Ingeniería de Flutter

Digital Version

Contents

Part I

Fundamentos de la Ingeniería Flutter

CHAPTER 2

Ingeniería Flutter: Conceptos Clave

Reviewers: Anna Leushchenko, Oleksandr Leushchenko

Bienvenido al emocionante mundo de la ingeniería Flutter. Este capítulo explora los principios y conceptos fundamentales que forman la base del desarrollo de software exitoso usando Flutter. A través de esta exploración, obtendrás valiosas percepciones sobre las perspectivas y enfoques únicos que distinguen a la ingeniería Flutter de la programación convencional, equipándote con el conocimiento y comprensión para crear aplicaciones impactantes y duraderas.

2.1 Ingeniería de Software con Flutter

A lo largo de mi carrera en ingeniería de software, adoptar Flutter ha marcado una evolución significativa en mi enfoque hacia la tecnología. Más que simplemente adquirir una nueva habilidad, ha implicado adoptar una estrategia integral que abarca todo el ciclo de vida del desarrollo de software, desde el diseño y desarrollo hasta las pruebas y mantenimiento.

Mi diverso trasfondo en varias tecnologías me ha ayudado a obtener una mejor perspectiva sobre Flutter, que veo tanto como una herramienta técnica como una forma de promover la innovación y creatividad en el desarrollo de software. La ingeniería Flutter adopta un enfoque holístico que equilibra cuidadosamente la experiencia del usuario, la gestión eficiente del tiempo, las consideraciones de escalabilidad y los compromisos requeridos para crear software impactante.

La arquitectura multiplataforma de Flutter permite a los desarrolladores concentrarse en crear una experiencia de usuario excepcional en lugar de adentrarse en los detalles específicos de la plataforma. A diferencia del desarrollo nativo, que se centra en adherirse a las directrices de la plataforma, Flutter prioriza la marca y la experiencia del usuario. Este enfoque alienta a los desarrolladores a priorizar la usabilidad universal sobre las limitaciones de la plataforma, llevando a una mentalidad más centrada en el usuario.

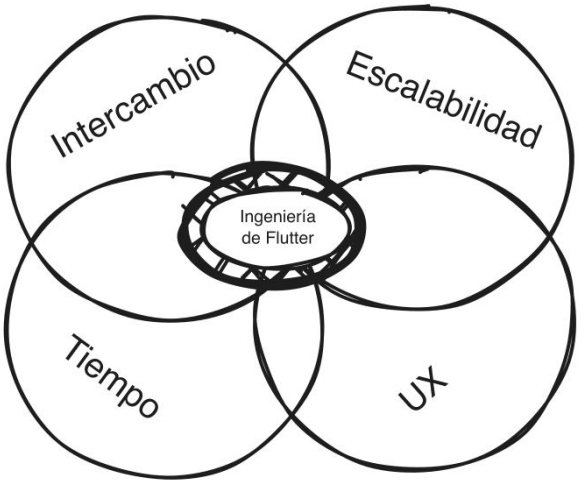

Figure 2.1: Pilares de ingeniería Flutter

La experiencia del usuario (UX) en la ingeniería Flutter es una lente crítica a través de la cual *debe ser* analizado cada proyecto. Constantemente me pregunto, "*£Qué pensará el usuario de esta característica? £Mejora o complica su experiencia?*" Por ejemplo, al crear una aplicación educativa basada en Flutter, responder estas preguntas ayuda a hacer el diseño intuitivo y atractivo para los aprendices; equilibrar la estética con la funcionalidad resulta en una experiencia encantadora. El desafío es alinear este enfoque centrado en el usuario con las capacidades técnicas de Flutter, asegurando que la aplicación sea tan agradable de interactuar como funcional.

El tiempo es un recurso vital en el desarrollo de software. Es una restricción finita que necesita ser gestionada eficientemente para cumplir con los plazos del proyecto y entregar valor a los interesados. La gestión del tiempo en el desarrollo de software va más allá de simplemente alcanzar hitos. El tiempo es un elemento multifacético y dinámico en el ámbito del desarrollo Flutter. Me lleva a preguntar frecuentemente, "*£Cuál es la expectativa de vida de mi código Flutter? £Cuánto va a durar esta aplicación? £Un año o una década? £Cuál es nuestro plazo de entrega?*" Estas preguntas van más allá de cumplir con los plazos del proyecto y se adentran en la prueba de futuro de la aplicación.

Por ejemplo, cuando trabajo en una aplicación de hogar inteligente basada en Flutter, me centro no solo en su lanzamiento inmediato sino también en su adaptabilidad a futuras tendencias de IoT y evoluciones tecnológicas. Este enfoque asegura que la aplicación permanezca relevante y funcional a lo largo del tiempo, adaptándose a los cambios en el comportamiento del usuario y la tecnología. También me recuerda incorporar el mecanismo para actualizar Flutter y otras dependencias de terceros.

El concepto de **escala** en la ingeniería Flutter es complejo y provocador. Al embarcarme en un nuevo proyecto, a menudo considero, "*£Cuántas personas están*

involucradas y qué roles juegan en el desarrollo y mantenimiento del proyecto? £Cuántos usuarios finales utilizarán esta aplicación más tarde?" Estas preguntas se vuelven particularmente relevantes en proyectos a gran escala como una aplicación logística integral desarrollada con Flutter. Aquí, el desafío radica en gestionar una base de código robusta y orquestar un equipo con diversa experiencia, asegurando un desarrollo cohesivo y eficiente en diferentes plataformas y dispositivos.

Los **compromisos** en la ingeniería Flutter implican tomar decisiones estratégicas que equilibren varios aspectos del proyecto. Por ejemplo, a menudo me enfrento a decisiones como, "*£Debería implementar una característica avanzada e intensiva en recursos que podría mejorar la experiencia del usuario pero también afectar el rendimiento de la aplicación en algunos dispositivos?*" Un ejemplo es elegir entre gráficos de alta resolución y un rendimiento suave en una aplicación de juegos. Otro es decidir entre implementar una animación avanzada que mejore la experiencia del usuario y mantener una aplicación ligera y de carga rápida, lo que ejemplifica el tipo de toma de decisiones estratégicas que define la ingeniería Flutter. Estas decisiones no son meramente técnicas, sino que también se alinean con los objetivos más amplios del proyecto y las expectativas de sus usuarios.

En mi experiencia, ingeniar software con Flutter es un proceso detallado de elaborar soluciones adaptables y escalables que resuenen con los usuarios finales. Implica una mezcla de habilidades técnicas, planificación estratégica y resolución creativa de problemas, todo dirigido a construir aplicaciones funcionales pero también atractivas y sostenibles en el mundo dinámico de la tecnología digital.

2.2 Desempacando los Principios Fundamentales

Para comprender completamente estos conceptos, exploremos el desarrollo de aplicaciones Flutter a través de la lente de los principios fundamentales de la ingeniería de software.

2.2.1 Paradigmas de Desarrollo

En el desarrollo de software, diversas ideologías y metodologías guían la construcción de sistemas. Estos principios rectores, o paradigmas de desarrollo, ofrecen distintas lentes a través de las cuales los desarrolladores abordan y dan forma al software.

Diferentes lenguajes de programación a menudo se asocian con paradigmas específicos, y la elección del lenguaje puede influir en cómo los desarrolladores piensan sobre y resuelven problemas. Algunos lenguajes, como Dart, admiten múltiples paradigmas.

