а платформенные сборки запекают CDN-URL, которые хост префетчит параллельно запуску приложения, ненадолго придерживая первый кадр, чтобы текст с самого начала появлялся в правильном шрифте. Если начертание опаздывает (плохая сеть), текст подменяется, когда оно доедет — это запасной путь, а не нормальный.

Выбор шрифта на элемент

`fontFamily` задаётся на каждый `UIText` (или через дефолт `theme({ fontFamily })` на всё приложение) — не полагайся на установку на контейнере. Для повторяющейся кастомной стилизации вынеси общий `Style<UIText>`.

registerFont — рантайм-лазейка

```ts
registerFont(fontFamily: string, url: string, options?: { weight?: number, style?: "normal" | "italic" }): Promise<void>
```

Только для URL, вычисляемых в рантайме (пикер шрифтов, шрифты от пользователя). Один вызов на начертание; промис разрешается, когда начертание пригодно, но раз `font()` покрывает и реестр, *и* файлы проекта, это почти никогда не нужно — и ждать его не нужно никогда.

Смотрите также

- Текст и элементы контента — `UIText`, стили `fontFamily` / `fontWeight`.
- Тема — `theme({ fontFamily })`, механизм дефолтного текста.

4.3 3D-движок

4.3.1 Scene

3D-мир: набор для отрисовки, `Camera`, источники света и окружение. Открытие сцены поднимает 3D-движок хоста — **рендерит и получает события указателя только активная сцена**. Для AR-сессии с камерой используй `ARScene` (подкласс).

Обзор

```ts
const scene = new Scene({ skybox: '#202030', bloom: true })
scene.add(
  Light.sun({ shadowsQuality: 2 }),
  Mesh.box({ material: Material.lit({ color: '#e33' }), position: [0, 0.5, -2] }),
)
scene.camera.position = [0, 1.5, 2]
scene.open()

scene.addEventListener('click', ev => {
  console.log('hit', ev.target?.name ?? 'nothing')
})
```

Создание

```ts
new Scene(options?: {
  ibl?: boolean                    // окружающий свет на основе изображения; по умолчанию true
  environmentIntensity?: number    // интенсивность IBL; по умолчанию 20000
  bloom?: boolean                  // пост-обработка bloom; по умолчанию false
  bloomIntensity?: number          // сила bloom; по умолчанию 0.2 (только с bloom: true)
  skybox?: ColorInput              // сплошной фон / цвет очистки
  antialias?: boolean              // 4× MSAA; по умолчанию false
})
```

Освещение задаётся явно: IBL по умолчанию даёт каждой сцене окружающий свет (поставь `ibl: false`, если освещаешь всё сам); добавь `Light.sun()` для прямого света и теней.

> **NOTE**
> опции доступны только в конструкторе, кроме фона (сеттер `scene.skybox = …`) и сглаживания (`setAntialias`). Рантайм-переключателя для `ibl` или `bloom` нет.

Добавление и удаление узлов

```ts
scene.add(...nodes: Node[]): this
scene.remove(...nodes: Node[]): this
```

Регистрирует узлы в наборе отрисовки этой сцены. Это отличается от *родительства* — `node.add(child)` строит иерархию (см. Node); `scene.add(node)` помещает его в мир.

Открытие и закрытие

```ts
scene.open(): void        // стать активной сценой (закрывает любую другую)
scene.close(): void       // разобрать 3D-вид
Scene.active              // статик: активная сейчас Scene или null
```

Активна одновременно только одна сцена; `open()` на второй сцене заменяет первую. После `close()` `Scene.active` — `null` (если это была активная). `ARScene.open()` — `async` — см. ARScene.

Оверлеи

```ts
scene.createOverlay(options?: SceneOptions): Scene
```

Вторая сцена, рисуемая поверх этой и **разделяющая её камеру** — 3D-слой HUD, всегда рендерящийся поверх контента родителя. Принимает те же опции, что конструктор; добавляй в неё узлы как в любую сцену. Она рендерится вместе с родителем — не вызывай `open()` на ней отдельно.

warmRender

```ts
scene.warmRender(): Promise<void>
```

Рендерит сцену один раз, чтобы шейдеры скомпилировались и ресурсы загрузились до показа. Если первый видимый кадр дёргается, сделай `await scene.warmRender()` на экране загрузки, затем `open()`. (`ARScene.open()` делает это автоматически.)

Ручки рендера

```ts
scene.skybox = '#87ceeb'                                  // цвет фона (только запись)
scene.setAntialias(enabled: boolean, scale = 4): void     // MSAA вкл/выкл + число сэмплов
scene.occlusionMaterial: Material                         // окклюзионный материал сцены (только чтение)
scene.setMaterialGlobalParameter(i, x, y, z, w): void     // общий для сцены vec4-слот i, читается кастомными шейдерами
```

`occlusionMaterial` — экземпляр `Material`; задавай его юниформы как у любого другого.

> **NOTE**
> `occlusionMaterial` гейтится хостом: веб-вьювер его не реализует (обращение бросает). Используй только на AR-способных нативных хостах.

События указателя на уровне сцены

```ts
scene.addEventListener('click', (ev: ClickEvent<Node | null>) => void): void
scene.addEventListener('touchstart', (ev: TouchStartEvent<Node | null>) => void): void
scene.removeEventListener(channel, callback): void
```

Слушатели сцены срабатывают на **каждый** тап по активной сцене; `ev.target` — задетый `Node` (с телом-пикером `Shape`) или `null` при промахе. Отдельные узлы тоже могут слушать — см. события Node и События указателя и жесты.

VRScene

```ts
const scene = new VRScene(options?: SceneOptions)
await scene.open()   // отклоняется на хостах без VR-рантайма ("VR is not supported on this device")
scene.close()
```

Подкласс `Scene`, рендерящийся в VR-шлеме (OpenXR — напр. Meta Quest). Камерой управляет шлем: оба глаза каждый кадр рендерятся из отслеживаемой позы головы, поэтому position / lookAt у `scene.camera` не действуют, пока сцена открыта. Начало мира — на уровне пола (stage space) — строй контент с землёй на `y = 0` и пользователем, стоящим в начале координат. Всё остальное (узлы, свет, материалы, аспекты) работает ровно как в обычной `Scene`.

Смотрите также

- ARScene — AR-сессия: async-открытие, якоря, управление размещением.

- Camera — `scene.camera`: fov, экранные лучи.
- Light — `Light.sun()` для прямого света + теней поверх окружающего IBL.
- Node — трансформ, иерархия, события.
- Material — что предоставляют `occlusionMaterial` и материалы мешей.

4.3.2 ARScene

`Scene`, рендеримая поверх видеопотока камеры устройства с трекингом мира — всё из `Scene` (опции, `add`/`remove`, события, оверлеи) применимо. Что она добавляет: **async** `open()`, запрашивающий камеру и запускающий трекинг, узлы-**якоря**, следующие за реальным миром, и готовые жесты размещения объектов через `addControls`.

Обзор

TS

```
const scene = new ARScene()
const model = await Model.load(asset('./chair.glb'))
scene.root.add(model)
scene.add(model)

const controls = scene.addControls(model)     // перетаскивание / щипок / поворот для
размещения
await scene.open()

model.addEventListener('track', () => console.log('anchored to the world'))
```

Создание и открытие

TS

```
new ARScene(options?: SceneOptions & {
  mode?: 'default' | 'markers' | 'arcore' | 'detached'   // режим трекинга; по умолчанию
'default'
})

scene.mode                          // только чтение: режим, с которым она создана
scene.useWarmRender = true          // open() сначала прогревает шейдеры (по умолчанию true)

await scene.open(): Promise<void>  // запрос камеры → прогрев рендера → запуск AR → стать
активной
scene.close(): void                // остановить AR-сессию и разобрать вид
```

`open()` асинхронный — он запрашивает разрешение на камеру, прогревает рендер (когда `useWarmRender` включён) и запускает трекинг до того, как сцена станет активной. Текущий пункт назначения (обычно экран загрузки) **остаётся видимым всё это время**: подмена — и

`transition`, если ты его передал — происходит только когда всё готово, так что экран загрузки переходит сразу в живой видеопоток камеры. Он **отклоняется**, если в камере отказано, AR не может запуститься или во время подготовки случилась другая навигация — во всех случаях текущий экран не тронут, поэтому делай `await` и обрабатывай сбой. `Router.push(arScene)` ждёт так же (для AR-страниц он возвращает промис).

Якоря

```ts
scene.root: Node                                                    // корневой якорь мира (только чтение)
scene.createAnchor(source: FetchResponse, physicalWidth = 0.2): Node   // якорь по изображению-цели
```

Якорь — это `Node`, чей трансформ управляется трекингом — подвешивай под него свой контент. `root` — якорь начала мира; `createAnchor` создаёт якорь, следующий за источником трекинга (напр. изображением-целью, скачанным через `fetch`), где `physicalWidth` — реальная ширина цели в **метрах** (по умолчанию 0.2).

Узлы-якоря сообщают состояние трекинга:

```ts
anchor.addEventListener('track', () => {})      // трекинг начался
anchor.addEventListener('untrack', () => {})    // трекинг потерян
anchor.isTracked                                // текущее состояние (false для не-якорных узлов)
```

> **NOTE**
>
> `root` **недоступен в режиме** `'markers'`. Поддержка якорей гейтится хостом — веб-вьювер не реализует `createAnchor`.

Управление размещением — `addControls`

```ts
scene.addControls(target: Node, options?: {
  minScale?: number   // наименьший равномерный масштаб при щипке; по умолчанию 0.08
  maxScale?: number   // наибольший; по умолчанию 2
  pan?: boolean       // перетаскивание одним пальцем; по умолчанию true
  pinch?: boolean     // масштаб двумя пальцами; по умолчанию true
  twist?: boolean     // поворот двумя пальцами; по умолчанию true
}): { remove(): void }
```

Классическое AR-взаимодействие размещения объекта, готовое:

- **Один палец** тащит цель по плоскости земли, относительно того, куда смотрит камера (вверх по экрану уводит её от тебя). Высота (мировой Y) остаётся фиксированной.
- **Два пальца** щипком масштабируют (равномерно, зажато `minScale`/`maxScale`) и поворотом вращают цель вокруг её **оси Y**.

Построено полностью на публичном touch-API (`touchstart` сцены + `ev.track`) — это удобство, а не примитив. Вызови `.remove()` на возвращённом хэндле, чтобы отсоединить слушатели.

> **NOTE**
> зажим щипка допускает лёгкий перелёт за заявленные границы — эффективный диапазон `[minScale × 0.4, maxScale × 1.5]`.

Смотрите также

- Scene — всё унаследованное: опции, add/remove, оверлеи, warmRender, события.
- Node — события `track`/`untrack`, `isTracked`, иерархия.
- Model — загрузка размещаемого GLB.
- События указателя и жесты — touch-поверхность, на которой построен `addControls`.

4.3.3 Node

Базовый узел 3D-сцены — пустой трансформ. `Mesh`, `Model`, `Light` и `Camera` расширяют его, поэтому всё здесь работает и на них. **Нативный движок владеет авторитетным трансформом**; SDK читает его лениво (физика может переписывать его каждый кадр), и каждый геттер трансформа возвращает **свежую копию** — см. семантику значений.

Обзор

TS

```
const pivot = new Node()
pivot.position = [0, 1, -2]
pivot.eulerAngles = [0, 45, 0]          // градусы
pivot.add(Mesh.box({ material: Material.lit({ color: '#38f' }) }))
scene.add(pivot)

setLoop(dt => { pivot.eulerAngles = pivot.eulerAngles.add([0, 90 * dt, 0]) })
```

Локальный трансформ

```ts
node.position = [x, y, z]         // Vec3Like; мировые единицы ≈ метры
node.x = 3                        // скалярные аксессоры: x / y / z
node.quaternion = Quat.fromEuler(0, 90, 0)
node.eulerAngles = [0, 90, 0]     // ГРАДУСЫ
node.scale = 2                    // равномерно или [sx, sy, sz]
node.matrix = Mat4.compose([0, 1, 0], Quat.identity, 1)   // полный локальный TRS
```

Геттеры возвращают свежие **копии** `Vec3` / `Quat` / `Mat4` — изменение одной никогда не трогает узел:

```ts
node.position.x = 3                                       // ✗ тихий no-op (меняет отброшенную копию)
node.x = 3                                                // ✓ сеттер одной оси
const p = node.position; p.y += 1; node.position = p      // ✓ поменять локальную, присвоить обратно
```

Мировое пространство

```ts
node.worldMatrix          // get/set: полный мировой трансформ (установка собирает через родителя)
node.worldPosition        // Vec3 (только чтение)
node.worldQuaternion      // Quat (только чтение)
node.worldEulerAngles     // Vec3, градусы (только чтение)
node.worldScale           // Vec3 (только чтение)
node.forward              // Vec3: ось вперёд узла (−Z) в мировом пространстве (только чтение)
```

Имя и видимость

```ts
node.name = 'door'        // хранится нативно; поиск детей Model сопоставляется по нему
node.visible = false      // скрывает узел (и то, что он рисует)
```

Иерархия

```ts
node.add(...children: Node[]): this                    // сделать детей потомками этого узла
node.setParent(parent: Node | null, worldPositionStays = false): this
node.parent                                            // Node | null
node.children                                          // Node[] (массив-снимок)
node.childCount
node.getChild(index: number): Node | null
node.traverse(callback: (node: Node) => void): void    // этот узел, затем все потомки
```

С `worldPositionStays: true` `setParent` держит узел там, где он в мире (его локальный трансформ пересчитывается); с дефолтным `false` локальный трансформ сохраняется как есть.

> **NOTE**
> родительство отдельно от членства в сцене — `scene.add(node)` кладёт узел в набор отрисовки, `node.add(child)` строит иерархию трансформов. См. Scene.

lookAt

```ts
node.lookAt(point: Vec3Like, mode = '-z', ortho = [0, 1, 0]): this
// mode: 'z' | '-z' | 'x' | '-x' | 'y' | '-y' — какая локальная ось смотрит на цель
```

Ориентирует узел так, чтобы выбранная локальная ось (по умолчанию `'-z'`, ось вперёд) смотрела на точку в **мировом пространстве**. `ortho` — подсказка «вверх» для разрешения крена. Масштаб сохраняется.

Аспекты

Возможности прикрепляются как аспекты и чейнятся (см. аспекты):

```ts
const ball = Mesh.sphere({ material: Material.lit() })
  .aspect(Shape, { sphere: 0.5 })                  // форма пикинга/коллизии → кликабелен, рейкастится
  .aspect(Physics, { motion: 'dynamic' })          // твёрдое тело → node.physics
```

См. 3D-физику и формы для `Shape`, `Physics`, `Trigger`, `CharacterController`.

События

```ts
node.addEventListener(channel, callback): void
node.removeEventListener(channel, callback): void
```

Канал	Срабатывает	Нужен
`click`	отпускание над узлом	аспект `Shape` (тело-пикер)
`touchstart`	нажатие над узлом; `ev.track()` начинает перетаскивание	аспект `Shape`
`enter` / `exit`	контакт физики / перекрытие триггера начались / закончились (`(other: Node)`)	`Shape` + `Physics`/`Trigger`
`loopReached`	зацикленный клип GLB обернулся (`(clip: number)`)	`ModelAnimation`
`completed`	незацикленный клип GLB завершился (`(clip: number)`)	`ModelAnimation`
`track` / `untrack`	AR-якорь получил / потерял трекинг (также `node.isTracked`)	якорь ARScene

Без аспекта `Shape` узел невидим для тапов — тач увидят только слушатели уровня сцены (с `ev.target === null`). Объекты событий и перетаскивания `ev.track()`: События указателя и жесты.

Жизненный цикл

```ts
node.destroy(): void      // освобождает нативную сущность; после этого узел непригоден
```

`destroy()` — единственный способ разбора, `dispose()` на узлах нет. Использование уничтоженного узла — неопределённое поведение.

Подводные камни

```ts
// ✗ изменение возвращённого значения геттера трансформа
node.position.x += speed * dt      // no-op: геттеры возвращают копии
// ✓
node.x += speed * dt

// ✗ запись трансформа динамического физического тела каждый кадр
setLoop(() => { node.position = target })     // борется с шагом физики
// ✓ управляй телом: node.physics.velocity / applyImpulse (см. physics.md)
```

Смотрите также

- Соглашения — семантика значений, градусы против радианов, единицы.
- Mesh / Model — рисуемые виды узлов.
- 3D-физика и формы — тела-пикеры `Shape`, `enter`/`exit`, твёрдые тела.
- События указателя и жесты — полезная нагрузка `click`/`touchstart`, перетаскивания.
- ARScene — узлы-якоря, `track`/`untrack`.

4.3.4 Camera

Камера сцены, доступная как `scene.camera`. Это `Node`, поэтому вся поверхность трансформа работает — `position`, `eulerAngles`, `lookAt`, `forward`. Сверх этого она предоставляет чтение проекции и помощники для лучей экран→мир для пикинга.

Ты её не конструируешь: каждая `Scene` создаёт свою камеру (оверлеи разделяют камеру родителя). В `ARScene` камерой управляет трекинг устройства — читай её, не размещай.

Обзор

TS

```
scene.camera.position = [0, 2, 4]
scene.camera.lookAt([0, 0.5, 0])

scene.addEventListener('touchstart', ev => {
  const ray = scene.camera.getRay(ev.clientX, ev.clientY)
  const hit = Physics.raycast(ray.origin, ray.dir)
  if (hit) console.log('tapped', hit.node?.name)
})
```

Чтение проекции

TS

```
camera.fov              // вертикальное поле зрения, ГРАДУСЫ (только чтение)
camera.horizontalFov    // горизонтальное поле зрения, градусы (только чтение)
camera.displaySize      // [ширина, высота] 3D-вьюпорта, px (только чтение)
camera.projectionMatrix // Float32Array(16), живая — хост обновляет её на месте при ресайзе/
смене fov
```

Лучи экран → мир

```ts
camera.getViewDirection(screenX, screenY): Vec3   // направление луча в мире через экранную точку
camera.getRay(screenX, screenY): Ray              // origin = camera.worldPosition, dir = направление взгляда
```

Экранные координаты — **логические px** — то же пространство, что `ev.clientX` / `ev.clientY`. `getRay` напрямую сочетается с `Physics.raycast` для пикинга по тапу; `Ray` описан в ray-plane-noise.

> **NOTE**
> помощника `worldToScreen` нет — обратная проекция доступна только сыро, через `projectionMatrix` + `worldMatrix` камеры.

Подводные камни

```ts
// ✗ двигать камеру в ARScene — трекинг перезаписывает её каждый кадр
arScene.camera.position = [0, 2, 4]
// ✓ двигай вместо этого МИР: подвесь контент под scene.root и размести его
```

Смотрите также

- Scene — откуда берётся камера; оверлеи её разделяют.
- Node — унаследованная поверхность трансформа (`position`, `lookAt`, `forward`).
- 3D-физика и формы — рейкаст лучей, которые ты строишь здесь.
- Ray, Plane и Noise — `Ray`, возвращаемый `getRay`.

4.3.5 Mesh

Рисуемый 3D-узел, построенный из сырой геометрии — примитивные формы (`Mesh.box()`, `Mesh.sphere()`, …) и `Mesh.from()` для кастомной `Geometry`. Расширяет `Node`, поэтому применима вся поверхность трансформа, иерархии, событий и аспектов. GLB-импорт — это другой вид узла — см. `Model.load`.

Обзор

```ts
const scene = new Scene()

const ground = Mesh.plane({
  material: Material.lit({ color: '#556655' }),
  normal: [0, 1, 0],                    // лицом вверх
  scale: 20,
  receiveShadows: true,
})
const crate = Mesh.box({
  material: Material.lit({ color: '#b5854b' }),
  size: [1, 1, 1],
  position: [0, 0.5, 0],
  castShadows: true,
})
scene.add(ground, crate)
scene.open()
```

Фабрики примитивов

Все фабрики делят одни базовые опции и возвращают готовый `Mesh`:

TS

```
type MeshOptions = {
  material?: Material          // по умолчанию Material.unlit()
  position?: Vec3Like
  eulerAngles?: Vec3Like       // градусы
  scale?: Vec3Like | number    // число = равномерно
  name?: string
  castShadows?: boolean
  receiveShadows?: boolean
}

Mesh.box(options?: MeshOptions & {
  size?: Vec3Like | number     // полные длины рёбер в мировых единицах; по умолчанию 1
(единичный куб)
})
Mesh.sphere(options?: MeshOptions & {
  radius?: number              // по умолчанию 0.5 (единичный диаметр)
  widthSegments?: number       // по умолчанию 32
  heightSegments?: number      // по умолчанию 32
})
Mesh.cylinder(options?: MeshOptions & {
  radius?: number              // по умолчанию 0.5; высота фиксирована на 1 (Y: -0.5..0.5)
  radiusTop?: number           // по умолчанию = radius (конус: поставь 0)
  radiusBottom?: number        // по умолчанию = radius
  edges?: number               // по умолчанию 32
  smooth?: boolean             // по умолчанию true — гладкие боковые нормали; false =
гранёные
})
Mesh.plane(options?: MeshOptions & {
  normal?: Vec3Like            // куда смотрит квад 1×1; по умолчанию [0, 0, 1] (к камере)
})
```

- Дефолты единичного размера: box 1×1×1, sphere диаметр 1, cylinder высота 1 / диаметр 1, plane 1×1. Задавай размер через `size` (box) или `scale`.
- `box.size` запекается в буферы геометрии; `scale` — трансформ узла. Оба рисуются одинаково, но запечённый `size` — то, что измеряет авто-бокс `Shape` — подходит любой, так как авто-бокс тоже умножает на мировой масштаб.

> **NOTE**
>
> плоскости-земле нужен `normal: [0, 1, 0]` — дефолтный квад стоит вертикально лицом к +Z.

Кастомная геометрия

```ts
new Mesh(geometry?: Geometry, material?: Material)
Mesh.from(geometry: Geometry, options?: MeshOptions): Mesh
```

`Geometry` хранит сырые буферы (`vertices`, `normals`, `indices`, `uv`). Построй её из билдеров примитивов (`Geometry.box()`, `.sphere(opts)`, `.cylinder(opts)`, `.plane(opts)` — те же опции, что у фабрик выше) и измени её, или собери из своих `Float32Array`:

```ts
const geo = Geometry.cylinder({ radiusTop: 0, edges: 6 })   // шестиугольный конус
  .scale(1, 3, 1)                                           // меняет буфер вершин
  .translate(0, 1.5, 0)                                     // сдвинуть пивот к основанию
const spike = Mesh.from(geo, { material, position: [2, 0, 0] })
```

`geometry.kind = 'triangles' | 'edges' | 'vertices'` переключает режим отрисовки (каркас / облако точек). Задай его до конструирования `Mesh`.

```ts
mesh.geometry   // Geometry | undefined — буферы, из которых построен этот меш
```

Геттер `geometry` — то, что авто-бокс `Shape` измеряет для формы коллизии (см. физику).

Материалы

```ts
mesh.material                                  // слот 0 — get/set
mesh.setMaterial(material, index = 0): this    // чейнится, любой слот
mesh.getMaterial(index = 0): Material | null   // null, если слот пуст
```

У примитивов один материал-слот. Не передав материал фабрике, получишь дефолтный `Material.unlit()` (плоский, без освещения, без отклика на свет). См. Material.

Флаги рендера

```ts
mesh.castShadows = true      // только запись
mesh.receiveShadows = true   // только запись
mesh.culling = false         // только запись; отключить отсечение по фрустуму (рисовать
всегда)
```

> **NOTE**
> это только сеттеры — чтения нет. Задавай при создании (опции фабрики) или храни свой флаг.

Подводные камни

```ts
// ✗ ждать, что плоскость ляжет плашмя по умолчанию
Mesh.plane({ scale: 10 })                       // стоит вертикально, лицом к +Z
// ✓ направь нормаль вверх для пола
Mesh.plane({ scale: 10, normal: [0, 1, 0] })

// ✗ изменение копии трансформа (семантика значений — см. соглашения)
crate.position.y = 2
// ✓
crate.y = 2
```

Смотрите также

- Node — трансформ, иерархия, события (унаследованы).
- Material — материалы lit/unlit/video.
- Model — GLB-импорты (другой рисуемый вид узла).
- Физика и формы — авто-боксы `Shape` из `mesh.geometry`.
- Соглашения — единицы, градусы, семантика значений.

4.3.6 Model и ModelAnimation

Загруженная GLB-модель — свой вид узла (GLB — это иерархия узлов с запечёнными клипами анимации), отличный от `Mesh` (сырые примитивы, без анимации). Расширяет `Node`. Каждая `Model` несёт аспект `ModelAnimation`, заранее прикреплённый на `model.anim` — сам ты его не прикрепляешь.

Обзор

```ts
const hero = await Model.load(asset('./hero.glb'))
hero.position = [0, 0, -2]
scene.add(hero)

hero.anim.play('Run', { loop: true })
hero.addEventListener('loopReached', clip => console.log('lap', clip))
```

Загрузка

```ts
Model.load(source: string | FetchResponse, options?: {
  culling?: boolean          // по умолчанию false — вся модель рисуется всегда
  onProgress?: (p: { loaded: number, total?: number }) => void
}): Promise<Model>
```

`source` — путь `asset('./file.glb')`, удалённый `https://…` URL или уже скачанный `FetchResponse`. Отклоняется при HTTP ≥ 400 или ошибке декодирования.

> **NOTE**
> отсечение по фрустуму по умолчанию **выключено** для моделей (скиннинг-меши двигаются за пределы своих границ времени импорта). Передай `culling: true` для крупных статичных пропов, чтобы части за кадром пропускали отрисовку.

Клонирование

```ts
const enemy = template.clone(): Model
```

`clone()` делает глубокую копию GLB — меши, скелет и запечённые клипы анимации — переиспользуя уже декодированный ассет (без повторного скачивания и парсинга), поэтому он сильно дешевле второго `Model.load`. Клон прикрепляется к родителю источника и попадает в его сцену, стартуя с текущего трансформа источника. У него своё **независимое** состояние анимации, доступное через `clone.anim`. Отсечение по умолчанию выключено, как у `Model.load`.

```ts
const template = await Model.load(asset('./enemy.glb'))   // декодируем один раз
const wave = Array.from({ length: 8 }, () => template.clone())  // дешёвые копии, без await
wave.forEach((e, i) => { e.x = i * 2; scene.add(e); e.anim.play('Walk', { loop: true }) })
```

Для пулов спавна предпочитай `clone()` повторной загрузке того же URL — один декод, затем N экземпляров.

Иерархия узлов

`Model` — корневой узел GLB; внутренние узлы файла висят под ним как обычные `Node`:

```ts
model.traverse(node => {                // этот узел, затем каждый потомок
  if (node.name === 'Sword') node.visible = false
})
model.children                          // прямые дети
```

Трансформ, `visible`, события, аспекты (`Shape`, `Physics`, …) работают и на корне, и на внутренних узлах.

Анимация — `model.anim`

```ts
model.anim.clips                      // { name: string, duration: number }[] — запечено в glb (секунды)

model.anim.play(clip?: string | number, options?: { loop?: boolean }): this
model.anim.stop(): this

model.anim.playing = true             // get/set — продолжить/приостановить текущий клип
model.anim.loop = true                // get/set — также задаётся через опции play()
model.anim.speed = 1.5                // множитель скорости (1 = авторская скорость)
model.anim.time = 0.5                 // get/set — перемотка внутри текущего клипа (секунды)
```

`play()` принимает **имя или индекс** клипа; без аргумента (пере)проигрывает текущий клип. Проигрывание по имени работает только когда клипы GLB осмысленно названы — проверь `model.anim.clips` (или назови их в DCC-инструменте/экспортёре).

> **NOTE**
> неизвестное имя клипа молча игнорируется — `play('Runn')` оставляет выбранный ранее клип и играет *его*. Валидируй по `clips`, если имя приходит из данных.

События конца клипа

Срабатывают на **узле** (не на `anim`), с индексом клипа как аргументом:

```ts
model.addEventListener('completed', clip => { /* незацикленный клип завершился */ })
model.addEventListener('loopReached', clip => { /* зацикленный клип обернулся */ })
```

После `completed` аспект переключается в `playing = false`; зацикленный клип идёт до `stop()`.

Ограничения

- **Нет кросс-фейда и блендинга** — `play()` жёстко переключается на новый клип в следующем кадре. Одновременно играет один клип; послойного/частичного микса тела нет.

Подводные камни

TS

```
// ✗ await свежей загрузки на каждый спавн — повторное скачивание + декод, дёргается в момент спавна
const enemy = await Model.load(url)          // на каждый спавн
// ✓ загрузи один раз, затем clone() — без await, без повторного декода
const template = await Model.load(url)
const enemy2 = template.clone()

// ✗ ждать плавного перехода
hero.anim.play('Idle')                       // резко — кросс-фейда не существует
```

Смотрите также

- Node — трансформ, иерархия, `traverse`, события (унаследованы).
- Mesh — примитивы / кастомная геометрия (другой рисуемый вид).
- Физика и формы — дать модели форму коллизии.
- Scene — добавление узлов, камера.

4.3.7 Geometry

Сырые данные меша — буферы вершин, нормалей, индексов и UV — плюс билдеры примитивов, стоящие за `Mesh.box()` и компанией. Тянись к ней, чтобы подправить примитив перед оборачиванием (масштаб, сдвинутый пивот, тайловые UV) или собрать полностью процедурные меши. `Geometry` рендерится через `Mesh.from()`; сама она не узел.

