23 · Custom Painting

Flutter Practice Notes

Contents

23 · Custom Painting

`CustomPainter` & `Canvas`

Paths, Shapes, Gradients & Clipping

Custom `RenderObject`s (Bespoke Layout + Paint)

Shaders & Effects (Fragment Shaders, `ImageFilter`, Blend Modes)

Custom Painting Performance

23 · Custom Painting

Introduction

When widgets can't express your visual — charts, gauges, signatures, custom shapes, backgrounds, effects — you drop to the **canvas**. This module covers `CustomPainter` / `Canvas`, paths/shapes/gradients, custom `RenderObject`s for bespoke layout/paint, shaders/effects, and keeping it all fast.

Why this module exists

Widget composition covers most UIs, but data-viz, drawing tools, and unique brand visuals need pixel-level control. Custom painting gives it — grounded in the paint phase (09 · paint_phase) — and must be done efficiently (`shouldRepaint`, isolation) to avoid jank.

Module map

#	File	Topic	Level
1	01_custompainter_and_canvas.md	`CustomPaint` / `CustomPainter` / `Canvas` / `Paint`	🔵
2	02_paths_shapes_gradients.md	`Path`, shapes, gradients, clipping	🔵
3	03_custom_renderobject.md	Bespoke layout + paint via `RenderObject`	🔴
4	04_shaders_and_effects.md	Fragment shaders, `ImageFilter`, blend modes	🔴
5	05_custom_painting_performance.md	Keeping custom paint fast	🔴

> **Cross-references:** Paint phase: 09 · paint_phase. Layout phase / render objects: 09 · layout_phase. Repaint isolation: 09 · compositing, 21 · jank_and_raster. Animation (driving repaints): Module 22. `dart:ui`: 10 · engine_internals.

Prerequisites

09 Rendering Pipeline (paint/layout), 22 Animations, 21 · jank_and_raster.

What you'll be able to do after this module

- Draw with `CustomPainter` / `Canvas` and implement `shouldRepaint` correctly.
- Build shapes/paths/gradients and clip regions.
- Write custom `RenderObject`s for bespoke layout + painting.
- Apply shaders/filters/blend modes.

- Keep custom painting smooth (isolation, caching, cheap paint).

Capstone

Animated gauge + signature pad: A `CustomPainter` gauge driven by an animation (correct `shouldRepaint`, isolated with `RepaintBoundary`), and a path-based signature pad capturing drag input — both profiled to hold 60fps.

Summary

Custom painting is `Canvas` drawing under a `CustomPainter` (or a full `RenderObject` for layout+paint). Master paths/gradients/shaders, implement `shouldRepaint`, isolate repaints, and cache expensive draws — the toolkit for charts, gauges, drawing tools, and bespoke visuals.

`CustomPainter` & `Canvas`

> `CustomPaint` hands a `CustomPainter` a `Canvas` and a `Size`; you issue draw commands (`drawRect` / `drawCircle` / `drawPath` / `drawText`) with a `Paint`, and implement `shouldRepaint` to control when it re-draws — the entry point to pixel-level drawing.

Introduction

`CustomPaint` is the widget that gives you a canvas; `CustomPainter` is where you draw. This file covers the `paint(canvas, size)` method, the `Paint` object (color/style/stroke), coordinate space, drawing text, and the critical `shouldRepaint`.

Why this concept exists

Widgets describe *composed* UI; some visuals (a ring progress, a chart, a waveform) are inherently drawn, not composed. `CustomPainter` exposes the same low-level canvas the framework uses in the paint phase (09 · paint_phase), letting you draw anything.

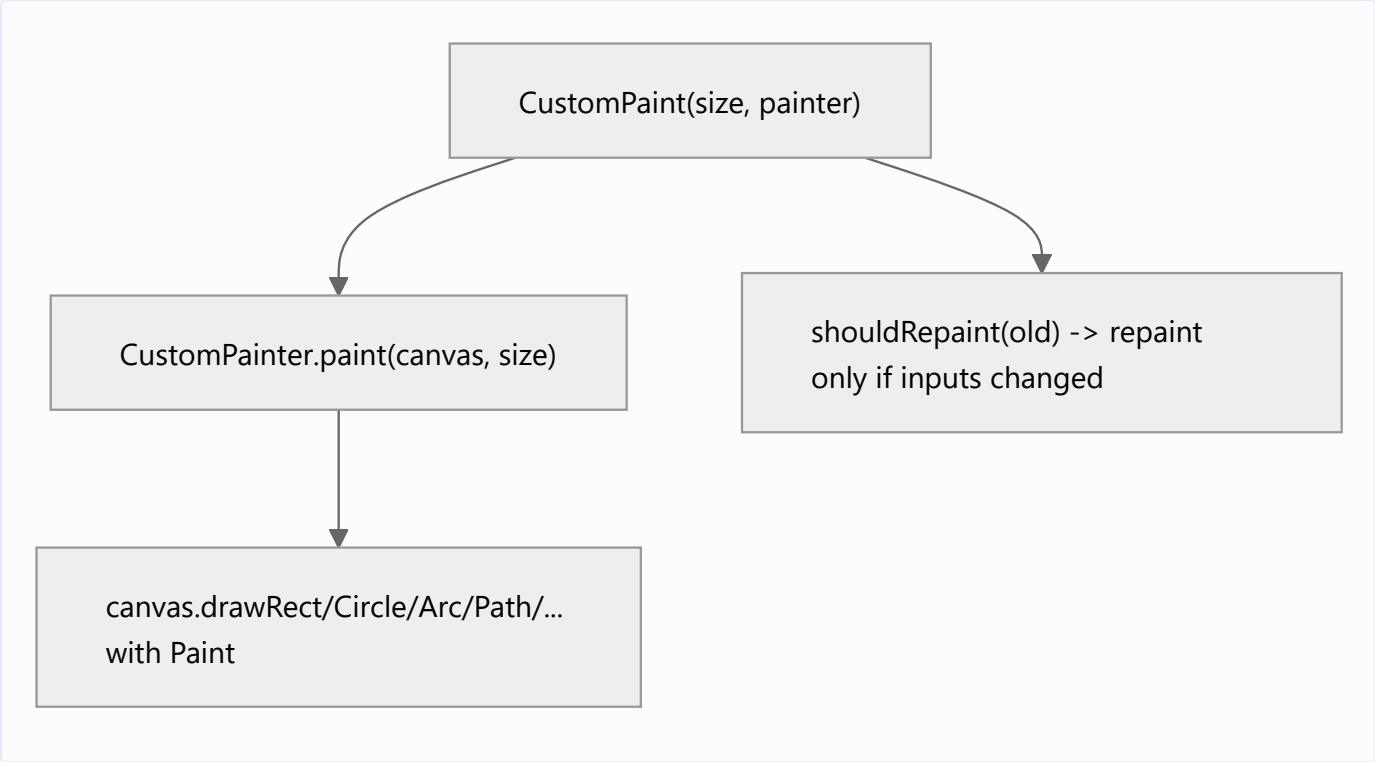
Real-world analogy

`CustomPaint` gives you a **blank canvas and brushes**; `CustomPainter.paint` is you **painting**. `Paint` is your **brush settings** (color, thickness, fill vs outline). `shouldRepaint` is deciding **whether the scene changed enough to repaint** rather than redoing the whole painting every frame.

Problem Statement

Draw a circular progress ring (arc) with a label, styled with stroke width and color, that repaints only when its progress value changes. You'll use `CustomPaint` / `CustomPainter`, a `Paint`, `drawArc`, text, and `shouldRepaint`.

Internal Working



- `CustomPaint(size:, painter:, foregroundPainter:, child:)` : allocates a canvas of `size` (or sizes to `child`); `painter` draws behind, `foregroundPainter` in front.
- `CustomPainter.paint(Canvas canvas, Size size)` : issue commands; `size` is the drawing area (top-left origin, y-down).
- `Paint` : `color`, `style` (`fill` / `stroke`), `strokeWidth`, `strokeCap`, `shader` (gradients — 02_paths_shapes_gradients.md), `blendMode`, `isAntiAlias`.
- **Canvas commands**: `drawRect`, `drawRRect`, `drawCircle`, `drawArc`, `drawLine`, `drawPath`, `drawImage`, plus `save` / `restore` / `translate` / `rotate` / `scale` / `clip*` for transforms.
- **Text**: use a `TextPainter` (`layout()` then `paint(canvas, offset)`) — `Canvas` has no direct text method.
- `shouldRepaint(oldDelegate)` : return `true` only when inputs changed (compare fields); `false`/default-true wrongly repaints every frame — a top perf bug (09 · paint_phase, 05_custom_painting_performance.md).
- Optional: `shouldRebuildSemantics`, `hitTest` for interactivity.

Memory Representation

The painter records draw ops into a layer (`Picture`) each paint; the framework rasterizes it (09 · rasterization). Reuse `Paint` / `TextPainter` objects to avoid per-frame allocation (05_custom_painting_performance.md).

Compiler Behavior

Not applicable.

Runtime Behavior

`paint` runs when the widget paints and whenever `shouldRepaint` returns true (or a repaint is triggered, e.g., by an animation via `repaint:`). Heavy paint ops cost raster time.

Flutter Engine Behavior

Draw commands become GPU operations via Skia/Impeller on the raster thread; complex paths/effects cost raster time (09 · rasterization).