A lo largo de la historia de la computación, han surgido varios paradigmas bien conocidos, cada uno dejando una huella significativa en el campo. Estos incluyen la Programación Procedural, la Programación Orientada a Objetos (OOP), la

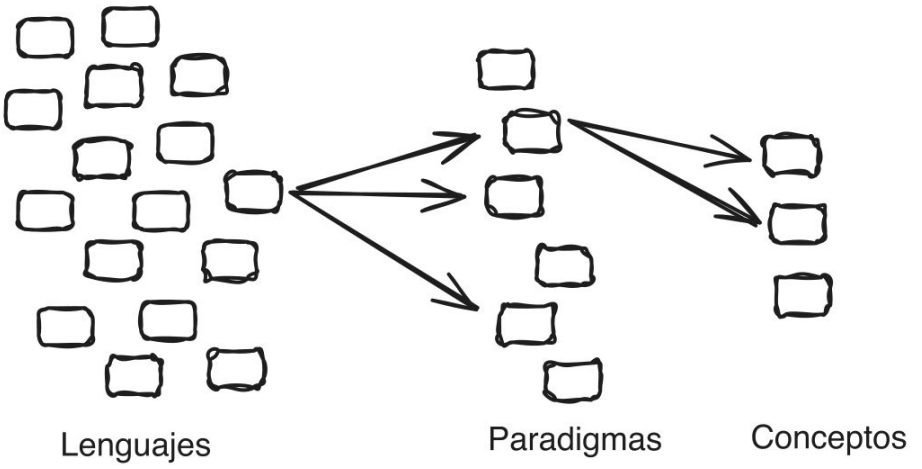

Figure 2.2: Paradigmas y conceptos de desarrollo

Programación Funcional, el Desarrollo Ágil, la programación Dirigida por Eventos, la programación Imperativa y Declarativa, etc.

Estos paradigmas, sin embargo, rara vez existen en aislamiento. Flutter adopta un entorno de programación **multiparadigma**, utilizando diversas técnicas de programación donde sus fortalezas son más beneficiosas. Exploraremos algunos de los hilos clave en este enfoque multifacético:

2.2.2 Constraint and Composition Programming

El corazón del diseño de Flutter radica en su uso de la composición. Este enfoque implica construir widgets complejos combinando otros más simples. Un ejemplo es el widget `TextButton`, una composición de otros widgets como `Material`, `InkWell`[1] y `Padding`[2].

Imagina tu aplicación como una obra maestra gigante de Lego. Cada **widget**, una pieza pequeña y especializada (texto, botones, imágenes), se ensambla, construyendo pantallas complejas. Este método de composición agresiva resulta en una UI altamente personalizable y flexible. Aprenderás más sobre esto en el capítulo 2.

En Flutter, el sistema de layout emplea una forma de programación de restricciones para establecer la geometría de los elementos de UI. Las restricciones sobre el tamaño, como el ancho y alto mínimo y máximo, se pasan de los widgets padres a sus hijos. Los widgets hijos luego ajustan sus tamaños para cumplir con estas re-

[1] https://api.flutter.dev/flutter/material/InkWell-class.html
[2] https://api.flutter.dev/flutter/widgets/Padding-class.html

stricciones, permitiendo a Flutter organizar toda la UI, a menudo en un solo paso. Este enfoque asegura un layout responsive y consistente en diferentes dispositivos.

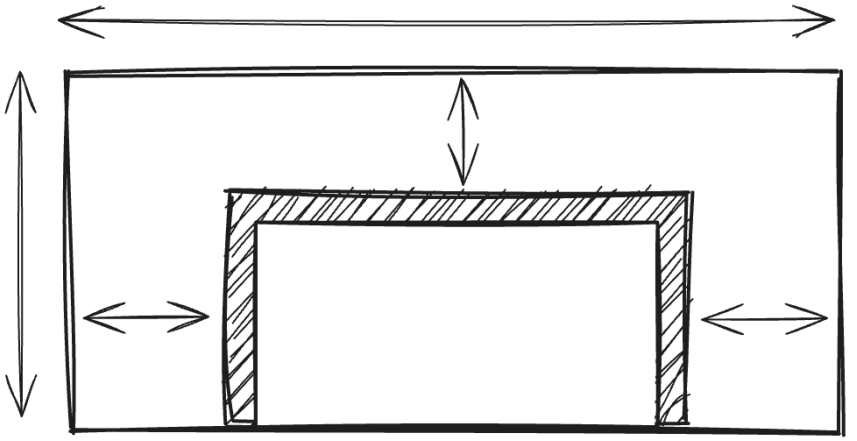

Figure 2.3: Las restricciones disminuyen. Los tamaños suben. El padre fija la posición.

2.2.3 Programación Imperativa y Declarativa

En Flutter, la programación imperativa se aplica en escenarios que requieren control directo de operaciones paso a paso. La lógica de negocios de la aplicación móvil frecuentemente implica secuencias de pasos, condiciones y bucles. La programación imperativa permite a los desarrolladores expresar estas secuencias de manera natural, facilitando la escritura y mantenimiento de la lógica.

Aquí hay un simple ejemplo de una función con estilo imperativo con declaraciones condicionales:

```
bool isPositive(int x) {
  if (x > 0) {
    print('x es positivo');
    return true;
  }
  print('x es negativo o cero');
  return false;
}
```

Otro ejemplo común de programación imperativa en Flutter se puede ver en las pruebas unitarias:

```
testWidgets(
  'CustomButton muestra una etiqueta',
```

```
  (WidgetTester tester) async {
    // Describe la situación bajo prueba
    await tester.pumpWidget(
        MaterialApp(home: CustomButton(label: 'Prueba')));

    // Lista las invariantes que la prueba debe coincidir
    expect(find.text('Prueba'), findsOneWidget);

    // Avanza el reloj o inserta eventos si es necesario
    await tester.tap(find.byType(CustomButton));
    await tester.pump();
  },
);
```

La programación declarativa es un aspecto clave del marco de Flutter, visto prominentemente en cómo se construyen los widgets. En Flutter, la UI se define típicamente usando la sintaxis declarativa de Dart, donde los métodos de construcción de widgets consisten en expresiones únicas con constructores anidados.

Considera el widget **`ListView`**:

```
ListView(
  children: [
    ListTile(title: Text('Item 1')),
    ListTile(title: Text('Item 2')),
    // Ítems de lista adicionales
  ],
)
```

En este ejemplo, **`ListView`** y sus hijos se definen de manera concisa y expresiva.

Este enfoque permite a los desarrolladores describir cómo debería verse la UI en lugar de cómo construirla paso a paso, como en la programación imperativa. El estilo declarativo en Flutter simplifica el proceso de construir UIs complejas y mejora la legibilidad y mantenibilidad del código. Además, este método puede combinarse sin problemas con la programación imperativa para escenarios donde un enfoque puramente declarativo podría ser limitado, ofreciendo la flexibilidad para construir UIs más dinámicas e interactivas.