Обзор

TS

```
// бокс, чей пивот у НИЖНЕЙ грани (чтобы y = 0 ставил его на пол)
const geo = Geometry.box().scale(1, 2, 1).translate(0, 1, 0)
const pillar = Mesh.from(geo, { material: Material.lit({ color: '#ccc' }) })
scene.add(pillar)
```

Примитивы

```ts
Geometry.box(): Geometry                        // единичный куб, центрирован на начале координат
Geometry.sphere(options?: {
  radius?: number          // по умолчанию 0.5
  widthSegments?: number   // по умолчанию 32
  heightSegments?: number  // по умолчанию 32
}): Geometry
Geometry.cylinder(options?: {
  radius?: number          // по умолчанию 0.5 (обе крышки)
  radiusTop?: number       // по умолчанию = radius (0 делает конус)
  radiusBottom?: number    // по умолчанию = radius
  edges?: number           // по умолчанию 32
  smooth?: boolean         // по умолчанию true — гладкие боковые нормали; false = гранёные
}): Geometry
Geometry.plane(options?: {
  normal?: Vec3Like        // направление лица; по умолчанию [0, 0, 1]
}): Geometry
```

Все примитивы единичного размера (1 мировая единица ≈ 1 м в поперечнике, высота 1 для цилиндра), центрированы на начале координат, с заполненными нормалями и UV. Плоскость — квад 1×1, перпендикулярный `normal`.

Буферы

```ts
geo.vertices   // Float32Array — тройки x,y,z
geo.normals    // Float32Array — тройки x,y,z (на вершину)
geo.indices    // Uint16Array — индексы треугольников (или рёбер/точек)
geo.uv         // Float32Array — пары u,v (на вершину)

new Geometry(vertices, normals, indices, uv)
```

Строй процедурные меши, заполняя буферы сам и конструируя напрямую.

> **NOTE**
>
> индексы — `Uint16Array`, поэтому одна геометрия ограничена 65 536 вершинами.

Преобразование данных

```ts
geo.translate(x, y, z): this     // сдвинуть каждую вершину — двигает пивот
geo.scale(x, y, z): this         // масштабировать каждую вершину
geo.scaleUV(x, y): this          // тайлить/растянуть текстурные координаты
```

Они меняют буферы на месте (чейнятся). Запекание трансформа в геометрию отличается от `node.scale`: физические формы и трансформы детей видят запечённый размер, а нормали *не* пересчитываются — неравномерный `scale` слегка скашивает освещение.

Режим отрисовки

```ts
geo.kind = 'triangles' | 'edges' | 'vertices'    // по умолчанию 'triangles' (только запись)
```

`'edges'` рендерит пары индексов как линии, `'vertices'` — как точки — для каркасных/отладочных видов.

Подводные камни

```ts
// ✗ редактирование Geometry после того, как из неё построен Mesh
mesh.geometry.translate(0, 1, 0)     // буферы уже загружены — меш не меняется
// ✓ сначала доделай геометрию, затем Mesh.from(geo)
```

Смотрите также

- Mesh — `Mesh.from(geometry)` и ярлыки фабрик примитивов.
- Material — как выглядит поверхность.
- 3D-физика и формы — авто-формы читают геометрию узла.

4.3.8 Material

Как выглядит 3D-поверхность. Встроенные виды — статические фабрики: `Material.lit` (PBR), `Material.unlit` (плоский), `Material.video`, `Material.shadow` — и кастомные скомпилированные шейдеры тоже грузятся как материалы. Униформы задаются через `.set()` / прокси `uniforms`, который приводит каждое JS-значение к нужному нативному вызову юниформа. Присваивай мешу через `mesh.material` — см. Mesh.

Обзор

```ts
const tex = await Texture.load(asset('./crate.jpg'))
const crate = Mesh.box({ material: Material.lit({ color: '#ffffff', map: tex }) })
const marker = Mesh.sphere({ material: Material.unlit({ color: '#0f0' }) })
scene.add(crate, marker)

crate.material.set('roughness', 0.3).set('metallic', 1)
```

Фабрики

```ts
Material.lit(options?: { color?: ColorInput, map?: Texture | Canvas | null,
                         roughness?: number, metallic?: number }): Material   // PBR с
освещением
Material.unlit(options?: { color?: ColorInput, map?: Texture | Canvas | null }): Material  //
плоский, игнорирует свет
Material.video(map?: Texture): Material              // сэмплит текстуру VideoPlayer
Material.shadow(color = '#000000aa'): Material       // ловец теней: невидим, кроме мест,
куда падают тени
Material.particles(options?: { map?: Texture | null, sheet?: number | [cols, rows],
                               emissive?: number, blend?: 'alpha' | 'add', soft?: boolean,
                               render?: 'point' | 'quad' | 'stretch' | 'ribbon', stretch?:
number }): Material
                                                     // спрайты/квады/ленты для Particles +
Trail (их дефолт)
Material.load(url: string): Promise<Material>        // кастомный скомпилированный шейдер
(.mat / filamat)
```

- `lit` требует света, чтобы быть видимым — IBL и/или `Light.sun()`. Его PBR-скаляры в 0–1: `roughness` 0 = зеркало, 1 = матовый; `metallic` 0 = диэлектрик, 1 = металл (оба — обычные юниформы: не задан — действует дефолт шейдера, а `.set('roughness', v)` меняет их позже).
- `video` берёт текстуру из `VideoPlayer.texture`.
- `shadow` — AR-классика: плоскость земли, показывающая только падающие на неё тени.
- `particles` — то, чем рисует система `Particles` (и `Trail`), когда ты не передаёшь кастомный материал — сам ты его конструируешь редко: опции `Particles`/`Trail` пробрасываются в него. Тинт/размер/поворот/непрозрачность/кадр каждой частицы приходят из кривых частиц; юниформы (`sheetSize`, `emissive`, `additive`, `softness`, плюс `mode`/`stretch` у quad-варианта) задают вид системы в целом, и опции фабрики ложатся на

них. `render: 'point'` использует `particles.filamat`; остальные режимы делят `particles-quad.filamat`.

Цвет и карта

```ts
material.color = '#ff8800'            // любой ColorInput
material.map = tex                    // Texture или Canvas (запекается в текстуру при присвоении)
material.map = null                   // убрать карту
```

`color` и `map` — удобства поверх нижележащих юниформов (`baseColor` / `baseColorMap` у lit/unlit, `color` у shadow). У lit/unlit шейдер их перемножает — `baseColor × baseColorMap`, оба по умолчанию белые (незаданную карту подменяет 1×1 белая текстура) — поэтому цвет сам по себе или карта сама по себе проходят без изменений, цвет, заданный **вместе** с картой, тонирует карту, а `map = null` сбрасывает текстуру обратно в белую. На **кастомном** материале они не знают имён юниформов — `color` тихий no-op, а `map` пишет `baseColorMap`; задавай свои юниформы напрямую.

Кастомные шейдеры и юниформы

```ts
const material = await Material.load(asset('./hologram.mat'))
// или из ответа, который у тебя уже есть:
const material = new Material(await fetch(asset('./hologram.mat'), { useOnce: true }))

material.set(key: string, value): this      // чейнится
material.uniforms[key] = value              // то же, в стиле свойства
material.uniforms.speed                     // чтение возвращает последнее ТВОЁ значение (нативного чтения нет)
```

Каждый тип значения приводится к конкретному вызову юниформа:

JS-значение	Приводится к
строка `'#rrggbb'`	RGB-цвет
строка `'#rrggbbaa'`	RGBA-цвет
`number`	float
`boolean`	bool
`Texture`	сэмплер (учитывает `wrapS`/`wrapT` текстуры)
`number[]` / `Float32Array`	массив float / вектор
`null`	очищает юниформ (текстура сбрасывается в 1×1 белую)

> **NOTE**
> строки цвета здесь должны быть `#`-hex — более широкие формы `ColorInput` (`rgba(…)`, упакованные int) понимает только сеттер `color`, не `set()`. И обёртка `Vec3`/`Quat` **не** приводится — разверни её в обычный массив: `material.set('dir', [...v])`.

Окклюзионный материал сцены

`scene.occlusionMaterial` отдаёт окклюзионный материал сцены как обычный `Material`, чтобы ты задавал на нём юниформы — см. Scene (гейтится хостом: недоступен в веб-вьювере).

Подводные камни

TS

```
// ✗ ждать, что material.color сработает на кастомном шейдере
customMat.color = '#f00'                  // no-op: SDK не знает имя твоего юниформа
// ✓ адресуй юниформ напрямую
customMat.set('baseColor', '#ff0000')

// ✗ передавать Vec3 как значение юниформа
mat.set('lightDir', dir)                  // не Array — приводится к NULL, не к vec3
// ✓
mat.set('lightDir', [...dir])
```

Смотрите также

- Mesh — присвоение материалов (`mesh.material`, `setMaterial(index)`).
- Texture — загрузка карт, режимы wrap, видео-текстуры.
- Light — на что откликаются материалы `lit`.
- Scene — `occlusionMaterial`, `setMaterialGlobalParameter` для кастомных шейдеров.

4.3.9 Texture

GPU-текстура для **3D-материалов**, декодируемая нативно 3D-движком. Не взаимозаменяема с `Texture2D` — та питает 2D-движок спрайтов; эта питает юниформы `Material`. Грузи одно и то же изображение в оба типа, если проект использует его в обоих мирах.

Обзор

TS

```
const tex = await Texture.load(asset('./crate.jpg'))
const crate = Mesh.box({ material: Material.lit({ map: tex }) })
scene.add(crate)
```

Загрузка

```ts
Texture.load(source: string | FetchResponse | File): Promise<Texture>
```

Принимает URL-строку (ассет/удалённый), уже скачанный `FetchResponse` или `File` (напр. из `openFilePicker`). Загрузки по URL отклоняются при HTTP-ошибках (статус ≥ 400) и ошибке декодирования.

```ts
Texture.fromCanvas(canvas: Canvas): Texture
```

Запекает поверхность `Canvas` в текстуру — синхронно, без декодирования. Текстура кэшируется на канвасе, поэтому повторные вызовы (или несколько материалов) делят одну GPU-текстуру, а `canvas.update()` перезаливает её. Сам ты это зовёшь редко: присвоение `Canvas` в `material.map` делает это за тебя.

Свойства

```ts
tex.width, tex.height     // пиксельные размеры (только чтение)
tex.wrapS = 1             // режим wrap по горизонтали (U)
tex.wrapT = 1             // режим wrap по вертикали (V)
```

`wrapS` / `wrapT` применяются при присвоении текстуры юниформу материала — задавай их *до* `material.set('baseColorMap', tex)`.

> **NOTE**
>
> у `Texture` нет `dispose()` — загруженную текстуру нельзя освободить явно, она живёт до разбора 3D-движка. Грузи большие текстуры один раз и переиспользуй.

Видео-текстуры

Текстуру нельзя создать из видеофайла напрямую — проигрывай его `VideoPlayer` и сэмпли его текстуру:

```ts
const player = new VideoPlayer(asset('./clip.mp4'))
const screen = Mesh.plane({ material: Material.video(player.texture) })
```

`player.texture` сообщает `width`/`height` равными `0` (размер видео неизвестен в момент создания текстуры). См. Медиа и `Material.video`.

Подводные камни

```ts
// ✗ скармливать 3D Texture в Sprite (или Texture2D в Material)
new Sprite({ texture: await Texture.load(url) })       // не тот движок — используй Texture2D.load
// ✓ каждый движок грузит свой тип
new Sprite({ texture: await Texture2D.load(url) })
```

Смотрите также

- Material — где текстуры потребляются (`map`, юниформы).
- Texture2D — аналог для 2D-движка.
- Медиа — `VideoPlayer.texture`.
- Canvas — рисованные поверхности как текстуры.

4.3.10 Light

Узел-источник света. Модель 3D-освещения SDK намеренно мала: каждая `Scene` получает **окружающий** свет на основе изображения (IBL) по умолчанию — настраивай его опцией сцены `environmentIntensity` — а `Light.sun()` добавляет единственный **прямой** свет: направленное солнце, которое ещё и отбрасывает тени. Фабрик точечного или прожекторного света нет.

`Light` расширяет `Node`; добавляй его в сцену как любой узел.

Обзор

```ts
const scene = new Scene()
scene.add(Light.sun({ direction: [1, -2, -1], intensity: 80000, shadowsQuality: 2 }))
scene.add(Mesh.box({ material: Material.lit({ color: '#e33' }), position: [0, 0.5, 0] }))
scene.open()
```

`Light.sun()`

```ts
Light.sun(options?: {
  direction?: Vec3Like          // куда свет ИДЁТ; по умолчанию [0.548, -0.474, -0.689] (угол ключевого света)
  intensity?: number            // сила свечения; по умолчанию 100000
  color?: ColorInput            // по умолчанию белый (0xffffff)
  shadowsQuality?: 0 | 1 | 2 | 3   // по умолчанию 1 — см. таблицу
}): Light
```

`shadowsQuality`	**Результат**
`0`	без теней
`1`	стандарт — shadow map 1024, жёсткие края PCF
`2`	высокое — 2048, стабильные, мягкие края с contact-hardening
`3`	лучшее — 4096, стабильные, мягкие тени PCSS

Более высокое качество стоит больше GPU и памяти.

> **NOTE**
> направление, интенсивность, цвет и качество теней — **опции времени создания** — `Light` не предоставляет для них сеттеров. Чтобы изменить солнце, `destroy()` его и создай новое.

> **NOTE**
> только меши с флагом `castShadows` отбрасывают тени, и только поверхности `receiveShadows` их показывают — см. Mesh. Плоскость `Material.shadow()` делает прозрачный ловец теней (AR-классика).

IBL против солнца

- **IBL (опция сцены)** — мягкий всенаправленный окружающий; включён по умолчанию, сила `environmentIntensity` (по умолчанию 20000), отключается через `ibl: false`. Без теней.
- `Light.sun()` — направленный ключевой свет; даёт поверхностям направление затенения и отбрасывает тени.

Типичная освещённая сцена использует оба. С `ibl: false` и без солнца поверхности `Material.lit()` рендерятся чёрными — используй `Material.unlit()`, если хочешь вообще без освещения.

Смотрите также

- Scene — опции `ibl` / `environmentIntensity`.
- Material — lit против unlit, материал-ловец теней.
- Mesh — `castShadows` / `receiveShadows` на узел.
- Node — унаследованная поверхность узла (`destroy`, членство в сцене).

4.3.11 3D-физика — Shape, Physics, Trigger, CharacterController

Физика на Jolt как аспекты узла. `Shape` — геометрия коллизии; в паре с `Physics` она даёт твёрдое тело, с `Trigger` — сенсорную зону, с `CharacterController` — кинематическую капсулу игрока. `Shape` сам по себе уже делает узел кликабельным указателем (`click` / `touchstart`).

Обзор

```ts
Physics.configure({ gravity: [0, -9.81, 0] })    // один раз, до создания тел

const ground = Mesh.plane({ material: mat, normal: [0, 1, 0], scale: 20 })
  .aspect(Shape, { box: [10, 0.1, 10] })
  .aspect(Physics, { motion: 'static' })

const crate = Mesh.box({ material: mat, position: [0, 4, 0] })
  .aspect(Shape, {})                             // авто-бокс из меша
  .aspect(Physics, { mass: 2 })                  // по умолчанию dynamic

crate.physics.applyImpulse([0, 6, 0])
crate.addEventListener('enter', other => console.log('hit', other.name))
scene.add(ground, crate)
```

Гейтинг хоста — `Physics.supported`

```ts
Physics.supported   // статик-булево — есть ли у этой сборки хоста 3D-физика
```

На сборке без физики каждый API этой страницы **молча делает no-op**: аспекты прикрепляются без ошибки, `id` остаётся `0`, `velocity` читается `[0,0,0]`, рейкасты промахиваются, события не срабатывают. Оберни зависящий от физики геймплей в `Physics.supported`.

Настройка мира (статически)

```ts
Physics.configure(config?: {
  gravity?: Vec3Like        // мировые единицы/с², Y вверх (вниз отрицательный); по умолчанию
[0, -9.81, 0]
  maxBodies?: number        // ёмкость мира; по умолчанию 4096
}): void

Physics.interpolation = true   // статик get/set; по умолчанию включено
```

- `configure()` должен выполниться **до существования любого тела** — `maxBodies` меняет размер мира и после игнорируется.
- `interpolation` сглаживает отрисованные трансформы тел между фиксированными шагами по 60 Гц. Выключи его, чтобы сэкономить покадровые записи трансформа при множестве движущихся тел; тогда они движутся дискретными шагами.

`Shape` — геометрия коллизии (и кликабельность)

```ts
node.aspect(Shape, {
  box?: Vec3Like                                    // ПОЛУ-экстенты [hx, hy, hz], мировые единицы
  sphere?: number                                   // радиус
  cylinder?: { halfHeight: number, radius: number } // выровнен по Y
  capsule?: { halfHeight: number, radius: number }  // выровнен по Y; halfHeight = цилиндрическая секция
  raycast?: boolean                                 // лучи указателя могут его задеть; по умолчанию true
})
// {} = авто: бокс из AABB меша × мировой масштаб
```

Чистая геометрия — `Shape` сам по себе не твёрдый и не сенсор. Выбери ровно один вид или передай `{}`, чтобы авто-подогнать бокс к `geometry` узла (его отрисованный размер, с учётом мирового масштаба; узлы без геометрии откатываются к полу-экстентам 0.5, всё равно умноженным на мировой масштаб узла).

> **NOTE**
>
> явные размеры — **мировые полу-экстенты**, не масштабируются узлом — `box: [0.5, 0.5, 0.5]` это коллайдер 1×1×1 при любом `scale` узла. `Mesh.box()` по умолчанию меш 1×1×1, поэтому его соответствующий коллайдер — `box: [0.5, 0.5, 0.5]` (или просто `{}`).

Кликабельность: `Shape` создаёт лёгкое тело только для пикинга (ни с чем не сталкивается), поэтому любой узел с формой получает `click` / `touchstart` — см. события указателя. Прикрепление `Physics` / `Trigger` заменяет это тело реальным (по-прежнему кликабельным). Поставь `raycast: false`, чтобы отказаться от пикинга (это также прячет его от `Physics.raycast`).

> **NOTE**
>
> тело только-для-пикинга — **статический снимок** на момент прикрепления. Для *движущегося* кликабельного объекта дай ему тело `Physics` (которое следует за симуляцией) — не полагайся на голое тело-пикер.

`Physics` — твёрдое тело

```ts
node.aspect(Physics, {
  motion?: 'static' | 'dynamic' | 'kinematic'   // по умолчанию 'dynamic'
  mass?: number                                 // кг, только dynamic; по умолчанию 1
})

node.physics.id                          // нативный id тела Jolt; 0 = не прикреплено / нет поддержки физики
node.physics.velocity                    // Vec3, мировые единицы/с — get (свежая копия) / set
node.physics.applyImpulse(v): this       // мгновенный импульс (кг·м/с), будит тело
node.physics.moveTo(p): this             // привести кинематическое тело (или телепорт) к мировой позиции
```

Порядок прикрепления важен: `Physics` требует, чтобы `Shape` уже был на узле — прикрепление его первым бросает `"Physics requires a Shape aspect"`. То же для `Trigger` и `CharacterController`.

Физика владеет трансформом `dynamic`-тела. Не задавай `node.position` каждый кадр — управляй им через `velocity` / `applyImpulse`; кинематическими телами — через `moveTo`. *Чтение* `node.position` / `node.worldPosition` всегда корректно (нативный слой пересинхронизирует). См. соглашения.

Типы движения: `static` никогда не двигается (стены, полы); `dynamic` полностью симулируется (гравитация, столкновения, масса); `kinematic` управляется скриптом через `moveTo` и толкает динамические тела, сам не толкаясь в ответ.

События контакта — `'enter'` / `'exit'`

```ts
node.addEventListener('enter', (other: Node) => { /* контакт / перекрытие начались */ })
node.addEventListener('exit',  (other: Node) => { /* контакт / перекрытие закончились  */ })
```

Срабатывают на **обоих** узлах контакта (твёрдое-твёрдое) или перекрытия (тело-триггер), каждый получает другой узел как аргумент. Узлу нужен `Shape` плюс тело `Physics` или `Trigger`.

`Trigger` — сенсорная зона

```ts
node.aspect(Trigger)          // требует Shape; без опций

node.trigger.id               // нативный id тела; 0 = нет поддержки физики
node.trigger.moveTo(p): this  // переместить зону в мировую точку
```

Создаёт **статическое сенсорное** тело из `Shape` узла: физические тела, перекрывающие его, эмитят `'enter'` / `'exit'` узла вместо столкновения. Триггер никогда не блокирует движение и остаётся кликабельным указателем.

TS

```
const goal = Mesh.box({ position: [0, 1, -6] })
  .aspect(Shape, { box: [1, 1, 0.2] })
  .aspect(Trigger)
goal.addEventListener('enter', other => win(other))
```

`CharacterController` — кинематический игрок

Jolt `CharacterVirtual` — контроллер collide-and-slide, **не** твёрдое тело: без отскоков и опрокидывания, чёткое управление, встроенный шаг по лестницам и обработка склонов. Требует `Shape` (рекомендуется капсула) и не использует `Physics`.

TS

```
node.aspect(CharacterController, {
  speed?: number       // горизонтальная скорость движения, мировые единицы/с; по умолчанию 5
  jumpSpeed?: number   // скорость отрыва прыжка, мировые единицы/с; по умолчанию 7
  gravity?: number     // гравитация только для персонажа, мировые единицы/с²; по умолчанию
-20 (отдельно от мировой)
  maxSlope?: number    // макс. проходимый склон в градусах; по умолчанию 45 — задаётся при
прикреплении, не позже
})

node.controller.move(x: number, z: number): void   // горизонтальное намерение, каждое в [-1,
1]; ЛИПКОЕ — шли 0, чтобы остановиться
node.controller.jump(): void                       // ставится в очередь; потребляется в
следующем кадре, если на земле
node.controller.teleport(x, y, z): this            // мировая позиция; сбрасывает
вертикальную скорость
node.controller.grounded                           // булево — стоит на проходимой земле
node.controller.groundState                        // 0 на земле, 1 крутой склон, 2 касается
неопираемого, 3 в воздухе
node.controller.velocity                           // Vec3, мировые единицы/с (свежая копия)
node.controller.id                                 // нативный id персонажа; 0 = не
прикреплён / нет поддержки
```

Ты подаёшь горизонтальное намерение + прыжок; контроллер каждый кадр интегрирует свою гравитацию, а Jolt разрешает столкновения с миром. Он обновляется в фазе аспектов EARLY, поэтому ввод, заданный в этом кадре, потребляется шагом этого кадра.

```ts
const hero = Mesh.cylinder({ material: mat, position: [0, 1, 0] })
  .aspect(Shape, { capsule: { halfHeight: 0.6, radius: 0.3 } })
  .aspect(CharacterController, { speed: 6, jumpSpeed: 8 })

setLoop(() => {
  const x = (Input.key('KeyD') ? 1 : 0) - (Input.key('KeyA') ? 1 : 0)
  const z = (Input.key('KeyS') ? 1 : 0) - (Input.key('KeyW') ? 1 : 0)
  hero.controller.move(x, z)
  if (Input.key('Space')) hero.controller.jump()
})
```

> **NOTE**
>
> контроллер сталкивается с твёрдыми телами, но **проходит сквозь триггеры**, и (v1) сам не детектируется триггерами и не кликабелен указателем, пока активен.

Рейкасты

```ts
Physics.raycast(origin: Vec3Like, dir: Vec3Like, maxDist = 1000): RayHit | null

interface RayHit {
  node: Node | null    // задетый узел (null, если он больше не зарегистрирован)
  point: Vec3          // точка попадания в мировом пространстве
  normal: Vec3          // нормаль поверхности в мировом пространстве
  fraction: number     // 0..1 вдоль луча (дистанция попадания = fraction × maxDist × |dir|)
}
```

Задевает ближайшее **кликабельное** тело (любой `Shape` с `raycast`, оставленным `true` — включая тела только-для-пикинга и триггеры). Сочетай с `camera.getRay` для выбора по тапу:

```ts
scene.addEventListener('click', ev => {
  const ray = scene.camera.getRay(ev.clientX, ev.clientY)
  const hit = Physics.raycast(ray.origin, ray.dir, 100)
  if (hit?.node) select(hit.node)
})
```

Подводные камни

```ts
// ✗ порядок прикрепления — Physics/Trigger/CharacterController до Shape бросает
node.aspect(Physics).aspect(Shape, {})
// ✓ сначала Shape
node.aspect(Shape, {}).aspect(Physics)

// ✗ запись трансформа динамического тела — им владеет физика
setLoop(() => { crate.y += 0.1 })
// ✓ управляй телом
crate.physics.velocity = [0, 3, 0]

// ✗ полные экстенты там, где ждут полу-экстенты — коллайдер вдвое больше меша
crate.aspect(Shape, { box: [1, 1, 1] })      // коллайдер 2×2×2 вокруг меша-бокса 1×1×1
// ✓
crate.aspect(Shape, { box: [0.5, 0.5, 0.5] })   // или {} для авто-подгонки
```

Смотрите также

- Аспекты — attach/get/has/removeAspect, фазы `update`.
- События указателя — что включает тело-пикер `Shape`.
- Node — трансформ, события.
- Ray, Plane и Noise — лучи камеры для `Physics.raycast`.
- Соглашения — «физика владеет динамическими трансформами», гейтинг хоста.

4.3.12 Particles

GPU-система частиц как узел. Добавь её в сцену — и она стримит частицы: дым, искры, пыль. Расширяет `Node`: двигай/вращай узел, чтобы двигать эмиттер (частицы симулируются в локальном пространстве эмиттера). Каждая опция также живой сеттер, поэтому `rate`, `color`, `gravity`, … можно менять в рантайме.

Частицы рисуются **материалом частиц по умолчанию** (`Material.particles()` — точечные спрайты, повёрнутые к камере; мягкие круглые точки, пока не дашь текстуру), если не передать свой скомпилированный шейдер через `material`.

Обзор

```ts
const sparks = new Particles({
  blend: 'add',                                                    // аддитивный — искры светятся
  rate: 80,                                                        // частиц/секунду
  shape: { type: 'point', v: [0, 0, 0] },
  startVelocity: { dir: [0, 1, 0], speed: { min: 2, max: 4 }, spread: 0.3 },
  gravity: 9.8,                                                    // положительное тянет ВНИЗ
  lifetime: { min: 0.6, max: 1.2 },
  size: { min: 0.05, max: 0.12 },
  color: colorCurve('#ffcc44').via(0.7, '#ffffff').to('#00000000'),  // затухание за время жизни
})
sparks.position = [0, 1, 0]
scene.add(sparks)

crate.addEventListener('click', () => sparks.spawn(40))         // всплеск поверх rate
```

Текстурный флипбук-эффект (лист спрайтов с кадрами):

```ts
const fire = new Particles({
  map: await Texture.load(asset('./flame-sheet.png')),
  sheet: 8,                    // 8×8 кадров; автоцикл один раз за жизнь частицы
  blend: 'add',
  emissive: 1.5,               // дополнительное свечение
  rate: 2,
  lifetime: 3,
  size: { min: 1.2, max: 2 },
})
```

По умолчанию каждая частица начинает цикл со **случайного кадра** — циклы вроде огня или дыма остаются рассинхронизированными. Для листа, который рассказывает историю по порядку (секвенция взрыва), начинай с 0; для неквадратного листа задай сетку как `[cols, rows]`:

TS

```
const boom = new Particles({
  map: await Texture.load(asset('./explosion.png')),
  sheet: [8, 4],               // 8 столбцов × 4 строки = 32 кадра…
  startFrame: 0,               // …проигрываются по порядку, один раз за жизнь частицы
  rate: 0,
  lifetime: 0.8,
})
boom.spawn(1)
```

Опции

TS

```
new Particles(options?: {
  // --- внешний вид (материал по умолчанию; см. Material.particles в material.md) ---
  map?: Texture                              // текстура спрайта; не задана = мягкая круглая точка
  sheet?: number | [cols, rows]              // сетка флипбука: N = N×N, или явно cols×rows (по умолчанию 1)
  startFrame?: number | 'random'             // с чего начинается дефолтный цикл флипбука (по умолчанию 'random')
  emissive?: number                          // дополнительная яркость для свечения (по умолчанию 0)
  blend?: 'alpha' | 'add'                    // смешивание: дым/пыль vs огонь/искры (по умолчанию 'alpha')
  soft?: boolean                             // радиальное затухание; по умолчанию: вкл без map, выкл с ней
  render?: 'point' | 'quad' | 'stretch'      // точечные спрайты (по умолчанию) / настоящие квады / квады, растянутые по скорости
  stretch?: number                           // ('stretch') доп. длина на единицу скорости (по умолчанию 0.05)
  material?: Material                        // свой скомпилированный шейдер — опции вида выше игнорируются

  // --- эмиттер ---
  maxParticles?: number                      // ёмкость пула частиц (по умолчанию 1000)
  rate?: number                              // частиц в секунду
  space?: 'local' | 'world'                  // 'world': частицы остаются там, где родились, пока эмиттер движется (по умолчанию 'local')
  inheritVelocity?: number                   // доля скорости эмиттера, передаваемая частицам при спавне
  rateOverDistance?: number                  // доп. частицы на мировую единицу ДВИЖЕНИЯ эмиттера (см. ниже)
  shape?: { type: 'point', v: Vec3Like }     // спавн со смещением (относительно эмиттера)
        | { type: 'box', min: Vec3Like, max: Vec3Like }        // случайно внутри бокса
        | { type: 'circle', center: Vec3Like, radius: number } // случайно на диске в плоскости XZ
  startVelocity?: Vec3Like | VelocityValue // см. ниже
  gravity?: number                           // мировые единицы/с² — ПОЛОЖИТЕЛЬНОЕ ТЯНЕТ ВНИЗ (сахар для acceleration: [0, -g, 0])
  acceleration?: Vec3Like                    // постоянный вектор ускорения (гравитация, ветер, восходящий поток…)
  drag?: number                              // затухание скорости — выше тормозит частицы быстрее
```

У всего есть дефолт — `new Particles()` уже эмитит мягкие белые точки. Поля типа `number | { min, max }` принимают плоское значение («всегда это») или диапазон («случайно на частицу»).