Dart VM Behavior

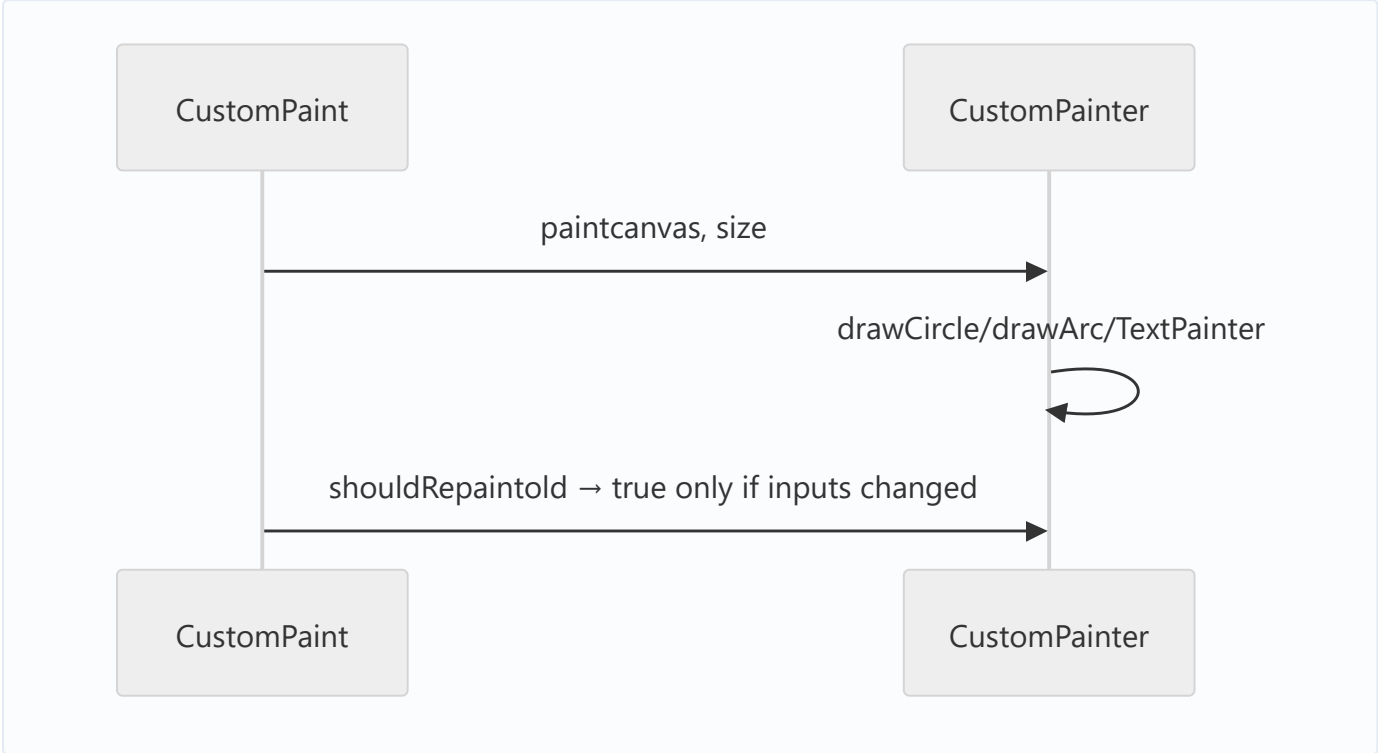
Not applicable.

Examples

```
import 'dart:math';
import 'package:flutter/material.dart';

class RingPainter extends CustomPainter {
  final double progress; // 0..1
  final Color color;
  RingPainter({required this.progress, required this.color});

  @override
  void paint(Canvas canvas, Size size) {
    final center = size.center(Offset.zero);
    final radius = min(size.width, size.height) / 2 - 6;

    final track = Paint()
      ..style = PaintingStyle.stroke ..strokeWidth = 8 ..color = color.withOpacity(0.2);
    final arc = Paint()
      ..style = PaintingStyle.stroke ..strokeWidth = 8 ..strokeCap = StrokeCap.round ..color = color;

    canvas.drawCircle(center, radius, track);                // background track
    canvas.drawArc(                                          // progress arc
      Rect.fromCircle(center: center, radius: radius),
```

```
      -pi / 2, 2 * pi * progress, false, arc,
    );

    // Text via TextPainter (Canvas has no drawText):
    final tp = TextPainter(
      text: TextSpan(text: '${(progress * 100).round()}%',
          style: TextStyle(color: color, fontSize: 20, fontWeight: FontWeight.bold)),
      textDirection: TextDirection.ltr,
    )..layout();
    tp.paint(canvas, center - Offset(tp.width / 2, tp.height / 2));
  }

  @override
  bool shouldRepaint(RingPainter old) =>
      old.progress != progress || old.color != color; // repaint ONLY on change
}

class ProgressRing extends StatelessWidget {
  final double progress;
  const ProgressRing({super.key, required this.progress});
  @override
  Widget build(BuildContext context) => CustomPaint(
        size: const Size(120, 120),
        painter: RingPainter(progress: progress, color: Colors.teal),
      );
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
`shouldRepaint => true` /default	Repaints every frame → jank	Compare inputs; return false when unchanged
Allocating `Paint` / `TextPainter` per paint	GC churn	Reuse/cache them (05_custom_painting_performance.md)
Using `canvas.drawText` -style call	No such method	Use `TextPainter.layout()` + `paint`
Ignoring `size` / hardcoding coords	Breaks on resize	Compute from `size`
Forgetting `save` / `restore` around transforms	State leaks between draws	Wrap transforms in `save` / `restore`
Heavy paint each frame	Raster jank	Cache/simplify; isolate with `RepaintBoundary`

Best Practices

- Implement `shouldRepaint` to repaint only on input changes (the #1 custom-paint perf rule).
- Compute geometry from `size` (resize-safe); wrap transforms in `save` / `restore`.
- Draw text with `TextPainter` (layout once, reuse where possible).
- Reuse `Paint` objects; keep `paint` **cheap**; **isolate** with `RepaintBoundary` (esp. when animated) (21 · jank_and_raster).

- Drive animated repaints via the painter's `repaint:` `Listenable` (e.g., an `AnimationController`) instead of rebuilding the widget.

Performance

Paint recording is UI-thread; the ops rasterize on the GPU. Correct `shouldRepaint` + isolation + cheap ops keep it smooth; complex paths/effects add raster cost (05_custom_painting_performance.md).

Advantages / Disadvantages

- **+** Total visual control (charts/gauges/drawings/backgrounds), efficient when done right, integrates with animation.
- **−** Lower-level/imperative, easy to over-repaint, manual geometry/text, requires perf discipline.

Interview Questions

1. 🟢 **What does `CustomPainter` give you?** — A `Canvas` + `Size` to issue low-level draw commands (shapes/paths/text) with a `Paint`.
2. 🟢 **What is `shouldRepaint` for?** — To tell the framework whether inputs changed so it repaints only when needed — returning true/default repaints every frame.
3. 🟡 **How do you draw text on a canvas?** — With a `TextPainter` (`layout()` then `paint(canvas, offset)`); `Canvas` has no text method.
4. 🟡 **What does `Paint` control?** — Color, fill/stroke style, stroke width/cap, shader (gradients), blend mode, anti-aliasing.
5. 🟡 **Why wrap transforms in `save` / `restore`?** — To isolate transform/clip state so it doesn't leak into subsequent draws.
6. 🔴 **How do you efficiently repaint an animated painter?** — Pass an `AnimationController` (a `Listenable`) as `CustomPainter(repaint: ...)` so it repaints on ticks without rebuilding the widget, and isolate with `RepaintBoundary`.
7. 🔴 **What's the top custom-paint perf bug?** — Missing/incorrect `shouldRepaint` (repainting every frame) — plus per-paint allocations.

Senior Engineer Tips

- Always define `shouldRepaint` by comparing the exact inputs; a wrong one silently repaints 60×/s.
- For animations, use `CustomPainter(repaint: controller)` + `RepaintBoundary` rather than `setState`-rebuilding the widget.
- Cache `Paint` / `TextPainter`/computed paths; recompute only when inputs change.

Architect Perspective

`CustomPainter` is the sanctioned drop-to-canvas for bespoke visuals, sitting on the paint phase. Wrapping custom visuals in reusable, correctly-`shouldRepaint`ing, isolated painter widgets gives data-viz/drawing features that are both flexible and performant — the base for charts, gauges, and drawing tools across an app (09 · paint_phase, Module 22).

Summary

- `CustomPaint` + `CustomPainter.paint(canvas, size)` draw with `Paint`; text via `TextPainter`; transforms via `save` / `restore`.
- Implement `shouldRepaint` to repaint only on change; drive animated repaints via `repaint:`; isolate + reuse objects.
- The entry point for charts/gauges/drawings/backgrounds — powerful but requires perf discipline.

Revision Notes

- `CustomPaint(size, painter, foregroundPainter, child)` → `paint(canvas, size)`; `Paint` (color/style/stroke/shader/blend).
- Text = `TextPainter.layout()` + `paint`; transforms in `save` / `restore`; geometry from `size`.
- `shouldRepaint(old)` = compare inputs (repaint only on change); animate via `repaint:` Listenable.
- Reuse Paint/TextPainter; isolate with `RepaintBoundary`.

Practice Questions

1. Why is a missing/incorrect `shouldRepaint` a perf bug?
2. How do you draw text on a `Canvas`?
3. How do you efficiently repaint an animated painter?

Coding Questions

1. Draw a progress ring (arc + track + % text) with correct `shouldRepaint`.
2. Draw a bar chart from a list of values scaled to `size`.
3. Animate a painter via `CustomPainter(repaint: controller)` inside a `RepaintBoundary`.

Mini Project

Progress ring widget (Flutter): Build a reusable `ProgressRing` (`CustomPainter`: track arc + progress arc + centered % text), driven by an `AnimationController` via `repaint:`, with correct `shouldRepaint`, reused `Paint`, and `RepaintBoundary`. Acceptance: resize-safe geometry; repaints only on value change; text via `TextPainter`; smooth animation; runs.

Paths, Shapes, Gradients & Clipping

> Beyond primitives, `Path` composes arbitrary vector shapes (lines/curves/arcs), `Gradient`s fill them via a `Paint.shader`, and clipping constrains drawing to a region — the toolkit for custom charts, curved containers, and complex visuals.

Introduction

This file covers building shapes with `Path` (`moveTo` / `lineTo` / `cubicTo` / `quadraticBezierTo` / `arcTo` / `close`), filling with **gradients** (linear/radial/sweep via a shader), and **clipping** (`clipPath` / `clipRRect` / `ClipPath` widget + `CustomClipper`). It's the expressive core of custom painting.

Why this concept exists

Rectangles and circles don't make a wave, a curved app-bar, a pie/line chart, or a speech bubble. `Path` provides arbitrary geometry; gradients provide rich fills; clipping shapes content to non-rectangular regions — enabling any 2D visual.

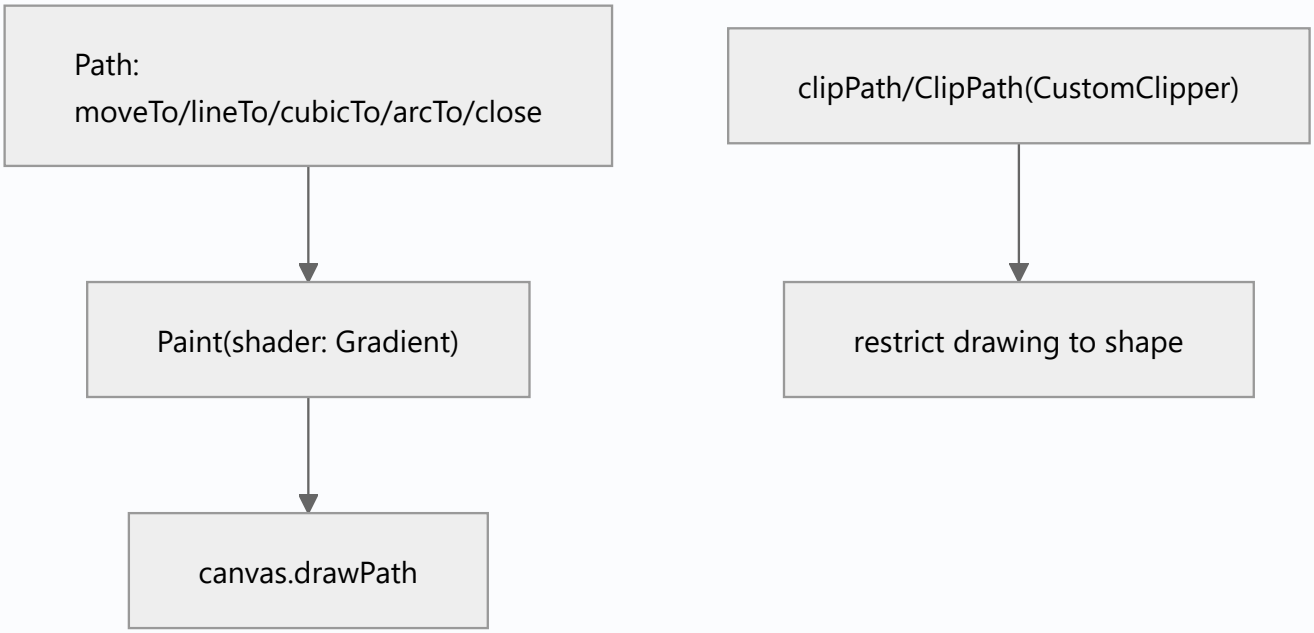
Real-world analogy

`Path` is **drawing an outline with a pen** (straight and curved strokes) then filling it; a gradient is **airbrushing a color blend** inside; clipping is a **stencil** that only lets paint through a chosen shape.

Problem Statement

Draw a gradient-filled wave background, a pie-chart slice, and clip an image into a curved shape. You'll build `Path`s (lines/curves/arcs), fill with a gradient shader, and clip.

Internal Working



- `Path` : build with `moveTo(x,y)` , `lineTo` , `quadraticBezierTo` / `cubicTo` (Bézier curves), `arcTo` / `arcToPoint` , `addRect/addOval/addRRect` , `close()` ; combine paths with `Path.combine` (union/difference/intersect). Draw via `canvas.drawPath(path, paint)` .
- **Gradients**: `LinearGradient` / `RadialGradient` / `SweepGradient` → `.createShader(rect)` assigned to `paint.shader` ; `SweepGradient` is great for pie/gauge arcs. (Widget-level: `BoxDecoration(gradient:)` .)
- **Clipping (canvas)**: `canvas.clipPath/clipRRect/clipRect` (within `save` / `restore`) restricts subsequent draws.
- **Clipping (widgets)**: `ClipRRect` / `ClipOval` / `ClipPath(clipper: CustomClipper<Path>)` clip a child to a shape — `CustomClipper.getClip(size)` returns the path.
- **Curves**: quadratic (1 control point) and cubic (2 control points) Béziers produce smooth curves (waves, bubbles); arcs for circular segments.

Memory Representation

Paths/shaders are objects; build once and reuse when inputs don't change (avoid rebuilding per paint) (05_custom_painting_performance.md). Clipping may allocate offscreen buffers (`saveLayer` -like cost — 21 · jank_and_raster).

Compiler Behavior

Not applicable.

Runtime Behavior

`drawPath` rasterizes the filled/stroked path; complex paths and anti-aliased clips cost more raster time. Clips constrain following draws until `restore`.

Flutter Engine Behavior

Paths/gradients/clips become GPU operations (Skia/Impeller); heavy/soft clips and large gradients add raster cost (09 · rasterization).

Dart VM Behavior

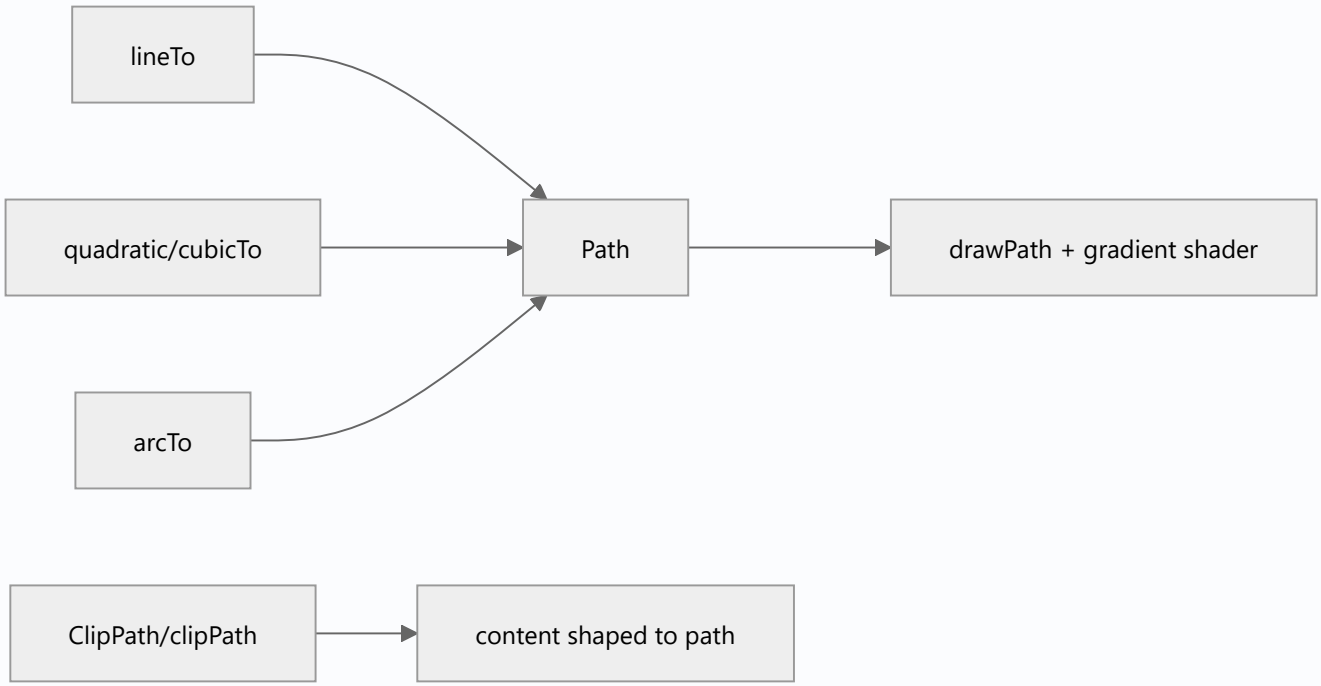
Not applicable.

Examples