Mirando este código, puedes ver la aplicación con una AppBar y un texto centrado. No contiene la lógica que especifica **cómo** se construirá la UI, sino solo la **declaración** de **qué** verá el usuario:

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
```

```
          title: Text('Programación Declarativa en Flutter'),
        ),
        body: Center(
          child: Text('Hola, Flutter!'),
        ),
      ),
    );
  }
}
```

2.2.4 Programación Funcional y Orientada a Objetos

One of the core concepts of functional programming is called "pure function." Given the same input, it's a function that will always produce the same output and has no observable side effects. The result of a pure function depends only on its input parameters, and it does not modify any external state, which significantly simplifies maintenance and opens doors for many optimizations.

Flutter also embraces functional programming, particularly in `StatelessWidgets`[3] that resemble pure functions. For instance, the `Icon`[4] widget can be viewed as a function mapping its parameters to visual output.

Flutter's emphasis on immutable data structures. The entire `Widget`[5] class hierarchy and supporting classes like `Rect`[6] and `TextStyle`[7] embrace this immutability, keeping your UI stable and reliable.

Dart's `Iterable`[8] API is another example of its functional programming characteristics. Remember those handy functions like `map`, `where`, and `reduce` your use in Dart? These are examples of the functional style frequently used to process lists of values in the framework.

Flutter's framework dances with both class inheritance and dynamic prototypes. Core APIs are built with class hierarchies, where base classes like `RenderObject`[9] define high-level functionalities that subclasses like `RenderBox`[10] specialize, adopting the Cartesian coordinate system for geometry. But it's not just static inheritance – the `ScrollPhysics`[11] class lets you chain instances dynamically at runtime, composing, for example, paging physics with platform-specific quirks, all without needing a pre-chosen platform. This blend of inheritance and dynamic flexibility gives Flutter apps the power to adapt and evolve like never before!

[3] https://api.flutter.dev/flutter/widgets/StatelessWidget-class.html
[4] https://api.flutter.dev/flutter/widgets/Icon-class.html
[5] https://api.flutter.dev/flutter/widgets/Widget-class.html
[6] https://api.flutter.dev/flutter/dart-ui/Rect-class.html
[7] https://api.flutter.dev/flutter/painting/TextStyle-class.html
[8] https://api.flutter.dev/flutter/dart-core/Iterable-class.html
[9] https://api.flutter.dev/flutter/rendering/RenderObject-class.html
[10] https://api.flutter.dev/flutter/rendering/RenderBox-class.html
[11] https://api.flutter.dev/flutter/widgets/ScrollPhysics-class.html

You will learn more about OOP in Dart in Chapter 4.

Uno de los conceptos centrales de la programación funcional se llama "función pura". Dado el mismo input, es una función que siempre producirá el mismo output y no tiene efectos secundarios observables. El resultado de una función pura depende únicamente de sus parámetros de entrada, y no modifica ningún estado externo, lo que simplifica significativamente el mantenimiento y abre puertas para muchas optimizaciones.

Flutter también adopta la programación funcional, particularmente en `StatelessWidgets`[12] que se asemejan a funciones puras. Por ejemplo, el widget `Icon`[13] puede verse como una función que mapea sus parámetros a un output visual.

El énfasis de Flutter en estructuras de datos inmutables. Toda la jerarquía de clases `Widget`[14] y clases de apoyo como `Rect`[15] y `TextStyle`[16] adoptan esta inmutabilidad, manteniendo tu UI estable y confiable.

La API `Iterable`[17] de Dart es otro ejemplo de sus características de programación funcional. £Recuerdas esas funciones útiles como `map`, `where` y `reduce` que usas en Dart? Estos son ejemplos del estilo funcional frecuentemente utilizado para procesar listas de valores en el marco.

El marco de Flutter baila con la herencia de clases y prototipos dinámicos. Las APIs centrales están construidas con jerarquías de clases, donde clases base como `RenderObject`[18] definen funcionalidades de alto nivel que las subclases como `RenderBox`[19] especializan, adoptando el sistema de coordenadas cartesianas para la geometría. Pero no es solo herencia estática – la clase `ScrollPhysics`[20] te permite encadenar instancias dinámicamente en tiempo de ejecución, componiendo, por ejemplo, físicas de paginación con peculiaridades específicas de la plataforma, todo sin necesidad de una plataforma pre-elegida. ąEsta mezcla de herencia y flexibilidad dinámica da a las aplicaciones Flutter el poder de adaptarse y evolucionar como nunca antes!

Aprenderás más sobre la OOP en Dart en el Capítulo 3.

[12] https://api.flutter.dev/flutter/widgets/StatelessWidget-class.html
[13] https://api.flutter.dev/flutter/widgets/Icon-class.html
[14] https://api.flutter.dev/flutter/widgets/Widget-class.html
[15] https://api.flutter.dev/flutter/dart-ui/Rect-class.html
[16] https://api.flutter.dev/flutter/painting/TextStyle-class.html
[17] https://api.flutter.dev/flutter/dart-core/Iterable-class.html
[18] https://api.flutter.dev/flutter/rendering/RenderObject-class.html
[19] https://api.flutter.dev/flutter/rendering/RenderBox-class.html
[20] https://api.flutter.dev/flutter/widgets/ScrollPhysics-class.html

2.2.5 Abstracción y La Encapsulación

2.2.6 Programación Dirigida por Eventos

La abstracción y la encapsulación son principios fundamentales en la ingeniería de software que Flutter utiliza efectivamente en su arquitectura centrada en widgets.

La **abstracción** trata sobre simplificar sistemas complejos en modelos más manejables, y la **encapsulación** implica agrupar datos y sus operaciones asociadas dentro de clases, protegiendo la integridad de los datos y previniendo el acceso inapropiado.

La abstracción simplifica elementos complejos de UI en widgets manejables, enfocándose en atributos y funcionalidades esenciales. Por ejemplo, un widget `ListView` en Flutter abstrae las complejas funcionalidades de una lista desplazable en un componente fácil de usar.

En el contexto de Flutter, la encapsulación se aplica al desarrollo de widgets, y el concepto es evidente en la implementación del widget `Container`. El widget `Container` encapsula varios atributos o propiedades que definen su apariencia y comportamiento. Estos atributos incluyen ancho, alto, color, padding, margen y más. Los desarrolladores interactúan con el Container usando un conjunto bien definido de propiedades y métodos. La encapsulación asegura que los detalles internos de cómo el `Container` gestiona estos atributos estén ocultos del mundo exterior.

Juntos, la abstracción y la encapsulación en Flutter contribuyen a un marco donde los diseños complejos de UI se simplifican en componentes manejables, y los estados internos de los widgets están bien protegidos, mejorando la usabilidad y mantenibilidad. Aprenderás más sobre estos temas en el Capítulo 3.

2.2.7 Programación Dirigida por Eventos

Las interacciones del usuario en Flutter se manejan mediante un enfoque dirigido por eventos.

Un ejemplo principal de esto en Flutter es el uso de la clase `Listenable`[21]. Esta clase sirve como la base para el sistema de animación en Flutter, donde los cambios en el estado de la animación se tratan como eventos. `Listenable` proporciona un modelo de suscripción, permitiendo que múltiples oyentes registren callbacks que se activan en respuesta a eventos específicos. Este mecanismo asegura que varias partes de la UI permanezcan actualizadas y sincronizadas con los cambios subyacentes de datos o estado, reflejando la naturaleza reactiva del marco.

Además, widgets como `GestureDetector`[22] y herramientas de gestión de estado utilizan eventos para responder a las entradas del usuario, ejemplificando la pro-

[21] https://api.flutter.dev/flutter/foundation/Listenable-class.html
[22] https://api.flutter.dev/flutter/widgets/GestureDetector-class.html

gramación dirigida por eventos en el marco. Aprenderás más sobre esto en la parte 2 de este libro.

2.2.8 Programación Reactiva

En Flutter, la programación reactiva es un concepto clave que impulsa la naturaleza dinámica del desarrollo de UI. Este paradigma es aparente en cómo los widgets reaccionan a los cambios, actualizando su estado y apariencia en respuesta a interacciones del usuario o cambios internos de datos.

En el sistema reactivo de Flutter, cualquier nueva entrada proporcionada en el constructor de un widget inmediatamente desencadena una reconstrucción de ese widget, propagando cambios a través del árbol de widgets. Por el contrario, los cambios en widgets de nivel inferior pueden propagarse hacia arriba a través de manejadores de eventos y actualizaciones de estado.

Flutter aprovecha el soporte de Dart para streams para proporcionar un modelo de programación reactivo, y **`StreamBuilder`** es un widget que juega un papel clave en este paradigma:

```
final StreamController<int> _controller = StreamController<int>();
//
StreamBuilder<int>(
  stream: _controller.stream,
  builder: (context, snapshot) {
    if (snapshot.hasData) {
      return Center(
        child: Text('Datos del stream: ${snapshot.data}'),
      );
    } else {
      return Center(
        child: Text('Esperando datos...'),
      );
    }
  },
)
```

La programación reactiva es un paradigma de programación que gira en torno a la propagación de cambios y el manejo de flujos de datos asincrónicos.

2.2.9 Programación Genérica

Flutter utiliza genéricos para mejorar la seguridad de tipo y reducir errores. Esto es visible en widgets como **`DropdownButton<T>`** donde T representa el tipo de

fuente de datos, o clases como `State`[23]`<T>` y `GlobalKey`[24]`<T>`, donde T representa el tipo de widget o estado con el que están asociados.

2.2.10 Programación Concurrente

La concurrencia en Flutter se maneja a través de las características async de Dart como `Future`[25]s y `Stream`[26]s. Esto es crucial en escenarios como la obtención de datos de una red o el trabajo con tareas de larga duración.

Aprenderás más sobre Concurrencia y Paralelismo en el capítulo 8.

2.2.11 Cohesión y Acoplamiento

En la ingeniería de software, la **cohesión** y el **acoplamiento** son principios fundamentales que pueden hacer o deshacer la mantenibilidad y eficiencia de un sistema.

La cohesión describe la fuerza interna de un módulo y cuán estrechamente relacionados están sus elementos con su propósito central. Idealmente, los módulos exhiben una alta cohesión, con componentes que trabajan juntos hacia un único objetivo. El acoplamiento, sin embargo, trata sobre el grado de interdependencia entre módulos. Esforzarse por un bajo acoplamiento asegura que los módulos interactúen mínimamente, minimizando los efectos en cascada cuando se realizan cambios.

En el mundo de Flutter, dos principios fundamentales definen una obra maestra mantenible: **bajo acoplamiento y alta cohesión**. Desglosemos sus pasos en el escenario de Flutter:

Alta Cohesión

Flutter logra una alta cohesión diseñando widgets enfocados en funcionalidades específicas. Por ejemplo, el widget `Text` es únicamente responsable de mostrar una cadena de texto con estilo básico. Sus responsabilidades son claras y bien definidas, haciéndolo altamente cohesivo. Otro ejemplo es el widget `Image`, dedicado a mostrar imágenes y que no se entremezcla con funcionalidades no relacionadas con imágenes.

Bajo Acoplamiento

Flutter mantiene un bajo acoplamiento permitiendo que los widgets funcionen de manera independiente con una mínima dependencia entre ellos. Por ejemplo, el widget `Scaffold`, que proporciona la estructura de diseño visual básica de material design, opera independientemente del widget `FloatingActionButton`

[23]https://api.flutter.dev/flutter/widgets/State-class.html
[24]https://api.flutter.dev/flutter/widgets/GlobalKey-class.html
[25]https://api.flutter.dev/flutter/dart-async/Future-class.html
[26]https://api.flutter.dev/flutter/dart-async/Stream-class.html

utilizado para botones de acción. Modificaciones a un `FloatingActionButton`, como cambiar su ícono o color, no afectan el layout o funcionamiento del `Scaffold`, demostrando un bajo acoplamiento entre estos componentes.

Puedes preguntarte ahora acerca de los Temas. Los temas afectan principalmente el estilo visual, manteniendo la funcionalidad separada. Los temas personalizables en diferentes niveles refuerzan el bajo acoplamiento, asegurando que los cambios no vinculen estrechamente los widgets.

Generalmente, los widgets deben depender de canales de comunicación establecidos como callbacks y eventos, minimizando los efectos en cascada cuando uno cambia de tono.

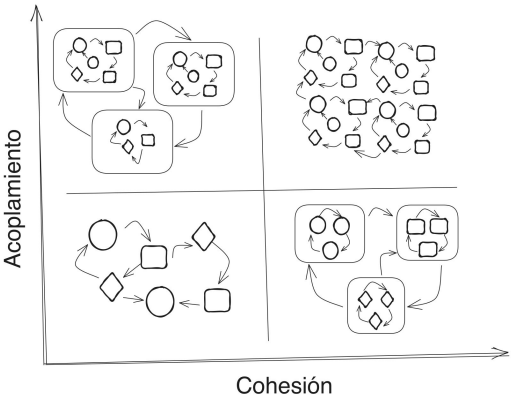

Figure 2.4: Acoplamiento y cohesión

Al desarrollar con Flutter, es crucial integrar los principios de "bajo acoplamiento, alta cohesión" para una arquitectura de app robusta. Crea widgets que operen de manera independiente; por ejemplo, un widget `PaymentProcessing` no debería estar intrincadamente vinculado a un widget `UserDashboard`, demostrando un bajo acoplamiento. Además, diseña cada widget con un rol enfocado, como un widget `ChatScreen` que maneja exclusivamente características de mensajería, asegurando una alta cohesión.

Mientras construyes, pregúntate regularmente: "£Cambiar un widget impacta innecesariamente a otros?" y "£Está bien definido el propósito y función de cada widget y es autocontenida?" Reflexionar sobre estas preguntas te guiará en la creación de una aplicación Flutter más eficiente y bien estructurada.

2.2.12 Separación de Concerns y Modularidad

La Separación de Concerns (SoC) y la Modularidad son conceptos fundamentales en la ingeniería de software que mejoran significativamente la organización del código, la mantenibilidad y la escalabilidad.

Separación de Concerns es un principio de diseño que implica desglosar una aplicación de software en secciones distintas, cada una abordando un aspecto o

preocupación específica. Este enfoque ayuda a simplificar la complejidad de un programa al permitir que los desarrolladores se centren en un área a la vez sin ser abrumados por otras. Facilita la reducción de interdependencias, lo que a su vez hace que la aplicación sea más flexible y fácil de mantener. **Modularidad**, sin embargo, se refiere a dividir un sistema de software en módulos separados e intercambiables, donde cada módulo encapsula una funcionalidad específica. Este enfoque de diseño facilita la prueba, depuración y actualización de componentes individuales, lo que lleva a un sistema más robusto y adaptable.

La arquitectura basada en widgets de Flutter es inherentemente modular, cada widget encapsulando un aspecto específico de UI o funcionalidad. Esto se alinea con el principio de SoC, donde preocupaciones como la interfaz de usuario, la lógica de negocio y la gestión de datos se mantienen distintas.

Como desarrollador de Flutter, puedes aprovechar estos principios para crear aplicaciones robustas y eficientes. Por ejemplo, en una aplicación de tareas basada en Flutter, puedes implementar SoC teniendo una capa de UI separada con widgets como `TaskListWidget` para mostrar tareas. La lógica de negocio puede encapsularse en una clase `TaskManager` que maneja operaciones relacionadas con tareas. Mientras tanto, la gestión de datos puede ser manejada por un `DatabaseService` responsable de almacenar y recuperar datos de tareas. La modularidad se puede lograr creando componentes reutilizables como un `LoginService` para autenticación de usuarios. Estos pueden usarse en diferentes partes de tu aplicación o incluso en otros proyectos, a menudo residiendo dentro de la carpeta `lib` o pueden ser extraídos y creados como paquetes `pub` individuales.

También es bueno saber que en Flutter, el concepto de Modularidad a menudo se cruza con la arquitectura "paquete por característica", siendo un aspecto único que los módulos a menudo pueden tomar la forma de widgets. Este enfoque organiza la aplicación en módulos basados en características específicas, donde cada módulo, o en muchos casos, cada widget, representa una funcionalidad distinta de la app.

Aprenderás más sobre esto en la Parte 2 de este libro, donde me adentro en la Arquitectura.

2.2.13

2.2.14 Patrones de Diseño y Estrategias

Los patrones de diseño en la ingeniería de software son soluciones establecidas para problemas comunes de diseño. Actúan como plantillas que se pueden aplicar a problemas recurrentes en el diseño de software, como gestionar la creación de objetos, facilitar la comunicación entre objetos y organizar interacciones complejas, permitiéndote escribir código que es:

- **Reutilizable:** Los patrones son reutilizables, ahorrando tiempo y esfuerzo mientras promueven la consistencia en toda tu aplicación.
- **Mantenible:** El código estructurado con patrones es más fácil de entender, modificar y extender en el futuro.

- **Flexible:** Los patrones se adaptan a diferentes contextos y requisitos, haciendo que tu código sea más versátil.

Flutter no dicta patrones específicos; sus características principales y arquitectura se prestan naturalmente a varios patrones. Un excelente ejemplo de un patrón de diseño utilizado dentro del marco de Flutter es el **Patrón de Constructor**. Un uso cotidiano del patrón de Constructor en Flutter es el widget `ListView.builder`. Este patrón se emplea frecuentemente en el proceso de creación de widgets de Flutter. El patrón de Constructor separa la construcción de un objeto complejo de su representación, permitiendo que el mismo proceso de construcción cree diferentes representaciones.

Aprenderás más sobre patrones de diseño en el Capítulo 5.

2.2.15 Eficiencia, Escalabilidad y Compensaciones

En la ingeniería de software, particularmente en el desarrollo con Flutter, comprender y navegar las complejidades de la eficiencia, escalabilidad y compensaciones es esencial. Estos conceptos se centran en cómo las aplicaciones utilizan los recursos (eficiencia), se adaptan al crecimiento (escalabilidad) y gestionan el delicado equilibrio entre necesidades competidoras (compensaciones). Estas elecciones no son solo sobre aspectos financieros sino que abarcan varios factores como la asignación de recursos, el esfuerzo del personal, etc.

Ir más allá de la mentalidad de "*porque todos los demás lo están haciendo*" y hacia un enfoque impulsado por el consenso que prioriza decisiones bien razonadas y específicas al contexto es esencial. Este enfoque es particularmente relevante en Flutter al sopesar opciones como técnicas de gestión de estado o la integración de paquetes externos.

Por ejemplo, elegir `setState` por su simplicidad podría llevar a desafíos de escalabilidad, mientras que métodos avanzados como BLoC, aunque inicialmente más complejos, ofrecen beneficios a largo plazo en escalabilidad y mantenibilidad. Del mismo modo, usar `cached_network_image` trae eficiencia y una mejor experiencia de usuario pero introduce complejidades como dependencias añadidas, que pueden afectar el mantenimiento a largo plazo y la compatibilidad con actualizaciones de Flutter.

En mi experiencia, "*Depende*" es particularmente significativo en la ingeniería de software, especialmente al trabajar con Flutter. Esto destaca la importancia de entender el contexto específico de cada elección tecnológica. Como desarrollador, constantemente equilibro factores como la facilidad de uso, escalabilidad y mantenibilidad futura. Estas decisiones son más que solo sobre resultados inmediatos; moldean la salud a largo plazo del proyecto. Esto requiere un profundo nivel de análisis crítico y previsión, enfatizando la necesidad de decisiones bien informadas y sostenibles en el campo de rápido desarrollo del software.

2.2.16 Verificación, Validación y "Shifting Left"

El modelo de Verificación y Validación en la ingeniería de software es un proceso utilizado para asegurar que un sistema cumpla todas sus especificaciones y cumpla con su propósito previsto. La "Verificación" implica verificar si el sistema está construido correctamente y cumple con los requisitos especificados. Esto a menudo se refiere como "Pruebas Estáticas". Por otro lado, la "Validación" verifica si se construyó el sistema correcto y cumple con las necesidades de los usuarios, conocido como "Pruebas Dinámicas". Este modelo es crucial para asegurar la calidad y fiabilidad de los sistemas de software.

En el contexto de Flutter, el modelo de Verificación y Validación (V&V) podría adaptarse a su ecosistema de la siguiente manera:

1. **Análisis de Requerimientos**: Entender lo que la aplicación busca lograr y el problema que resuelve para los usuarios.
2. **Arquitectura de la Aplicación**: Definir la estructura general de la aplicación, incluyendo estrategias de gestión de estado y navegación.
3. **Diseño de Características**: Detallar el plan de implementación de cada característica de la aplicación, abarcando la lógica de negocio y la interfaz frontend.
4. **Diseño de Unidades**: Desglosar las características en unidades más pequeñas y testeables, típicamente funciones individuales o widgets.

Las fases de pruebas correspondientes son:

1. **Pruebas Unitarias**: Verificar la funcionalidad de unidades o componentes individuales, especialmente la lógica de negocio.
2. **Pruebas de Widgets**: Asegurar que los widgets de Flutter se rendericen correctamente e interactúen como se espera, para que la composición general de widgets como característica funcione.
3. **Pruebas de Integración**: Evaluar la interacción entre unidades o widgets combinados dentro de la aplicación para que la arquitectura general funcione.
4. **Pruebas de Aceptación del Usuario**: Validar la aplicación contra los requerimientos del usuario, a menudo a través de pruebas manuales, para asegurar que cumpla con sus expectativas.

El modelo V&V asegura que la aplicación Flutter se desarrolle correctamente y cumpla con sus necesidades diseñadas en cada etapa, desde los requisitos hasta las pruebas de aceptación.

El concepto de "Shifting Left" en el desarrollo de software, particularmente dentro de Flutter, sugiere que invertir tiempo en las etapas iniciales del ciclo de vida del desarrollo, como diseño, desarrollo y pruebas iniciales, es más rentable que abordar problemas más adelante en la cadena. Cuanto más cerca se descubre un problema de su introducción (típicamente en el lado izquierdo del cronograma de desarrollo), menos costoso es de corregir. Esto se debe a que los problemas encontrados durante etapas como staging o producción (en el lado derecho) pueden ser significativamente más costosos y consumir más tiempo para resolver debido

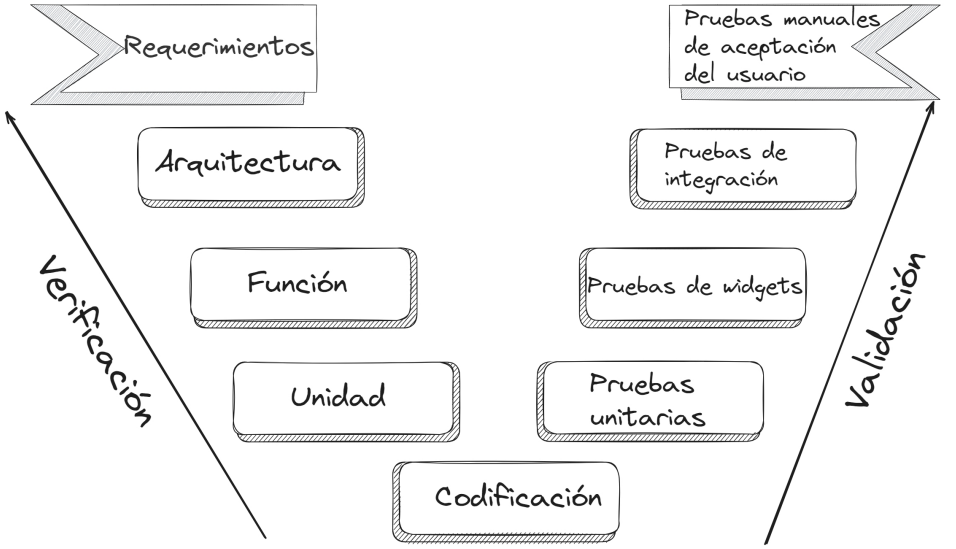

Figure 2.5: Flutter Verification and Validation Model

a la complejidad de la depuración y el impacto potencial en la experiencia del usuario.

"Shifting Left" en Flutter significa efectivamente incorporar prácticas como el análisis de código estático para detectar errores sintácticos y posibles bugs temprano. Las revisiones de código son esenciales para asegurar la calidad y captar problemas que las herramientas automatizadas podrían pasar por alto. Integrar la automatización dentro de las pipelines CI permite ejecutar consistentemente pruebas unitarias, de widgets e integración, asegurando que las nuevas adiciones de código cumplan con los estándares de calidad antes de fusionar. Además, emplear banderas de características y pruebas A/B permite a los desarrolladores probar nuevas características selectivamente en entornos de producción, reduciendo el riesgo de problemas generalizados.

Al incorporar estas prácticas tempranamente y a lo largo del proceso de desarrollo de Flutter, los equipos pueden mitigar riesgos, reducir el costo de la remediación de defectos en etapas tardías y entregar aplicaciones robustas y de alta calidad de manera eficiente.