`startVelocity`

Простой `Vec3Like` — фиксированная скорость запуска (её длина — это speed); объект выбирает режим направления плюс опциональную рандомизацию:

```ts
startVelocity: {
  dir?: Vec3Like                                    // запуск вдоль этого направления, или…
  from?: Vec3Like                                   // …прочь от этой точки, или…
  to?: Vec3Like                                     // …к этой точке
  speed?: number | { min: number, max: number }     // масштабирует направление; диапазон =
на частицу
  spread?: number                                   // полуугол конуса (радианы) — дрожание
в его пределах
  randomizeAngle?: { min, max }                     // асимметричное дрожание: два угла
(x/y), радианы
}
```

```ts
sparks.startVelocity = { dir: [0, 1, 0], speed: { min: 2, max: 5 }, spread: 0.25 }
```

Движущиеся эмиттеры — шлейфы

По умолчанию вся система едет на своём узле: подвинь эмиттер — и каждая живая частица сдвинется вместе с ним (факел, который несут по уровню). Для **шлейфов** — дым из-под колёс, грязь, кильватер, выхлоп, пыль от шагов — нужно обратное: частицы остаются в мире там, где родились, а эмиттер едет дальше. Это `space: 'world'` плюс два компаньона:

- **`rateOverDistance`** эмитит на мировую единицу *пройденного пути* (поверх `rate`), с точками спавна, равномерно распределёнными вдоль пути — плотность шлейфа не зависит от скорости, без покадровых комков.
- **`inheritVelocity`** отдаёт каждой частице долю скорости эмиттера при спавне, чтобы дым выглядел *сорванным* с движущейся машины, а не возникшим из ниоткуда.

```ts
// дым дрифта у колеса: прикрепи к узлу колеса и просто езжай
const smoke = new Particles({
  map: smokeTex,
  sheet: 8,
  space: 'world',
  rate: 0,
  rateOverDistance: 12,            // 12 клубов на метр дрифта, на любой скорости
  inheritVelocity: 0.4,            // уносится вместе, потом остаётся позади
  shape: { type: 'circle', center: [0, 0, 0], radius: 0.15 },
  startVelocity: { dir: [0, 1, 0], speed: 0.5, spread: 0.6 },
  lifetime: { min: 1, max: 2 },
  size: curve({ min: 0.4, max: 0.7 }).to(2),
  opacity: curve(0.5).fade(0.1, 0.5),
})
wheel.add(smoke)

// модулируй количество из геймплея — оба живые сеттеры:
smoke.rateOverDistance = 12 * driftAmount
```

Переключение `space` в рантайме сбрасывает живые частицы (они хранятся в старом пространстве). Оба компаньона имеют смысл только со `space: 'world'` — в локальном пространстве эмиттер никогда не «движется» относительно своих частиц.

Режимы отрисовки

- `'point'` (по умолчанию) — GPU-спрайты точек: самый дешёвый режим на каждом бэкенде, но драйверы ограничивают максимальный экранный размер (экстремальные крупные планы упираются в потолок), а вращение обрезает углы спрайта.
- `'quad'` — настоящие квады, повёрнутые к камере: без ограничения размера, честное геометрическое вращение. Чуть больше CPU/загрузки (4 вершины на частицу).
- `'stretch'` — квады, растянутые вдоль скорости каждой частицы, в проекции на экран: искры, дождь, полосы скорости. `stretch` добавляет `stretch · |velocity|` мировых единиц длины поверх `size`.

```ts
const sparks = new Particles({
  render: 'stretch',
  stretch: 0.08,                    // искра на 4 ед/с рисуется ~на 0.3 м.е. длиннее своей ширины
  blend: 'add',
  startVelocity: { dir: [0, 1, 0], speed: { min: 2, max: 5 }, spread: 0.4 },
  gravity: 9.8,
  size: 0.05,
  lifetime: { min: 0.4, max: 0.9 },
})
```

Ленты — `Trail`

`Trail` — не эмиттер, а лента, следующая за своим узлом по миру: дуга взмаха меча, следы шин, шлейфы ракет. Двигай узел (или что угодно, чему он дочерний): лента кладёт точку через каждые `minDistance` движения, каждая точка живёт `time` секунд, а головой лента остаётся приклеенной к узлу.

```ts
const slash = new Trail({
  time: 0.25,                       // как долго лента держится
  width: curve(0.3).to(0),          // полная ширина у клинка, сужается в ничто к хвосту
  color: '#8df0ff',
  blend: 'add',
})
swordTip.add(slash)
// взмахни мечом — дуга появится сама; между взмахами перестань класть точки:
slash.emitting = false              // существующие точки всё равно стареют, так что дуга гаснет естественно
```

`width` / `opacity` / `color` принимают те же кривые, что и `Particles`, вычисляемые по жизни каждой точки — то есть вдоль ленты от головы (свежая) к хвосту (умирающая). Дефолтная `opacity` уже гасит хвост. Живые сеттеры: `time`, `width`, `opacity`, `color`, `minDistance`, `emitting`. Опции также принимают общие поля вида (`map`/`sheet`/`emissive`/`blend`/`soft`); ось u у `map` идёт вдоль ленты, голова → хвост.

Кривые за время жизни частицы — `curve()` / `colorCurve()`

`size`, `rotation`, `opacity`, `frame` и `color` могут анимироваться за время жизни каждой частицы. Кривая читается как путь: **`.from(v)`** при рождении → точки **`.via(t, v)`** → **`.to(v)`** при смерти; `t` идёт `0..1` за жизнь частицы.

```ts
curve(base?: number | { min, max },        // значение частицы (диапазон = случайно на частицу)
      mode?: 'multiply' | 'add')           // кривая умножает базу (по умолчанию) или прибавляет к ней
  .from(v)                                 // значение при t = 0 — должно идти первым
  .via(t, v)                               // значение при t (по возрастанию)
  .to(v)                                   // значение при t = 1
  .fade(in?, out?)                         // классический конверт: рост за `in`, спад за последние `out`

colorCurve(base?: ColorInput | { min, max })   // всегда умножает базовый цвет
  .from(c).via(t, c).to(c)
```

Любое значение стопа может быть диапазоном `{ min, max }` — рандомизируется на частицу. Цепочка без `.from` начинается с нейтрального значения (1 для multiply, 0 для add); без `.to` держит последнее значение.

```ts
sparks.size = curve(0.1).to(4)                          // вырасти до 4× к концу жизни
sparks.size = curve(0.3).to(0)                          // сжаться в ничто
sparks.opacity = curve(0.3).fade(0.15, 0.4)             // = .from(0).via(0.15,1).via(0.6,1).to(0)
sparks.rotation = curve({ min: 0, max: 6.28 }, 'add').to({ min: -2, max: 2 })   // вращение ±2 рад за жизнь
sparks.color = colorCurve('#ffaa33').via(0.7, '#ffffff').to('#000000')          // затухание в чёрный
```

`colorCurve` умножает базу, поэтому его главное применение — затухание яркости/альфы к смерти. Держи стопов мало (максимум 8), а `t` — по возрастанию: они запекаются в короткую нативную кривую, а не в произвольный сплайн; невалидная кривая отклоняется с предупреждением в консоли, прежняя остаётся.

Живые сеттеры и методы

```ts
sparks.spawn(count): this      // эмитировать всплеск прямо сейчас, поверх `rate` — чейнится
sparks.material                // get/set — сменить материал отрисовки на лету

// живые сеттеры только-для-записи (те же типы, что опции):
sparks.rate = 120
sparks.space = 'world'                 // переключение сбрасывает живые частицы
sparks.inheritVelocity = 0.5
sparks.rateOverDistance = 12
sparks.shape = { type: 'box', min: [-1, 0, -1], max: [1, 0, 1] }
sparks.startVelocity = [0, 3, 0]
sparks.gravity = 9.8                   // положительное = вниз
sparks.acceleration = [0.5, 0, 0]      // ветер
sparks.drag = 0.5
sparks.lifetime = { min: 0.5, max: 1 }
sparks.color = '#88ccff'
sparks.size = curve(0.1).to(4)
sparks.rotation = { min: 0, max: 6.28 }
sparks.noise = { strength: 2 }         // или null, чтобы отключить

sparks.opacity = curve().fade(0.2)     // появление и затухание
sparks.frame = curve({ min: 0, max: 63 }, 'add').to(128)  // флипбук: 2 цикла за жизнь

sparks.custom[0] = 0.2                 // сырые слоты параметров кривой 0..3;
size/rotation/opacity/frame — алиасы 0–3
```

Жизненный цикл: система эмитит непрерывно с конструирования; `sparks.destroy()` (из `Node`) убирает её и освобождает её GPU-буферы. Для одноразового эффекта поставь `rate: 0` и вызывай `spawn(n)`.

Подводные камни

```ts
// ✗ покадровый spawn() для имитации частоты эмиссии
setLoop(() => sparks.spawn(1))
// ✓ для этого есть rate
sparks.rate = 60

// ✗ ждать, что curve(2).to(4) анимирует 2 → 4
sparks.size = curve(2).to(4)         // база 2 × кривая(1 → 4): анимирует 2 → 8
// ✓ база — значение частицы; сам путь клади в кривую
sparks.size = curve().from(2).to(4)  // анимирует 2 → 4
```

Смотрите также

- Material — `Material.particles({...})` (материал отрисовки по умолчанию) и свои шейдеры
- Node — трансформ (эмиттер), `destroy()`
- Scene — где живёт система

4.3.13 Ray, Plane и Noise

Три небольших класса 3D-математики/утилит. `Ray` + `Plane` — набор для tap-to-place: преврати экранный тап в луч камеры, пересеки его с плоскостью земли, размести в точке попадания. Для рейкаста против *тел*, а не математической плоскости, используй `Physics.raycast` (он принимает `origin`/`dir` у `Ray`). `Noise` — нативный сэмплер Перлина/фрактала для рельефа, покачивания и органичного движения.

Обзор

```ts
const ground = new Plane([0, 1, 0], [0, 0, 0])       // Y-up плоскость через начало координат

scene.addEventListener('click', ev => {
  const ray = scene.camera.getRay(ev.clientX, ev.clientY)
  const t = ground.intersectRay(ray)
  if (t !== null) marker.position = ray.getPoint(t)  // размести в точке тапа по полу
})
```

`Ray`

```ts
new Ray(origin: Vec3Like, dir: Vec3Like)   // dir не обязан быть нормализован

ray.origin           // Vec3
ray.dir              // Vec3
ray.getPoint(t)      // Vec3 — origin + t·dir
```

Обычный источник — камера: `scene.camera.getRay(screenX, screenY)` строит луч пикинга от камеры через экранный пиксель (логические px — `ev.clientX/clientY` подставляются напрямую). См. Camera.

> **NOTE**
>
> параметр `getPoint(t)` — в единицах длины `dir`, мировые единицы только когда `dir` нормализован. `dir` лучей камеры берётся из направления взгляда хоста; нормализуй (`ray.dir.normalize()`) прежде, чем трактовать `t` как дистанцию в метрах.

`Plane`

Бесконечная геометрическая плоскость, хранимая как единичная `normal` + скалярное смещение `d` (`dot(normal, point)`).

```ts
new Plane(normal: Vec3Like, point: Vec3Like)       // normal нормализуется за тебя
Plane.fromPoints(a, b, c): Plane                   // через три точки (нормаль из обхода)
Plane.fromCoefficients(a, b, c, d): Plane          // из ax + by + cz = d

plane.normal                          // Vec3, единичной длины (только чтение)
plane.d                               // number (только чтение)

plane.intersectRay(ray): number | null    // параметр t ≥ 0 или null (параллельно / позади начала)
plane.intersectLine(ray): number | null   // то же, но попадания позади начала разрешены (t может быть < 0)
plane.signedDistance(point): number       // положительное на стороне нормали
plane.projectPoint(point): Vec3           // ближайшая точка на плоскости
```

Оба метода пересечения возвращают **параметр** луча `t`, не точку — передай его в `ray.getPoint(t)`.

TS

```
// держим персонажа над наклонной платформой
const surface = Plane.fromPoints(p0, p1, p2)
if (surface.signedDistance(hero.position) < 0) {
  hero.position = surface.projectPoint(hero.position)
}
```

`Noise`

Нативный сэмплер FastNoise Перлина/фрактала. Детерминирован: одни и те же координаты всегда возвращают одно значение, поэтому анимируй, двигая точку сэмпла (напр. используй время как одну ось).

TS

```
const noise = new Noise()

noise.get(x, y)          // сэмпл 2D-шума
noise.get3d(x, y, z)     // сэмпл 3D-шума

noise.frequency = 0.01           // get/set; по умолчанию 0.01 — пространственный масштаб
(выше = мельче)
noise.octaves = 4                // get/set; по умолчанию 1 — фрактальные слои (>1 добавляет
детали)
noise.fractalLacunarity = 2      // get/set; по умолчанию 2 — шаг частоты между октавами
noise.fractalGain = 0.5          // get/set; по умолчанию 0.5 — шаг амплитуды между октавами
```

TS

```
// мягкое покачивание и качание в покое
const n = new Noise()
n.frequency = 0.5
let t = 0
setLoop(dt => {
  t += dt
  ghost.y = 1 + n.get(t, 0) * 0.2
  ghost.eulerAngles = [0, n.get(t, 100) * 15, 0]
})
```

Подводные камни

```ts
// ✗ трактовать результат intersectRay как точку попадания
marker.position = ground.intersectRay(ray)      // это скаляр t
// ✓
const t = ground.intersectRay(ray)
if (t !== null) marker.position = ray.getPoint(t)

// ✗ ждать, что плоскость заблокирует Physics.raycast — Plane это математика, не тело
// ✓ дай полу Shape (+ Physics { motion: 'static' }), чтобы сделать его кликабельным/твёрдым
```

Смотрите также

- Физика и формы — `Physics.raycast` против реальных тел, `RayHit`.
- Camera — `scene.camera.getRay(x, y)`.
- События указателя — откуда берутся `clientX/clientY`.
- Vec3 и математика — семантика значений возвращаемых векторов.

4.3.14 Файлы сцен (defineScene)

Сцена как **данные**: один вызов `defineScene({...})`, по конвенции — дефолтный экспорт файла `.scene.ts`. Аргумент — простой литерал: узлы по именам, у каждого один блок-источник (`mesh` / `model` / `light`, или никакого = узел-группа), трансформ, опциональный `material`, `aspects: [use(Ctor, props), …]` и `children`. Всё это опускается до обычных вызовов SDK (фабрики `Mesh`, `node.aspect()`, `scene.add`), поэтому файл сцены выполняется на каждой платформе как любой другой код. Визуальный редактор сцен le.codes читает и пишет эти файлы; писать их руками так же нормально.

Обзор

TS

```ts
// city.scene.ts
import { Spinner } from './spinner'

export default defineScene({
  env: { skybox: '#10131a', bloom: true },
  camera: { position: [0, 6, 12], target: [0, 0, 0] },
  nodes: {
    ground: {
      mesh: { kind: 'box', size: [20, 1, 20] },
      material: { lit: { color: '#444444' } },
      position: [0, -0.5, 0],
      aspects: [use(Shape, { box: [10, 0.5, 10] }), use(Physics, { motion: 'static' })],
    },
    hero: {
      model: asset('./hero.glb'),
      position: [2, 0, 0],
      aspects: [use(Spinner, { speed: 2 })],
      children: {
        halo: { mesh: { kind: 'sphere', radius: 0.2 }, position: [0, 2, 0] },
      },
    },
    sun: { light: { kind: 'sun', shadowsQuality: 2 } },
  },
})
```

TS

```ts
// main.ts
import city from './city.scene'

const { scene, nodes } = await city.open()
nodes.hero.anim.play('idle')        // nodes.<имя> типизирован по своему блоку-источнику
nodes.ground.physics                // …и несёт свои use()-аспекты
```

Хэндл

`defineScene` возвращает `SceneHandle`:

Член	Что делает
`load()`	Инстанцировать сцену (async — модели загружаются). Идемпотентно: один экземпляр на хэндл.
`open()`	`load()` + сделать сцену активной. Возвращает те же `{ scene, nodes }`.
`def`	Литерал, который ты передал.

`nodes` — **плоская** карта имя → узел по всему дереву (включая `children`), каждая запись типизирована по своему источнику: `mesh` → `Mesh`, `model` → `Model`, `light` → `Light`, никакого → `Node` — с `use(...)`-аспектами, отражёнными в типе. Имена узлов должны быть уникальны в пределах файла сцены.

Записи узлов

Ключ	Описание
`mesh`	`{ kind: 'box' | 'sphere' | 'cylinder' | 'plane', …опции геометрии }`
`model`	URL GLB — `asset('./hero.glb')`
`light`	`{ kind: 'sun', …SunOptions }`
`make`	Поддерево, построенное кодом — `make(factoryFn, { …литеральные аргументы })` (ниже)
`prefab`	Другой файл сцены как переиспользуемая композиция — его импортированный хэндл (ниже)
`camera`	`{}` — этот узел И ЕСТЬ камера сцены (ниже)
`material`	Для `mesh`: `{ lit: { color?, roughness?, metallic? } }`, `{ unlit: { color? } }`, `{ shadow: color }` или разделяемый экземпляр `Material`
`position`, `eulerAngles`, `scale`, `visible`	Трансформ / видимость
`locked`	Только в редакторе: гизмо трансформа не нацеливается на этот узел (поля по-прежнему редактируются). Нет эффекта в рантайме
`castShadows`, `receiveShadows`	Флаги отрисовки меша
`aspects`	`[use(Ctor, props), …]` — порядок прикрепления важен (например, `Shape` до `Physics`)
`children`	Вложенные записи узлов, родитель — этот узел
`overrides`	Для источников model/prefab: переопределения трансформов **внутренних** узлов (ниже)

Не больше одного блока-источника на узел. Каждый узел дефа попадает в набор отрисовки сцены (членство и родительство — раздельны, см. node.md).

`overrides` — позирование внутренних узлов GLB

Узел-модель может переставить узлы *внутри* своего GLB, не трогая файл. Ключи — **пути частей**: имена узлов от корня модели вниз, соединённые `/`; каждое значение принимает `position`, `eulerAngles`, `scale`, `visible`:

```ts
robot: {
  model: asset('./robot.glb'),
  position: [2, 0, 0],
  overrides: {
    'Body/Head': { eulerAngles: [0, 30, 0] },
    'Body/ArmL': { position: [-0.7, 0.2, 0], visible: false },
  },
},
```

Сегмент — имя ребёнка; дубликаты среди соседей различаются как `name[i]`, безымянные узлы — как `[i]` (индекс внутри группы с одним именем). Редактор сцен пишет эти пути за тебя — раскрой узел-модель в дереве и потяни часть. Пути, которые больше не резолвятся (GLB изменился), молча пропускаются. Переопределения применяются один раз при загрузке, после инстанцирования модели; играющий анимационный клип перезапишет трансформы суставов, которыми он управляет.

Узлы-камеры

`camera: {}` делает узел камерой сцены: пока сцена запущена, вид каждый кадр следует за мировым трансформом узла (позиция + ориентация; камера смотрит вдоль локальной **−Z** узла). Поскольку это обычный узел, всё компонуется — делай его дочерним, анимируй сценарными аспектами (`FollowPath` на узле-камере — это облёт) или веди его из своего аспекта:

```ts
camera: { camera: {}, position: [0, 5, 10], eulerAngles: [-26, 0, 0] },
```

Побеждает первый узел-камера в порядке файла; узлы-камеры внутри `prefab` игнорируются (как и собственные блоки `camera:`/`env:` префаба). Старый верхнеуровневый блок `camera: { position, target }` продолжает работать как фолбэк, когда узла-камеры нет — он один раз задаёт вид, и дальше ничто его не отслеживает. В визуальном редакторе узел-камера рисуется маркером-фрустумом, собственная орбитальная камера редактора остаётся независимой, кнопка инспектора **Set from current view** ставит узел из твоей точки обзора, а последний узел-камеру нельзя удалить.

Empty (простые узлы-группы)

Узел **без** блока-источника — группа, невидимый трансформ. Редактор называет такие узлы **Empty** и рисует для них маленький маркер-крест из осей во время редактирования (маркеры

существуют только в редакторе и никогда не входят в запущенную сцену). Empty — рабочий материал no-code-сцен: цели `MoveTo`, вейпоинты `FollowPath` (дети узла-пути), объекты `LookAt` и просто папки для организации `children`.

Заголовок сгенерированного файла

Каждая запись из визуального редактора начинает файл двухстрочным комментарием о том, что файл управляется редактором. Ручные правки остаются первоклассными — значения вне литеральной грамматики сохраняются дословно и показываются в инспекторе только для чтения — но форматирование, порядок ключей и округление чисел (4 знака) нормализуются при следующей записи редактором. Написанные вручную файлы сцен получают заголовок при первой записи редактором.

`make(fn, args)` — узлы, построенные кодом

Когда поддерево — естественный результат функции (линия забора, винтовая лестница, процедурный кластер), сошлись на фабрику вместо ручного описания узлов:

```ts
// props.ts — обычная функция, возвращающая Node
export const buildFence = (args: { posts?: number, gap?: number }): Node => { … }

// city.scene.ts
fence: {
  make: make(buildFence, { posts: 6, gap: 1.2 }),
  position: [-5.5, 0, 3.5],
},
```

Фабрика выполняется после того, как существует каждый узел сцены (поэтому `ref()`-аргументы резолвятся, включая ссылки вперёд), и может быть async; возвращённое поддерево монтируется под узел дефа. Разделение владения — в этом суть:

- **Деф владеет трансформом.** Подвинуть/повернуть узел — обычная правка трансформа, фабрика для этого не перевызывается.
- **Аргументы владеют содержимым.** Аргументы должны быть литеральными данными (та же грамматика, что у пропсов аспектов, включая `ref()`); в редакторе сцен каждый аргумент — поле инспектора, и правка одного **перевызывает фабрику вживую** — без компиляции, ведь функция уже в запущенном бандле. `ref()`-аргумент также перевызывает при изменении узла, на который он ссылается.

Держи фабрики чистыми строителями: одни аргументы → одно поддерево, никаких побочных эффектов вне возвращаемых узлов. `make` намеренно зеркалит `use(Ctor, props)` — вызываемый по идентификатору, литеральные пропсы — именно это делает вызов редактируемым; вызов, записанный любым другим способом (вычисленные аргументы, инлайн-функция), сохраняется дословно, но показывается как «задано в коде».

`prefab` — сцена в сцене

Файл `.scene.ts` уже *есть* композиция, описанная данными — поэтому файлы сцен могут инстанцировать друг друга. Импортируй хэндл другой сцены и используй его как блок-источник:

```ts
import streetlamp from './streetlamp.scene'

lamp: {
  prefab: streetlamp,
  position: [4, 0, 2],
  overrides: { head: { visible: false } },
},
```

Каждый экземпляр строит узлы префаба заново под простым узлом-обёрткой — экземпляры независимы, а правка *файла* префаба меняет каждый экземпляр при следующей загрузке. Семантика:

- **Обёртка — обычный узел**: трансформ, `visible`, `locked`, `aspects`, `children` работают; внутренности экземпляра живут под ней.
- **`ref()` внутри префаба резолвятся локально для файла, на экземпляр** — аспекты и `make()`-фабрики префаба видят узлы своего экземпляра и никогда — узлы инстанцирующей сцены. Внутренности префаба не появляются в плоской карте `nodes` инстанцирующего хэндла (имена не могут сталкиваться между экземплярами).
- **`overrides` позирует внутренности** ровно как у GLB — та же грамматика путей частей, адресуемая именами узлов префаба (`'pole/bulb'`); вложенные модели внутри префаба сверлят глубже. В редакторе сцен раскрой узел-префаб в дереве и потяни часть — правка сохранится как переопределение.
- **`env`/`camera` префаба игнорируются** при инстанцировании — сценой владеет инстанцирующий файл.
- **Префабы могут вкладываться**; цикл импортов (сцена, инстанцирующая себя напрямую или транзитивно) обнаруживается при загрузке, и такой экземпляр пропускается с ошибкой в консоли.

`use(Ctor, props)`

Ссылается на аспект **по классу** — импорт и есть регистрация, а `props` проверяется типами по полям аспекта ровно как `node.aspect(Ctor, props)`. Поведение никогда не живёт в файле сцены: напиши свой аспект в обычном `.ts`-файле и прикрепи его через `use(...)`.

`ref(name)` — ссылки на узлы в пропсах аспектов

Пропс аспекта может указывать на другой узел сцены по имени:

```ts
class Road extends Aspect<'road'> {
  from: Node | null = null
  to: Node | null = null
}

// в файле сцены — `road` может ссылаться на узлы, объявленные ниже него
road:   { aspects: [use(Road, { from: ref('pointA'), to: ref('pointB') })] },
pointA: { position: [1, 0, 0] },
pointB: { position: [5, 0, 0] },
```

Сначала инстанцируются узлы, аспекты прикрепляются после, поэтому **порядок объявления не важен**. Маркеры `ref()` резолвятся в живые узлы при прикреплении (верхнеуровневые пропсы и один уровень массива вглубь — `waypoints: [ref('a'), ref('b')]` работает); неизвестное имя резолвится в `null`, поэтому типизируй такие поля `Node | null`. Только для файлов сцен — рукописный код передаёт узлы напрямую: `node.aspect(Road, { from: nodes.pointA })`. Редактор сцен показывает `ref`-поля как дропдаун узлов с кнопкой выбора во вьюпорте (достаточно аннотации типа `Node | null`; `editor: 'node'` в `static fields` тоже работает).

Метаданные инспектора (`static fields`)

Визуальный редактор выводит виджет для каждого публичного поля из его значения по умолчанию (число, булев переключатель, цвет `'#rrggbb'`, vec3 `[x, y, z]`, строка). Опциональный `static fields` на классе аспекта уточняет это — диапазоны, подписи, варианты дропдауна или скрытие поля:

```ts
class Spinner extends Aspect<'spinner'> {
  speed = 1
  mode: 'local' | 'world' = 'local'

  static fields: FieldMeta<Spinner> = {
    speed: { min: 0, max: 20, step: 0.1 },
    mode: { options: ['local', 'world'] },
  }
}
```

Аспекты-генераторы (`static editor`)

Аспект, который *выводит* содержимое сцены из своих пропсов — дорога между двумя узлами, забор вдоль пути — может подписаться на выполнение **во время редактирования сцены**:

```ts
class Road extends Aspect<'road'> {
  from: Node | null = null
  to: Node | null = null
  width = 2

  static editor = { rebuild: true }

  onAttach() { this.rebuild() }
  rebuild() {
    this.generated.clear()                      // идемпотентно: очистить, затем создать
    if (!this.from || !this.to) return
    // …создаём меши через this.generated.add(mesh)…
  }
}
```

Контракт:

- `rebuild()` перегенерирует вывод под **`this.generated`** — добавленный в сцену узел-контейнер, предоставляемый до запуска `onAttach`/`rebuild` (узлы, `add()`-нутые в него, попадают в набор отрисовки автоматически; `clear()` уничтожает предыдущий вывод). Предоставляется для прикреплений из файла сцены; прикреплённые вручную аспекты его не получают.
- **Режим Play**: ничего особенного — `onAttach` выполняется как обычно и сам вызывает `rebuild()`.
- **Режим редактирования**: класс инстанцируется по-настоящему (в отличие от обычных аспектов, которые остаются инертными данными) — `node`/`generated` установлены, `ref()`-пропсы разрезолвлены, `rebuild()` вызван — но **никогда `onAttach`** (никакой физики, таймеров и циклов во время редактирования). Затем редактор перезапускает `rebuild()` при каждом изменении пропса в инспекторе или движении узла, на который ссылается `ref()`-поле — потяни `pointA`, и дорога последует. Бросающий `rebuild()` логируется и пропускается; уронить редактор он не может.
- Сгенерированный вывод — **производный, никогда не сохраняется**: он не появляется ни в файле сцены, ни в дереве узлов редактора; файл хранит рецепт (запись аспекта), вьюпорт показывает результат.

Свои карточки инспектора (`static inspector`)

Для полного контроля над тем, как аспект выглядит в инспекторе редактора сцен, объяви `static inspector(ui, aspect)` — **immediate-mode**-функцию (думай imgui): она перезапускается при каждой правке или взаимодействии и описывает карточку вызовами `InspectorUI`. Без неё редактор показывает выведенные поля; `ui.auto()` эмитит те же поля,

поэтому своя карточка обычно начинается с него и добавляет строки статуса, кнопки или динамические дропдауны:

```ts
class Road extends Aspect<'road'> {
  from: Node | null = null
  to: Node | null = null
  width = 2
  static editor = { rebuild: true }

  static inspector(ui: InspectorUI, road: Road) {
    ui.auto()                                    // выведенные поля, как обычно
    if (road.from && road.to) ui.info(`${road.generated.children.length} pieces`)
    else ui.warn('Assign both endpoints')
    if (ui.button('Shuffle')) road.rebuild()     // true на прогоне с кликом — действуй прямо
тут
  }
}
```

Словарь `InspectorUI`: поля — `number` / `slider` / `text` / `color` / `switch` / `select(key, options)` / `vec2` / `vec3` / `vec4` / `node` (каждое эмитит виджет И возвращает текущее значение); действия и текст — `button(label)` (true на прогоне, потребившем клик), `header`, `info`, `warn`; и `auto(...keys)` для выведенных полей.

Правило привязки — один словарь, два времени жизни: поле, чей ключ — **объявленное поле класса**, привязано к документу (редактор сохраняет правки в файл сцены, с undo); любой другой ключ — временное **состояние редактора**, хранится на карточку и никуда не пишется (выбор клипа для превью, размер кисти). Опции `select` можно вычислять заново на каждом прогоне, так что живые данные дают дропдауны бесплатно.