```
import 'dart:math';
import 'package:flutter/material.dart';

// Gradient-filled wave background
class WavePainter extends CustomPainter {
  @override
  void paint(Canvas canvas, Size size) {
    final path = Path()
      ..moveTo(0, size.height * 0.7)
      ..quadraticBezierTo(                                  // smooth curve
        size.width * 0.25, size.height * 0.6,
        size.width * 0.5, size.height * 0.7)
      ..quadraticBezierTo(
        size.width * 0.75, size.height * 0.8,
        size.width, size.height * 0.7)
      ..lineTo(size.width, size.height)
      ..lineTo(0, size.height)
      ..close();

    final paint = Paint()
      ..shader = const LinearGradient(                      // gradient fill
        colors: [Colors.teal, Colors.indigo],
        begin: Alignment.topLeft, end: Alignment.bottomRight,
      ).createShader(Offset.zero & size);
```

```
    canvas.drawPath(path, paint);
  }
  @override bool shouldRepaint(WavePainter old) => false;    // static
}

// Pie-chart slice (arc as a wedge path with sweep gradient)
class SlicePainter extends CustomPainter {
  final double startAngle, sweep;
  SlicePainter(this.startAngle, this.sweep);
  @override
  void paint(Canvas canvas, Size size) {
    final center = size.center(Offset.zero);
    final r = min(size.width, size.height) / 2;
    final path = Path()
      ..moveTo(center.dx, center.dy)
      ..arcTo(Rect.fromCircle(center: center, radius: r), startAngle, sweep, false)
      ..close();
    final paint = Paint()..shader =
        SweepGradient(colors: const [Colors.orange, Colors.red]).createShader(Offset.zero & size);
    canvas.drawPath(path, paint);
  }
  @override bool shouldRepaint(SlicePainter o) => o.startAngle != startAngle || o.sweep != sweep;
}

// Clip a child into a custom (rounded-top) shape
class RoundedTopClipper extends CustomClipper<Path> {
  @override
  Path getClip(Size size) => Path()
    ..moveTo(0, 40)
    ..quadraticBezierTo(size.width / 2, 0, size.width, 40)
    ..lineTo(size.width, size.height)
    ..lineTo(0, size.height)
    ..close();
  @override bool shouldReclip(RoundedTopClipper old) => false;
}
// Usage: ClipPath(clipper: RoundedTopClipper(), child: Image.network(...))
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Rebuilding paths/shaders every paint	GC churn / cost	Cache; rebuild only on input change
Forgetting `close()` on a fill	Open path fills oddly	`close()` filled shapes
Heavy anti-aliased clips per frame	Raster cost (offscreen)	`Clip.hardEdge` when acceptable; limit/cache
Not scaling geometry to `size`	Breaks on resize	Compute from `size`
Wrong angle units/direction (arcs)	Wrong shape	Radians, clockwise from +x; offset by `-pi/2` for top-start
Overusing `ClipPath` for simple rounding	Cost	Use `ClipRRect` / `BorderRadius`

Best Practices

- Build `Path`s from `size` (resize-safe); **cache** paths/shaders and rebuild only when inputs change.
- Use the right **gradient** (linear/radial/**sweep** for arcs) via `paint.shader`.
- Prefer `ClipRRect` / `ClipOval` for simple rounding; reserve `ClipPath` + `CustomClipper` for genuine custom shapes; implement `shouldReclip`.

- Watch **clip cost** (anti-aliased/soft clips use offscreen buffers) — use `Clip.hardEdge` when quality allows (21 · jank_and_raster).
- `close()` filled shapes; use radians and account for start-angle offset for arcs.

Performance

`drawPath` and clipping cost raster time; heavy/soft clips are `saveLayer`-like. Cache paths/gradients; simplify shapes; use hard-edge clips where possible; isolate animated paths with `RepaintBoundary` (05_custom_painting_performance.md).