El concepto de "Shifting Left" enfatiza la integración de estos procesos al principio del ciclo de desarrollo. Para los desarrolladores de Flutter, significa realizar pruebas y verificaciones de calidad desde las etapas iniciales. Esta intervención temprana ayuda a detectar y corregir problemas prontamente, reduciendo el costo y tiempo típicamente asociados con la depuración en etapas posteriores. Implementar estas prácticas en Flutter mejora la calidad del código y realza la fiabilidad de la aplicación y la experiencia del usuario.

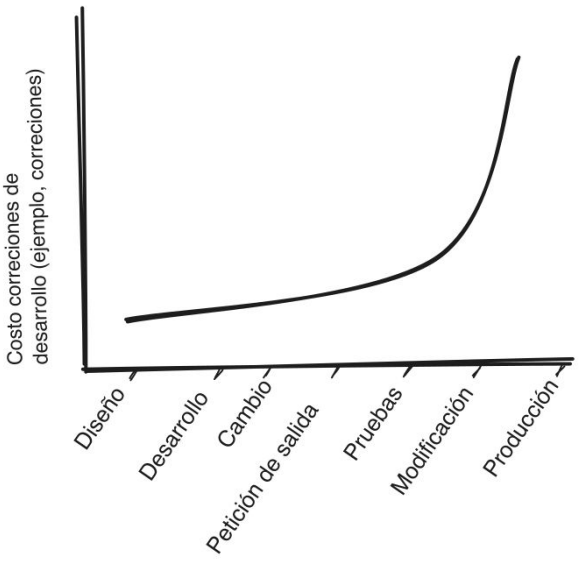

Figure 2.6: Concepto de Shifting Left en el desarrollo de Flutter

2.2.17 Toma de Decisiones Informadas en el Desarrollo

En Flutter y el desarrollo de software, tomar decisiones informadas a menudo implica sopesar factores cuantificables contra aspectos más matizados y no cuantificables. Por ejemplo, un desarrollador puede necesitar elegir entre usar soluciones de gestión de estado como BLoC, que ofrece escalabilidad pero con una complejidad añadida, versus opciones más sencillas como `setState`, que son más fáciles de implementar pero pueden no escalar tan bien para aplicaciones más grandes.

Además, las decisiones en Flutter a veces son sobre algo más que elementos medibles. Considera la implementación de un widget personalizado versus el uso de un widget de terceros existente. La decisión abarca no solo la funcionalidad inmediata sino también factores como el mantenimiento a largo plazo, la fiabilidad del paquete de terceros y su alineación con las necesidades evolutivas de la aplicación.

Equilibrar estos aspectos requiere una cuidadosa consideración tanto de los impactos cuantificables como de las implicaciones a largo plazo menos tangibles, pero igualmente importantes, de las elecciones de desarrollo en Flutter.

2.3 Ciclo de Vida del Desarrollo de Flutter

Antes de adaptarlo para el desarrollo de Flutter, entendamos el Ciclo de Vida del Desarrollo de Software (SDLC). SDLC es un marco estructurado que define una serie de etapas para construir y entregar aplicaciones de software. Proporciona una hoja de ruta para desarrolladores y partes interesadas, asegurando calidad, eficiencia y previsibilidad a lo largo del desarrollo.

Hay varios modelos de SDLC, cada uno con sus etapas específicas y énfasis. Algunos modelos populares incluyen:

- **Modelo Waterfall:** Este modelo lineal y secuencial sigue un enfoque estricto de etapas, donde cada etapa debe completarse antes de pasar a la siguiente. Es eficiente para requisitos precisos y entornos controlados.
- **Modelo Ágil:** Este modelo iterativo e incremental enfatiza la flexibilidad y adaptabilidad. Descompone el desarrollo en ciclos más pequeños (sprints), permitiendo retroalimentación continua y entrega de software funcional.
- **Modelo Espiral:** Este modelo impulsado por riesgos combina la naturaleza iterativa de Ágil con el control de Waterfall. Implica evaluación de riesgos a lo largo del ciclo de desarrollo, haciéndolo adecuado para proyectos de alto riesgo.

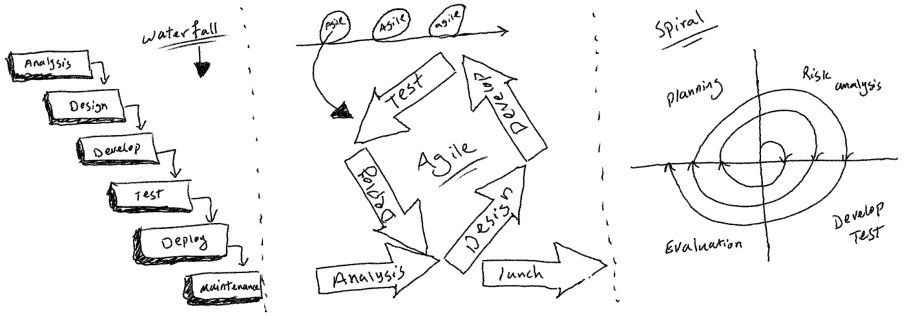

Figure 2.7: Waterfall vs. Ágil vs. Espiral

Independientemente del modelo elegido, las etapas centrales de un SDLC típicamente incluyen:

1. **Análisis**: Esta fase implica comprender las necesidades y objetivos específicos de la aplicación Flutter. Incluye la recolección de requisitos detallados de las partes interesadas y la definición del alcance de la aplicación.
2. **Diseño**: Basado en los requisitos, se diseña la arquitectura general del sistema para la aplicación Flutter. Esto incluye decidir sobre el flujo de navegación de la aplicación, el enfoque de gestión de estado y el diseño general de UI/UX.
3. **Desarrollo**: Aquí tiene lugar la codificación real de la aplicación Flutter. Los desarrolladores escriben código Dart para implementar las funcionalidades definidas, adhiriéndose a las especificaciones de diseño. Se debería escribir pruebas unitarias y de widgets como parte de las mejores prácticas al escribir código.
4. **Pruebas**: En esta fase crítica, la aplicación Flutter se somete a diversas pruebas para asegurar la calidad y el rendimiento. Esto incluye pruebas de integración y potencialmente pruebas de aceptación del usuario para validar todos los aspectos de la aplicación.

5. **Despliegue**: Una vez que las pruebas están completas y la aplicación está libre de errores, se despliega en las plataformas apropiadas (por ejemplo, Google Play Store, Apple App Store). Esto podría involucrar la configuración de pipelines de CI/CD para procesos de despliegue eficientes.
6. **Mantenimiento**: Post-despliegue, la aplicación entra en la fase de mantenimiento, donde se actualiza regularmente, se arreglan errores y se agregan nuevas características según la retroalimentación de los usuarios o requisitos cambiantes. La monitorización de la aplicación es otra parte de esta fase. La monitorización está relacionada con la notificación de crashes y errores, análisis, medición de rendimiento, etc.

Al adaptar el Ciclo de Vida del Desarrollo de Software (SDLC) para el desarrollo de Flutter, entran en juego algunas consideraciones específicas para aprovechar las características únicas del marco. Durante la fase de Análisis de Requerimientos, un enfoque centrado en móviles es clave, pero con un ojo en la posible expansión a web y escritorio, gracias a la versatilidad de Flutter. La característica de recarga en caliente de Flutter facilita la prototipación rápida y la retroalimentación iterativa, mientras que los requisitos de rendimiento para animaciones y capacidad de respuesta son cruciales para la compatibilidad con diversos dispositivos.

A medida que el proceso avanza hacia el Diseño del Sistema, Desarrollo, Pruebas y Despliegue, la selección de widgets apropiados y soluciones de gestión de estado adaptadas a la complejidad de la aplicación se vuelve vital. Características del lenguaje Dart, como la seguridad de nulos y las mejores prácticas en jerarquía de widgets y organización del código, aseguran claridad y eficiencia. Las pruebas, una fase crítica, abarcan pruebas unitarias, de widgets e integración para asegurar estabilidad y amigabilidad al usuario, con pruebas de rendimiento para optimizar la app en dispositivos. Finalmente, la fase de despliegue se beneficia de la capacidad de Flutter para compartir bases de código a través de plataformas, facilitado a través de pipelines CI/CD para lanzamientos eficientes en múltiples plataformas.

Recuerda, tus adaptaciones específicas dependerán del tamaño, complejidad y requisitos del proyecto. Elige las herramientas y prácticas que mejor se adapten a tu equipo de desarrollo y a los objetivos de la aplicación.

2.4 Ingeniería Flutter vs. Programación

Hasta este punto, hemos explorado varias facetas de la ingeniería de software, y en esta etapa, deberías haber obtenido una comprensión más completa de este tema. Sin embargo, me gustaría elaborar más y compartir mi perspectiva.

En el desarrollo de software, la "ingeniería Flutter" y la "programación" representan roles y responsabilidades distintos dentro de un proyecto. La programación implica principalmente escribir código para implementar funcionalidades específicas, centrándose en la implementación de código y la resolución de problemas. Los programadores son responsables de traducir el diseño y los requisitos en código ejecutable. En contraste, la ingeniería Flutter abarca un rol más integral. Los

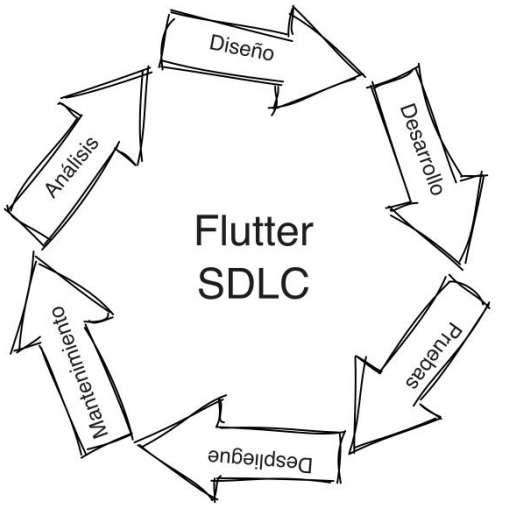

Figure 2.8: Ciclo de vida del desarrollo de software de Flutter

ingenieros de Flutter escriben código, diseñan la arquitectura del sistema y la interfaz de usuario, y toman decisiones estratégicas sobre la estructura y escalabilidad del proyecto. Se enfocan en la calidad del código, la gestión del proyecto y la innovación, desempeñando un papel fundamental en el proceso de desarrollo. Entender estas diferencias es esencial para gestionar efectivamente un proyecto Flutter y ensamblar el equipo adecuado para el éxito.