Аргумент `aspect` — живой редакторный экземпляр для классов-генераторов (`static editor`) или превью-экземпляр (с установленным `node`, разрезолвленными ref'ами, никогда не прикреплённый) для обычных аспектов. Бросающий инспектор логируется и рендерит строку-предупреждение — уронить редактор он не может. Узлы-модели получают встроенную карточку Animation (клип / скорость / луп, Play/Stop), построенную на этом же протоколе.

Код только для редактора (`EDITOR` + `*.editor.ts`)

`EDITOR` — булева константа времени компиляции: `true` в бандлах редактора сцен, `false` в отгружаемых — и в продакшене компилятор сворачивает её в константу, поэтому ветки `if (EDITOR) { … }` (и всё, на что ссылаются только они) удаляются целиком. Оборачивай в неё отладочные хелперы, редакторные предупреждения или дорогую валидацию — по нулевой цене в поставке:

```ts
if (EDITOR && this.segments > 500) console.warn("road: very dense — consider fewer segments")
```

`*.editor.ts` — файловая форма того же: такие файлы компилируются и выполняются **только** в бандлах редактора (импортируются автоматически, после файла сцены) и невидимы для продакшен-компиляций — продакшен-сборка их даже не парсит, и определение точек входа их пропускает. Плагины редактора (окна, инструменты вьюпорта) живут в этих файлах — см. editor-plugins.md.

Режим редактирования (как редактор сцен запускает файл сцены)

Когда редактор сцен le.codes запускает бандл сцены, он выставляет флаг хоста до выполнения: источники инстанцируются по-настоящему (вьюпорт показывает сцену), но аспекты удерживаются как данные **без прикрепления** — никаких побочных эффектов `onAttach`, никаких тиков `update()`, физика инертна. Нажатие Play запускает проект нормально. Рукописный код никогда не видит этот режим.

4.3.15 Сценарные аспекты (no-code-поведения)

Пять аспектов, закрывающих базовые сценарии «пусть сцена ЧТО-ТО ДЕЛАЕТ» без написания кода: доехать до точки, пройти круг по вейпоинтам, вращаться, смотреть на цель, запустить GLB-анимацию. Они спроектированы для визуального редактора сцен (прикрепи из меню Aspects, привяжи цели выбором узлов во вьюпорте), но это обычные аспекты — `node.aspect(MoveTo, { target })` из рукописного кода работает так же.

Всё движение выполняется покадрово в режиме Play. Пока сцена *редактируется*, аспекты инертны — но аспекты движения рисуют свои пути редакторными линиями (синие для движения, янтарные для look-at), так что маршрут можно видеть и настраивать, не запуская сцену.

Обычный рецепт сочетает их с узлами **Empty** (простые узлы-группы — см. scene-files.md): поставь Empty туда, куда что-то должно попасть, и наведи на него аспект. На узле-камере те же аспекты становятся движениями камеры.

`MoveTo`

Движение от стартовой позиции узла к узлу-цели.

Поле	По умолчанию	Описание
`target`	`null`	Узел назначения (`ref('name')` в файлах сцен)
`duration`	`2`	Секунды на весь путь
`delay`	`0`	Секунды ожидания перед стартом
`mode`	`'once'`	`'once'` остановиться там · `'loop'` заново со стартовой точки · `'pingpong'` туда-обратно
`easing`	`'smooth'`	`'smooth'` ease-in-out · `'linear'` постоянная скорость

Цель отслеживается вживую — если она движется, путь изгибается к её новой позиции (а `once` держит узел прибитым к ней после прибытия).

TS

```
crate: {
  mesh: { kind: 'box', size: [1, 1, 1] },
  position: [0, 0.5, 0],
  aspects: [use(MoveTo, { target: ref('dropPoint'), duration: 3, mode: 'pingpong' })],
},
dropPoint: { position: [4, 0.5, -2] },   // Empty
```

`FollowPath`

Движение по **детям** узла-пути, в порядке файла сцены — добавляй по Empty на вейпоинт.

Поле	По умолчанию	Описание
`path`	`null`	Узел, чьи дети — вейпоинты
`duration`	`6`	Секунды на один полный проход
`delay`	`0`	Секунды ожидания перед стартом
`mode`	`'loop'`	`'loop'` замкнутый круг (последний → первый) · `'once'` остановиться на последней точке · `'pingpong'` туда-обратно по разомкнутому пути
`orient`	`true`	Поворачиваться по направлению движения
`forward`	`'-z'`	Какая локальная ось ведёт при ориентации

Скорость постоянна вдоль всего пути (длины сегментов измеряются вживую, поэтому перетаскивание вейпоинта корректно перетаймит проход). GLB-модели обычно смотрят в `+z` — переверни `forward`, если твоя модель едет задом наперёд.

```ts
bus: {
  model: asset('./bus.glb'),
  aspects: [use(FollowPath, { path: ref('route'), duration: 12, forward: 'z' })],
},
route: {
  position: [0, 0, 0],
  children: {
    stop1: { position: [0, 0, 0] },
    stop2: { position: [8, 0, 2] },
    stop3: { position: [5, 0, 8] },
  },
},
```

`Spin`

Непрерывное вращение вокруг локальной оси — `speed` градусов в секунду, `axis` `'x' | 'y' | 'z'` (по умолчанию `'y'`). Компонуется в пространстве кватернионов, поэтому корректен на заранее повёрнутых узлах.

`LookAt`

Держит узел повёрнутым к узлу-цели каждый кадр.

Поле	По умолчанию	Описание
`target`	`null`	Узел, на который смотреть
`forward`	`'-z'`	Какая локальная ось указывает на цель
`smoothing`	`0`	`0` — мгновенный снап; иначе секунды запаздывания поворота (больше = медленнее)

Турель, следящая за игроком, указатель или — на узле-камере — камера, которая держит движущегося героя в кадре, пока `FollowPath` ведёт кран.

`PlayAnimation`

Запускает GLB-клип при старте сцены. Прикрепляй к узлу с `model:`.

Поле	По умолчанию	Описание
`clip`	`''`	Имя клипа; пустое = первый клип модели
`loop`	`true`	Зациклить клип
`speed`	`1`	Скорость воспроизведения

В редакторе его карточка инспектора показывает клипы модели в дропдауне и умеет **Preview**/**Stop** клипа вживую во время редактирования. Для управления в рантайме (переключение клипов из своего кода, перемотка, события) используй `model.anim` напрямую — этот аспект только нажимает play при загрузке.

4.3.16 Плагины редактора (окна + инструменты вьюпорта)

Расширяй **редактор сцен** le.codes из кода проекта: оверлейные панели и инструменты вьюпорта, автоматизирующие рутинную сборку сцен (рассыпание пропов, связывание узлов, измерения). Регистрации живут в файлах `*.editor.ts` — они компилируются и выполняются **только** в бандлах редактора (файлы сцен), поэтому плагины стоят ноль байт в поставке.

TS

```
// scatter.editor.ts — только бандлы редактора
const state = { model: "", count: 5 }

registerEditorWindow("Scatter", (ui, editor) => {
  state.model = ui.asset("model")
  state.count = ui.slider("count", { min: 1, max: 20, step: 1, value: 5 })
  ui.toolButton("Paint", "scatter")
  ui.info(`${editor.nodes().length} nodes in the scene`)
})

registerEditorTool("scatter", {
  cursor: "copy",
  onViewportClick(hit, editor) {
    editor.transact(() => {                       // вся покраска = ОДИН шаг undo
      for (let i = 0; i < state.count; i++) {
        editor.addNode(editor.uniqueName("prop"), {
          model: state.model,
          position: [ hit.point[0] + Math.random(), hit.point[1], hit.point[2] +
Math.random() ],
          eulerAngles: [ 0, Math.random() * 360, 0 ],
        })
      }
    })
  },
})
```

Плагины пишут в документ, никогда — в живое состояние

Каждая запись через `editor` попадает в **файл сцены** через обычный путь коммита редактора — поэтому действие плагина отменяемо (один стек doc-патчей для людей и плагинов), выглядит обычным git-диффом и применяется к запущенному вьюпорту на самом дешёвом

живом уровне. Багованный плагин в худшем случае запишет плохие данные; испортить живое состояние движка он не может.

Окна

`registerEditorWindow(title, (ui, editor) => …)` добавляет сворачиваемую панель, пришвартованную поверх вьюпорта (рейка инспектора остаётся привязанной к выделению — окна глобальны для редактора). Окно, эмитящее `ui.toolButton` для инструмента, — **панель настроек** этого инструмента: активация инструмента раскрывает и подсвечивает её, так что настройки лежат ровно там, где инструмент используется. Функция рендера — **immediate-mode**, тот же протокол `InspectorUI`, что у своих карточек инспектора: она перезапускается при каждом взаимодействии (и при каждом изменении документа) и описывает панель вызовами виджетов. Все ключи полей окна — временное **состояние редактора** (у окон нет записи в документе): размеры кистей, выбранные ассеты; они переживают перерендеры, но никогда не пишутся в файл. К словарю добавляются два оконных виджета:

Виджет	Что делает
`ui.asset(key)`	Дропдаун ассетов проекта (GLB-пути проекта). Возвращает путь, `""` = ничего.
`ui.toolButton(label, tool)`	Переключает именованный инструмент вьюпорта — рисуется нажатым, пока тот активен.

Чтобы поделиться значениями виджетов с инструментом, копируй их в состояние модуля в рендере (пример выше) — immediate-mode возвращает текущее значение на каждом прогоне.

Инструменты вьюпорта

`registerEditorTool(name, hooks)` добавляет запись в тулбар рядом с move/rotate/scale (также активируется через `ui.toolButton`, деактивируется Esc или W/E/R). Пока инструмент активен, гизмо трансформа скрыто, клики выделения приостановлены, и каждый клик по вьюпорту вызывает `onViewportClick(hit, editor)`:

```ts
type EditorRayHit = {
  point: [number, number, number]    // точка попадания в мировых координатах
  normal: [number, number, number]
  node: string | null                // узел файла сцены, в который попали; null = плоскость земли
}
```

На сборках с физикой попадание берётся из точного рейкаста по мешам сцены; иначе (и при промахах) идёт фолбэк на плоскость земли (y = 0). `hooks.cursor` задаёт CSS-курсор вьюпорта (по умолчанию `"crosshair"`); `hooks.icon` задаёт глиф кнопки в тулбаре (один символ / эмодзи — инструменты без него делят общую «волшебную палочку», так что задавай его всякий раз,

когда редактор регистрирует два и больше инструментов). Бросающий хук логируется и пропускается — уронить редактор он не может.

API `editor`

Передаётся рендерам окон и хукам инструментов:

Член	Что делает
`selection` / `select(name)`	Имя выделенного узла (чтение / смена).
`nodes()`	Узлы документа: `{ name, kind }[]` (mesh / model / light / group).
`uniqueName(base)`	Первое незанятое имя `base`, `base2`, ….
`addNode(name, def)`	Добавить узел из простых данных дефа (грамматика файла сцены; строковое значение `model` — путь ассета).
`setProp(name, key, value)`	Записать один проп дефа — трансформы применяются вживую, остальное патчится.
`removeNode(name)` / `duplicate(name)`	Удалить / дублировать (возвращает имя копии).
`raycast(x, y)`	Тот же рейкаст вьюпорта, который получают клики инструментов.
`transact(fn)`	Сгруппировать все правки документа внутри `fn` в ОДИН шаг undo.

Дефы с `model`-ассетом перезапускают компиляцию сцены (бандл должен завендорить URL ассета); дефы mesh/light/group применяются живыми патчами без компиляции.

Реестр `__editorPlugins`, который питают регистрации, — внутренний: его читает обвязка редактора сцен; никогда не трогай его напрямую.

4.4 2D-движок

4.4.1 Scene2D

2D-мир: набор для отрисовки плюс ортографическая `Camera2D`. Открытие сцены поднимает 2D-движок на канвасе и запускает рендер; активна одновременно только одна сцена. Не путать со `Scene` (3D-мир).

Обзор

```ts
const scene = new Scene2D({ background: '#10131a' })
scene.layer(1).ySort()                 // слой 1 сортирует сверху вниз по Y

const tex = await Texture2D.load(asset('./hero.png'))
const hero = new Sprite({ texture: tex, anchor: [0.5, 1], layer: 1 })
scene.add(hero)

scene.addEventListener('click', ev => {
  console.log('tapped', ev.target?.id ?? 'empty space', 'at world', ev.worldX, ev.worldY)
})
scene.open()
```

Создание

```ts
new Scene2D(options?: {
  background?: ColorInput          // цвет очистки
  filter?: 'nearest' | 'linear'    // сэмплинг текстур; по умолчанию nearest (чёткий пиксель-арт)
})

scene.background = '#10131a'       // можно задать и позже
scene.camera                       // Camera2D сцены (только чтение)
```

> **NOTE**
>
> `filter` — **глобальная** настройка рендерера — один сэмплер на все текстуры, не на каждую отдельно. Она (пере)применяется на `open()`, поэтому побеждает выбор открытой сейчас сцены. Используй `'linear'` для hi-res / не-пиксельарт-ассетов; опусти, чтобы оставить текущую настройку.

Открытие и закрытие

```ts
scene.open(): this          // сделать активной: поднимает 2D-движок, закрывает любую другую сцену
scene.close(): this         // остановить рендер этой сцены
scene.destroy(): void       // закрыть + освободить нативную сцену
Scene2D.active              // статик: открытая сейчас Scene2D или null
```

Узлы

```ts
scene.add(...nodes: Node2D[]): this
scene.remove(...nodes: Node2D[]): this
```

Добавляет/удаляет корневые `Node2D`. Дочерние узлы идут вместе с родителем — `scene.add` нужен только корням.

Слои и Y-сортировка

```ts
scene.layer(n).ySort(enabled = true): LayerHandle
```

Узлы выбирают слой через `node.layer` (более высокие слои рисуются поверх; внутри слоя порядок задаёт `node.z`). Вызов `.ySort()` на слое переключает его на глубинную сортировку сверху вниз: спрайты ниже по экрану перекрывают спрайтов выше — классический вид «персонажи заходят за деревья».

> **NOTE**
> Y-сортировка сравнивает позиции узлов, то есть точки якоря. Дай персонажам и объектам на Y-сортируемом слое якорь низ-центр (`anchor: [0.5, 1]` — «ноги»), чтобы глубина сортировалась там, где они касаются земли.

Пикинг

```ts
scene.pick(worldPoint: Vec2Like): Node2D | null
```

Самый верхний **видимый спрайт**, чьи мировые границы содержат точку — хит-тест без тела, физика не нужна. Сочетай с `scene.camera.screenToWorld()`, чтобы преобразовать позицию тапа.

Для автоматической доставки событий указателя узлам дай узлу физическое тело — см. пикинг в 2D-физике.

События указателя на уровне сцены

```ts
scene.addEventListener('click', (ev: ClickEvent) => void)          // отпускание где угодно в
сцене
scene.addEventListener('touchstart', (ev: TouchStartEvent) => void) // нажатие; ev.track()
для перетаскиваний
scene.removeEventListener(channel, callback)
```

Обработчик срабатывает на **каждый** тап/нажатие в сцене; `ev.target` — задетый узел с физическим телом или `null` (пустое место). `ev.worldX` / `ev.worldY` дают точку попадания в мировом пространстве. На `'touchstart'` вызови `ev.track({...})`, чтобы захватить перетаскивание — см. События указателя и жесты.

Смотрите также

- Node2D — трансформ, иерархия, слой/z, события.
- Camera2D — зум, преобразование экран↔мир.
- Sprite — текстурированные узлы, якоря.
- 2D-физика — что делает узел кликабельным указателем.
- События указателя и жесты — `ClickEvent`, `ev.track()`.

4.4.2 Node2D

Базовый узел 2D-сцены — пустой трансформ. `Sprite` и `Tilemap` расширяют его, поэтому всё здесь (трансформ, иерархия, события, аспекты) работает и на них. Голый `Node2D` полезен как группирующий родитель, маркер/точка спавна в мировом пространстве или физический/триггерный объект (прикрепи `Shape2D` плюс `Physics2D` или `Trigger2D`).

Обзор

```ts
const door = new Node2D()
  .aspect(Shape2D, { box: [16, 32] })
  .aspect(Trigger2D)
door.position = [320, 64]
door.addEventListener('enter', other => console.log('entered by', other.id))
scene.add(door)

const cart = new Node2D()
cart.add(wheelA, wheelB)        // дети двигаются вместе с тележкой
cart.x += 40
```

Трансформ

Локальный относительно родителя, **Y вверх**; 1 единица = 1 логический px при зуме камеры 1. См. Соглашения.

```ts
node.x, node.y                      // сеттеры одной оси
node.position = [120, 64]           // Vec2Like; геттер возвращает свежий Vec2
node.rotation = 45                  // градусы, CCW
node.scale = 2                      // число (равномерно) или Vec2Like — только масштаб отрисовки
node.layer = 1                      // слой отрисовки (int); более высокие слои рисуются поверх
node.z = 5                          // внутрислойная глубина на не-Y-сортируемых слоях (больше = поверх)
node.visible = false
```

> **NOTE**
>
> Геттеры `position` и `scale` возвращают **свежие копии** — `node.position.x = 3` это тихий no-op. Используй `node.x = 3` или поменяй локальную переменную и присвой обратно. См. семантику значений в математике.

> **NOTE**
>
> `scale` влияет только на отрисовку — физические формы его никогда не читают. Увеличенный спрайт сохраняет объявленный тобой коллайдер. См. 2D-физику.

Иерархия

```ts
node.parent                                 // Node2D | null (можно присваивать — сахар для setParent)
node.children                               // readonly Node2D[]
node.add(...children: Node2D[]): this       // прикрепить (каждый сохраняет свой мировой трансформ)
node.remove(child: Node2D): this            // отсоединить прямого ребёнка → корень, сохранив мировой трансформ
node.setParent(parent: Node2D | null, keepWorld = true): this
```

`setParent` с `keepWorld: true` (по умолчанию) пересчитывает локальный трансформ, чтобы ничего не сдвинулось на экране; передай `false`, чтобы сохранить локальные значения и прыгнуть в систему нового родителя. Сам себя и потомков в родители не пускает (без циклов).

```ts
node.worldPosition                    // Vec2 — собирает всех предков
node.worldToLocal(p: Vec2Like): Vec2  // мировая точка → локальное пространство этого узла
node.localToWorld(p: Vec2Like): Vec2  // и обратно
```

Трансформы под управлением физики

Как только у узла есть `dynamic`- или `kinematic`-тело `Physics2D`, нативный шаг физики владеет его трансформом. *Чтение* `node.position` / `worldPosition` всегда корректно — узел автоматически пересинхронизируется с нативным слоем. **Не** пиши `node.position` каждый кадр; управляй телом через `velocity` / `applyImpulse` (кинематическое: `moveTo`).

События

```ts
node.addEventListener(channel, callback)
node.removeEventListener(channel, callback)
```

Канал	Срабатывает, когда	Аргумент колбэка
`click`	отпускание над физической формой этого узла	`ClickEvent`
`touchstart`	нажатие на его форму — `ev.track()` для перетаскиваний	`TouchStartEvent`
`enter` / `exit`	контакт физики / перекрытие сенсора начались / закончились	другой `Node2D`
`loopReached`	зацикленный клип `SpriteAnimation` обернулся	имя клипа (строка)
`completed`	незацикленный клип завершился	имя клипа (строка)

`click` / `touchstart` требуют, чтобы узел был кликабельным: `Shape2D` плюс `Physics2D` или `Trigger2D`. См. События указателя и жесты и пикинг в 2D-физике.

Уничтожение

```ts
node.destroy(): void
```

Освобождает нативную сущность и отсоединяет от родителя. **Дети переподвешиваются к корню**, сохраняя мировые трансформы — они не уничтожаются вместе с родителем. Использование уничтоженного узла после — неопределённое поведение.

Подводные камни

TS

```
// ✗ изменение копии трансформа
node.position.x = 3
// ✓ сеттер одной оси или присвоение целого значения
node.x = 3

// ✗ борьба с шагом физики на динамическом теле
setLoop(() => { node.y += 2 })
// ✓ управляй телом вместо этого
node.physics.velocity = [0, 120]
```

Смотрите также

- Scene2D — добавление узлов, слои, Y-сортировка.
- Sprite — текстурированный узел.
- 2D-физика — `Shape2D`, тела, триггеры, `enter`/`exit`.
- События указателя и жесты — `click`, `touchstart`, перетаскивания.
- Соглашения — единицы, семантика значений, жизненный цикл.

4.4.3 Camera2D

Ортографическая камера `Scene2D`. Каждая сцена владеет одной — обращайся к ней как `scene.camera`; сам `Camera2D` ты не конструируешь.

Обзор

TS

```
const scene = new Scene2D()
scene.camera.zoom = 2                          // 2× увеличение — дружелюбно к пиксель-арту

// держим камеру на герое, с лёгким сглаживанием
setLoop(dt => {
  scene.camera.position = scene.camera.position.lerp(hero.worldPosition, 10 * dt)
})
```

Трансформ

```ts
camera.position = [x, y]     // мировые единицы — точка в центре экрана
camera.zoom = 2              // увеличение; при 1, 1 мировая единица = 1 логический px. По умолчанию 1
camera.rotation = 15         // градусы, CCW. По умолчанию 0
```

Геттеры возвращают свежие значения (`camera.position.x = 3` — no-op — см. Соглашения).

> **NOTE**
> встроенного `follow()` нет. Следи за узлом из `setLoop` (как выше) или маленьким аспектом с `update(dt)` — нечего освобождать, когда цель исчезает.

Экран ↔ мир

```ts
camera.screenToWorld(screenX, screenY): Vec2   // логические px → мировые единицы
camera.worldToScreen(worldX, worldY): Vec2     // мировые единицы → логические px
```

`screenToWorld` — пара для обработки тапов: преобразуй `ev.clientX/clientY` и передай результат в `scene.pick()` или `Physics2D.overlapPoint()`. Помни про смену знака: экранный Y смотрит вниз, мировой Y — вверх — преобразование делает это за тебя.

Смотрите также

- Scene2D — сцена-владелец, `pick()`.
- 2D-физика — `overlapPoint`, `raycast`.
- События указателя и жесты — `clientX/clientY` в логических px.

4.4.4 Sprite

Текстурированный 2D-узел — рабочая лошадка 2D-сцены. Расширяет `Node2D`, поэтому имеет полный трансформ, иерархию, события и поверхность аспектов. Анимация спрайт-листа не встроена: прикрепи аспект `SpriteAnimation`.

Обзор

```ts
const tex = await Texture2D.load(asset('./hero.png'))

const hero = new Sprite({ texture: tex, anchor: [0.5, 1], layer: 1 })
  .aspect(SpriteAnimation, { size: [32, 48], fps: 10, clips: { idle: [0], walk: [1, 2, 3, 4] } })
hero.position = [120, 64]
hero.anim.play('walk')
scene.add(hero)
```

Создание

```ts
new Sprite(options?: {
  texture?: Texture2D | Canvas   // Canvas запекается в текстуру при присвоении
  anchor?: Vec2Like              // пивот: [0,0] верх-лево … [1,1] низ-право; по умолчанию [0.5, 0.5]
  size?: Vec2Like                // мировой размер; по умолчанию = пиксельные размеры текстуры
  frame?: [u0, v0, u1, v1]       // нормализованный UV-подпрямоугольник
  color?: ColorInput             // тинт, умножается с текстурой
  opacity?: number               // 0..1
  layer?: number                 // слой отрисовки (см. слои Scene2D)
  position?: Vec2Like
})
```

> **NOTE**
> пространство якоря — Y вниз, хотя мир Y вверх: `[0.5, 1]` — низ-центр спрайта («ноги»). Используй его для всего на Y-сортируемом слое, чтобы глубина сортировалась по ногам.

Текстура и размер

```ts
sprite.texture = tex             // Texture2D или Canvas (запекается при присвоении)
sprite.size = [w, h]             // мировые единицы; сбрасывает нативный дефолт при смене текстуры
```

- Размер по умолчанию — пиксельные размеры текстуры — 1 тексель = 1 мировая единица = 1 логический px при зуме 1.
- Спрайт на основе `Canvas` по умолчанию берёт **логический** размер канваса, поэтому `pixelRatio` канваса влияет только на чёткость, никогда на экранный размер. См. Canvas.

Кадры (атласы / спрайт-листы)

```ts
sprite.frame = [u0, v0, u1, v1]        // нормализованный UV-подпрямоугольник
sprite.setFramePx(x, y, w, h): this    // подпрямоугольник в пикселях текстуры — чейнится
```

`setFramePx` — то, что стоит использовать с атласом: координаты совпадают с тем, что показывает графический редактор. Смена кадра не меняет `size` — задай `size` один раз равным пиксельным размерам кадра, если хочешь вывод без масштабирования.

Для покадровой *анимации* не гоняй `frame` сам — прикрепи `SpriteAnimation` и определи клипы.

Внешний вид

```ts
sprite.color = '#ff8800'    // тинт (умножается); по умолчанию белый = без тинта
sprite.opacity = 0.5        // 0..1
sprite.flipX = true         // отразить по горизонтали (лицом влево/вправо без отрицательного масштаба)
sprite.flipY = true
```

Подводные камни

```ts
// ✗ ждать, что scale повлияет на физику — экстенты Shape2D это сырые мировые единицы, без масштаба
sprite.scale = 2   // рисует 2×, но его физический бокс остаётся объявленного размера

// ✗ покадровая смена текстуры для анимации
setLoop(() => sprite.setFramePx(...))   // работает, но плохо переизобретает SpriteAnimation
```

Смотрите также

- Node2D — трансформ, иерархия, события (унаследованы).
- SpriteAnimation — клипы, fps, события play/loop.
- Texture2D — загрузка изображений.
- Scene2D — слои, Y-сортировка, добавление узлов.

4.4.5 SpriteAnimation

Анимация спрайт-листа как аспект на `Sprite` — аксессор `node.anim`. Клипы объявляются как данные (размер ячейки кадра плюс именованные списки индексов кадров) и затем управляются по имени; после определения воспроизведение идёт полностью в нативном коде.

Обзор

```ts
const hero = new Sprite({ texture: tex, anchor: [0.5, 1] })
  .aspect(SpriteAnimation, {
    size: [32, 48],                                  // одна ячейка кадра, в пикселях текстуры
    fps: 10,
    clips: {
      idle: [0, 1],
      walk: [4, 5, 6, 7],
      die:  { frames: [8, 9, 10], fps: 6, loop: false },
    },
  })

hero.anim.play('walk')
hero.addEventListener('completed', clip => { if (clip === 'die') hero.destroy() })
```

Настройка

```ts
sprite.aspect(SpriteAnimation, {
  size?: [w, h]                    // размер ячейки кадра в пикселях текстуры; по умолчанию = вся текстура
  fps?: number                     // fps по умолчанию для клипов; по умолчанию 12
  loop?: boolean                   // loop по умолчанию для клипов; по умолчанию true
  clips?: Record<string, Clip>     // Clip = number[] | { frames: number[], fps?, loop? }
})
```

Индексы кадров считаются **построчно** по сетке листа: сетка имеет `floor(texture.width / cellW)` столбцов, индекс 0 — верхняя-левая ячейка, индексы продолжаются слева направо, затем вниз. Клип — это либо простой массив индексов (наследует `fps`/`loop`), либо объект с переопределениями на клип.

> **NOTE**
>
> у спрайта уже должна быть `texture`, когда аспект прикрепляется — сетка нарезается из пиксельного размера текстуры. Прикрепление без неё бросает.

> **NOTE**
>
> передача `size` также задаёт мировой `size` спрайта равным размеру ячейки, поэтому одна ячейка рисуется 1 тексель = 1 мировая единица.

```ts
sprite.anim.define(): this    // пересобрать нативные клипы после изменения anim.clips
```

Воспроизведение

```ts
sprite.anim.play(name: string): this   // бросает на неизвестное имя клипа
sprite.anim.stop(): this
sprite.anim.speed = 2                  // множитель скорости воспроизведения (только запись)
sprite.anim.current                    // имя играющего клипа или null
sprite.anim.frame                      // текущий индекс кадра (только чтение)
```

Повторный запуск уже активного клипа — no-op — безопасно звать `play('walk')` каждый кадр из кода ввода, не перезапуская анимацию.

События

События анимации эмитятся на **узле**, с **именем** клипа как аргументом:

```ts
hero.addEventListener('loopReached', clip => {})   // зацикленный клип обернулся (каждый цикл)
hero.addEventListener('completed', clip => {})     // незацикленный клип завершился
```

Смотрите также

- Sprite — узел-хост; ручные `frame` / `setFramePx`.
- Texture2D — загрузка листа.
- Node2D — `addEventListener`.

4.4.6 2D-физика

Физика на Box2D как аспекты на `Node2D`. Четыре части, зеркалящие 3D-семейство:

- **`Shape2D`** — чистая геометрия (аксессор `node.shape`). Сама по себе ничего не делает; две ниже превращают её в фикстуру.
- **`Physics2D`** — твёрдое тело (`node.physics`): static, kinematic или dynamic.
- **`Trigger2D`** — статическая сенсорная зона (`node.trigger`): эмитит `enter`/`exit`, ничего не блокирует.
- **`CharacterController2D`** — помощник для платформеров (`node.controller`) поверх динамического тела.

Обзор

```ts
Physics2D.configure({ gravity: [0, -980] })       // один раз, ДО создания тел

const ground = new Node2D()
  .aspect(Shape2D, { segment: { from: [-500, 0], to: [500, 0] } })
  .aspect(Physics2D, { motion: 'static' })

const hero = new Sprite({ texture: tex, anchor: [0.5, 1] })
  .aspect(Shape2D, { capsule: { from: [0, 6], to: [0, 26], radius: 6 } })
  .aspect(Physics2D, { motion: 'dynamic', fixedRotation: true })
hero.addEventListener('enter', other => console.log('touched', other.id))
hero.physics.applyImpulse([0, 400])               // прыжок

scene.add(ground, hero)
```

Гейтинг хоста

```ts
Physics2D.supported    // статик: поставляет ли эта сборка 2D-физику?
```

На сборке без поддержки физики всё семейство **молча инертно**: тела получают `id` 0, каждый вызов — no-op, события не срабатывают — и пикинг узлов указателем (ниже) никогда не попадает. Оберни зависящий от физики геймплей в `Physics2D.supported`.