Advantages / Disadvantages

- + Arbitrary vector shapes, rich gradient fills, custom-shaped content — full 2D expressiveness.
- − Imperative geometry/math, raster cost for complex paths/soft clips, easy to over-allocate/over-clip.

Interview Questions

1. 🟢 **How do you draw an arbitrary shape?** — Build a `Path` (`moveTo` / `lineTo` / `cubicTo` / `arcTo` / `close`) and `canvas.drawPath(path, paint)`.
2. 🟢 **How do you fill a shape with a gradient?** — Set `paint.shader` from a `LinearGradient` / `RadialGradient` / `SweepGradient` via `.createShader(rect)`.
3. 🟡 **When use `ClipRRect` vs `ClipPath`?** — `ClipRRect` / `ClipOval` for simple rounded/oval clips; `ClipPath` + `CustomClipper` for arbitrary custom shapes.
4. 🟡 **Quadratic vs cubic Bézier?** — Quadratic has one control point; cubic has two (more expressive curves) — used for waves/bubbles/smooth outlines.
5. 🟡 **Why is clipping potentially expensive?** — Anti-aliased/soft clips allocate offscreen buffers on the raster thread (like `saveLayer`).
6. 🔴 **How do you keep path drawing efficient?** — Cache paths/shaders (rebuild only on input change), simplify geometry, prefer hard-edge clips, and isolate animated paths with `RepaintBoundary`.
7. 🔴 **Which gradient suits a pie/gauge arc?** — `SweepGradient` (angular color sweep around a center).

Senior Engineer Tips

- Precompute paths/gradients in the painter's fields (or memoize by inputs); building them inside `paint` every frame is a common cost.
- For charts, map data→geometry from `size` so it's responsive; use `Path.combine` for composite shapes.

- Prefer widget clippers (`ClipRRect`) for simple cases; drop to `ClipPath` /canvas clips only when needed, and mind `shouldReclip` .

Architect Perspective

Paths/gradients/clipping are the expressive vocabulary of custom visuals (charts, branded shapes, drawing tools). Encapsulated in reusable painters/clippers with caching and cost-awareness, they enable rich data-viz and unique UI while respecting the raster budget (09 · rasterization, 21 · jank_and_raster).

Summary

- `Path` builds arbitrary shapes (lines/curves/arcs); gradients fill via `paint.shader` ; clipping shapes content to a region.
- Cache paths/shaders, compute from `size` , prefer simple widget clippers, mind clip/raster cost.
- The expressive core for charts, curved UI, and drawing — with performance discipline.

Revision Notes

- `Path` : `moveTo/lineTo/quadraticBezierTo/cubicTo/arcTo/close` , `Path.combine` ; `canvas.drawPath` .
- Gradients: Linear/Radial/Sweep → `paint.shader = g.createShader(rect)` ; Sweep for arcs/pies.
- Clip: `ClipRRect/ClipOval` (simple), `ClipPath(CustomClipper)` (custom, `shouldReclip`); soft clips = offscreen cost.
- Cache paths/shaders; geometry from `size` ; radians for arcs; isolate animated paths.

Practice Questions

1. How do you draw a wave (curve) background?
2. Which gradient for a pie slice, and why?
3. Why is `ClipPath` costlier than `ClipRRect` ?

Coding Questions

1. Draw a gradient-filled wave via quadratic Béziers.
2. Draw a pie chart from data (arc wedges + sweep gradient).
3. Clip an image into a rounded-top shape with a `CustomClipper` .

Mini Project

Mini pie chart (Flutter): Build a `CustomPainter` pie chart from a list of values — arc wedges via `Path` + `arcTo`, sweep/solid fills, computed from `size`, with cached paths and correct `shouldRepaint`; add a `ClipRRect` rounded container. Acceptance: correct proportional slices; responsive to size; cached geometry; repaint only on data change; runs.

Custom `RenderObject`s (Bespoke Layout + Paint)

> When you need custom **layout** (sizing/positioning children) *and* painting that `CustomPainter` (paint-only) or existing widgets can't express, drop to a custom `RenderObject` — implementing `performLayout` and `paint` directly on the render tree.

Introduction

`CustomPainter` draws but doesn't do layout or manage children. For bespoke layout algorithms (custom flex, radial menus, flow layouts) plus painting, you write a `RenderObject` (usually via `RenderBox`) and expose it with a `RenderObjectWidget`. This file covers when and how — the rarest, most advanced custom-visual tier.

Why this concept exists

The framework's layout widgets and `CustomPainter` cover most needs, but some UIs need a genuinely custom **layout protocol** (how children are measured/placed) combined with painting — e.g., a chart that lays out labels around data, a masonry/flow grid, a custom slider track+thumb. Only a `RenderObject` gives full control of layout *and* paint (09 · layout_phase).

Why this concept exists (when NOT to)

Most of the time you **don't** need this — compose widgets, use `CustomPainter`, `Flow`, `CustomMultiChildLayout`, or `LayoutBuilder` first. Custom render objects are low-level and easy to get wrong; reserve them for real layout+paint needs.

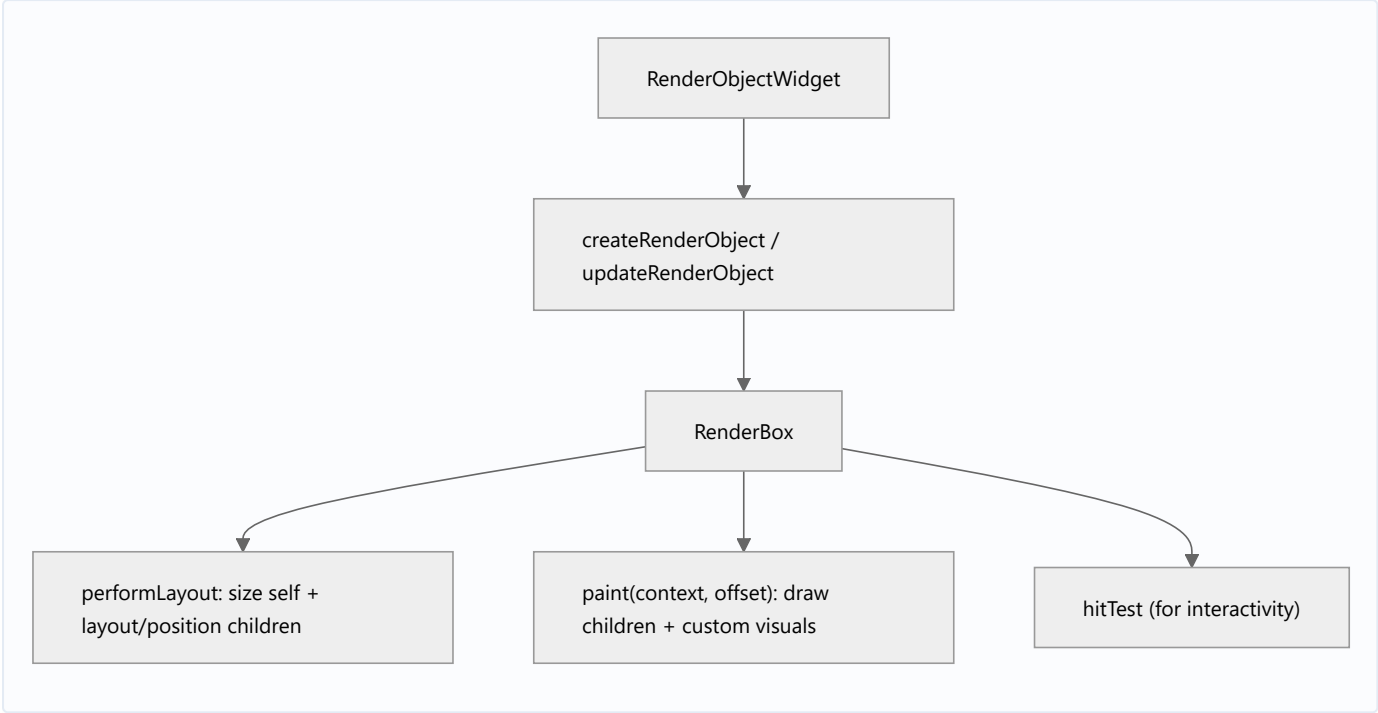
Real-world analogy

Widgets are **prefab furniture**; `CustomPainter` is **painting a wall**; a custom `RenderObject` is **being the architect and builder** — deciding room sizes/positions (layout) *and* the finish (paint). Powerful, but you own the structural engineering (the layout contract).

Problem Statement

Build a widget that lays out children in a circle (radial layout) and paints connecting lines — layout the framework doesn't provide. You'll implement a custom `RenderBox` with `performLayout` + `paint`.

Internal Working



- **Two options**:
  - `SingleChildRenderObjectWidget` / `MultiChildRenderObjectWidget` + a custom `RenderBox` — full control of layout + paint.
  - `LeafRenderObjectWidget` for no children (pure custom-drawn+laid-out leaf).
- `RenderBox.performLayout()` : read `constraints` , call `child.layout(childConstraints, parentUsesSize:)` , set `size` , and set each child's offset (`parentData`) — the layout contract (09 · layout_phase).
- `paint(PaintingContext context, Offset offset)` : draw via `context.canvas` ; paint children with `context.paintChild(child, offset + childOffset)` .
- `RenderObjectWidget` : `createRenderObject(context)` builds it; `updateRenderObject` pushes new config (mark `markNeedsLayout` / `markNeedsPaint` on changes).
- **Parent data**: use `ContainerRenderObjectMixin` / `RenderBoxContainerDefaultsMixin` + a `ParentData` subclass for multi-child positioning.
- **Interactivity**: implement `hitTestChildren` / `hitTestSelf` for gestures.
- **Simpler alternatives** to consider first: `CustomMultiChildLayout` + `MultiChildLayoutDelegate` , `Flow` + `FlowDelegate` , or `CustomPainter` (paint-only).

Memory Representation

A persistent node in the render tree holding size/offset/parentData; children are laid out/positioned relative to it (09 · layout_phase).

Compiler Behavior

Not applicable.

Runtime Behavior

`performLayout` runs in the layout phase (respect the constraints protocol); `paint` in the paint phase. Config changes must call `markNeedsLayout` / `markNeedsPaint` to schedule updates.

Flutter Engine Behavior

Produces layers/draw ops rasterized by the engine; incorrect layout (not calling `child.layout`, wrong `size`) breaks rendering (09 · pipeline_overview).