Мир: `Physics2D.configure`

```ts
Physics2D.configure(config?: {
  gravity?: Vec2Like        // мировые единицы/с², Y вверх (вниз = отрицательный Y); по умолчанию [0, -980]
  pixelsPerMeter?: number   // подстраивает внутренние допуски Box2D; ты всё равно задаёшь в мировых единицах; по умолчанию 64
  subSteps?: number         // под-шаги солвера на фиксированный шаг; по умолчанию 4
})
```

Создаёт — или **сбрасывает** — мир физики. Вызывай один раз, до создания любых тел; повторный вызов сбрасывает мир, осиротив существующие тела.

Симуляция шагает с фиксированными 60 Гц. Отрисованные трансформы интерполируются между шагами:

```ts
Physics2D.interpolation = false   // статик, глобально; по умолчанию true
```

Выключи интерполяцию, чтобы сэкономить покадровые записи трансформа при множестве движущихся тел (тогда они движутся дискретными шагами по 60 Гц).

Shape2D — геометрия

```ts
node.aspect(Shape2D, {
  box?: [hw, hh]                                          // ПОЛУ-экстенты, мировые единицы
  circle?: number                                         // радиус
  capsule?: { from: Vec2Like, to: Vec2Like, radius: number }   // между двумя локальными точками
  segment?: { from: Vec2Like, to: Vec2Like }              // тонкая грань — статичная земля/стены/склоны
  polygon?: Vec2Like[]                                    // выпуклый, до 8 локальных точек (оболочка вычисляется)
  offset?: Vec2Like                                       // локальное смещение от начала узла (box/circle)
})
```

Задай ровно один из `box` / `circle` / `capsule` / `segment` / `polygon`. **Без единого** форма выводит бокс из `size` спрайта узла (полуширина × полувысота; запасной 16×16 для голого `Node2D`).

- `box` принимает **полу**-экстенты: `box: [12, 20]` — это бокс 24×40.
- Точки `capsule`, `segment` и `polygon` — в локальном пространстве узла и игнорируют `offset`.
- `segment` — только для **статичной** твёрдой геометрии — у него нет площади (нет плотности) и он никогда не сенсор; не используй его с `Trigger2D` или на динамических телах.

> **NOTE**
>
> формы никогда не читают `scale` отрисовки узла — все экстенты это сырые мировые единицы. Масштабирование спрайта 2× не масштабирует его коллайдер.

Physics2D — твёрдые тела

```ts
node.aspect(Physics2D, {
  motion?: 'static' | 'kinematic' | 'dynamic'   // по умолчанию 'dynamic'
  density?: number                              // по умолчанию 1
  friction?: number                             // по умолчанию 0.3
  restitution?: number                          // упругость; по умолчанию 0
  fixedRotation?: boolean                       // блокировать вращение (персонажи платформеров); по умолчанию false
  bullet?: boolean                              // непрерывная коллизия для быстрых объектов; по умолчанию false
  gravityScale?: number                         // множитель гравитации на тело; по умолчанию 1
})
```

Требует `Shape2D` на том же узле — прикрепи её **первой** (иначе `Physics2D` бросает).

```ts
node.physics.velocity = [vx, vy]        // линейная скорость, мировые единицы/с (геттер: свежий Vec2)
node.physics.angularVelocity = 90       // градусы/с (только запись)
node.physics.gravityScale = 0           // 1 = полная гравитация, 0 = парит, <0 = отталкивается
node.physics.linearDamping = 0.5        // только запись
node.physics.enabled = false            // только запись: приостановить это тело
node.physics.applyImpulse([x, y]): this // мгновенный импульс; будит тело
node.physics.applyForce([x, y]): this   // непрерывная сила; будит тело
node.physics.moveTo(p, deg = 0): this   // телепорт / привести кинематическое тело к мировой позе
node.physics.id                         // нативный id тела (0 = отсоединено / нет поддержки физики)
```

Физика владеет трансформом

Для `dynamic`- или `kinematic`-тела нативный шаг физики пишет трансформ узла. Не задавай `node.position` каждый кадр — управляй динамическими телами через `velocity` / `applyImpulse`, кинематическими — через `moveTo`. *Чтение* `node.position` / `worldPosition` всегда корректно; узел пересинхронизируется с нативным слоем при чтении. См. Node2D.

> **NOTE**
>
> держи тела на **корневых** узлах. Телу под движущимся родителем шаг физики пишет трансформ в мировых терминах — движение родителя и физика будут бороться.

Trigger2D — сенсорные зоны

```ts
const goal = new Node2D()
  .aspect(Shape2D, { box: [32, 64] })
  .aspect(Trigger2D)                        // без опций
goal.addEventListener('enter', other => win(other))

goal.trigger.moveTo(p: Vec2Like): this     // переместить зону в мировую точку
goal.trigger.id                            // нативный id тела (0 = нет поддержки физики)
```

Триггер — это **статическое сенсорное** тело, построенное из `Shape2D` узла: перекрывающиеся тела эмитят события `enter`/`exit` узла вместо столкновения — оно никогда не блокирует движение. Перемещай его через `trigger.moveTo` (статическое тело не следует за поздними перемещениями узла).

События `enter` / `exit`

Начало/конец контакта (твёрдое против твёрдого) и начало/конец перекрытия сенсора (триггер) оба приходят как события `enter`/`exit` узла, доставляемые **обоим** узлам с другим узлом как аргументом:

```ts
hero.addEventListener('enter', other => {})   // коснулся стены или вошёл в триггер
hero.addEventListener('exit',  other => {})
```

CharacterController2D — движение платформера

```ts
node.aspect(CharacterController2D, {
  speed?: number         // горизонтальная скорость движения, мировые единицы/с; по умолчанию
200
  jumpSpeed?: number     // скорость отрыва прыжка, мировые единицы/с; по умолчанию 500
  groundProbe?: number   // доп. дистанция луча под ногами для проверки «на земле»; по
умолчанию 6
  footOffset?: number    // расстояние от начала узла до ног; по умолчанию = половина высоты
спрайта
})

node.controller.move(dir: number)   // горизонтальное намерение в [-1, 1]; ЛИПКОЕ — задай 0,
чтобы остановиться
node.controller.jump()              // ставится в очередь; потребляется в следующем кадре,
если на земле
node.controller.grounded            // true, пока стоит на чём-то (обновляется каждый кадр)
```

Построен на управлении скоростью: каждый кадр он задаёт горизонтальную скорость тела равной `dir * speed` и отдаёт вертикаль Box2D (гравитация, прыжок, приземление, отклик стен/пола). `grounded` берётся из короткого луча вниз от ног.

Если у узла их нет, прикрепление контроллера авто-добавляет `Shape2D` (авто-бокс из спрайта) и `Physics2D` с `{ motion: 'dynamic', fixedRotation: true }`.

TS

```
setLoop(() => {
  hero.controller.move((Input.key('ArrowRight') ? 1 : 0) - (Input.key('ArrowLeft') ? 1 : 0))
  if (Input.key('Space')) hero.controller.jump()
})
```

Запросы (статические)

TS

```
Physics2D.raycast(from: Vec2Like, to: Vec2Like): RayHit | null
// RayHit = { node: Node2D | null, point: Vec2, normal: Vec2, fraction: number /* 0..1 вдоль луча */ }

Physics2D.overlapPoint(p: Vec2Like): Node2D | null
```

`raycast` возвращает **ближайшее** тело, задетое отрезком; `overlapPoint` — самое верхнее тело — твёрдое или сенсор — чья форма содержит мировую точку.

Пикинг указателем

Узел с `Shape2D` **плюс** `Physics2D` или `Trigger2D` кликабелен указателем: тапы хит-тестятся через `Physics2D.overlapPoint`, поэтому узел получает события `click` / `touchstart`. См. События указателя и жесты. Для пикинга без тел используй `scene.pick()` (границы спрайта).

Подводные камни

```ts
// ✗ полные экстенты — box принимает ПОЛУ-экстенты
hero.aspect(Shape2D, { box: [24, 40] })    // это бокс 48×80
// ✓ половина размера спрайта
hero.aspect(Shape2D, { box: [12, 20] })

// ✗ конфигурация после создания тел — сбрасывает мир
ground.aspect(Shape2D, {}).aspect(Physics2D, { motion: 'static' })
Physics2D.configure({ gravity: [0, -500] })
// ✓ сначала конфигурируй, потом создавай тела

// ✗ двигать динамическое тело записью трансформа
setLoop(() => { hero.x += 2 })
// ✓ управляй телом
hero.physics.velocity = [120, hero.physics.velocity.y]
```

Смотрите также

- Node2D — слушатели `enter`/`exit`, владение трансформом.
- События указателя и жесты — `click`/`touchstart` на кликабельных узлах.
- Scene2D — `pick()` без тел, события указателя на уровне сцены.
- Sprite — масштаб отрисовки против размера коллайдера.

4.4.7 Tilemap

Узел тайловой сетки: атлас-текстура, нарезанная на ячейки, разложенная на равномерной сетке, загруженная один раз и отрисованная нативно с покадровым отсечением по виду — рисуй большой уровень одним узлом. Расширяет `Node2D`, поэтому имеет полный трансформ, иерархию и поверхность слоёв.

Обзор

```ts
const tiles = await Texture2D.load(asset('./tiles.png'))

const map = new Tilemap({
  texture: tiles,
  cols: 20, rows: 5,          // размер карты в тайлах
  tile: 32,                   // каждая ячейка 32×32 мировые единицы
  atlas: [8, 4],              // текстура содержит 8×4 тайла
  data: [                     // строка 0 = ВЕРХНЯЯ строка; -1 = пусто
    -1, -1, -1, /* … cols*rows элементов … */
     0,  1,  1, /* … */
  ],
})
map.position = [0, 0]         // НИЖНИЙ-ЛЕВЫЙ угол карты
scene.add(map)
```

Создание

```ts
new Tilemap(options: {
  texture: Texture2D
  cols: number                    // ширина карты в тайлах
  rows: number                    // высота карты в тайлах
  tile: number | [w, h]           // экранный размер ячейки в мировых единицах (число = квадрат)
  atlas: [atlasCols, atlasRows]   // как текстура нарезана на тайлы
  data: number[] | Int32Array     // построчные индексы атласа, длина cols*rows; -1 = пусто
})

map.cols, map.rows                // только чтение
```

- `data` — **построчно, строка 0 сверху** карты (читается как текст уровня в исходнике), хотя мировой Y смотрит вверх.
- Индексы атласа тоже считаются построчно по сетке атласа: 0 — верхний-левый тайл атласа.
- `position` узла — **нижний-левый** угол карты.

Редактирование ячеек

```ts
map.setTile(x: number, y: number, index: number): this   // координаты сетки как в data (строка 0 = верх)
```

Устанавливает одну ячейку в индекс атласа (`-1` очищает). Дёшево — сетка живёт нативно.

Ограничения

Tilemap — это **статическая равномерная сетка**. Не предоставляется (на момент текущего исходника):

- анимированные тайлы — меняй ячейки сам через `setTile` при необходимости
- коллизия на тайл — строй коллизию уровня отдельно, напр. невидимые `Node2D` с боксами/сегментами `Shape2D` и статическими телами `Physics2D`
- отражение/поворот на тайл — запеки зеркальные/повёрнутые варианты в атлас

Смотрите также

- Node2D — трансформ, слой (унаследованы).
- Texture2D — загрузка атласа.
- 2D-физика — статическая геометрия коллизий для уровней.
- Scene2D — слои, Y-сортировка.

4.4.8 Texture2D

GPU-текстура для 2D-движка — изображение за `Sprite` или `Tilemap`. Сырые байты файла декодируются и загружаются нативно. Не взаимозаменяема с 3D `Texture` (изображения материалов).

Обзор

```ts
const tex = await Texture2D.load(asset('./hero.png'))
const hero = new Sprite({ texture: tex })
console.log(tex.width, tex.height)      // пиксельные размеры
```

Загрузка

```ts
Texture2D.load(source: string | FetchResponse | File): Promise<Texture2D>
```

- **URL-строка** — путь `asset('./file.png')` или удалённый `https://…` URL; скачивается, затем декодируется. Отклоняется при HTTP-статусе ≥ 400 или ошибке декодирования.
- `FetchResponse` — уже скачанный ответ (пропускает лишний запрос).

- **`File`** — напр. из `Canvas.toFile()` или `openFilePicker`.

```ts
Texture2D.fromCanvas(canvas: Canvas): Texture2D
```

Оборачивает запечённую поверхность `Canvas` как текстуру — синхронно (уже растрирована, без декодирования). Её `width`/`height` — **физические** пиксели канваса (`логический размер × pixelRatio`). Присвоение `Canvas` напрямую в `sprite.texture` делает это за тебя.

Свойства и жизненный цикл

```ts
tex.width, tex.height    // пиксели (только чтение)
tex.id                   // нативный хэндл текстуры (только чтение)
tex.destroy(): void      // освободить GPU-текстуру
```

Текстуры свободно разделяются — много спрайтов могут использовать один `Texture2D`. `destroy()` освобождает GPU-ресурс; спрайты, всё ещё ссылающиеся на неё — неопределённое поведение.

> **NOTE**
> сэмплинг (nearest против linear) не на текстуру — это глобальная настройка рендерера, выбираемая на сцену через опцию `filter` у `Scene2D`.

Смотрите также

- Sprite — использование текстуры, кадры/атласы.
- Tilemap — сетки атласа.
- SpriteAnimation — нарезка листа на клипы.
- Scene2D — глобальная настройка `filter`.

4.5 Ядро

4.5.1 Сигналы (реактивность)

Точечная реактивность для состояния приложения. `signal` хранит значение; чтение `.value` внутри `effect`, `computed` или **UI-привязки** подписывает её, а запись `.value` заново выполняет ровно этих подписчиков. Никакого virtual DOM и никакого ре-рендера: запись сигнала ложится точечным обновлением свойства на удержанном дереве UI.

Обзор

```ts
const count = signal(0)
const label = computed(() => `Count: ${count.value}`)

const screen = UIScreen(
  UIColumn(
    UIText(() => label.value)                        // привязка текста — обновляется сама
      .style({ color: "white", fontSize: 20 }),
    UIButton(UIText("+1").style({ color: "white" }))
      .style({
        height: 40, borderRadius: 8, justifyContent: "center",
        bgColor: () => (count.value > 9 ? "#22c55e" : "#FF4032"),   // привязка стиля
      })
      .onClick(() => count.value++),
  ).style({ gap: 8, p: 16 }),
)
Router.push(screen)
```

Нигде нет ручного `label.text = …` — привязки перевыполняются, когда меняется `count`.

API

```ts
const s = signal(0)          // Signal<number>
s.value                      // чтение (подписывает внутри effect/computed/привязки)
s.value = 1                  // запись — уведомляет подписчиков, пакетно за микрозадачу
s.peek()                     // чтение БЕЗ подписки

const c = computed(() => s.value * 2)   // Computed<number> — ленивый, кэшируется до смены зависимости
c.value

const dispose = effect(() => {          // выполняется сейчас и снова после смены любого прочитанного сигнала
  console.log("count is", s.value)
})
dispose()                               // остановить; неостановленные effect живут весь запуск приложения
```

- Записи **пакетируются**: несколько записей `.value =` за один тик дают одно перевыполнение на каждый затронутый effect (на следующей микрозадаче). Запись

равного значения (`Object.is`) — no-op.

- Зависимости отслеживаются на каждый прогон: ветка, переставшая читать сигнал, отписывается от него.
- `effect` возвращает свою функцию остановки. UI-привязкам остановка не нужна — они живут и умирают вместе со своим элементом.

UI-привязки

Везде, где UI-элемент принимает текстовую строку или примитивное значение стиля, функция `() => value` вместо этого создаёт привязку:

```ts
UIText(() => `${items.value.length} items`)          // реактивный текст
el.style({ opacity: () => (open.value ? 1 : 0) })    // реактивное свойство стиля (форма слияния)
el.style.transform = () => `translateY(${y.value}px)` // реактивная запись одного свойства
```

Правила:

- Привязки выполняются один раз синхронно при создании, поэтому элемент сразу держит конкретное значение.
- Последующий `.style({ prop: staticValue })` **заменяет** привязку по этому ключу; последующий `.style({ prop: () => … })` перепривязывает её. (Прямая запись `el.style.prop = staticValue` *не* снимает привязку — следующая смена сигнала перезапишет её.)
- Только верхний уровень: функции-значения внутри вложенных блоков состояний (`onPressed: { … }`, `onLandscape: { … }`) — не привязки.
- Ручная установка `.text` на привязанном `UIText` работает, но перезаписывается при следующей смене сигнала.

Реактивные дети

Контейнер также принимает **функцию** в позиции детей — список сам себя перерисовывает, когда меняется прочитанный им сигнал:

```ts
const todos = signal([ { title: "one", done: false } ])

UIColumn(() => todos.value.map(todo =>
  UIRow(UIText(todo.title))
))

// условный рендер — ложные записи просто исчезают:
UIColumn(() => [ header, expanded.value && details ])

todos.value = [ ...todos.value, { title: "two", done: false } ]   // вставлен один узел, остальные не тронуты
```

Обновления **минимальны**: рантайм сравнивает по идентичности элементов — ушедшие узлы удаляются, новые вставляются на свою позицию, неизменные остаются смонтированными. Обычный `.map(…)` внутри функции-детей мемоизируется поэлементно компилятором (через внутренний хелпер `__uiMap` — сам его никогда не зови), поэтому объект-элемент, оставшийся в списке, сохраняет свой элемент между обновлениями. Отсюда правила:

- **Идентичность элемента — это ссылка на объект.** Обновляй списки, записывая новый массив, *переиспользующий* неизменные объекты-элементы (`[ ...items.value, added ]`, `.filter(…)`). Пересоздание каждого объекта-элемента при каждой записи (`items.value.map(i => ({ ...i }))`) вызывает полную перерисовку.
- Колбэк рендера выполняется **один раз на новый элемент** (как компонент), а не на каждое изменение — не клади туда побочные эффекты в расчёте на прогон при каждом обновлении.
- Контейнер с функцией-детьми принадлежит этой привязке — не вызывай на нём ещё и `.append()` / `.remove()` вручную.
- *Переставленный* элемент удаляется и вставляется заново (его состояние — фокус ввода, скролл — сбрасывается).
- Для сотен строк используй `UIVirtualizedList`; функции-дети — для маленьких и средних списков.

`el.setContent(() => …)` привязывается так же.

Реактивные классы: `el.class`

Привяжи `$`-класс-стиля к сигналу, присвоив функцию — класс тогда отслеживает всё, что функция читает, а переходы, объявленные на классе, применяются автоматически:

```ts
row.style({ $done: { opacity: 0.4, duration: 150 } })
   .class({ done: () => todo.done.value })

// или как запись одного ключа:
row.class.done = () => todo.done.value
```

Последующая обычная запись (`row.class.done = false`) заменяет привязку — побеждает последняя запись.

Автоматическая обёртка (что компилятор делает за тебя)

В коде проекта текстовый аргумент `UIText`, значение стиля верхнего уровня или значение `.class({ … })`, читающее `.value` сигнала, оборачивается в привязку на этапе компиляции — поэтому явная форма `() =>` в этих позициях необязательна:

```ts
UIText(`Count: ${count.value}`)                    // компилируется в UIText(() => `…`)
btn.style({ bgColor: active.value ? "#f43" : "#333" })   // компилируется в привязку стиля
row.class({ done: todo.done.value })               // компилируется в привязку класса
```

Формы с `() =>` остаются валидными и ведут себя идентично. Чтения внутри твоих собственных функций-хелперов (`UIText(fmt())`) *не* распознаются — там используй явную стрелку.

Подводные камни

```ts
// ✗ сначала прочитать .value в обычную переменную — чтение происходит один раз, дальше
ничего не обновляется
const n = count.value
UIText(`Count: ${n}`)
// ✓ читай .value прямо в текстовом выражении (авто-обёртка) или пиши привязку явно
UIText(`Count: ${count.value}`)
UIText(() => `Count: ${count.value}`)

// ✗ сигнал с массивом, изменённым на месте — та же ссылка, уведомления нет
items.value.push(x)
// ✓ запиши новую ссылку
items.value = [ ...items.value, x ]

// ✗ ждать, что DOM изменится синхронно после записи (обновления сбрасываются на микрозадаче)
count.value = 5; /* label ещё не перерисован на этой строке */
```

Смотрите также

- Стилизация UI — свойства стиля, на которые могут нацеливаться привязки.
- Элементы контента — `UIText`.
- animate — твины по времени (сигналы — про состояние, не про время).

4.5.2 Аспекты

Прикрепляемые возможности узла — композиция с лаконичностью методов. Всё, что узел *умеет* сверх своего трансформа (анимация, коллайдеры, физика, твоя игровая логика), прикрепляется как аспект, а не раздувает класс узла. Встроенные `SpriteAnimation`, `Shape2D`, `Physics2D` — это аспекты; твои поведения расширяют `Aspect` тем же способом.

Обзор

TS

```
class Health extends Aspect<'health', Sprite> {
  hp = 100
  hurt(n: number) {
    this.hp -= n
    if (this.hp <= 0) this.node.destroy()
  }
}

const hero = new Sprite({ texture })
  .aspect(SpriteAnimation, { size: [32, 48], fps: 10, clips: { walk: [1, 2, 3] } })
  .aspect(Health, { hp: 80 })          // прикрепить + настроить; чейнится — возвращает узел
hero.anim.play('walk')                 // каждый аспект добавляет именованный аксессор
hero.health.hurt(10)
```

Прикрепление, доступ, открепление

TS

```
node.aspect(Class, opts?): node & { name: instance }   // прикрепить + настроить (или перенастроить)
node.get(Class): instance | undefined                  // безопасный доступ — undefined, если не прикреплён
node.has(Class): boolean                               // проверка наличия И type guard
node.removeAspect(Class): node                         // открепить (запускает onDetach); чейнится
```

- `aspect()` возвращает узел, **типизированный как уже имеющий этот аспект**, поэтому именованный аксессор работает без guard'а сразу после цепочки.

- Именованный аксессор (`hero.anim`, `hero.health`, …) берётся из класса аспекта: встроенные объявляют имя через `static aspect`; для твоих аспектов сборка извлекает имя из первого дженерик-параметра (`Aspect<'health', …>` → `node.health`).
- `has()` — это type guard: внутри `if (node.has(Physics2D))` тип `node.physics` считается присутствующим.

> **NOTE**
>
> вызов `.aspect(Class, opts)` на уже прикреплённом аспекте **не** прикрепляет заново — он просто делает `Object.assign` опций на существующий экземпляр. `onAttach` выполняется один раз, только при первом прикреплении.

Свой аспект

```ts
class Follow extends Aspect<'follow', Sprite> {   // <имя аксессора, вид целевого узла>
  target?: Node2D
  speed = 4                                       // поля класса = настраиваемые значения по умолчанию
  onAttach() { /* node установлен; opts уже присвоены */ }
  onDetach() { /* очистка: таймеры, слушатели */ }
  update(dt: number) {                            // dt в секундах
    if (this.target) this.node.position = this.node.position.lerp(this.target.position, this.speed * dt)
  }
}
sprite.aspect(Follow, { target: hero, speed: 6 })
```

- `this.node` — узел, к которому прикреплён аспект, типизированный по второму дженерик-параметру. Прикрепление к узлу неверного вида — ошибка компиляции.
- Аспекты создаёт движок через `node.aspect()`, **никогда не** `new` — не пиши конструктор. Инициализируй в `onAttach` (к этому моменту узел и опции уже установлены); настраиваемые параметры объявляй полями класса со значениями по умолчанию.

Покадровый `update(dt)`

Аспект, определяющий `update(dt)`, тикает каждый кадр, пока прикреплён (`dt` = **секунды** с прошлого кадра). Обновления идут в одной из двух фаз:

- **LATE (по умолчанию)** — после шага физики и синхронизации трансформов, прямо перед отрисовкой кадра. Чтение `node.worldPosition` даёт финальную отрисованную позицию, поэтому камеры и «преследователи» встают точно, без отставания на кадр.
- **EARLY** (`updateBeforePhysics = true`) — до шага физики, поэтому записи скорости, сил или кинематических трансформов учитываются шагом *того же* кадра (нулевая задержка ввода).

```ts
class Steer extends Aspect<'steer', Sprite> {
  updateBeforePhysics = true      // этот аспект ПИТАЕТ симуляцию — выполнять до шага
  order = -10                     // порядок тика внутри фазы: по возрастанию; по умолчанию
0, при равенстве — порядок прикрепления
  update(dt: number) { /* пиши this.node.physics.velocity здесь */ }
}
```

Правило: аспекты, которые **пишут** в симуляцию, идут в EARLY; аспекты, которые **читают** результаты (камеры, преследователи, синхронизация UI), остаются в фазе LATE по умолчанию.

> **NOTE**
>
> `updateBeforePhysics` и `order` читаются **один раз, при прикреплении** — задавай их полями класса (или в опциях прикрепления), не позже.

> **NOT IMPLEMENTED**
>
> `updateWhenVisible` объявлен, чтобы аспекты могли подписаться на отсечение по видимости, но сейчас это заглушка — каждый зарегистрированный аспект тикает независимо от того, на экране он или нет.

`With<N, A>` — типизация переменной с аспектами

```ts
let boss: With<Sprite, Health | Physics2D>    // Sprite, про который ИЗВЕСТНО, что у него оба
аспекта
boss.health.hurt(10)                          // guard не нужен
```

Объединяй аспекты через union (не кортеж) — так читается естественно и повторяет рантайм-guard `node.has(Health)`. Используй для полей и параметров функций, принимающих узлы, собранные в другом месте.

Смотрите также

- Соглашения — стиль цепочек, в который вписываются аспекты.
- События — узлы также являются эмиттерами (`addEventListener`).
- Sprite — типичный хост для аспектов.

4.5.3 Canvas и Bitmap

2D-поверхность для рисования, запекающаяся в текстуру — способ поместить текст и произвольную векторную графику в спрайты, 3D-материалы и UI-изображения («canvas — это просто текстура»). API повторяет 2D-контекст браузера, но **записывает** команды вместо

немедленной отрисовки: весь буфер растрируется один раз за запекание, на нативном 2D-стеке платформы.

Обзор

```ts
const c = new Canvas(160, 40, { pixelRatio: 2 })       // логические 160×40, запекается в 320×80
c.font = 'bold 24px Inter'
const w = c.measureText('Alice').width + 24
c.resize(w, 40)
c.fillStyle = '#000a'
c.roundRect(0, 0, w, 40, 12).fill()
c.fillStyle = '#fff'
c.textAlign = 'center'
c.fillText('Alice', w / 2, 27)

const tag = new Sprite({ texture: c })   // Canvas — валидный источник текстуры где угодно
```

Создание

```ts
new Canvas(width, height, opts?: {
  pixelRatio?: number    // множитель device-пикселей; по умолчанию 1
})
```

Всё рисование задаётся в логических единицах. `width`/`height` — логический размер; запечённая текстура имеет `width × pixelRatio` на `height × pixelRatio` device-пикселей. `pixelRatio` влияет только на чёткость, никогда на экранный размер — `Sprite` на основе Canvas по умолчанию берёт **логический** размер как мировой. Используй `pixelRatio: 2` (или `device.pixelRatio`) для чёткого текста на retina-экранах.

> **NOTE**
> цвета и шрифты — это CSS-строки, разрешаемые платформой. Всё рисование выполняй **после** `await registerFont(...)` для любого своего шрифта — незагруженный шрифт измеряется неверно.

Состояние рисования

Свойства чтения/записи, как в браузере. Установка одного записывает операцию состояния *и* запоминает значение, чтобы геттер вернул его обратно.

```ts
c.fillStyle = '#ff8800'          // любая CSS-строка цвета; по умолчанию '#000000'
c.strokeStyle = '#fff'
c.lineWidth = 2                  // по умолчанию 1
c.lineJoin = 'miter'             // 'miter' | 'round' | 'bevel'
c.lineCap = 'butt'               // 'butt' | 'round' | 'square'
c.globalAlpha = 0.5              // 0..1; по умолчанию 1
c.font = '16px sans-serif'       // CSS-строка шрифта; по умолчанию '10px sans-serif'
c.textAlign = 'start'            // 'left' | 'center' | 'right' | 'start' | 'end'
c.textBaseline = 'alphabetic'    // 'alphabetic' | 'top' | 'middle' | 'bottom' | 'hanging' |
'ideographic'
```

Трансформы и стек состояний

Все методы рисования возвращают `this`, поэтому вызовы чейнятся.

```ts
c.save(): this                   // положить состояние (трансформ + состояние рисования)
c.restore(): this                // снять его
c.translate(x, y): this
c.scale(sx, sy): this            // трансформ — не связан с pixelRatio
c.rotate(rad): this              // радианы
```

Пути

```ts
c.beginPath(): this
c.moveTo(x, y): this
c.lineTo(x, y): this
c.quadraticCurveTo(cx, cy, x, y): this
c.bezierCurveTo(c1x, c1y, c2x, c2y, x, y): this
c.arc(x, y, r, a0, a1, ccw?): this      // углы в радианах; ccw по умолчанию false
c.rect(x, y, w, h): this
c.roundRect(x, y, w, h, r): this        // один радиус на все углы
c.closePath(): this
c.fill(evenOdd?): this                  // evenOdd по умолчанию false (ненулевое правило
обхода)
c.stroke(): this
```

Сахар для прямоугольников и текста

```ts
c.fillRect(x, y, w, h): this
c.strokeRect(x, y, w, h): this
c.clearRect(x, y, w, h): this
c.fillText(text, x, y, maxWidth?): this      // maxWidth 0 = без ограничения (по умолчанию)
c.strokeText(text, x, y, maxWidth?): this
c.measureText(text): { width, ascent, descent }   // измеряет в ТЕКУЩЕМ шрифте, логические px
```

`measureText` — единственный синхронный вызов к платформе; всё остальное только записывает.

Изображения: `drawImage` и `Bitmap`

`drawImage` копирует `Bitmap` — снимок, сделанный через `toBitmap()`. Он не принимает `Texture2D` или другой `Canvas`.

```ts
c.drawImage(bmp, dx, dy): this                              // весь bitmap в натуральном (логическом) размере
c.drawImage(bmp, dx, dy, dw, dh): this                      // весь bitmap, масштабированный в dw×dh
c.drawImage(bmp, sx, sy, sw, sh, dx, dy, dw, dh): this    // подпрямоугольник, масштабированный в dw×dh
```

Все координаты **логические** — включая исходный подпрямоугольник в форме с 9 аргументами. Копирование записывается, как и всё остальное, поэтому держи `Bitmap` живым до следующего запекания (`update()` / `texture()` / `toBitmap()`).

```ts
canvas.toBitmap(): Bitmap    // неизменяемая копия текущих пикселей, независимая от canvas
```

```ts
bmp.width, bmp.height              // логический размер
bmp.pixelWidth, bmp.pixelHeight    // device-пиксели (логический × pixelRatio)
bmp.toFile(name?, type?): Promise<File>   // кодировать: 'image/png' (по умолчанию) или 'image/jpeg'
bmp.destroy(): void                // освобождает нативную поверхность; после этого bitmap непригоден
```

Паттерн «сплющивания» — держать повторные запекания O(1) на постоянно растущем рисунке:

```ts
const snap = canvas.toBitmap()
canvas.reset()
canvas.drawImage(snap, 0, 0)   // одна операция заменяет всю историю; продолжай рисовать поверх
```

Запекание в текстуры

```ts
canvas.texture(): Texture2D    // запечь + вернуть 2D-текстуру (создаётся один раз, затем кэшируется)
canvas.update(): this          // перерастрировать и перезалить в каждую текстуру, произведённую этим canvas
```

`texture()` сам вызываешь редко — присваивание canvas туда, где ждут текстуру (`new Sprite({ texture: c })`, `UIImage`, карта `Material`), запекает его. После перерисовки динамического canvas (`reset()` + новые команды) вызови `update()`, чтобы протолкнуть новые пиксели всем его потребителям.

```ts
canvas.toFile(name?, type?): Promise<File>   // закодировать текущие пиксели; png (по умолчанию) или jpeg
canvas.destroy(): void                       // освободить нативную поверхность + её кэшированную 2D-текстуру
```

> **NOTE**
> после `destroy()` сам canvas и любой спрайт/материал, всё ещё сэмплящий его текстуру, недействительны. Текстура, запечённая в 3D-движок, освобождается при разрушении этого движка, а не здесь.

`reset()` и `resize()`

```ts
canvas.reset(): this           // сбросить записанный рисунок, чтобы начать новый
canvas.resize(w, h): this      // изменить логический размер; вступает в силу при следующем запекании
```

> **NOTE**
>
> `reset()` очищает и *записанное* состояние — `font`, `fillStyle`, `textAlign`, каждая заданная тобой настройка исчезает из следующего запекания, которое откатывается к значениям по умолчанию. Геттеры свойств всё ещё сообщают старые значения (и `measureText` всё ещё измеряет старым шрифтом), из-за чего это легко упустить. **Переустанавливай `font` и компанию после каждого `reset()`.**

`resize()` не очищает записанные команды; сочетай с `reset()`, когда начинаешь заново в новом размере. Схема «измерить, потом изменить размер» (см. *Обзор*) работает, потому что до первого запекания ничего не растрируется.

Подводные камни

TS

```
// ✗ перерисовка после reset() без переустановки состояния — текст запекается шрифтом по умолчанию 10px
c.reset(); c.fillText('99', 10, 20)
// ✓ состояние — часть записи; задай его снова
c.reset(); c.font = 'bold 24px Inter'; c.fillStyle = '#fff'; c.fillText('99', 10, 20)

// ✗ перерисовка в расчёте, что живые потребители изменятся — спрайт держит старые пиксели
c.reset(); c.fillRect(0, 0, 40, 40)
// ✓ протолкни новые пиксели в каждую текстуру, произведённую этим canvas
c.reset(); c.fillRect(0, 0, 40, 40); c.update()

// ✗ задавать размер спрайта через pixelRatio — pixelRatio это чёткость, не размер
new Sprite({ texture: c, size: [c.width * c.pixelRatio, c.height * c.pixelRatio] })
// ✓ мировой размер по умолчанию уже равен логическому
new Sprite({ texture: c })
```

Смотрите также

- Sprite — Canvas как текстура спрайта.
- Соглашения — логические px, строки цветов.

4.5.4 События (паттерн Emitter)

Паттерн слушателей, общий для всего SDK. Нет глобального `Emitter`, который нужно конструировать — эта страница описывает два метода, которые предоставляет каждый объект с событиями, и как они себя ведут. UI-элементы — исключение: они используют чейнящиеся методы `on*` (см. ниже).

Обзор

```ts
scene.addEventListener('click', ev => console.log(ev.clientX, ev.clientY))


const onDone = () => next()
player.addEventListener('completed', onDone)
player.removeEventListener('completed', onDone)   // удаление по ТОЙ ЖЕ ссылке на функцию
```

API

```ts
obj.addEventListener(channel, callback): void
obj.removeEventListener(channel, callback): void
```

Каналы и сигнатуры колбэков типизированы для каждого объекта (`'click'` узла доставляет `ClickEvent`, `'completed'` медиаплеера не доставляет ничего). Удаление сопоставляется по идентичности функции — храни ссылку на любой колбэк, который придётся удалять.

> **NOTE**
> нет ни `once()`, ни публичной отправки — только сам объект-владелец эмитит свои события. Добавление одного колбэка дважды вызывает его дважды. Удаление слушателя изнутри колбэка безопасно (отправка итерируется по снимку).

У кого он есть

- 3D `Scene` / `Node`, 2D `Scene2D` / `Node2D` — события указателя (`click`, `touchstart`), события контактов физики
- `device` — `resize`
- `AudioPlayer` / `VideoPlayer` — `completed`, `loopReached`
- `WebSocket` — `open`, `message`, `close`, `error`

UI устроен иначе: методы `on*`

UI-элементы не используют `addEventListener`; они принимают колбэки через чейнящиеся методы `on*`, чтобы обработчики вставали в цепочку сборки:

```ts
UIButton('Play').onClick(() => start())          // UI: чейнящийся сеттер on*
ball.addEventListener('click', () => kick())     // сцены/узлы: addEventListener
```

Смотрите также

- События указателя и жесты — объекты событий `click` / `touchstart`.
- Соглашения — правило событий в контексте.

4.6 Рантайм

4.6.1 Устройство

Информация о платформе, текущий размер дисплея, связность и событие ресайза от хоста. `device` — обычный глобальный объект, ничего создавать не нужно.

Обзор

```ts
if (device.platform === 'ios') toast('running on iOS')

const layout = (w: number, h: number) => {
  console.log('display is', w, 'x', h, 'logical px')
}
layout(device.width, device.height)               // читается в любой момент
device.addEventListener('resize', layout)         // и снова при каждом ресайзе
```

Инфо о платформе

```ts
device.platform     // "web" | "android" | "ios" | string  (только чтение)
device.language     // код языка хоста, напр. "en"          (только чтение)
device.pixelRatio   // физических px на логический px; 1 на обычных дисплеях, 2–3 на retina/iOS
```

`pixelRatio` — коэффициент, с которым запекать `Canvas` для чёткого вывода без хардкода: `new Canvas(w, h, { pixelRatio: device.pixelRatio })`. Откатывается к `1`, если хост его не сообщает.

> **NOTE**
> веб-поверхность 2D/GL сейчас рендерит в логическом разрешении, поэтому на вебе `pixelRatio` помогает только UI-канвасам; на iOS (физическая поверхность) он делает 2D-канвасы чёткими.

Размер дисплея

```ts
device.width        // текущая ширина дисплея, логические px  (только чтение)
device.height       // текущая высота дисплея, логические px (только чтение)
```

Те же значения, что доставляет событие `resize`, но читаемые в любой момент — не только внутри слушателя. Оба — **логические px** (как `clientX/clientY`), никогда не физические пиксели.

> **NOTE**
> оба равны `0`, пока хост не сообщил размер. Не строй layout из `device.width` в начале файла, который выполняется до первого кадра — подпишись ещё и на `resize`.

Событие `resize`

```ts
device.addEventListener('resize', (width, height) => { })   // логические px
device.removeEventListener('resize', callback)              // та же ссылка на функцию
```

Срабатывает при любом изменении вьюпорта хоста (поворот, ресайз окна).

Связность: `device.online` и события `online` / `offline`

```ts
device.online                                       // boolean (только чтение)
device.addEventListener('offline', () => toast('No connection'))
device.addEventListener('online', () => sync())
```

Семантика `navigator.onLine` по мере возможностей: `false` — только когда платформа **уверена**, что сети нет; `true` не гарантирует, что интернет доступен — относись к нему как к подсказке для UI (баннеры «ты офлайн», откладывание синхронизации) и всё равно обрабатывай ошибки `fetch`.

> **NOTE**
> гейтится хостом — читается как `true`, и события никогда не срабатывают на хостах, которые не отслеживают связность (headless).

Точный тач

```ts
device.setPreciseTouch(enabled: boolean)    // ВЫКЛ по умолчанию
```

Подключи систему точного тача там, где платформа её поддерживает. Когда включено, быстрые штрихи сэмплируются на полной частоте дигитайзера (объединённые касания iOS, ≈120–240 Гц), а не раз в кадр дисплея (~60 Гц) — приложение с активным вводом (рисование, рукописный ввод, перетаскивание) получает больше точек и более гладкие линии. Для tap/кнопочных UI оставь выключенным.

> **NOTE**
> гейтится хостом — тихий no-op на хостах без понятия объединённого ввода (веб уже объединяет pointermove; headless).

Хаптика

```ts
device.vibrate()             // удар "medium" по умолчанию
device.vibrate("selection")  // лёгкий тик для смены значения
```

Выдай одиночный хаптик **семантического** `style` (по умолчанию `"medium"`):

Группа	Стили	Смысл
Impact	`"light"` `"medium"` `"heavy"` `"soft"` `"rigid"`	физический «тап» разного веса
Notification	`"success"` `"warning"` `"error"`	сигнал об исходе
Selection	`"selection"`	лёгкий тик для меняющегося значения

API семантический, а не длительность/паттерн, намеренно: это единственный словарь, который ощущается нативно **и** на iOS (Taptic Engine / `UIFeedbackGenerator`), и на Android (`HapticFeedbackConstants` / `VibrationEffect`). Длительность `vibrate(ms)` была бы честной только на вебе/Android — у iOS нет публичного API для вибрации произвольной длины.

> **NOTE**
> гейтится хостом — тихий no-op там, где нет хаптического железа (iPad, старые iPhone, веб, headless). Также уважает системную настройку хаптики пользователя.

Движение

Сфьюженный датчик ориентации устройства (гироскоп + акселерометр) — для управления наклоном/рулением и magic-window / 360°-панорам.

```ts
await device.motion.start()          // начать обновления (батарея); Promise<boolean> — false
= нет сенсора

setLoop(() => {
  scene.camera.quaternion = device.motion.attitude   // камера 360°-панорамы, одна строка
  // или 2D-игра с наклоном:
  ball.x += device.motion.gravity.x
  ball.y += device.motion.gravity.y
})

device.motion.recenter()             // сделать «здесь» направлением вперёд (только yaw)
device.motion.stop()                 // освободить сенсор
```

Член	Тип	Примечания
`start(options?)`	`Promise<boolean>`	Начать обновления. `false` = нет сенсора / в разрешении отказано.
`stop()`	`void`	Остановить обновления, освободить сенсор.
`recenter()`	`void`	Обнулить текущее направление (pitch/roll остаются привязаны к гравитации).
`available`	`boolean`	Есть ли у устройства гироскоп вообще?
`enabled`	`boolean`	Идут ли обновления сейчас?
`attitude`	`Quat`	Текущая ориентация. По умолчанию **мировой фрейм** — присваивай прямо в `quaternion` узла/камеры.
`gravity`	`Vec3`	Направление гравитации в **экранном пространстве** (x → вправо, y → вниз); читай `x`/`y` для 2D-наклона.

`start(options)` принимает `{ interval?: number, frame?: "world" | "device" }`. `interval` (секунды, по умолчанию `1/60`) — нижняя граница: сенсор фьюзит в фоне на частоте не ниже её, а ты опрашиваешь самый свежий сэмпл каждый кадр, так что частота сенсора не обязана совпадать с частотой кадров. `frame` по умолчанию `"world"`: `attitude`/`gravity` приходят уже преобразованными в Y-up мировое пространство движка (а `gravity` — в экранное) и **знают об ориентации** — поверни телефон в ландшафт, и камера останется вертикальной автоматически. Передай `"device"` для сырого фрейма сенсора, если хочешь считать сам.

Опрашивай внутри `setLoop`; читаемые поля всегда возвращают последний сфьюженный сэмпл. `attitude` скорректирован от дрейфа, так что горизонт панорамы остаётся ровным бесконечно.

> **NOTE**
>
> гейтится хостом — тихий no-op там, где нет гироскопа (старые iPad, веб без разрешения, headless). `available` скажет заранее; `attitude` читается как `Quat.identity`, а `gravity` — как `(0,0,0)`, когда обновления не идут.

Смотрите также

- События указателя и жесты — события, которые питает точный тач.
- Соглашения — логические px, паттерн `addEventListener`.

4.6.2 Жизненный цикл приложения, буфер обмена и openURL

Собственный жизненный цикл приложения (передний план/фон), URL, с которым его открыли (диплинки), системный буфер обмена и открытие внешних URL. `app` и `clipboard` — обычные глобальные объекты, ничего создавать не нужно.

Обзор

```ts
app.addEventListener('pause', () => saveGame())      // пользователь ушёл — сохраняйся сейчас
app.addEventListener('resume', () => refreshFeed())

if (app.launchUrl) openFromLink(app.launchUrl)       // диплинк холодного старта
app.addEventListener('url', url => openFromLink(url)) // …и ссылки, приходящие во время работы

clipboard.write('LECODES-42')
openURL('https://le.codes/docs')
```

Жизненный цикл: `app.state`, `pause` / `resume`

```ts
app.state                                  // "active" | "background"  (только чтение)
app.addEventListener('pause', () => { })    // приложение ушло с переднего плана
app.addEventListener('resume', () => { })   // приложение вернулось
app.removeEventListener('pause', callback)  // та же ссылка на функцию
```

`"pause"` срабатывает, когда приложение перестаёт быть приложением переднего плана — кнопка «домой», другое приложение сверху, скрытая вкладка браузера. Оно доставляется **до** того, как хост остановит цикл кадров, так что это последний надёжный момент сохранить состояние, поставить музыку на паузу или остановить работу, которой нельзя идти вслепую (watch у `Geolocation`, таймеры опроса). `"resume"` срабатывает, когда кадры снова идут.

События семантические и одинаковые на каждой платформе — словарь Activity/Scene ты не видишь никогда. `app.state` — та же информация в виде pull, читается в любой момент.

> **NOTE**
> не рассчитывай, что код работает, *пока* приложение в фоне — на мобильных цикл кадров останавливается и таймеры замирают до resume. `pause` — для сохранения, а не для планирования фоновой работы.

Диплинки: `app.launchUrl` и событие `url`

```ts
app.launchUrl                              // string | null  (только чтение)
app.addEventListener('url', (url: string) => { })
```

`launchUrl` — URL, с которым приложение было открыто (в последний раз) — `null` при обычном запуске. Когда ссылка приходит, пока приложение уже работает (*тёплая* ссылка),

`launchUrl` обновляется на новое значение и с ним срабатывает событие `"url"`. Обрабатывай оба случая, именно в этом порядке:

```ts
const openFromLink = (url: string) => {
  const screen = new URL(url).searchParams.get('screen')
  if (screen === 'stats') router.push(StatsScreen())
}
if (app.launchUrl) openFromLink(app.launchUrl)
app.addEventListener('url', openFromLink)
```

> **NOTE**
> гейтится хостом — `launchUrl` читается как `null`, а `"url"` никогда не срабатывает на хостах без понятия ссылки (headless).

Управление миром: `app.restart` / `app.quit`

```ts
app.restart(url?)     // перезапустить бандл приложения в свежем мире — location.reload()
app.quit()            // уйти в лаунчер LeCodes (домой) — в остальных случаях no-op
```

`restart()` перезапускает приложение с нуля: свежий мир, тот же код. Передай `url`, чтобы перезапуститься с другими аргументами запуска (новый запуск читает их через `app.launchUrl`), передай `null`, чтобы перезапуститься с очищенным, или опусти его, чтобы повторить текущий — аргумент зеркалит роль `launchUrl` как «командной строки» приложения. Несохранённое состояние пропадает, ровно как при перезагрузке страницы.

`quit()` возвращает пользователя в лаунчер LeCodes. В standalone-сборке лаунчера нет, поэтому он ничего не делает (платформы запрещают программный выход) — безопасно вешать на кнопку «назад в LeCodes» безусловно.

```ts
UIButton(UIText('Start over')).onClick(() => app.restart(null))
UIButton(UIText('Exit')).onClick(() => app.quit())
```

Клавиатура: `app.keyboardHeight` и событие `keyboard` {#keyboard}

```ts
app.keyboardHeight                          // number  (только чтение, логические px; 0,
когда скрыта)
app.addEventListener('keyboard', (height: number, duration: number) => { })
```

`keyboardHeight` — **сырое перекрытие** экранной клавиатуры с вьюпортом приложения, независимо от политики `keyboardShrink` сфокусированного инпута. С политикой сжатия по

умолчанию layout и так избегает клавиатуры, так что оно нужно редко; главный потребитель — **режим оверлея** (`keyboardShrink: false`), где приложение само управляет своим инсетом. Событие срабатывает на каждое изменение фрейма клавиатуры; `duration` — длительность анимации клавиатуры на платформе в мс (`0`, где её нет) — передай её в `animateTo`, чтобы двигать UI синхронно:

```ts
// композер чата: клавиатура перекрывает UI, композер сам сдвигается вверх
input.style({ keyboardShrink: false })
app.addEventListener('keyboard', (height, duration) => {
  composer.animateTo({ transform: `translateY(${-height}px)`, duration })
})
```

> **NOTE**
> гейтится хостом — читается как `0` и никогда не срабатывает там, где экранной клавиатуры нет (macOS, десктоп, аппаратные клавиатуры).

`clipboard`

```ts
clipboard.write(text: string): void         // выстрелил и забыл
clipboard.read(): Promise<string>           // отклоняется: нет доступа / отказано / нечего читать
```

`write` кладёт текст в системный буфер обмена (тихий no-op там, где буфера обмена нет). `read` асинхронный, потому что веб-хост гейтит его за запросом разрешения; ожидай отклонение и обрабатывай его:

```ts
clipboard.write(inviteCode)
toast('Copied!')

try { input.value = await clipboard.read() }
catch { toast('Nothing to paste') }
```

`openURL`

```ts
openURL(url: string): void
```

Открой `url` в системном браузере / внешнем обработчике — выстрелил и забыл, как `share`. Используй для «Условий использования», «Оцените нас», ссылок `mailto:`. Тихий no-op на хостах, где такого нет (headless).

```ts
UIButton('Privacy policy').onClick(() => openURL('https://le.codes/privacy'))
```

Смотрите также

- Устройство — `device.online` + события `online`/`offline` (связность).
- Geolocation — останавливай watch'и в `pause`.
- Хранилище — что сохранять, когда срабатывает `pause`.

4.6.3 Input

Опрос клавиатуры. `Input` отвечает на один вопрос — *зажата ли сейчас эта клавиша?* — и ты спрашиваешь его каждый кадр внутри `setLoop`. **Событий клавиш нет** (никаких keydown/keyup-слушателей); для тапов и перетаскиваний по объектам используй события указателя.

Обзор

```ts
setLoop(dt => {
  if (Input.key('ArrowRight')) player.x += 200 * dt
  if (Input.key('ArrowLeft'))  player.x -= 200 * dt
})
```

Опрос

```ts
Input.key(code: string): boolean    // true, пока клавиша зажата
```

`code` — строка в стиле `KeyboardEvent.code` — это **физическая** клавиша, а не введённый символ: `'ArrowRight'`, `'KeyW'`, `'Space'`, `'Enter'`, `'ShiftLeft'`. WASD — это `'KeyW'`/`'KeyA'`/`'KeyS'`/`'KeyD'` независимо от раскладки.

> **NOTE**
> опрос, а не события — умножай перемещение на `dt` (секунды), чтобы скорость не зависела от частоты кадров.

«Нажата в этом кадре» — детект фронта

`Input.key` срабатывает по уровню (true каждый кадр, пока клавиша зажата). Для действий, которые должны срабатывать раз на нажатие (прыжок, выстрел), держи карту `wasDown` и защёлкивай её в **конце** кадра:

```ts
const wasDown: Record<string, boolean> = {}
const pressed = (code: string) => Input.key(code) && !wasDown[code]

setLoop(dt => {
  if (pressed('Space')) player.jump()               // срабатывает раз на нажатие
  if (Input.key('ArrowRight')) player.x += 200 * dt // срабатывает каждый кадр, пока зажато

  wasDown['Space'] = Input.key('Space')             // защёлкнуть в конце кадра
})
```

Защёлкивай каждую клавишу, для которой ловишь фронт — по одной строке на клавишу в конце цикла.

Подводные камни

```ts
// ✗ проверка по уровню для одноразового действия — прыгает каждый кадр, пока зажат Space
if (Input.key('Space')) player.jump()
// ✓ лови фронт по рецепту с wasDown выше
if (pressed('Space')) player.jump()

// ✗ символьные коды — зависят от раскладки и не то, что ждёт Input.key
Input.key('w')
// ✓ физические коды клавиш
Input.key('KeyW')
```

Смотрите также

- События указателя и жесты — тапы касанием/мышью и трекинг перетаскивания.
- Соглашения — `setLoop` и `dt` в секундах.

4.6.4 События указателя и жесты

Общие объекты событий указателя. Каждая тапабельная поверхность в SDK — 3D-узлы, 2D-узлы, сцены и UI-элементы — доставляет одни и те же два события: `click` (отпускание над целью) и `touchstart` (нажатие, а также точка входа для жестов перетаскивания через `ev.track()`).

Обзор

```ts
// тап
ball.addEventListener('click', ev => {
  console.log('tapped at', ev.clientX, ev.clientY)
})

// перетаскивание — захвати указатель на touchstart, затем следуй за ним
stick.onTouchStart(ev => {
  const startX = ev.clientX, startY = ev.clientY
  ev.track({
    claim: true,                                  // это наше перетаскивание — не отдавай его скроллу
    onMove: p => place(p.clientX - startX, p.clientY - startY),   // суммарное смещение от точки нажатия
    onEnd: () => rest(),
    onCancel: () => rest(),
  })
})
```

Где срабатывают события

Поверхность	Как цель становится кликабельной	Слушать через
3D `Node`	есть аспект `Shape` (тело для пикинга)	`node.addEventListener('click' | 'touchstart', cb)`
3D `Scene`	всегда (цель = задетый узел или `null`)	`scene.addEventListener(…)`
2D `Node2D`	есть `Shape2D` + `Physics2D`/`Trigger2D` (тело для хит-теста)	`node.addEventListener(…)`
2D `Scene2D`	всегда (цель = задетый узел или `null`)	`scene.addEventListener(…)` или `scene.pick(worldPoint)` для ручных хит-тестов
UI	только `UIButton`, `UIScreen`, `UIWidget`	`.onClick(cb)`, `.onTouchStart(cb)`

> **NOTE**
>
> `click` отправляется на отпускании в точке отпускания, независимо от активного трека — завершённое перетаскивание всё равно заканчивается `click` по тому, что под пальцем.

ClickEvent / TouchStartEvent / LongPressEvent

```ts
ev.clientX, ev.clientY    // позиция указателя в логических px (экранное пространство)
ev.pointerId              // стабильный id этого пальца
ev.target                 // задетый Node / Node2D или null (слушатели сцены)
ev.worldX, ev.worldY      // только 2D-сцены: точка попадания в мировом пространстве
(undefined в 3D)
```

Все три несут одни и те же поля. `LongPressEvent` — только у `UIButton`: он приходит в `onLongPress`, когда палец удержан дольше порога долгого нажатия, и, как `TouchStartEvent`, умеет `track()` (ниже) — чтобы утащить элемент прямо из удержания.

Перетаскивание: `ev.track(handler)`

Трекать может только `TouchStartEvent` или `LongPressEvent`. Вызов `ev.track()` захватывает все последующие перемещения этого указателя и направляет их в твой обработчик, пока палец не поднимется:

```ts
ev.track({
  onMove(pos)   { },   // pos: { clientX, clientY, deltaX, deltaY }
  onEnd(pos)    { },   // палец поднят нормально
  onCancel()    { },   // жест отобрали (системный жест, teardown) — всегда прибирайся и
здесь
  claim: 'pan-y',      // см. ниже
})
```

> **NOTE**
>
> `deltaX` / `deltaY` — **дельты на каждое перемещение** — движение с *предыдущего* события перемещения, а не с момента нажатия. Для суммарного смещения перетаскивания храни стартовую точку сам: `pos.clientX - ev.clientX`.

`claim` — победа над скроллерами

Пока твой элемент находится внутри чего-то, что тоже хочет жест (`UIScrollable`, `UIVirtualizedList`), `claim` объявляет, какое движение принадлежит тебе:

```ts
claim: true         // забрать указатель безусловно (свободное 2D-перетаскивание: джойстик,
рисование по канвасу)
claim: 'pan-x'      // забрать горизонтальное движение, вертикальное пусть начинает скролл
claim: 'pan-y'      // забрать вертикальное движение (нижние шторки)
claim: 'pan-up' | 'pan-down' | 'pan-left' | 'pan-right'   // только одно направление
```

Без claim скролл-контейнер, распознавший свой пан, может забрать указатель у тебя — ты получишь `onCancel`.

Подводные камни

```ts
// ✗ читать deltaX как «расстояние от места, где началось касание»
onMove: p => knob.style({ transform: `translate(${p.deltaX}px, 0)` })   // дёргается около нуля
// ✓ дельты — на каждое перемещение; накапливай или считай разницу со стартовой точкой
onMove: p => knob.style({ transform: `translate(${p.clientX - ev.clientX}px, 0)` })

// ✗ очистка только в onEnd — забранный системный жест её пропустит
ev.track({ onEnd: stop })
// ✓ onCancel срабатывает вместо onEnd, когда жест отбирают
ev.track({ onEnd: stop, onCancel: stop })
```

Смотрите также

- 2D-физика и пикинг — что делает `Node2D` кликабельным.
- 3D-физика и формы — тела для пикинга на 3D-узлах.
- Кнопки и поля ввода — `UIButton.onClick`, отклик на нажатие.
- Input — опрос клавиатуры.

4.6.5 Сеть

HTTP и сокеты через мост хоста. Читаются как веб-API, но это **не** они — одно, что нужно усвоить: после того как `await fetch(...)` разрешился, тело уже на хосте, поэтому **`res.json()` / `res.text()` синхронны** — второго `await` нет.

Обзор

```ts
const res = await fetch('https://api.example.com/items')
if (res.status === 200) {
  const items = res.json<{ id: number, name: string }[]>()   // синхронно — без await
  console.log(items.length)
}
res.dispose()
```

`fetch`

```ts
fetch(url: string, options?: {
  method?: string                          // 'GET' (по умолчанию), 'POST', …
  headers?: Record<string, string>
  body?: any                               // строка или FormData
  useOnce?: boolean                        // одноразовое тело — хост может освободить его
после чтения
  onProgress?: (p: { loaded: number, total?: number }) => void   // прогресс загрузки, байты
}): Promise<FetchResponse>
```

Промис разрешается, как только у хоста есть весь ответ — статус *и* тело. HTTP-статус ошибки (404, 500) всё равно **разрешается**; проверяй `res.status`. Отклонение означает, что сам запрос провалился (нет соединения, плохой URL).

`onProgress` сообщает скачанные байты; `total` отсутствует, когда сервер не присылает длину. Передавай `useOnce: true` для запросов «выстрелил и забыл», чьё тело ты читаешь ровно один раз — так делают собственные загрузчики SDK (`Texture2D.load`, `Model.load`).

`FetchResponse`

```ts
res.status          // HTTP-код статуса (только чтение)
res.json<T>(): T    // распарсить тело как JSON — СИНХРОННО
res.text(): string  // тело как текст — СИНХРОННО
res.dispose()       // освободить буфер тела на стороне хоста
```

Тело живёт в хранилище на стороне хоста; `json()`/`text()` перетягивают его. `FetchResponse` также принимается напрямую как источник изображения (`UIImage(res)`) и как значение `FormData`.

> **NOTE**
>
> `json()`/`text()` — обычные синхронные вызовы. `await res.json()` «работает» (await обычного значения), но выдаёт неверную ментальную модель — не пиши так.

> **NOTE**
>
> владение — хост держит тело, пока ты не вызовешь `dispose()` (или ты сделал fetch с `useOnce: true`). Освобождай ответы, которые держишь, когда закончил с ними.

`fetchLocal`

```ts
fetchLocal(path: string): FetchResponse    // синхронно, статус всегда 200
```

Читай собранный ресурс проекта — без промиса, без сети. Используй с макросом `asset()` для файлов данных, которые поставляешь:

```ts
const config = fetchLocal(asset('./config.json')).json<{ levels: number }>()
```

`File`

```ts
file.name    // имя файла (только чтение)
file.size    // байты (только чтение)
```

Хэндл выбранного или снятого файла. Ты их не конструируешь — они приходят из `openFilePicker` и `takePhoto()` у `CameraView`. Добавь в `FormData` для загрузки, используй как источник изображения или `share()` его.