Dart VM Behavior

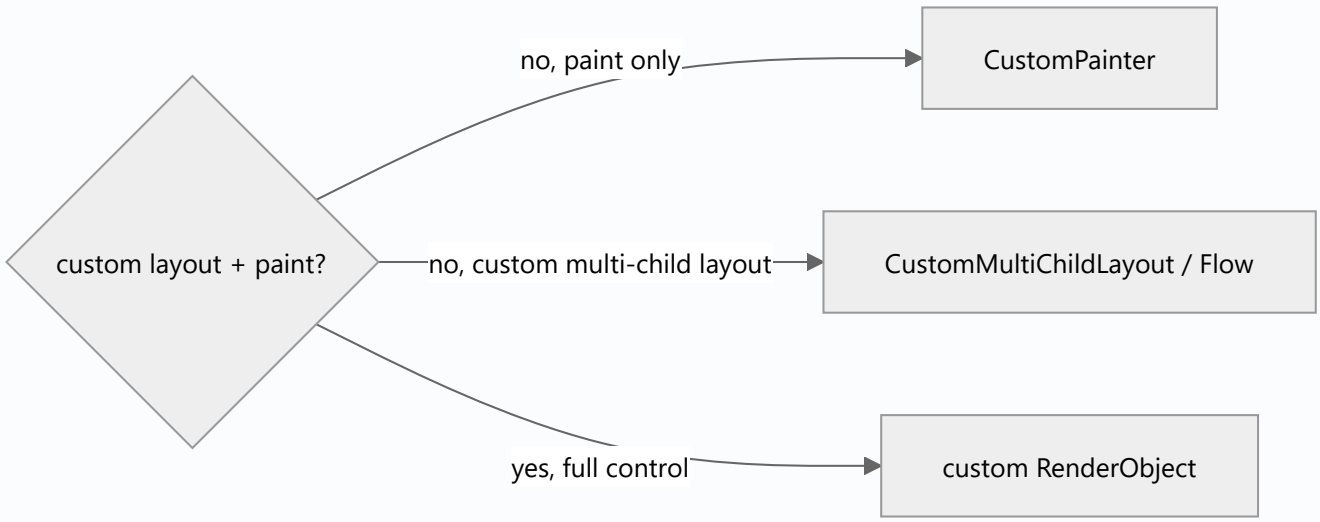
Not applicable.

Examples

```
import 'dart:math';
import 'package:flutter/material.dart';
import 'package:flutter/rendering.dart';

// Radial layout: place children evenly on a circle, paint spokes.
class RadialLayout extends MultiChildRenderObjectWidget {
  const RadialLayout({super.key, required super.children});
  @override
  RenderObject createRenderObject(BuildContext context) => _RenderRadial();
}

class _RadialParentData extends ContainerBoxParentData<RenderBox> {}

class _RenderRadial extends RenderBox
    with ContainerRenderObjectMixin<RenderBox, _RadialParentData>,
         RenderBoxContainerDefaultsMixin<RenderBox, _RadialParentData> {
  @override
  void setupParentData(RenderBox child) {
    if (child.parentData is! _RadialParentData) child.parentData = _RadialParentData();
  }
```

```
  @override
  void performLayout() {
    size = constraints.biggest;                       // fill available
    final center = size.center(Offset.zero);
    final radius = min(size.width, size.height) / 2 - 40;
    final n = childCount;
    var child = firstChild;
    var i = 0;
    while (child != null) {
      child.layout(const BoxConstraints.tightFor(width: 60, height: 60), parentUsesSize: true);
      final angle = 2 * pi * i / n - pi / 2;          // place on circle
      final pd = child.parentData! as _RadialParentData;
      pd.offset = center + Offset(cos(angle), sin(angle)) * radius - const Offset(30, 30);
      child = pd.nextSibling; i++;
    }
  }

  @override
  void paint(PaintingContext context, Offset offset) {
    final center = offset + size.center(Offset.zero);
    final paint = Paint()..color = Colors.grey ..strokeWidth = 1;
    var child = firstChild;
    while (child != null) {
      final pd = child.parentData! as _RadialParentData;
      context.canvas.drawLine(center, offset + pd.offset + const Offset(30, 30), paint); // spoke
      context.paintChild(child, offset + pd.offset);   // paint the child
      child = pd.nextSibling;
    }
  }
}
// Usage: RadialLayout(children: [ ...avatars... ])
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Writing a RenderObject when a delegate/painter suffices	Needless complexity/bugs	Use `CustomPainter` / `CustomMultiChildLayout` / `Flow` first
Not calling `child.layout` / not setting `size`	Broken layout	Follow the layout contract exactly
Forgetting `markNeedsLayout` / `markNeedsPaint` on config change	Stale rendering	Mark dirty in `updateRenderObject` /setters
Wrong `parentData` setup	Positioning breaks	Use the container mixins + a `ParentData` subclass
Ignoring constraints	Overflow/errors	Honor incoming `constraints`
No `hitTest` for interactive widgets	Gestures don't work	Implement `hitTestChildren` / `hitTestSelf`

Best Practices

- **Exhaust simpler options first**: composition, `CustomPainter` (paint-only), `CustomMultiChildLayout` / `Flow` (custom multi-child layout without a full RenderObject), `LayoutBuilder`.
- If you must: follow the **layout contract** (`constraints` → `child.layout` →set `size` →position children); mark dirty (`markNeedsLayout` / `markNeedsPaint`) on changes.
- Use **container mixins** + a `ParentData` subclass for multi-child positioning; implement `hitTest` for interactivity.

- Keep `performLayout` / `paint` efficient (relayout/repaint boundaries — 09 · layout_phase).

Performance

Custom render objects run in layout/paint each relevant frame — respect relayout/repaint boundaries and keep the algorithms O(children). Correctness (constraints/contract) matters as much as speed (09 · layout_phase, 05_custom_painting_performance.md).

Advantages / Disadvantages

- **+** Total control of layout *and* paint (impossible-otherwise layouts), integrates natively with the render tree.
- **−** Low-level, verbose, error-prone (layout contract/parentData/hitTest), rarely necessary — high maintenance cost.

Interview Questions

1. 🟢 **When do you need a custom `RenderObject` vs `CustomPainter`?** — When you need custom **layout** (sizing/positioning children), not just painting; `CustomPainter` is paint-only.
2. 🟢 **What must `performLayout` do?** — Read `constraints`, lay out children (`child.layout`), set the render object's `size`, and position children (offsets) — honoring the layout contract.
3. 🟡 **What simpler options should you try first?** — Composition, `CustomPainter`, `CustomMultiChildLayout` / `MultiChildLayoutDelegate`, and `Flow` / `FlowDelegate`.
4. 🟡 **How does a `RenderObjectWidget` connect to the render object?** — `createRenderObject` builds it; `updateRenderObject` pushes new config (marking layout/paint dirty as needed).
5. 🟡 **How do you position multiple children?** — Use `ContainerRenderObjectMixin` + a `ParentData` subclass storing each child's offset.
6. 🔴 **What triggers re-layout/re-paint after a config change?** — Calling `markNeedsLayout()` / `markNeedsPaint()` (typically in setters/`updateRenderObject`).
7. 🔴 **How do you make a custom render object interactive?** — Implement `hitTestSelf` / `hitTestChildren` so it participates in the gesture hit-test.

Senior Engineer Tips

- The right answer is usually **"don't"** — reach for `Flow` / `CustomMultiChildLayout` / `CustomPainter` before a full `RenderObject`; they cover most "custom layout/paint" needs with far less risk.
- If you do write one, mirror an existing framework `RenderBox` and respect the constraints protocol exactly; subtle contract violations cause baffling bugs.
- Mark dirty correctly on every config change; forgetting it yields stale, confusing renders.

Architect Perspective

Custom `RenderObject`s are the escape hatch for genuinely novel layout+paint (advanced charts, editors, custom layout engines). They're powerful but high-cost/maintenance; architecturally, prefer composition/delegates/painters and reserve render objects for a small, well-encapsulated set of truly-custom widgets (09 · layout_phase).

Summary

- Custom `RenderObject` = full control of layout + paint (via `RenderBox`: `performLayout` + `paint`), exposed by a `RenderObjectWidget`.
- Honor the constraints/layout contract, use container mixins + parentData for children, mark dirty on changes, implement `hitTest` for interactivity.
- Rare/advanced — exhaust composition/ `CustomPainter` / `Flow` / `CustomMultiChildLayout` first.

Revision Notes

- Need custom layout+paint → `RenderBox` (`performLayout`: constraints→child.layout→size→offsets; `paint`: context.paintChild + custom draws).
- `RenderObjectWidget`: `createRenderObject` / `updateRenderObject`; mark `markNeedsLayout` / `markNeedsPaint`.
- Multi-child: `ContainerRenderObjectMixin` + `ParentData`; interactivity: `hitTest*`.
- Prefer `CustomPainter` / `Flow` / `CustomMultiChildLayout` first.

Practice Questions

1. When is a custom `RenderObject` justified over `CustomPainter`?
2. What are the steps of `performLayout`?
3. What must you call after a config change, and why?

Coding Questions

1. Implement a leaf `RenderBox` that sizes to a fixed aspect ratio and paints a shape.
2. Build a radial multi-child layout (children on a circle) + spokes.
3. Rewrite a custom layout using `CustomMultiChildLayout` instead and compare complexity.

Mini Project

Radial menu (Flutter): Implement a `RadialLayout` custom `MultiChildRenderObjectWidget` placing children evenly on a circle with painted spokes (correct `performLayout` / `paint` /parentData, `markNeedsLayout` on change, basic `hitTest`). Then reimplement with `CustomMultiChildLayout` and note

the tradeoff. Acceptance: correct layout contract; children positioned + spokes drawn; interactive; documented "prefer delegate" comparison; runs.

Shaders & Effects (Fragment Shaders, `ImageFilter`, Blend Modes)

> For effects beyond shapes/gradients — glows, distortions, procedural patterns, blurs, color blends — use GPU **fragment shaders** (GLSL via `FragmentProgram`), `ImageFilter` (blur/matrix), `ShaderMask`, and `BlendMode` — powerful visuals that run on the GPU but carry real raster cost.

Introduction

This file covers advanced visual effects: custom **fragment shaders** (`.frag` GLSL compiled to a `FragmentProgram`, applied as a `Paint.shader`), built-in **`ImageFilter`s** (blur, matrix) via `BackdropFilter` / `ImageFiltered`, **`ShaderMask`** (mask a child with a shader/gradient), and **`BlendMode`** (how colors combine). It's the top tier of custom visuals.

Why this concept exists

Some looks — animated gradients, noise, ripples, glassmorphism blur, gradient-text, glow/duotone — can't be drawn with paths/solid fills alone. Shaders compute per-pixel color on the GPU; filters/blends compose effects. They enable rich, modern visuals Flutter's engine can render efficiently (with care).

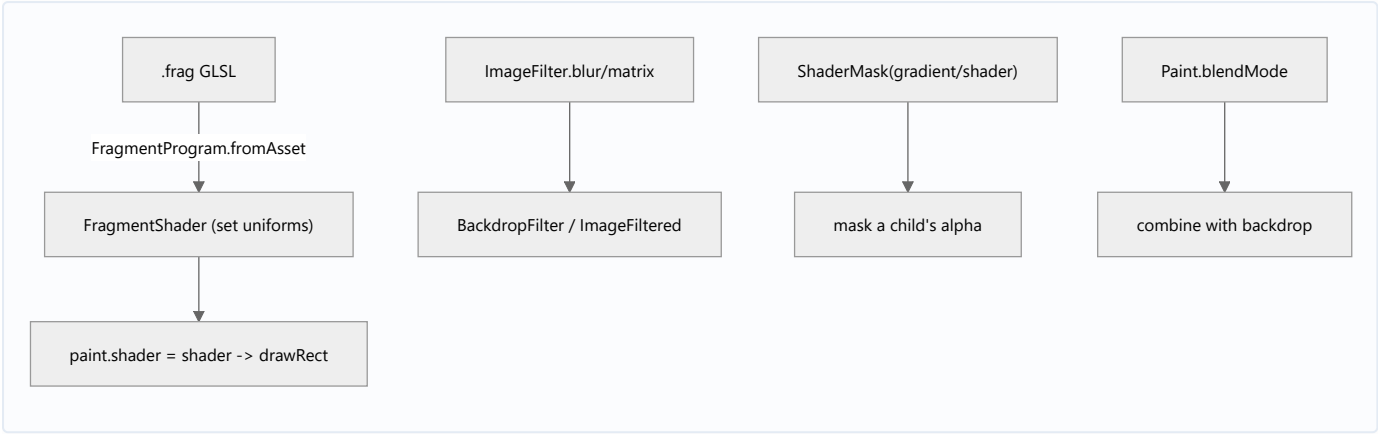
Real-world analogy

Fragment shaders are **a tiny program run for every pixel** (like a per-pixel recipe) executed on the GPU's many cores; `ImageFilter`/blend modes are **post-processing filters** (blur, tint, overlay) applied to already-drawn content — the difference between painting each dot vs applying an Instagram filter.

Problem Statement

Add a gradient-masked headline (`ShaderMask`), a frosted-glass panel (`BackdropFilter` blur), and an animated custom fragment-shader background. You'll use `ShaderMask`, `ImageFilter.blur`, and a `FragmentProgram`.

Internal Working



- **Fragment shaders** (Flutter): author a GLSL `.frag`, declare it under `flutter: shaders:` in `pubspec.yaml`, load with `FragmentProgram.fromAsset(...)`, create a `FragmentShader`, set **uniforms** (`setFloat` / `setImageSampler`) — including time/resolution for animation — and assign to `paint.shader` for `drawRect` / `drawPaint`. Runs per-pixel on the GPU.
- `ImageFilter`: `ImageFilter.blur(sigmaX, sigmaY)` (and matrix/compose) used by `BackdropFilter` (blur what's *behind* — glassmorphism) or `ImageFiltered` (filter the child).
- `ShaderMask`: applies a shader/gradient as a **mask** over its child (e.g., gradient text, fade edges) via a blend mode.
- `BlendMode`: how source pixels combine with destination (`srcOver`, `multiply`, `screen`, `overlay`, etc.) — set on `Paint.blendMode` or in `ShaderMask`.
- **Cost**: shaders/filters run on the raster thread; blurs/backdrop are expensive (offscreen buffers); **shader compilation** can cause first-run jank (Impeller precompiles) (21 · jank_and_raster).

Memory Representation

Filters/backdrops allocate offscreen GPU buffers; shaders use GPU program + uniform state. Large/animated blurs are GPU-memory and time heavy (09 · compositing).

Compiler Behavior

`.frag` shaders are compiled/bundled (declared in `pubspec`); Impeller precompiles shaders (no first-run jank); Skia compiles lazily.

Runtime Behavior

Shaders execute per-pixel per frame; animating uniforms (time) repaints each frame. Filters/backdrops recompute each frame they're active — costly if large/continuous.

Flutter Engine Behavior

All run on the GPU via the engine's rasterizer; Impeller vs Skia affects shader-compilation behavior and predictability (09 · rasterization).

Dart VM Behavior

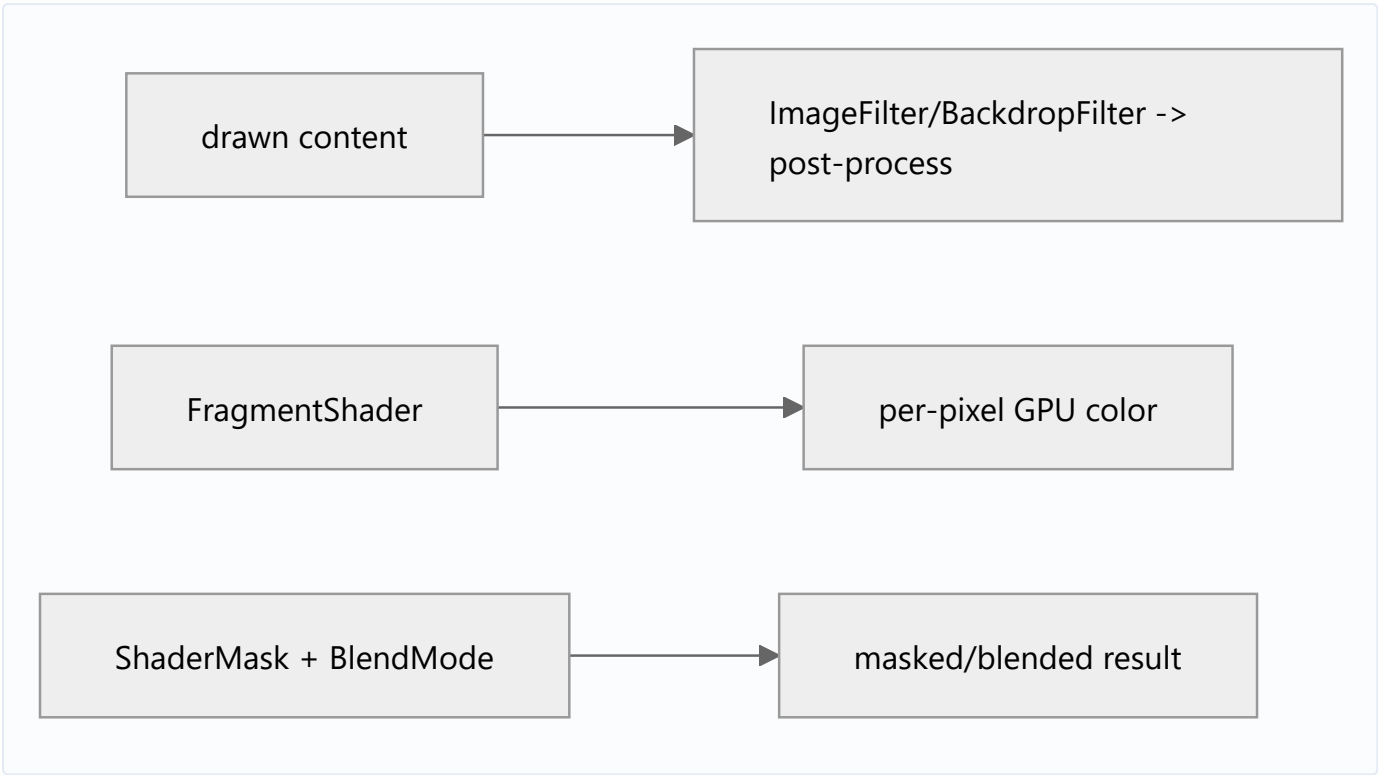
Not applicable (GPU-side).

Examples