`FormData`

```ts
const form = new FormData()
form.append(name: string, value: string | number | boolean | File | FetchResponse,
            filename?: string)     // filename по умолчанию — собственное имя File
form.delete(name: string)          // удаляет первое поле с этим именем
```

Собери multipart-тело, затем передай его как `body` у `fetch`. Это минимальное подмножество — нет `get`, `has`, `set` или итерации.

```ts
const file = await openFilePicker({ accept: 'image/*' })
if (file) {
  const form = new FormData()
  form.append('avatar', file)
  form.append('userId', 123)
  await fetch('https://api.example.com/upload', { method: 'POST', body: form })
}
```

`WebSocket`

```ts
const ws = new WebSocket(url: string, headers?: Record<string, string>)  // подключается
сразу
ws.send(message: string | ArrayBuffer)
ws.close()
```

Событийная поверхность как в браузере, с одним нестандартным дополнением: необязательный аргумент `headers` идёт на upgrade-запрос (удобно для токенов авторизации). Слушай через `addEventListener` / `removeEventListener` (общий паттерн эмиттера — см. Соглашения):

```ts
ws.addEventListener('open',    () => ws.send('hello'))
ws.addEventListener('message', data => console.log('got', data))   // текстовый или бинарный фрейм
ws.addEventListener('close',   code => console.log('closed', code))
ws.addEventListener('error',   () => console.log('socket error'))
```

После `close()` сокет инертен — дальнейшие вызовы `send()` — тихие no-op. Закрывай сокеты в `onClose` экрана, иначе они продолжат работать при навигации.

Подводные камни

```ts
// ✗ привычки веб-fetch — читатели тела здесь не асинхронны
const data = await (await fetch(url)).json()
// ✓ дождись fetch один раз, затем читай синхронно
const res = await fetch(url); const data = res.json()

// ✗ считать 404 отклонением
try { await fetch(url) } catch { /* ждём тут 404 — он не придёт */ }
// ✓ HTTP-статусы разрешаются; проверяй статус сам
const res = await fetch(url); if (res.status !== 200) handleError(res.status)
```

Смотрите также

- Файлы и шеринг — откуда берутся хэндлы `File`.
- Хранилище — хранить небольшое состояние локально вместо сервера.
- Соглашения — события, макрос `asset()`.

4.6.6 Хранилище

Постоянное key-value хранилище на стороне хоста. Привычная форма веб-`localStorage`, сведённая к трём важным методам. Значения переживают перезагрузки.

Обзор

```ts
// сохранить
localStorage.setItem('save', JSON.stringify({ level: 3, coins: 120 }))

// загрузить (со значением по умолчанию)
const raw = localStorage.getItem('save')
const save = raw ? JSON.parse(raw) : { level: 1, coins: 0 }

// очистить
localStorage.removeItem('save')
```

API

```ts
localStorage.getItem(key: string): string | null   // null, если ключ никогда не задавали
localStorage.setItem(key: string, value: string)   // сохранить / перезаписать
localStorage.removeItem(key: string)               // удалить
```

Это вся поверхность — нет `clear()`, нет `length`, нет итерации `key(i)`.

> **NOTE**
> значения — **только строки**. Всё структурированное — через `JSON.stringify` / `JSON.parse`.

Смотрите также

- Сеть — `fetch`, если состояние должно жить на сервере.

4.6.7 Медиа

`AudioPlayer` проигрывает звук; `VideoPlayer` проигрывает видео и дополнительно предоставляет 3D `Texture`. Оба делят одну поверхность воспроизведения (внутренняя база `MediaPlayer`), поэтому play/pause, громкость, зацикливание, перемотка и события у них идентичны.

Обзор

```ts
const music = new AudioPlayer(asset('./theme.mp3'))
music.loop = true
music.volume = 0.4
music.play()

const sting = new AudioPlayer(asset('./win.wav'))
sting.addEventListener('completed', () => sting.dispose())
sting.play()
```

Создание

```ts
new AudioPlayer(src?: string)   // asset('./sound.mp3') или удалённый URL
new VideoPlayer(src?: string)
```

Воспроизведение

```ts
player.play()               // старт / продолжить
player.pause()              // пауза с сохранением позиции
player.playing = true       // булево-сеттер — play()/pause() это его алиасы
player.volume = 0.5         // 0..1; по умолчанию 1
player.loop = true          // зациклить в конце вместо остановки; по умолчанию false
player.time = 12.5          // текущая позиция в СЕКУНДАХ — задай для перемотки
player.duration             // полная длина в секундах (только чтение); 0, пока не загрузятся
метаданные
```

События

```ts
player.addEventListener('completed',   () => { })   // достигнут конец при loop === false;
playing становится false
player.addEventListener('loopReached', () => { })   // срабатывает на КАЖДОЙ границе цикла
при loop === true
```

Ровно одно из двух срабатывает за проигрывание, решается по `loop` в момент достижения конца: зацикленные плееры эмитят `loopReached` раз на круг и продолжают играть; незацикленные останавливаются (`playing` переключается в `false`) и эмитят `completed` один раз.

Жизненный цикл

```ts
player.dispose()    // освободить медиаресурс хоста (декодер, сеть, буферы); идемпотентно
player.disposed     // true после того, как dispose() отработал (только чтение)
```

`dispose()` также сбрасывает все слушатели событий. Плееры не освобождаются сами — освобождай всё созданное при закрытии его экрана.

> **NOTE**
>
> после `dispose()` плеер мёртв — обращение к `play()`, `volume`, `time` и т. п. **бросает** (`"MediaPlayer has been disposed"`). Повторный вызов `dispose()` — нормально.

Видео-текстура

```ts
const movie = new VideoPlayer(asset('./clip.mp4'))
movie.loop = true
movie.play()
const tex = movie.texture                                    // 3D Texture, сэмплящая видео
const screen = Mesh.plane({ material: Material.video(tex) })
```

`texture` возвращает 3D `Texture`, сэмплящую живое видео — передай её в `Material.video(...)`, чтобы положить видео на 3D-поверхность. Для видео в UI используй `UIVideo(player)` — см. Текст, изображения, видео.

> **NOTE**
>
> каждое чтение `texture` конструирует новую обёртку `Texture` — прочитай один раз и держи ссылку, а не обращайся к `player.texture` каждый кадр.

Смотрите также

- Текст, изображения, видео — `UIVideo` показывает `VideoPlayer` в UI.
- Файлы и шеринг — шеринг снятых/скачанных медиафайлов.

4.6.8 Файлы и шеринг

Как заводить файлы в приложение и отдавать их: `openFilePicker` открывает нативный пикер хоста и разрешается в хэндл(ы) `File`; `share` отдаёт медиафайл в системный шеринг ОС.

Обзор

```ts
const file = await openFilePicker({ accept: 'image/*' })
if (file) {
  const form = new FormData()
  form.append('photo', file)
  await fetch('https://api.example.com/upload', { method: 'POST', body: form })
}
```

`openFilePicker`

```ts
openFilePicker(options?: { accept?: string, multiple?: false }): Promise<File | null>
openFilePicker(options:  { accept?: string, multiple: true  }): Promise<File[]>
```

По умолчанию одиночный выбор: разрешается в один `File` или `null`, если пользователь отменил. С `multiple: true` разрешается в `File[]` — пустой при отмене, никогда не `null`. `accept` фильтрует, что можно выбрать, та же идея, что у HTML `<input accept>` (`'image/*'`, `'.pdf'`).

```ts
const many = await openFilePicker({ accept: 'image/*', multiple: true })
console.log(`picked ${many.length}`)
```

Возвращаемые хэндлы несут только `name` / `size` — нет пути и нет прямого доступа к байтам из JS; байты остаются на стороне хоста, за непрозрачным хэндлом. Используй `File` там, где SDK его принимает: загрузки через `FormData`, источники изображений (`UIImage(file)`, `Texture2D.load(file)`) и `share()`.

`share`

```ts
share(media: File | FetchResponse, text?: string): void
```

Поделиться медиафайлом (изображение/видео) через системный шеринг ОС. `media` — это `File` из пикера/камеры или скачанный `FetchResponse`; `text` — необязательная подпись/сообщение. Выстрелил и забыл — результат не возвращается.

Поведение по платформам:

- **iOS** — открывает системный шеринг; он также позволяет сохранить в Файлы/Фото.
- **web** — сохраняет (скачивает) файл.

```ts
const photo = await camera.takePhoto()   // camera = CameraView(), открыт раньше
share(photo, 'Look at this')
```

Смотрите также

- Сеть — класс `File`, `FormData`, `FetchResponse`.
- CameraView — другой источник хэндлов `File`.

4.6.9 Toast и SvgSource

Два небольших глобала платформы: временное уведомление и обёртка, превращающая сырой SVG XML в источник изображения.

`toast`

```ts
toast(msg: string): void
```

Показать временное toast-уведомление — быстрое «Сохранено», «Скопировано», «Нет соединения». Выстрелил и забыл; хост показывает и убирает его.

```ts
toast('Saved!')
```

`SvgSource`

```ts
SvgSource(svg: string): { readonly svg: string, tintColor: string | null }
```

Оберни сырую строку SVG XML, чтобы её можно было использовать везде, где принимается источник изображения (напр. `src` у `UIImage` — см. Текст, изображения, видео). Задай `tintColor` на возвращённом значении, чтобы перекрасить SVG; `null` (по умолчанию) означает без окраски.

```ts
const icon = SvgSource('<svg viewBox="0 0 24 24"><path d="M12 2 L22 22 H2 Z"/></svg>')
icon.tintColor = '#4caf50'
const img = UIImage(icon)
```

> **NOTE**
> импорт `.svg`-файла из проекта делает это за тебя — компилятор превращает `import icon from './icon.svg'` в готовое значение `SvgSource`.

Смотрите также

- Текст, изображения, видео — `UIImage` и полный набор источников изображений.
- Сеть — `FetchResponse`/`File` как другие источники изображений.

4.6.10 Даты

`date(...)` оборачивает момент времени в стиле dayjs: создаёте значение, цепляете арифметику, завершаете `format` или `timeAgo`. Оба понимают **русский** и **английский** и по умолчанию берут текущий язык устройства (`device.language`), если не передать его явно. Форматирование — чистый JS, без `Intl`, поэтому вывод одинаков на любой платформе.

```ts
date(value?: Date | number | string | DateValue): DateValue
```

Аргументом может быть `Date`, таймстемп в миллисекундах (`Date.now()`), разбираемая строка или другой `DateValue`; **без аргумента** — текущее время.

```ts
date().format('dddd, D MMMM')       // "пятница, 10 июля"  · "Friday, 10 July"
date(ts).format('DD.MM.YYYY HH:mm') // "10.07.2026 14:05"
date(ts).timeAgo()                  // "5 минут назад"     · "5 minutes ago"
```

Возвращаемый `DateValue` **иммутабелен** — каждый метод возвращает новое значение, так что его безопасно хранить и переиспользовать. Через `valueOf()` он приводится к epoch-ms, поэтому `+date(x)`, `date(a) - date(b)` и `date(a) < date(b)` просто работают.

Форматирование — `.format`

```ts
.format(pattern = 'D MMMM YYYY', locale?: 'en' | 'ru'): string
```

Форматирует по шаблону из токенов в стиле `day.js` (по умолчанию — длинная дата).

`YYYY` `YY`	год	`2026`, `26`
`MMMM` `MMM` `MM` `M`	месяц	`июля`, `июл`, `07`, `7`
`DD` `D`	день месяца	`05`, `5`
`dddd` `ddd`	день недели	`пятница`, `пт`
`HH` `H`	час (24)	`08`, `8`
`hh` `h`	час (12)	`08`, `8`
`mm` `m`	минута	`05`, `5`
`ss` `s`	секунда	`09`, `9`
`A` `a`	AM/PM	`PM`, `pm`

Символы вне токенов (пробелы, `.`, `:`, `,`) копируются в вывод как есть.

> **ПАДЕЖ МЕСЯЦА**
> Русский пишет месяц по-разному в зависимости от контекста: родительный рядом с числом (`10 июля`), именительный сам по себе (`Июль 2026`). `.format` выбирает форму автоматически — по наличию токена дня (`D`) в шаблоне.

Относительное время — `.timeAgo`

TS

```
.timeAgo(locale?: 'en' | 'ru', now?: Date | number | DateValue): string
```

Человеческое относительное время от `now` (по умолчанию — текущий момент). Выбирает самую крупную подходящую единицу, до ~45 секунд читается как «только что» / “just now” и правильно согласует русские числительные (`1 минуту` / `2 минуты` / `5 минут`).

TS

```
date(ts).timeAgo()      // "5 минут назад"  ·  "5 minutes ago"
date(future).timeAgo()  // "через 2 дня"    ·  "in 2 days"
```

Арифметика — цепочки, иммутабельность

TS

```
.add(n, unit)      .subtract(n, unit)      // unit:
'ms'|'second'|'minute'|'hour'|'day'|'week'|'month'|'year'
.startOf(unit)     .endOf(unit)            // к началу/концу единицы (локальное время)
```

TS

```
date().add(3, 'day').startOf('day').format('D MMMM')   // "13 июля"
date(ts).endOf('month').format('D MMMM')               // "31 июля"
```

`month`/`year` учитывают календарь и обрезаются на коротких месяцах (`Jan 31 + 1 month → Feb 28`) — как в day.js.

Сравнение и извлечение — скаляры

TS

```
.diff(other, unit = 'ms'): number     // целые единицы со знаком, усечение к нулю
.isBefore(other) .isAfter(other) .isSame(other): boolean
.valueOf(): number   // epoch ms      .unix(): number   // epoch seconds
.year() .month() .day() .weekday() .hour() .minute() .second(): number
.toDate(): Date
```

TS

```
date(a).diff(b, 'hour')   // например, -3
date(a).isBefore(b)       // true
```

`.month()` начинается с 0, а `.weekday()` — 0 (воскресенье)–6, оба как у нативных геттеров `Date`.

Смотрите также

- device — `device.language`, источник локали по умолчанию

4.7 Анимация

4.7.1 animate

Одноразовый твин значения на кадровых часах хоста: интерполирует `from → to` за `duration` и вызывает `onUpdate` с текущим значением каждый кадр. Он анимирует *значения*, а не объекты — значение ты применяешь сам в `onUpdate`. Для бесконечного покадрового движения используй `setLoop` (Глобалы хоста).

Обзор

TS

```
const id = animate({
  from: 0,
  to: 1,
  duration: 400,                    // миллисекунды
  easing: easeOut,
  onUpdate: v => { ghost.opacity = v },
  onComplete: () => console.log('faded in'),
})
```

Запуск твина

```ts
animate({
  from: T,                  // начальное значение
  to: T,                    // конечное значение — его форма выбирает интерполяцию (см. ниже)
  duration: number,         // МИЛЛИСЕКУНДЫ
  onUpdate(val: T): void,   // вызывается каждый кадр с интерполированным значением
  onComplete?(): void,      // вызывается один раз, после финального onUpdate
  easing?: Easing,          // отображает линейный прогресс 0..1; по умолчанию линейный
}): number                  // id анимации для stop/pause/resume
```

> **NOTE**
> `duration` — **миллисекунды** (кадровый `dt` — секунды; длительности UI `.animateTo()` тоже в мс). См. Соглашения.

`easing` получает сырой линейный прогресс `0..1` и возвращает сглаженный — см. Функции сглаживания. `onComplete` срабатывает, когда прогресс достигает 1; твин, остановленный раньше, его не вызывает.

Виды значений

Форма `to` выбирает интерполяцию:

`to`	Интерполяция	`onUpdate` получает
`number`	lerp	`number`
массив длины 2 (как `Vec2`)	покомпонентный lerp	`Float32Array` (переиспользуется)
массив длины 3 (как `Vec3`)	покомпонентный lerp	`Float32Array` (переиспользуется)
массив длины 4 (как `Quat`)	**slerp** кватерниона	`Float32Array` (переиспользуется)

> **NOTE**
> для массивных видов `onUpdate` получает **один и тот же `Float32Array` каждый кадр** — он перезаписывается на месте. Не храни ссылку; копируй (`[...v]`), если нужно сохранить значение кадра.

> **NOTE**
> массивы длины 4 всегда трактуются как кватернионы (slerp с переворотом полушария) — не анимируй RGBA-кортеж как массив длины 4.

> **NOT IMPLEMENTED**
> строковые значения (цвета). Ветка цвета в исходнике — мёртвый код (сравнивает значение, а не тип), поэтому строковый `to` проваливается в проверки длины массива и не даёт ничего полезного — NaN на строках из 2/3/4 символов, полный no-op в остальных случаях. Исключение не бросается. Анимируй цвет, твиня число `0..1` и смешивая два цвета сам в `onUpdate`.

Управление запущенным твином

```ts
stopAnimation(id: number): void     // остановить навсегда — больше нет onUpdate, нет onComplete
pauseAnimation(id: number): void    // заморозить прогресс
resumeAnimation(id: number): void   // продолжить приостановленный твин
```

animateMat4 — твин матрицы трансформа

```ts
animateMat4({
  from: Mat4Like,           // Float32Array | number[], 16 элементов
  to: Mat4Like,
  duration: number,         // миллисекунды
  onUpdate(m: Float32Array): void,
  onComplete?(): void,
  easing?: Easing,
}): number
```

Поэлементный lerp матрицы дал бы скос; `animateMat4` вместо этого раскладывает обе матрицы **один раз в момент вызова** на позицию / поворот / масштаб, lerp’ит позицию и масштаб, slerp’ит поворот и пересобирает каждый кадр. `onUpdate` получает переиспользуемый `Float32Array` из 16 элементов — то же правило «не храни ссылку», что и у векторных видов.

Подводные камни

```ts
// ✗ duration в секундах — твин завершится меньше чем за кадр
animate({ from: 0, to: 100, duration: 0.5, onUpdate: v => { node.x = v } })
// ✓ миллисекунды
animate({ from: 0, to: 100, duration: 500, onUpdate: v => { node.x = v } })

// ✗ хранение vec-значения — все записи окажутся одним (финальным) массивом
animate({ from: [0, 0], to: [10, 10], duration: 300, onUpdate: v => path.push(v) })
// ✓ копируй значение кадра
animate({ from: [0, 0], to: [10, 10], duration: 300, onUpdate: v => path.push([...v]) })
```

Смотрите также

- Функции сглаживания — `easeIn` / `easeOut` / `easeInOut`, `cubicBezier`, свои функции.
- Соглашения — время — где SDK использует секунды, а где миллисекунды.
- Глобалы хоста — `setLoop` для бесконечной покадровой анимации.

4.7.2 Функции сглаживания

Функции сглаживания для `animate`. Сглаживание — это просто функция:

```ts
type Easing = (t: number) => number   // линейный прогресс 0..1 → сглаженный прогресс
```

Всё такой формы работает как `animate({ easing })` — встроенные ниже покрывают частые случаи, `cubicBezier` строит остальное, а свои можно написать руками.

Обзор

```ts
animate({ from: 0, to: 300, duration: 400, easing: easeOut,
          onUpdate: v => { card.x = v } })

const overshoot = cubicBezier(0.34, 1.56, 0.64, 1)     // back-out: проходит цель, затем оседает
animate({ from: 0.5, to: 1, duration: 250, easing: overshoot,
          onUpdate: v => { card.scale = v } })
```

Встроенные функции

```ts
easeIn(t): number       // квадратичное ускорение: t²  — медленный старт, быстрый финиш
easeOut(t): number      // квадратичное замедление: 1 - (1 - t)²  — быстрый старт, мягкое оседание
easeInOut(t): number    // easeIn для первой половины, зеркально для второй
```

Все три — обычные квадратичные кривые. `easeOut` — то, что нужно большинству UI-движений (отклик сразу, мягкое оседание).

cubicBezier

```ts
cubicBezier(x1: number, y1: number, x2: number, y2: number): Easing
```

Строит сглаживание из двух контрольных точек — те же параметры, что у CSS `cubic-bezier(x1, y1, x2, y2)`, поэтому любая кривая из галереи CSS-easing переносится напрямую. Возвращаемая функция переиспользуема; создавай её один раз, а не на каждый твин.

- `x1` / `x2` должны оставаться в `0..1` (они параметризуют время).
- `y1` / `y2` могут выходить за `0..1` — так получается overshoot (ощущения back/spring): сглаженный прогресс временно превышает 1, поэтому анимируемое значение проходит `to` и возвращается.
- Концы точны: вход `0` возвращает `0`, вход `1` возвращает `1`.
- `cubicBezier(a, a, b, b)` (контрольные точки на диагонали) возвращает обычное линейное сглаживание.

Смотрите также

- animate — куда подключаются функции сглаживания.

4.8 Математика

4.8.1 Mathf

Скалярная математика — функции «только с числами», к которым тянешься внутри кадрового цикла. `Mathf` — это обычный объект с функциями и константами, ничего создавать не нужно. Векторы, кватернионы и матрицы живут на `Vec2` / `Vec3`, `Quat` и `Mat4`, а не здесь.

Обзор

```ts
const camX = Mathf.smoothDamp(0, 0.15)          // следящий сглаживатель — создаётся ОДИН раз, вне цикла

setLoop(dt => {
  cam.x = camX(player.x, dt)                    // критически задемпфированное слежение камеры
  zoom = Mathf.damp(zoom, targetZoom, 10, dt)   // экспоненциальное сглаживание без состояния
  health = Mathf.clamp(health, 0, 100)
  bar.width = Mathf.lerp(0, 200, Mathf.inverseLerp(0, maxCharge, charge))
})
```

Константы и перевод углов

```ts
Mathf.PI            // Math.PI
Mathf.TAU           // 2π — полный оборот в радианах
Mathf.EPSILON       // 1e-6
Mathf.DEG2RAD       // π / 180
Mathf.RAD2DEG       // 180 / π
Mathf.deg2rad(deg): number
Mathf.rad2deg(rad): number
```

> **ЗАМЕТКА**
>
> `DEG2RAD` / `RAD2DEG` доступны и как «голые» глобалы времени компиляции (без префикса `Mathf.`). Какие API принимают градусы, а какие радианы — смотрите в разделе **Соглашения**.

Диапазоны и интерполяция

```ts
Mathf.clamp(v, min, max): number
Mathf.clamp01(v): number                                  // зажать в [0, 1]
Mathf.lerp(a, b, t): number                               // a + (b − a)·t
Mathf.inverseLerp(a, b, v): number                        // где v лежит в [a, b] как 0..1 (0,
если a === b)
Mathf.remap(v, inMin, inMax, outMin, outMax): number  // перенести v из одного диапазона в
другой
Mathf.sign(x): number                                     // -1, 0 или 1 (Math.sign)
Mathf.repeat(t, length): number                           // зациклить t в [0, length)
Mathf.pingPong(t, length): number                         // t «отскакивает» туда-обратно в [0,
length]
```

Сглаживание к значению

```ts
Mathf.moveTowards(a, b, maxDelta): number         // линейный шаг, никогда не перелетает цель
Mathf.damp(a, b, lambda, dt): number              // экспоненциальное приближение, не зависит
от FPS
```

`damp` — это безопасная по частоте кадров замена `a = lerp(a, b, 0.1)` в цикле: `lambda` — скорость (больше = резче; хорошие значения 5…15), `dt` — секунды на кадр. Функция без состояния — передавайте ей обратно значение с прошлого кадра.

`smoothDamp` — критически задемпфированное слежение (как SmoothDamp в Unity)

```ts
Mathf.smoothDamp(initial: number,   smoothTime, maxSpeed?): Damper<number>
Mathf.smoothDamp(initial: number[], smoothTime, maxSpeed?): Damper<number[]>  // массивы
vec2/vec3

const damper = Mathf.smoothDamp([0, 0], 0.15)
damper(target, dt)          // вызывайте каждый кадр → новое сглаженное значение
damper.setValue(v)          // задать внутреннее состояние (телепорт) — сбрасывает «резинку»
```

Возвращает **сглаживатель с состоянием**, который сам отслеживает свою скорость, поэтому движение плавно разгоняется и тормозит без перелёта. `smoothTime` — примерно время выхода на цель в секундах; `maxSpeed` (по умолчанию `Infinity`) ограничивает скорость — для массивов ограничение применяется к длине вектора, а не покомпонентно.

```ts
const follow = Mathf.smoothDamp([0, 0], 0.15)
setLoop(dt => { scene.camera.position = follow([player.x, player.y], dt) })
```

> **ВНИМАНИЕ**
> Создавайте сглаживатель один раз, вне цикла. Новый сглаживатель каждый кадр сбрасывает скорость и сводит сглаживание на нет. При телепортах вызывайте `setValue`, чтобы следящий объект не гнался за целью со старого места.

Случайные числа

```ts
Mathf.random(min?, max?): number      // дробное в [min, max); по умолчанию 0..1
Mathf.randomInt(min, max): number     // целое, ВКЛЮЧАЯ оба конца
```

Смотрите также

- Vec2 и Vec3 — векторные `lerp` / расстояние / нормализация живут на типах векторов.
- **Соглашения** — `dt` в секундах; правила «градусы против радианов».

4.8.2 Vec2 и Vec3

Векторные типы SDK. Поля — обычные изменяемые числа (`v.x = 3` работает), но **каждый метод чистый**: `a.add(b)` возвращает НОВЫЙ вектор и никогда не трогает `a`. Единственные мутаторы — `set` и `copy`. Оба типа делят один и тот же API; `Vec3` добавляет операции только для 3D (векторное произведение как вектор, преобразования кватернионом и матрицей).

Везде, где принимается вектор, подойдёт и «сырой» кортеж (`Vec2Like` / `Vec3Like`): обёртка, кортеж `[x, y]` / `[x, y, z]`, `number[]` или `Float32Array`. Векторы итерируемы — их можно разворачивать и деструктурировать как кортежи.

Обзор

```ts
const toTarget = new Vec3(target.position).sub(pos)
const dir = toTarget.normalize()                    // чисто — toTarget не меняется
pos = pos.add(dir.scale(speed * dt))                // переприсваиваем; add() возвращает новый вектор

node.position = [0, 1, 0]                           // сырые кортежи работают везде, где ждут Vec3
const [x, y, z] = node.position                     // векторы деструктурируются как кортежи
if (toTarget.lengthSq() < 4) attack()               // дешевле, чем length(), для проверки дистанции
```

Создание

```ts
new Vec2(x?, y?)                // компоненты; по умолчанию (0, 0)
new Vec2([3, 4])                // или копия любого Vec2Like
new Vec3(x?, y?, z?)            // по умолчанию (0, 0, 0)
Vec3.from(v: Vec3Like): Vec3

Vec2.zero / .one / .up / .down / .left / .right
Vec3.zero / .one / .up / .down / .left / .right / .forward / .back
```

> **ЗАМЕТКА**
>
> Статические константы — это геттеры: каждое обращение возвращает свежий экземпляр, поэтому менять его безопасно. `Vec3.forward` — это **−Z** (камеры смотрят вдоль −Z), `Vec3.back` — это +Z. `Vec2.up` — это +Y (2D-мир имеет ось Y вверх).

Мутация — только эти два метода меняют `this`

```ts
v.set(x, y): this               // Vec3: set(x, y, z)
v.copy(other: Vec2Like): this
```

Арифметика (чистая — каждый метод возвращает новый вектор)

```ts
a.add(b)   a.sub(b)   a.mul(b)   a.div(b)     // покомпонентно; b может быть кортежем
a.scale(s)                                     // умножить на скаляр
a.scaleAndAdd(b, s)                            // a + b·s за один вызов
a.negate()
a.withX(x)  a.withY(y)                         // копия с заменой одной компоненты (Vec3:
withZ)
```

Произведения и меры

```ts
a.dot(b): number
a.length(): number          a.lengthSq(): number
a.distanceTo(b): number     a.distanceSqTo(b): number
a.angle(b): number          // угол между векторами, РАДИАНЫ (0, если один из них нулевой)
a.normalize(): Vec          // единичный вектор; нулевой вектор нормализуется в нулевой
```

Для сравнений предпочитайте варианты с `Sq` — они пропускают извлечение корня.

Интерполяция и зажатие

```ts
a.lerp(b, t)                // покомпонентное смешивание, t обычно 0..1
a.clamp(min, max)           // покомпонентный зажим между двумя векторами
a.min(b)   a.max(b)         // покомпонентный минимум / максимум
```

Только Vec2

```ts
a.cross(b): number          // скалярная z-компонента 3D-векторного произведения — знаковая площадь
a.perp(): Vec2              // (-y, x): перпендикуляр, повёрнутый на 90° против часовой
a.rotate(rad, origin?)      // поворот против часовой на РАДИАНЫ вокруг origin (по умолчанию [0, 0])
a.heading(): number         // atan2(y, x) — угол вектора в РАДИАНАХ
```

Только Vec3

```ts
a.cross(b): Vec3
a.reflect(normal)           // отражение относительно ЕДИНИЧНОЙ нормали: v − 2(v·n)n
a.project(onto)             // проекция a на `onto` (нулевой `onto` → нулевой вектор)
a.rotateX(rad, origin?)     // поворот точки вокруг оси через origin, РАДИАНЫ
a.rotateY(rad, origin?)
a.rotateZ(rad, origin?)
a.rotate(q)                 // поворот кватернионом Quat или сырым [x, y, z, w]
a.transform(m)              // преобразование как ТОЧКИ матрицей Mat4 / сырым column-major массивом на 16
                            // — применяет перенос и перспективное деление
```

Для направления (без переноса) используйте `Mat4.transformDirection`, а не `transform`.