```
import 'dart:ui' as ui;
import 'package:flutter/material.dart';

// ShaderMask: gradient-filled text (mask child alpha with a gradient)
Widget gradientText(String text) => ShaderMask(
      shaderCallback: (rect) => const LinearGradient(
        colors: [Colors.purple, Colors.orange],
      ).createShader(rect),
      blendMode: BlendMode.srcIn,           // paint gradient where text is opaque
      child: Text(text, style: const TextStyle(fontSize: 40, color: Colors.white)),
    );

// BackdropFilter: frosted-glass panel (blur what's behind)
Widget frostedPanel(Widget child) => ClipRRect(
      borderRadius: BorderRadius.circular(16),
      child: BackdropFilter(
        filter: ui.ImageFilter.blur(sigmaX: 12, sigmaY: 12), // blur backdrop
        child: Container(color: Colors.white.withOpacity(0.15), child: child),
      ),
    );

// Custom fragment shader (animated) — pubspec: flutter: shaders: - shaders/wave.frag
class ShaderBackground extends StatefulWidget {
  const ShaderBackground({super.key});
  @override State<ShaderBackground> createState() => _ShaderBackgroundState();
}
class _ShaderBackgroundState extends State<ShaderBackground> with SingleTickerProviderStateMixin {
  ui.FragmentShader? _shader;
  late final _c = AnimationController(vsync: this, duration: const Duration(seconds: 4))..repeat();
```

```
  @override
  void initState() {
    super.initState();
    ui.FragmentProgram.fromAsset('shaders/wave.frag').then((p) => setState(() => _shader =
p.fragmentShader()));
  }
  @override void dispose() { _c.dispose(); super.dispose(); }

  @override
  Widget build(BuildContext context) {
    if (_shader == null) return const SizedBox.expand();
    return AnimatedBuilder(
      animation: _c,
      builder: (_, __) => CustomPaint(size: Size.infinite, painter: _ShaderPainter(_shader!, _c.value)),
    );
  }
}
class _ShaderPainter extends CustomPainter {
  final ui.FragmentShader shader; final double t;
  _ShaderPainter(this.shader, this.t);
  @override
  void paint(Canvas canvas, Size size) {
    shader
      ..setFloat(0, size.width) ..setFloat(1, size.height) ..setFloat(2, t); // uniforms: resolution + time
    canvas.drawRect(Offset.zero & size, Paint()..shader = shader);
  }
  @override bool shouldRepaint(_ShaderPainter o) => o.t != t;
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Large/continuous `BackdropFilter` blurs	Very expensive (offscreen, per frame)	Limit area/sigma; static/cached where possible
Shader not declared in `pubspec`	Fails to load	Add under `flutter: shaders:`
Ignoring shader-compilation jank	First-run stutter	Impeller / warm-up (21 · jank_and_raster)
Wrong `BlendMode` in `ShaderMask`	Wrong masking (e.g., not `srcIn`)	Pick the mode for the effect
Animating heavy effects unbounded	Raster jank	Bound size/area; isolate; profile
Not disposing FragmentShader/controller	Leak	Dispose

Best Practices

- Use built-ins first: `ShaderMask` (gradient/mask), `ImageFilter` / `BackdropFilter` (blur), `BlendMode` — reserve **custom fragment shaders** for genuinely procedural/animated effects.
- **Bound cost**: limit blur `sigma`/area; avoid large continuous backdrops; cache static effects; isolate with `RepaintBoundary`.
- Address **shader-compilation jank** via **Impeller** (or SkSL warm-up on Skia).
- Declare `.frag` in `pubspec`; set uniforms correctly; dispose shaders/controllers.

- **Profile** effects in release (raster-bound) on a low-end device.

Performance

Shaders/filters/blends are raster-thread work; blurs/backdrops are the most expensive (offscreen buffers per frame). Impeller removes shader-compilation jank; still budget effect cost, cache, and isolate (21 · jank_and_raster).

Advantages / Disadvantages

- **+** Rich, modern, GPU-accelerated visuals (glass/glow/procedural/animated) not possible with shapes alone.
- **−** Expensive (esp. blurs/backdrops), shader-compilation jank (Skia), GLSL complexity, GPU-memory cost — needs profiling/restraint.

Interview Questions

1. 🟢 **What's a fragment shader in Flutter?** — A GLSL program run per-pixel on the GPU, loaded via `FragmentProgram.fromAsset`, applied as a `Paint.shader` (with uniforms).
2. 🟢 **What does `ShaderMask` do?** — Masks a child using a shader/gradient + blend mode (e.g., gradient text, edge fades).
3. 🟡 **`BackdropFilter` vs `ImageFiltered`?** — `BackdropFilter` filters what's *behind* it (glassmorphism blur); `ImageFiltered` filters the child itself.
4. 🟡 **Why are blurs/backdrops expensive?** — They allocate offscreen buffers and recompute per frame on the raster thread.
5. 🟡 **How do you animate a fragment shader?** — Pass a time uniform (updated each frame via an `AnimationController`) and repaint (`shouldRepaint` on time).
6. 🔴 **What causes first-run effect stutter and the fix?** — Shader compilation (Skia); fix with Impeller (precompiled) or SkSL warm-up.
7. 🔴 **How do you keep effects performant?** — Prefer built-ins, bound blur size/area, cache static effects, isolate with `RepaintBoundary`, and profile raster time in release.

Senior Engineer Tips

- Treat blur/backdrop as premium-cost; use small areas, cache when static, and never animate full-screen heavy blurs without profiling.
- Reach for `ShaderMask`/gradients/`BlendMode` before writing GLSL — most "shader-looking" effects are achievable with built-ins.
- If you ship custom shaders, plan for Impeller (or warm-up) and test first-run on a low-end device.

Architect Perspective

Shaders/effects are the highest-cost visual tier — powerful for brand/immersive UI and data-viz, but raster-heavy. Architecturally, prefer built-in effects, budget/cached/isolated usage, and Impeller for predictability; reserve custom GLSL for signature effects, encapsulated and profiled (09 · rasterization, 21 · jank_and_raster).

Summary

- Effects: fragment shaders (per-pixel GPU), `ImageFilter` / `BackdropFilter` (blur), `ShaderMask` (masking), `BlendMode` (compositing).
- Prefer built-ins; bound blur/backdrop cost; cache/isolate; address shader-compilation jank via Impeller.
- Rich modern visuals with real raster cost — profile in release and use restraint.

Revision Notes

- Fragment shader: `.frag` (pubspec) → `FragmentProgram.fromAsset` → `FragmentShader` (set uniforms) → `paint.shader`.
- `ShaderMask` (gradient/mask + blend), `ImageFilter.blur` via `BackdropFilter` (behind)/ `ImageFiltered` (child), `BlendMode`.
- Blurs/backdrops expensive (offscreen/frame); Impeller precompiles shaders (no first-run jank).
- Bound size/area; cache/isolate; dispose; profile raster in release.

Practice Questions

1. `BackdropFilter` vs `ImageFiltered` — difference?
2. Why are blurs expensive and how do you mitigate?
3. How do you animate a fragment shader?

Coding Questions

1. Build gradient text with `ShaderMask` (correct `BlendMode`).
2. Make a frosted-glass panel with `BackdropFilter` (bounded blur).
3. Load and animate a simple fragment shader with a time uniform.

Mini Project — Module capstone

Effects showcase (Flutter): Build a screen with gradient-masked headline (`ShaderMask`), a frosted-glass card (`BackdropFilter` , bounded blur, `ClipRRect`), and an animated fragment-shader background (time uniform, `shouldRepaint` , `RepaintBoundary`). Profile raster time in release; note Impeller for shader jank. Acceptance: effects render correctly; blur cost bounded/isolated; shader animates; disposed; measured raster within budget. (Combines with the gauge/signature capstone in README.)

Custom Painting Performance

> Custom paint can jank easily because it can run every frame: implement `shouldRepaint` correctly, **cache** expensive computations (paths/`Paint`/pictures), **isolate** with `RepaintBoundary`, keep `paint` **cheap**, and budget **raster-heavy** ops (soft clips/blurs/complex paths) — verified in release.

Introduction

This file consolidates the performance rules for custom painting, connecting `CustomPainter`/paths/shaders to the paint/rasterization phases (Module 09) and the performance module (Module 21). It's the discipline that makes charts/gauges/drawings smooth.

Why this concept exists

A painter with a wrong `shouldRepaint` or per-frame allocations repaints 60–120×/s, and complex paths/blurs stress the raster thread. Custom visuals are precisely where naive code janks — targeted rules keep them fast.

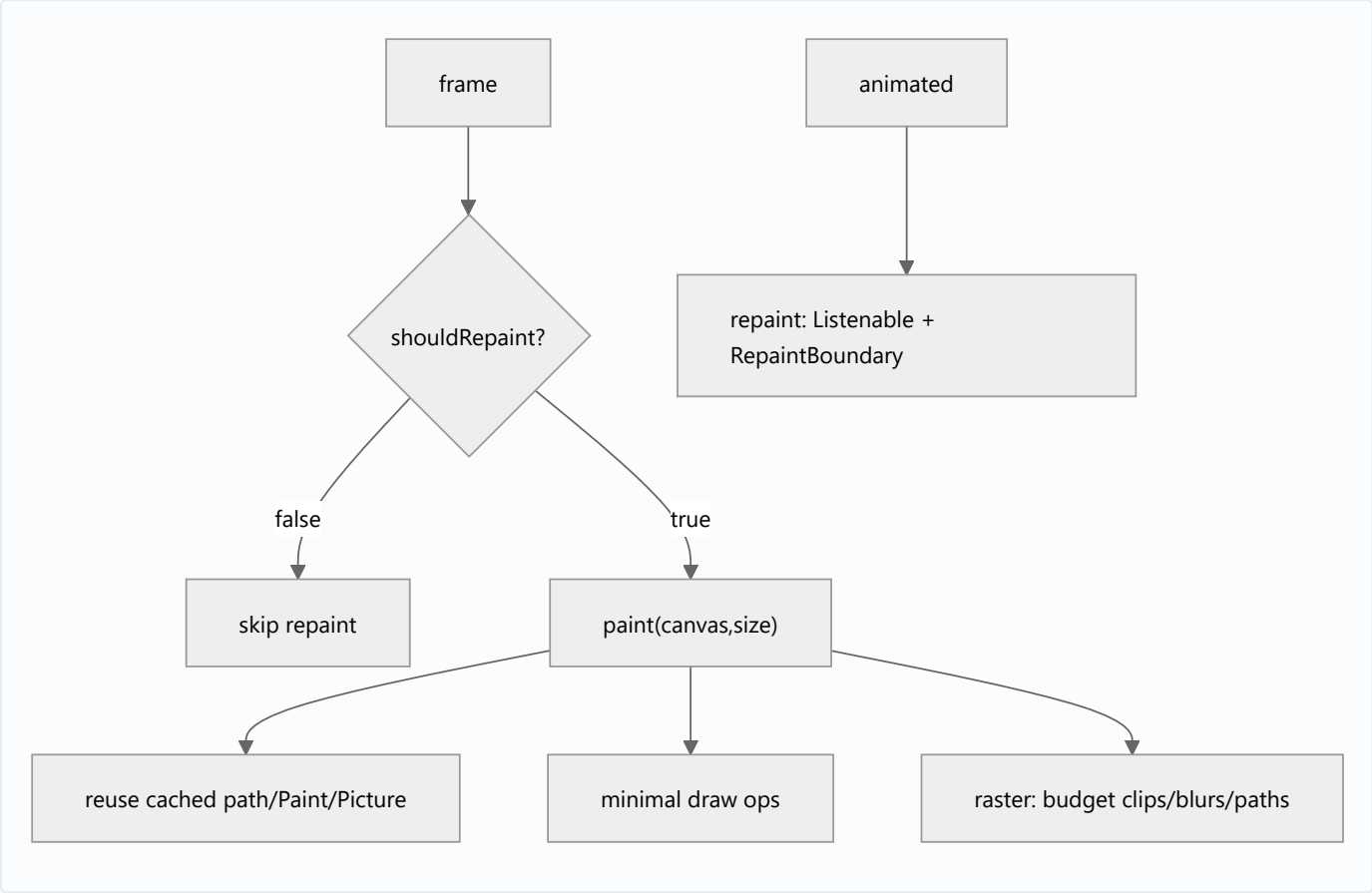
Real-world analogy

An artist who **redraws the whole mural every second** (bad `shouldRepaint`), **mixes fresh paint each stroke** (per-frame allocation), and **uses slow airbrush everywhere** (blurs) will never keep up. The fast artist redraws only what changed, reuses mixed paint, and reserves the airbrush for small areas.

Problem Statement

An animated chart janks: it repaints every frame, rebuilds its path/`Paint` each paint, isn't isolated, and uses a soft clip. You'll fix each and confirm 60fps in release.

Internal Working



- `shouldRepaint`: return **false** unless inputs changed (compare fields) — the single biggest win; default/true repaints every frame (01_custompainter_and_canvas.md).
- **Cache expensive work**: build `Path`s, `Paint`s, gradients, and `TextPainter`s **once** (fields/memoized by inputs), not inside every `paint`. For static complex drawings, record a `ui.Picture` (`PictureRecorder`) once and `drawPicture` it.
- **Isolate**: wrap the painter (and cache expensive static neighbors) in `RepaintBoundary` so its repaints don't affect others — essential for animated painters (09 · compositing).
- **Animate efficiently**: drive repaints via `CustomPainter(repaint: controller)` (a `Listenable`) — repaints without rebuilding the widget — and keep the animated part small.
- **Budget raster ops**: complex paths, **soft/anti-aliased clips**, and **blurs** cost raster time (offscreen buffers) — simplify, use `Clip.hardEdge` where acceptable, limit blur area, and consider **Impeller** for shader jank (04_shaders_and_effects.md, 21 · jank_and_raster).
- **Profile**: check `rasterDuration` vs `buildDuration` during the paint/animation in release on a low-end device (21 · profiling_and_frame_budget).