Совместимость

```ts
a.equals(b, eps? /* 1e-6 */): boolean   // сравнение с относительным эпсилон
a.clone(): Vec
a.toArray(): [number, number]           // Vec3: [number, number, number]
const [x, y] = a; [...a]                // итерируемый
```

Подводные камни

```ts
// ✗ ждать, что методы мутируют — они этого не делают
dir.normalize()                          // результат отброшен; dir не изменился
dir = dir.normalize()                    // ✓ переприсвоить

// ✗ менять копию трансформа узла — геттеры возвращают свежие значения
node.position.x = 3                      // тихий no-op (см. Соглашения)
node.x = 3                               // ✓
```

Смотрите также

- Mathf — скалярные помощники (`clamp`, `damp`, `smoothDamp` для массивов).
- Quat — повороты; `Vec3.rotate(q)` — это `Quat.rotateVec3` с другой стороны.
- Mat4 — полные преобразования; `transformPoint` / `transformDirection`.

4.8.3 Quat

Кватернион `(x, y, z, w)` — тип поворота за 3D-API узлов (`node.quaternion`). Тот же контракт, что у `Vec3`: изменяемые поля, **чистые методы** (каждый возвращает новый `Quat`; мутируют только `set` / `copy`), а сырой массив `[x, y, z, w]` подойдёт везде, где принимается `Quat` (`QuatLike`). Правая система координат — как в gl-matrix / three.js.

Для повседневных поворотов ты выставляешь `node.eulerAngles` (градусы) и `Quat` не трогаешь вовсе; тянешься к нему, когда надо комбинировать повороты, делать slerp или прицеливаться (`lookRotation`, `fromTo`).

Обзор

```ts
const target = Quat.lookRotation(enemy.position.sub(turret.position))
setLoop(dt => {
  turret.quaternion = new Quat(turret.quaternion).slerp(target, 1 - Math.exp(-8 * dt))
})

const tilt = Quat.fromEuler(0, 0, 15)                    // ГРАДУСЫ
const spin = Quat.fromAxisAngle([0, 1, 0], Math.PI / 2)  // РАДИАНЫ
node.quaternion = spin.mul(tilt)                         // сначала tilt, затем spin
```

Создание

```ts
new Quat()                          // единичный (0, 0, 0, 1)
new Quat(x, y, z, w)
new Quat([x, y, z, w])              // копия любого QuatLike
Quat.from(q: QuatLike): Quat
Quat.identity                       // геттер — свежий единичный при каждом обращении

Quat.fromEuler(x, y, z, order? /* "YXZ" */): Quat     // углы в ГРАДУСАХ
Quat.fromAxisAngle(axis, rad): Quat                   // угол в РАДИАНАХ; ось нормализуется за тебя
Quat.fromTo(a, b): Quat                               // кратчайший поворот, переводящий направление a в b
Quat.lookRotation(forward, up? /* Vec3.up */): Quat   // ориентация так, чтобы локальная −Z смотрела вдоль `forward`
```

> **NOTE**
>
> деление на градусы/радианы следует общему правилу SDK: эйлеровы фабрики принимают **градусы**, аргументы-«сырые углы» — **радианы**. `EulerOrder` — одно из `"XYZ" | "YXZ" | "ZXY" | "ZYX" | "YZX" | "XZY"`; по умолчанию `"YXZ"` — это yaw-pitch-roll (рыскание-тангаж-крен) и совпадает с `node.eulerAngles`.

Композиция и применение

```ts
a.mul(b): Quat              // произведение Гамильтона a ⊗ b — применяет b ПЕРВЫМ, затем a
a.invert(): Quat            // обратный поворот (= conjugate() для единичных кватернионов)
a.conjugate(): Quat
a.normalize(): Quat         // ре-нормализация после накопления ошибки (ноль → единичный)
a.rotateVec3(v): Vec3       // повернуть вектор; то же, что new Vec3(v).rotate(a)
```

Порядок умножения читается справа налево, как у матриц: `yaw.mul(pitch)` сначала наклоняет (pitch) в локальном пространстве, затем рыскает (yaw).

Интерполяция и сравнение

```ts
a.slerp(b, t): Quat          // сферическая интерполяция, t ∈ [0, 1]; идёт коротким путём
a.angle(b): number           // угловое расстояние между двумя единичными кватернионами, РАДИАНЫ
a.dot(b): number
a.equals(b, eps? /* 1e-6 */): boolean       // покомпонентно — считает q и -q РАЗНЫМИ
a.sameRotation(b, eps? /* 1e-6 */): boolean // true, если это один и тот же поворот (±q равны)
```

> **NOTE**
> `q` и `-q` кодируют один и тот же поворот. Для вопроса «смотрит ли туда же» используй `sameRotation`, а не `equals`.

Извлечение углов Эйлера

```ts
q.toEuler(order? /* "YXZ" */): Vec3   // углы в ГРАДУСАХ; обратная к Quat.fromEuler
```

Совместимость

```ts
q.set(x, y, z, w): this      // единственные мутаторы
q.copy(other): this
q.clone(): Quat
q.toArray(): [number, number, number, number]
const [x, y, z, w] = q       // итерируемый
```

Подводные камни

```ts
// ✗ градусы в fromAxisAngle (он принимает радианы)
Quat.fromAxisAngle([0, 1, 0], 90)
Quat.fromAxisAngle([0, 1, 0], 90 * DEG2RAD)     // ✓ — или используй Quat.fromEuler(0, 90, 0)

// ✗ ждать, что a.mul(b) применит сначала a — он применяет сначала b, затем a
world = local.mul(delta)     // поворот на delta в локальном пространстве
world = delta.mul(local)     // ✓ если нужно применить delta в мировом пространстве
```

Смотрите также

- Vec2 и Vec3 — `Vec3.rotate(q)`, помощники направлений (`Vec3.forward` = −Z).

- Mat4 — `Mat4.fromQuat`, `mat.rotation` извлекает `Quat`.
- Соглашения — градусы против радианов, −Z вперёд.

4.8.4 Mat4

Матрица 4×4 в **column-major** (по столбцам) — тип преобразований/проекций в SDK. Тот же контракт, что у `Vec3` / `Quat`: все методы чистые (каждый возвращает новый `Mat4`; мутируют только `set` / `copy`), а везде, где принимается матрица, подойдёт и сырой массив на 16 элементов или `Float32Array` (`Mat4Like`).

В отличие от векторов, внутреннее хранилище открыто: `.m` — это сырой `number[]` в column-major (длина 16), его можно читать и писать напрямую.

Обзор

```ts
const world = Mat4.compose([0, 1, -3], Quat.fromEuler(0, 45, 0), 1.5)  // T · R · S
const p = world.transformPoint([0, 0, -1])          // локальная точка → мир
const { position, rotation, scale } = world.decompose()

const view = Mat4.lookAt([0, 2, 5], [0, 0, 0])      // камера в (0,2,5) смотрит в начало координат
const proj = Mat4.perspective(60 * DEG2RAD, w / h, 0.1, 100)
const mvp = proj.mul(view).mul(world)               // справа налево: world, затем view, затем proj
```

Сырой массив: `.m`

```ts
mat.m: number[]             // длина 16, column-major; перенос лежит в m[12..14]
mat.m[12] += vx * dt        // прямая правка чисел — .m открыт намеренно
mat.toFloat32Array()        // выгрузить копию как Float32Array для хоста/GPU
```

Текучие методы никогда не меняют `.m` — они возвращают новый `Mat4`, — поэтому запись в `.m` это единственный способ изменить матрицу на месте помимо `set` / `copy`.

Создание

```ts
new Mat4()                          // единичная
new Mat4(src: Mat4Like)             // копирует src (Mat4, его .m или сырой массив на 16)
Mat4.identity(): Mat4
Mat4.from(src: Mat4Like): Mat4

Mat4.compose(position, rotation, scale? /* 1 */): Mat4  // T · R · S; scale: вектор или скаляр
Mat4.fromTranslation(v): Mat4
Mat4.fromScale(v /* vector or scalar */): Mat4
Mat4.fromQuat(q): Mat4
Mat4.fromEuler(x, y, z, order? /* "YXZ" */): Mat4       // углы в ГРАДУСАХ
```

Композиция

```ts
a.mul(b): Mat4              // a · b — применяет b ПЕРВЫМ, затем a (справа налево, как в glsl)
a.premul(b): Mat4           // b · a — с другой стороны
a.invert(): Mat4            // вырожденные матрицы возвращают единичную
a.transpose(): Mat4
a.determinant(): number
a.equals(b, eps? /* 1e-6 */): boolean
```

Построение преобразований (локальное пространство, пост-умножение)

```ts
m.translate(v): Mat4            // m · T(v)
m.rotate(rad, axis): Mat4       // m · R(axis, rad) — угол в РАДИАНАХ, ось нормализуется за тебя
m.rotateX(rad)  m.rotateY(rad)  m.rotateZ(rad)
m.scale(v /* vector or scalar */): Mat4
```

Каждый применяет своё преобразование в **локальном** пространстве матрицы (пост-умножение), поэтому цепочка читается как спуск по графу сцены: `Mat4.fromTranslation(pos).rotateY(a).scale(2)`.

Преобразование векторов

```ts
m.transformPoint(v): Vec3         // полное преобразование: перенос + перспективное деление
m.transformDirection(v): Vec3     // только поворот/масштаб — без переноса и деления
```

`Vec3.transform(m)` — это `transformPoint` со стороны вектора.

Разложение

```ts
m.position: Vec3                  // геттер — столбец переноса (m[12..14])
m.scaling: Vec3                   // масштаб по осям (длины базисных векторов)
m.rotation: Quat                  // поворот с вычтенным масштабом
m.eulerAngles: Vec3               // ГРАДУСЫ, порядок "YXZ"
m.toEuler(order? /* "YXZ" */): Vec3
m.decompose(): { position: Vec3, rotation: Quat, scale: Vec3 }
m.basisX / m.basisY / m.basisZ: Vec3   // локальные оси в мировом пространстве (столбцы
0/1/2)
```

Все геттеры возвращают свежие значения — изменение `m.position` не записывается обратно в матрицу (для этого пиши прямо в `m.m[12..14]`).

Камеры и проекция

```ts
Mat4.lookAt(eye, center, up? /* Vec3.up */): Mat4     // матрица ВИДА (мир → камера)
Mat4.targetTo(eye, target, up? /* Vec3.up */): Mat4   // МИРОВАЯ матрица: помещает объект в
eye,
                                                      // ориентируя его лицом к target
Mat4.perspective(fovy, aspect, near, far): Mat4       // fovy в РАДИАНАХ; far может быть
Infinity
Mat4.ortho(left, right, bottom, top, near, far): Mat4
```

> **NOTE**
>
> проекции используют clip-конвенцию WebGL/OpenGL (NDC z ∈ [−1, 1]). `lookAt` против `targetTo` — классическая ловушка: `lookAt` строит *обратную* матрицу (вида); чтобы развернуть узел к чему-то, нужен `targetTo` (или `Quat.lookRotation`).

Совместимость

```ts
m.set(values: ArrayLike<number>): this   // единственные мутаторы (помимо записи в .m)
m.copy(src: Mat4Like): this
m.clone(): Mat4
m.toArray(): number[]                    // копия .m
m.toFloat32Array(): Float32Array
```

Подводные камни

```ts
// ✗ градусы в rotate/perspective — аргументы-«сырые углы» это радианы
m.rotateY(90);            Mat4.perspective(60, aspect, 0.1, 100)
m.rotateY(90 * DEG2RAD);  Mat4.perspective(60 * DEG2RAD, aspect, 0.1, 100)   // ✓

// ✗ считать .m row-major — он column-major; перенос это m[12], m[13], m[14]
m.m[3] = x
m.m[12] = x   // ✓
```

Смотрите также

- Vec2 и Vec3 — `Vec3.transform(m)`, направления базиса.
- Quat — `Quat.fromEuler` / `lookRotation`; `mat.rotation` совместим с `Mat4.fromQuat` в обе стороны.
- Соглашения — градусы против радианов, −Z вперёд, Y вверх.

4.8.5 Color

Общий парсер/конвертер цвета. Напрямую его зовёшь редко: везде, где SDK принимает цвет (`sprite.color`, цвета материалов, UI-стили), берётся `ColorInput` и внутренне конвертируется через этот модуль. Тянешься к `Color` напрямую, когда нужна конкретная числовая форма самому — например, скормить упакованный int в uniform шейдера или смешать каналы вручную.

Обзор

```ts
sprite.color = '#e33'                      // на цветовых свойствах работает любой ColorInput

const [r, g, b] = Color.toRgb01('#10131a')       // [0.062…, 0.074…, 0.101…]
const packed = Color.toPackedRgb([1, 0.5, 0])    // 0xff8000
const css = Color.toHexString(0xff8000)          // '#ff8000'
```

Допустимые входы (`ColorInput`)

```ts
'#e33'  '#e33c'                  // сокращённый hex — #rgb / #rgba, каждая цифра удваивается
'#ff3333'  '#ff3333cc'           // hex — #rrggbb / #rrggbbaa (ведущий '#' необязателен)
0xff3333                         // упакованный int 0xRRGGBB — альфа всегда 1
[1, 0.2, 0.2]                    // [r, g, b] дробные, 0..1
[1, 0.2, 0.2, 0.8]               // [r, g, b, a] дробные, 0..1
```

> **NOTE**
> парсятся только hex-строки. CSS-функциональная строка (`'rgba(255, 51, 51, 0.8)'`, `'red'`) считается некорректной и откатывается к **непрозрачному чёрному** — исключения не бросает. Для альфы в строке используй форму `#rrggbbaa` или массив дробных.

> **NOTE**
> упакованный int на *входе* не несёт альфу — `0xff3333cc` читается как 24-битный RGB, не RGBA. Чтобы передать альфу, используй строку (`'#ff3333cc'`) или массив из 4 элементов.

Конвертеры

```ts
Color.toRgb01(c): [r, g, b]              // дробные 0..1 (альфа отброшена)
Color.toRgba01(c): [r, g, b, a]          // дробные 0..1; входы-массивы проходят без потерь
Color.toPackedRgb(c): number             // 0xRRGGBB
Color.toPackedRgba(c): number            // 0xRRGGBBAA, беззнаковый (безопасно сравнивать/сдвигать)
Color.toHexString(c): string             // '#rrggbb' или '#rrggbbaa', когда альфа < 1
```

Смотрите также

- Соглашения — где принимается `ColorInput`.
- Mathf — `lerp` / `clamp01` для смешивания каналов, вытянутых через `toRgba01`.

4.9 Плагины

4.9.1 CameraView

Камера устройства — живой предпросмотр плюс съёмка фото — как Presentable: открой её на весь экран, сделай `Router.push` или встрой среди детей экрана как узел живого предпросмотра. `takePhoto()` возвращает `File` (хэндл буфера хоста), поэтому снимок можно сразу отправить в текстуру (`Texture2D.load(file)`), на загрузку (`FormData`) или в шторку шеринга (`share(file)`). **Опционален по хосту** — проверяй через `CameraView.isSupported`.

Обзор

```ts
const camera = CameraView()               // задняя камера по умолчанию
await camera.open()                       // спрашивает разрешение, показывает предпросмотр
const photo = await camera.takePhoto()
camera.close()

share(photo, 'My shot')
```

API

```ts
CameraView.isSupported: boolean                          // статик — есть ли у этого хоста камера?

CameraView(options?: { facingMode?: 'front' | 'back' }) // по умолчанию 'back'
camera.open(opts?: { transition }): Promise<void>       // запросить разрешение, показать предпросмотр
camera.setFacingMode(mode: 'front' | 'back'): Promise<void>   // переключить, пока предпросмотр живой
camera.takePhoto(): Promise<File>                        // фото, закодированное хостом
camera.close(): void                                     // закрыть, освободить камеру
```

- `open()` следует контракту «подготовить, затем показать» (как `ARScene` / `QRScanner`): **текущий пункт назначения остаётся видимым**, пока запрашивается разрешение на камеру; переключение — и `transition`, если ты его передал — происходит, когда камера готова. Промис отклоняется, если в разрешении отказано, у хоста нет камеры или его опередила другая навигация — обработай это.
- Камера — полноценный `NativeView` поверх зарегистрированного хостом вью `"camera"`: `Router.push(camera)` работает, и она встраивается как узел лэйаута — живой видоискатель внутри твоего экрана, размером по своим стилям. `isSupported` — ровно «зарегистрировал ли этот хост это вью».
- Каждый `CameraView()` — отдельный экземпляр, но камера устройства — одна сессия: не держи открытыми две сразу.

> **NOTE**
>
> жизненный цикл — камера остаётся занятой до `close()`. Всегда закрывай при уходе с экрана (`onClose`), иначе камера (и её системный индикатор) останется включённой.

Смотрите также

- QRScanner — сканирующая разновидность камеры.

- Файлы и шеринг — `share()` снятого фото; `openFilePicker` как не-камерный способ получить изображение.
- Сеть — хэндл `File` и загрузка через `FormData`.

4.9.2 QRScanner

QR-сканер камеры хоста как Presentable: открой его на весь экран, сделай `Router.push` или встрой среди детей экрана как узел живого предпросмотра. **Опционален по хосту** — всегда проверяй через `QRScanner.isSupported`, прежде чем предлагать эту возможность.

Обзор

```ts
if (QRScanner.isSupported) {
  const scanner = QRScanner().onScan(data => {
    if (data === null) return          // кадр камеры без читаемого кода
    console.log('scanned:', data)
    scanner.close()
  })
  await scanner.open()                 // спрашивает разрешение на камеру, затем показывает сканер
}
```

API

```ts
QRScanner.isSupported: boolean                          // статик — умеет ли этот хост вообще сканировать?

QRScanner(): QRScanner                                  // создать вью сканера
scanner.onScan(cb: (data: string | null) => void): this   // срабатывает на каждый декодированный кадр
scanner.open(opts?: { transition }): Promise<void>   // запросить доступ к камере, показать сканер
scanner.close(): void                                   // закрыть, освободить камеру
```

- `onScan` срабатывает на каждый кадр, который декодирует сканер; `null` означает кадр без читаемого кода, поэтому жди его повторно, пока пользователь наводит. Один и тот же код может сообщаться не раз — делай дебаунс/`close()` сам, когда получил нужное.
- `open()` следует контракту «подготовить, затем показать» (как `ARScene`): **текущий пункт назначения остаётся видимым**, пока запрашивается разрешение на камеру; переключение — и `transition`, если ты его передал — происходит, когда сканер готов. Промис отклоняется, если в камере отказано, хост не умеет сканировать или пока он готовился, случилась другая навигация.

- Сканер — полноценный `NativeView` поверх зарегистрированного хостом вью `"qrScanner"`: `Router.push(scanner)` работает (жест «назад» его выталкивает), и он встраивается как узел лэйаута — помести его среди детей экрана, размером по своим стилям. `isSupported` — ровно «зарегистрировал ли этот хост это вью».

> **NOTE**
>
> в отличие от физики (которая молча делает no-op при отсутствии), `open()` **бросает** `Error` на хостах без сканера. `isSupported` — это защита — проверяй её, прежде чем вообще показывать кнопку «Сканировать». См. гейтинг хоста.

> **NOTE**
>
> жизненный цикл — сканер держит камеру до `close()` (или пока роутер его не вытолкнет). Если открываешь его из флоу экранов, закрывай после успешного скана, иначе камера останется занятой.

Смотрите также

- CameraView — обычный предпросмотр камеры + съёмка фото.
- Экраны и Router — пункты назначения, переходы, `Router.push`.
- Соглашения — гейтинг хоста — другие защиты возможностей (`Physics.supported`, `Physics2D.supported`).

4.9.3 Geolocation

Позиция устройства — разовые фиксы и непрерывное слежение. В отличие от QRScanner / CameraView, здесь нет вьюхи, которую нужно показывать: Geolocation — *сервисный* плагин, типизированная обёртка над зарегистрированным хостом сервисом `"geolocation"`. **Опционален по хосту** — всегда проверяй `Geolocation.isSupported`, прежде чем предлагать функции геолокации.

Обзор

```ts
if (Geolocation.isSupported) {
  const pos = await Geolocation.getCurrent()
  console.log(pos.latitude, pos.longitude, '±' + pos.accuracy + 'm')
}
```

API

```ts
Geolocation.isSupported: boolean            // статик — зарегистрировал ли этот хост сервис?

Geolocation.getCurrent(options?): Promise<GeoPosition>
Geolocation.watch(cb: (pos: GeoPosition) => void, options?): Promise<{ stop(): void }>

interface GeoPosition {
  latitude: number; longitude: number
  accuracy: number                 // горизонтальный радиус, метры
  altitude: number | null         // метры, null если неизвестно
  heading: number | null           // градусы по часовой от севера, null в покое
  speed: number | null             // м/с, null если неизвестно
  timestamp: number                // когда получен фикс, мс epoch
}
interface GeoOptions {
  highAccuracy?: boolean           // предпочесть GPS: медленнее первый фикс, больше батареи
(по умолчанию false)
  timeout?: number                 // только getCurrent: отклонить "timeout" спустя столько мс
}
```

- **Системный запрос разрешения происходит при первом вызове** (`getCurrent` или `watch`) — и никогда раньше. Отказ — это reject (`Error("denied")`); жди также `"unavailable"` (нет провайдера) и `"timeout"`. На хостах без сервиса оба метода отклоняются сразу, до любого пути с разрешением.
- `watch()` разрешается, когда хост действительно начал слежение (разрешение выдано, провайдер запущен), поэтому отказ — это reject, а не молча мёртвый колбэк. Дальше обновления приходят с той частотой, с которой их отдаёт провайдер платформы.
- Одно хостовое слежение обслуживает всех подписчиков: два `watch()` стоят одной GPS-сессии, и побеждают опции того, который её запустил. У каждой подписки свой `stop()`; последний `stop()` освобождает провайдер.

> **NOTE**
> жизненный цикл — работающий watch держит железо геолокации (и индикатор ОС) включённым. Всегда вызывай `stop()`, уходя с экрана (`onClose`) — то же правило, что закрытие камеры.

Слежение за позицией

```ts
const screen = UIScreen(
  UIText(() => label.value).style({ fontSize: 20 }),
)
const label = signal('locating…')

const watch = await Geolocation.watch(pos => {
  label.value = pos.latitude.toFixed(5) + ', ' + pos.longitude.toFixed(5)
})
screen.onClose(() => watch.stop())
```

Смотрите также

- Соглашения — гейтинг хоста — защиты `isSupported` (`Physics.supported`, `QRScanner.isSupported`)
- QRScanner / CameraView — плагины в форме вьюхи; Geolocation — та же идея минус UI-поверхность

4.9.4 Push

Удалённые push-уведомления. Как и Geolocation, это *сервисный* плагин — типизированная обёртка над зарегистрированным хостом сервисом `"push"` — но большая часть механики живёт вне твоего приложения: **хост** владеет системным токеном, запросом разрешения и регистрацией на бэкенде le.codes; **le.codes ретранслирует твои отправки** в APNs/FCM, так что в схеме по умолчанию (приложение LeCodes) ты вообще не касаешься учётных данных Apple/Google. **Опционален по хосту** — всегда проверяй `Push.isSupported`.

Обзор

```ts
if (Push.isSupported) {
  const { address } = await Push.register({ user: 'u42' })
  // отдай `address` своему бэкенду — или пропусти его вовсе и таргетируй по id `user`
}
Push.addEventListener('message', p => toast(p.title))   // пришло, пока приложение открыто
```

Дальше твой сервер (или curl) отправляет через реле:

```
POST https://le.codes/api/projects/<projectUuid>/push/send
Authorization: Bearer lecodes_pk_…        ← a project push key le.codes minted for you (see below)
{ "title": "Order shipped", "users": ["u42"], "url": "myapp://orders/42" }
```

API

```ts
Push.isSupported: boolean                   // зарегистрировал ли этот хост сервис?

Push.getStatus(): Promise<PushStatus>       // никогда не показывает запрос разрешения
Push.register(options?: { user?: string }): Promise<{ address: string }>
Push.unregister(): Promise<void>
Push.getLaunch(): Promise<PushPayload | null>
Push.addEventListener('message' | 'tap', cb: (p: PushPayload) => void): void
Push.removeEventListener('message' | 'tap', cb): void

interface PushPayload {
  title: string
  body?: string
  url?: string                    // deep link — также приходит через app.launchUrl / событие
"url"
  data?: Record<string, any>      // твой JSON, ≤ 2 КБ в сериализованном виде
  badge?: number
}
interface PushStatus {
  permission: 'granted' | 'denied' | 'prompt'
  registered: boolean
}
```

- **Системный запрос разрешения происходит внутри `register()`** — и никогда раньше. `register()` идемпотентен (безопасен на каждом запуске) и отклоняется с `"denied"` / `"unavailable"`. На хостах без сервиса каждый метод отклоняется сразу, до любого пути с разрешением.
- Когда приходит push, твоё приложение **не запущено** — уведомление показывает ОС. Приложение видит ровно три пути доставки: событие `"message"` (пришло, пока приложение было открыто — без баннера), событие `"tap"` (тапнули, пока приложение работало) и `Push.getLaunch()` (уведомление, холодным стартом запустившее этот сеанс).
- **`url` в полезной нагрузке — одновременно deep link**: он доставляется и через `app.launchUrl` / событие приложения `"url"`, так что приложения с маршрутизацией по ссылкам получают маршрутизацию уведомлений, не трогая `Push`.

Идентификация ТВОИХ пользователей

`register({ user })` помечает устройство собственным id пользователя твоего приложения, а API отправки принимает `users: ["u42"]` — один пользователь разворачивается во все его устройства, и твой бэкенд никогда не хранит адреса. Повторная регистрация без `user` снимает метку (логаут).

> **NOTE**
> id пользователя **утверждается клиентом** — любое устройство с твоим приложением может заявить любой id и заодно получать уведомления этого пользователя. Годится для «заказ отправлен»; никогда не таргетируй секреты по id пользователя.

Отправка

Во время разработки учётные данные не нужны вовсе — CLI отправляет от твоего логина:

```
lecodes pn devices                                  # did the device actually register?
lecodes pn send --title "Order shipped" --user u42
```

Для собственного сервера используй **проектный push-ключ** (`lecodes_pk_…`). le.codes его *генерирует* — выбрать свой нельзя, и показывается он ровно один раз, так что сохрани его при создании:

```
lecodes pn keys new "prod server" --env             # writes LECODES_PUSH_KEY= into .env
```

или настройки проекта → **Push notifications** → *New key* (только владелец; панель заодно даёт готовый запрос). Ключ может отправлять push-уведомления ТОЛЬКО для своего проекта — намеренно бесполезен для чего-либо ещё, в отличие от персонального токена доступа, поэтому деплоить нужно именно его. Потерянные ключи не восстанавливаются: выпусти новый и отзови старый. Персональные токены с доступом разработчика тоже работают на этом маршруте (именно их использует `lecodes pn send`).

Таргетинг: `to: [address…]`, `users: [id…]` или оба сразу — ответ `{ sent, failed: [{address, reason}] }`. Неизвестные адреса отклоняют весь батч; пользователь без устройств — просто ноль отправок.

> **NOTE**
> push нужен *атрибутированный* мир — приложение LeCodes знает, какой проект оно запускает, только когда это объявил доверенный лончер (запуски опубликованных проектов по QR / ссылке / из онбординга). Бандлы `lecodes dev` не атрибутированы, поэтому `Push.register()` там отклоняется с `"unavailable"` — тестируй push через опубликованный проект, открытый по QR.

Собственное приложение (standalone-оболочка)

Тот же бандл и тот же код работают в оболочке `lecodes app`: оболочка регистрирует тот же сервис `"push"` со своим bundle id и твоим собственным ключом APNs (загруженным в настройки проекта), а твой сервер шлёт точно такой же запрос — реле подбирает учётные данные по регистрации.

Смотрите также

- Geolocation — паттерн сервисного плагина, которому следует Push
- app — `app.launchUrl` + событие `"url"` (путь deep link)
- Соглашения — гейтинг хоста

4.9.5 Service

Прямой доступ к зарегистрированному хостом **headless-сервису** — собрату `NativeView` без UI. Сервис — это нативный код, который хост зарегистрировал под именем (`engine.registerService("app.battery") { params, channel in … }` на iOS); приложение общается с ним через JSON-сессию вызовов/событий. Первопартийные сервисы поставляются с типизированными обёртками (Geolocation, Push) — `Service` для всего остального: **нативного кода, который ты написал в собственной оболочке приложения**, и сторонних плагинов без библиотеки-обёртки.

Опционален по хосту, как и любой плагин: веб-редактор не регистрирует ни одного сервиса, поэтому всегда проверяй `Service.isSupported(name)`.

TS

```
if (Service.isSupported("app.battery")) {
  const battery = Service("app.battery")
  battery.on("change", data => { levelText.text = data.level + "%" })
  const { level, charging } = await battery.call("level")
}
```

API

- `Service(name, params?)` — клиент именованного сервиса. Сессия хоста открывается лениво при первом `call()`; в этот момент `params` передаются фабрике хоста.
- `Service.isSupported(name)` — зарегистрировал ли этот хост фабрику под именем `name`.
- `.call(method, ...args)` — вызвать метод; аргументы и результат сериализуются в JSON. Отклоняется, если у хоста нет такого сервиса или нативная сторона сообщила об ошибке.
- `.on(event, cb)` / `.off(event, cb)` — подписка на события, которые сервис шлёт, пока открыта сессия. Полезные нагрузки приходят уже декодированными из JSON.
- `.close()` — закрыть сессию хоста (нативная сторона останавливает свою работу). Слушатели остаются зарегистрированными; последующий `call()` открывает свежую сессию, которая снова доставляет им события.

Именование

Первопартийные сервисы владеют голыми именами (`geolocation`, `push`). Локальные сервисы приложения называй `app.*` (`app.battery`), сервисы сторонних плагинов — с вендорным префиксом (`acme.bluetooth`) — голые имена зарезервированы за SDK.

Для разового **синхронного значения** (без событий, без async) хостовый `engine.registerCallback("native.thing") { … }`, внедряющий обычный глобал, может быть проще сервиса — см. документацию оболочки приложения. Берись за `Service`, как только нужны события, асинхронная работа или состояние сессии.