Memory Representation

Cached paths/`Paint`/`Picture` live in painter fields (reused); per-frame allocation causes GC churn. Boundaries/offscreen buffers cost GPU memory — use where measured (02 · memory_and_gc).

Compiler Behavior

Not applicable (Impeller precompiles shaders).

Runtime Behavior

`paint` runs on repaint (gated by `shouldRepaint` / `repaint` Listenable). Cached objects skip rebuild cost; heavy ops rasterize slowly.

Flutter Engine Behavior

Draw ops rasterize on the GPU; soft clips/blurs/complex paths are the costly ones; `drawPicture` replays a cached recording efficiently (09 · rasterization).

Dart VM Behavior

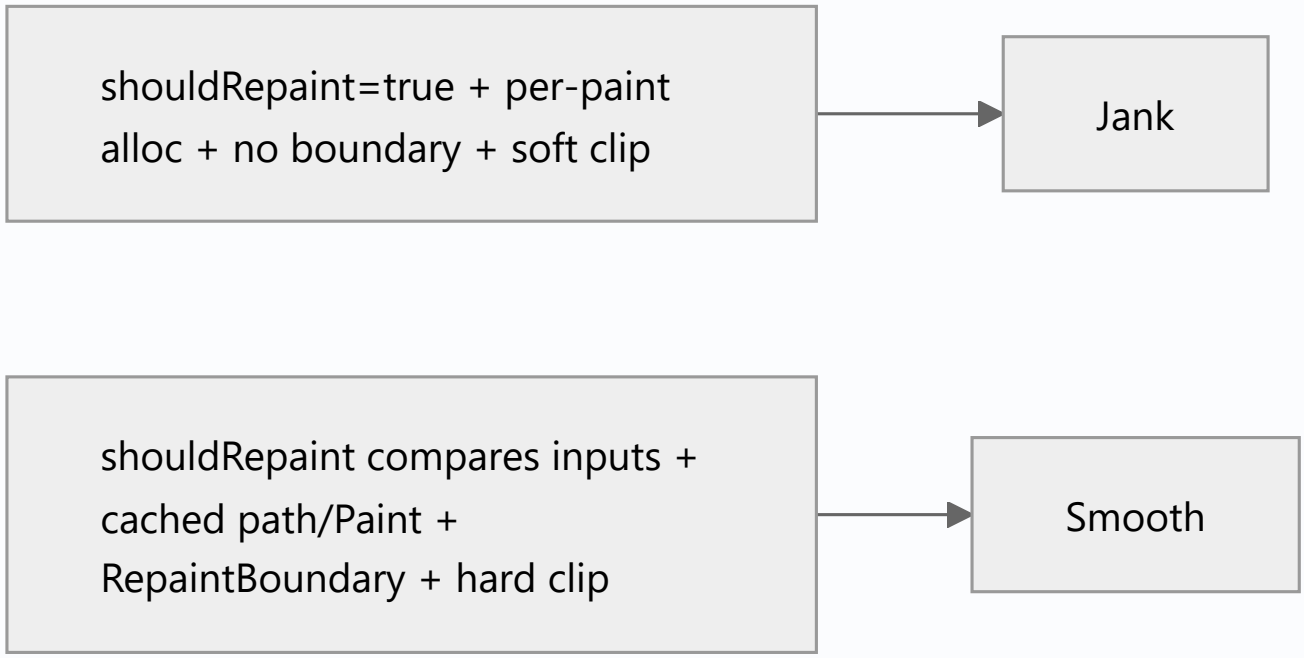
Heavy per-paint Dart (path/geometry computation) blocks the UI thread — precompute/memoize (02 · isolates).

Examples

```
import 'package:flutter/material.dart';


class ChartPainter extends CustomPainter {
  final List<double> data;
  // Cache Paint objects (reused across paints):
  static final _linePaint = Paint()
    ..style = PaintingStyle.stroke ..strokeWidth = 2 ..color = Colors.teal;
  Path? _cachedPath;
  Size? _cachedSize;


  ChartPainter(this.data, {Listenable? repaint}) : super(repaint: repaint);


  @override
  void paint(Canvas canvas, Size size) {
    // Rebuild the path only when inputs (data/size) change:
    if (_cachedPath == null || _cachedSize != size) {
      _cachedPath = _buildPath(size);
      _cachedSize = size;
    }
    canvas.drawPath(_cachedPath!, _linePaint); // cheap: draw cached path + reused Paint
```

```
  }

  Path _buildPath(Size size) {
    final path = Path();
    for (var i = 0; i < data.length; i++) {
      final x = size.width * i / (data.length - 1);
      final y = size.height * (1 - data[i]);
      i == 0 ? path.moveTo(x, y) : path.lineTo(x, y);
    }
    return path;
  }

  @override
  bool shouldRepaint(ChartPainter old) => old.data != data; // repaint only on data change
}

// Usage: isolate + drive animation efficiently
// RepaintBoundary(child: CustomPaint(painter: ChartPainter(data, repaint: controller)))
```

Diagrams



Common Mistakes

Mistake	Why	Fix
`shouldRepaint => true` /default	Repaints every frame	Compare inputs; false when unchanged
Building path/ `Paint` /gradient in `paint`	Per-frame allocation/cost	Cache in fields; rebuild on input change
No `RepaintBoundary` around animated painter	Repaints neighbors	Isolate it
`setState` -rebuild for animation	Whole widget rebuilds	`CustomPainter(repaint: controller)`
Soft/anti-aliased clips + blurs each frame	Raster cost (offscreen)	Hard-edge clips; limit blur; cache
Recomputing static drawing every frame	Wasted work	Record once to `ui.Picture` ; `drawPicture`
Profiling in debug	Unrepresentative	Profile in release/low-end

Best Practices

- **Implement** `shouldRepaint` comparing inputs (repaint only on change) — the top rule.
- **Cache** paths/ `Paint` /gradients/ `TextPainter` (and static drawings as a `ui.Picture`); recompute only when inputs change.
- **Isolate** with `RepaintBoundary` ; **animate** via `repaint: Listenable` (not widget rebuilds); keep the animated area small.
- **Budget raster ops**: simplify paths, prefer hard-edge clips, limit blur area, adopt Impeller for shader jank.
- **Profile in release** (build vs raster) on a low-end device; verify before/after.

Performance

The rules target each phase: repaint gating (`shouldRepaint`), UI-thread cost (caching/cheap paint), raster cost (simpler ops/clips/Impeller), and isolation (`RepaintBoundary`). Together they keep custom visuals at 60/120fps (21 · profiling_and_frame_budget).

Advantages / Disadvantages

- + Smooth, efficient custom visuals; scalable to animated charts/gauges/drawings.
- − Requires discipline (`shouldRepaint` /caching/isolation/raster budgeting) and release profiling; boundaries cost GPU memory.

Interview Questions

1. 🟢 **What's the #1 custom-paint perf rule?** — Implement `shouldRepaint` to repaint only when inputs change (never default/true).

2. 🟢 **Why cache paths/`Paint`?** — Building them inside `paint` allocates every frame (GC churn) and wastes CPU; cache and rebuild only on input change.
3. 🟡 **How do you animate a painter efficiently?** — Pass an `AnimationController` as `CustomPainter(repaint:)` (repaints on ticks without rebuilding the widget) and isolate with `RepaintBoundary`.
4. 🟡 **What custom-paint ops are raster-expensive?** — Complex paths, soft/anti-aliased clips, and blurs (offscreen buffers) — simplify/hard-edge/limit and consider Impeller.
5. 🟡 **How do you draw a complex static picture cheaply?** — Record it once with `PictureRecorder` / `ui.Picture` and `drawPicture` it (no per-frame rebuild).
6. 🔴 **How can a painter be raster-bound with cheap `paint`?** — Heavy raster ops (blurs/soft clips) or shader compilation cost GPU time regardless — check `rasterDuration`.
7. 🔴 **Why isolate an animated painter?** — So its per-frame repaints don't force neighbors/backgrounds to repaint (localizes cost).

Senior Engineer Tips

- Default recipe: correct `shouldRepaint` + cached path/`Paint` + `CustomPainter(repaint: controller)` + `RepaintBoundary` — solves most custom-paint jank.
- Cache static complex drawings as a `ui.Picture` and replay it; recompute only on real changes.
- Profile the *specific* painter in release; attribute UI vs raster before optimizing.

Architect Perspective

Custom-paint performance is where the drawing tools meet the pipeline: gate repaints, cache work, isolate, and budget raster. Encapsulating these in reusable painter widgets yields data-viz/drawing features that stay smooth as data/animation scale — the payoff of understanding the paint/raster phases (Module 09, Module 21).

Summary

- Custom paint runs per frame — implement `shouldRepaint`, cache expensive work (paths/`Paint`/`Picture`), isolate with `RepaintBoundary`, animate via `repaint:`, budget raster ops.
- Profile in release (build vs raster) on low-end; verify.
- These rules keep charts/gauges/drawings/effects at 60/120fps.

Revision Notes

- `shouldRepaint` compares inputs (top rule); cache paths/`Paint`/gradients/`TextPainter`/`ui.Picture`.
- Animate via `CustomPainter(repaint: controller)` + `RepaintBoundary`; keep animated area small.
- Raster budget: simpler paths, hard-edge clips, limited blur, Impeller.

- Profile release/low-end (build vs raster); before/after.

Practice Questions

1. Why is a wrong `shouldRepaint` the top perf bug?
2. How do you animate a painter without rebuilding the widget?
3. How do you draw a complex static picture cheaply each frame?

Coding Questions

1. Add correct `shouldRepaint` + cached path/`Paint` to a chart painter.
2. Drive a painter animation via `repaint:` inside a `RepaintBoundary`.
3. Record a static complex drawing to a `ui.Picture` and replay it.

Mini Project

Fast animated chart (Flutter): Optimize a janky animated line chart: correct `shouldRepaint`, cached path/`Paint`, `CustomPainter(repaint: controller)` + `RepaintBoundary`, hard-edge clip, and a `ui.Picture` for the static grid. Profile release before/after (build vs raster). Acceptance: repaints only on data change; no per-frame allocation; isolated; measured 60fps; runs.