2.5 La Posición de Flutter en la Evolución Tecnológica

Al concluir este capítulo, me gustaría expresar mi perspectiva sobre la posición de Flutter en el paisaje tecnológico en constante evolución. Flutter ocupa una posición verdaderamente única y emocionante en el mundo tecnológico.

2.5.1 Enfoques Multiplataforma

Flutter emerge como un verdadero disruptor en la industria en una era donde el desarrollo multiplataforma ha ganado importancia primordial. Su capacidad para permitir a los desarrolladores crear aplicaciones de alta calidad y visualmente cautivadoras en diversas plataformas representa un cambio de paradigma. El enfoque del marco en la productividad, creatividad y eficiencia ha revolucionado nuestro enfoque hacia el desarrollo de aplicaciones, democratizando el acceso para desarrolladores y negocios de todas las escalas. La noción de que “donde haya un píxel, se puede encontrar Flutter” ha alterado fundamentalmente nuestras

percepciones sobre la construcción de software multiplataforma desde una base de código única.

En este contexto, Flutter es un habilitador tecnológico y un catalizador para el crecimiento de los desarrolladores. Anima a los desarrolladores a ampliar sus conocimientos y habilidades a través de varias plataformas, cada una con atributos únicos. Este enfoque mejora la calidad de las aplicaciones, elevando la experiencia del usuario y fomentando la evolución de los desarrolladores en profesionales experimentados dentro de su campo.

2.5.2 El Rol de Flutter en el Ecosistema Tecnológico Más Amplio

Además, el papel de Flutter en el ecosistema tecnológico más amplio tiene una importancia significativa. Simplifica las complejidades del desarrollo multiplataforma, catalizando la innovación al promover la creación de experiencias de usuario visualmente atractivas, receptivas y consistentemente excepcionales. A medida que profundizamos en el mundo del desarrollo de Flutter en los próximos capítulos, se hace evidente que Flutter no es meramente una herramienta; es una fuerza impulsora que empuja los límites de lo que es alcanzable en el desarrollo de software.

Los métodos innovadores de Flutter para el desarrollo multiplataforma permiten a los desarrolladores centrarse en crear experiencias de usuario en lugar de en tecnologías o plataformas específicas. Además, la vibrante comunidad de Flutter juega un papel fundamental en la configuración de la tecnología, fomentando una mayor demanda e innovación. Flutter es un modelo a seguir notable para sus pares y la industria tecnológica en general, ya que continúa evolucionando y dejando una marca duradera en el mundo del desarrollo de software.

Flutter es relevante en el presente y está preparado para dar forma al futuro.

2.6 Conclusión

Este capítulo ha explorado de manera integral los principios fundamentales y la filosofía única que impulsan el desarrollo de alta calidad en Flutter. Hemos profundizado en paradigmas críticos, incluyendo abstracción, encapsulación, patrones de diseño y consideraciones de eficiencia y escalabilidad. Se ha enfatizado el concepto de "shifting left" a través de la verificación y validación tempranas, estableciendo el escenario para la toma de decisiones informada a lo largo del ciclo de desarrollo.

Además, hemos subrayado la distinción entre programación e ingeniería, mostrando cómo Flutter promueve la modularidad, separación de preocupaciones y análisis reflexivo de compensaciones. También hemos situado a Flutter dentro del paisaje tecnológico más amplio, arrojando luz sobre sus fortalezas y el impacto potencial en comparación con otros enfoques multiplataforma.

Al concluir este capítulo, debemos hacernos la siguiente pregunta: £Cómo podemos aprovechar estos conocimientos fundamentales para crear aplicaciones Flutter

de alto rendimiento, mantenibles y estéticamente agradables? Esta pregunta será nuestra estrella guía mientras nos embarcamos en el emocionante viaje de aplicar estos principios en la práctica y crear experiencias Flutter excepcionales en los siguientes capítulos.

CHAPTER 27

Custom Painters y Shaders

Reviewer: Renan C. Araújo

El mundo de la pintura personalizada y los shaders en Flutter es vasto e ilimitado. Ofrece una flexibilidad notable que va más allá de los marcos convencionales. El propósito de este capítulo no es solo ayudarte a convertirte en un desarrollador profesional de Flutter que puede usar canvas y shaders para crear arte animado generativo, sino también proporcionar una comprensión integral de los aspectos de ingeniería de estas características. Exploraremos cuándo y cómo puedes utilizar estas poderosas herramientas en Flutter para mejorar tus aplicaciones mientras equilibras la capacidad técnica con la expresión creativa. Se requeriría un libro o más para cubrir a fondo estos temas, que podrían ser el foco de mi próximo libro. Alternativamente, por favor envíame un correo electrónico si conoces estas áreas y te gustaría colaborar conmigo en escribir sobre ellas.

Comencemos ahora.

27.1 El Arte de CustomPainter

`CustomPainter` es un lienzo para dibujar diseños personalizados en una aplicación Flutter. El widget `CustomPaint` en Flutter es una puerta de entrada para crear interfaces de usuario visualmente impresionantes y únicas. En su núcleo, `CustomPaint` es un widget que proporciona un lienzo para dibujar gráficos personalizados. Hace de puente entre el mundo de alto nivel de los widgets de Flutter y las operaciones de bajo nivel de dibujo y renderizado.

£Pero por qué y cuándo deberías usar `CustomPainter`? La clave reside en su flexibilidad y control. Es ideal para escenarios en los que debes crear gráficos personalizados complejos que no se pueden lograr con widgets estándar. Esto incluye escenarios como generar formas dinámicas, crear animaciones intrincadas o implementar elementos de UI personalizados que deben ser visualmente distintos e interactivos. `CustomPainter` brilla en aplicaciones que requieren un alto grado de personalización en la UI, como juegos, herramientas de visualización de datos, o cualquier aplicación que quiera destacar con una identidad visual única.

A veces, es posible que necesites usar `CustomPainter` para optimizar la UI de tu aplicación. Por ejemplo, si tienes una UI compleja que se puede crear usando widgets estándar, podría causar retrasos o consumir mucha energía y CPU. Usar APIs de bajo nivel en `CustomPainter` puede ayudarte a optimizar aún más tu aplicación, aunque esto puede conllevar un mayor grado de complejidad en la comprensión y escritura de código. Por lo tanto, si necesitas mejorar el rendimiento de tu aplicación, `CustomPainter` es una excelente opción.

En resumen, CustomPainter se trata de pintar puntos en la pantalla como se desee. CustomPainter se desempeña mejor que otros widgets en Flutter porque evita el complejo mecanismo de diseño de widgets que Flutter suele usar. Esto permite al autor controlar lo que hará el lienzo. Un concepto similar se puede encontrar en los fragment shaders, donde el autor evita el marco y el motor de Flutter. Pero recuerda, como dijo una vez Winston Churchill: "Donde hay un gran poder, hay una gran responsabilidad."

27.1.1 Widget CustomPaint

Para entender mejor CustomPaint en Flutter, revisemos su estructura fundamental y uso. La base se construye sobre una clase personalizada que extiende `CustomPainter`, como se muestra en el fragmento:

```
class DrawingPainter extends CustomPainter {
  const DrawingPainter();

  @override
  void paint(Canvas canvas, Size size) {}

  @override
  bool shouldRepaint(
    covariant CustomPainter oldDelegate,
  ) =>
      false;
}
```

Esta clase `DrawingPainter` proporciona acceso a un objeto `Canvas`, tu área de juego, para dibujos personalizados. El método `paint` es donde ocurre toda la magia. Aquí, puedes dibujar cualquier cosa, desde formas simples hasta gráficos complejos en el lienzo utilizando varios métodos de dibujo proporcionados por la API de `Canvas`. El parámetro size te da las dimensiones del área que tienes para dibujar.

El método `shouldRepaint`, el otro método, es crucial para optimizar el rendimiento del widget. Determina si el `CustomPainter` debería repintarse a sí mismo. Por ejemplo, devolver `false` significa que el lienzo no se repintará a menos que se le indique explícitamente, lo cual es beneficioso para gráficos estáticos.

Ahora, para usar este pintor personalizado, lo envuelves dentro de un widget `CustomPaint`, así:

```
CustomPaint(
  painter: const DrawingPainter(),
)
```

El widget `CustomPaint` integra tu dibujo personalizado (`DrawingPainter` en este caso) en el árbol de widgets de Flutter. Cuando usas `CustomPaint` y pasas tu `DrawingPainter`, Flutter sabe que debe llamar al método `paint` de `DrawingPainter` cada vez que necesita renderizar el widget.

27.1.2 Aplicación de dibujo

En este ejemplo, quiero asegurarme de que entiendas lo fácil que es usar Canvas, incluso si es tu primer uso. ąNo tengas miedo de intentarlo!

Paso 1: Define el Custom Painter (`DrawingPainter`)

```
class DrawingPainter se extiende de CustomPainter {
  List<Offset> points;

  DrawingPainter(this.points);

  @override
  void paint(Canvas canvas, Size size) {
    final pencil = Paint()
      ..color = Colors.black
      ..strokeWidth = 4
      ..isAntiAlias = true
      ..strokeCap = StrokeCap.round;

    for (int i = 0; i < points.length - 1; i++) {
      canvas.drawLine(
        points[i],
        points[i + 1],
        pencil,
      );
    }
  }

  @override
  bool shouldRepaint(
    DrawingPainter oldDelegate,
  ) => true;

  @override
  bool shouldRebuildSemantics(
    DrawingPainter oldDelegate,
  ) => false;
```

```
  @override
  bool? hitTest(Offset position) {
    return super.hitTest(position);
  }
}
```

`DrawingPainter` se extiende de `CustomPainter` y es responsable de renderizar el dibujo en el canvas. El Constructor toma una lista de puntos `Offset`, representando las posiciones donde el usuario ha tocado la pantalla.

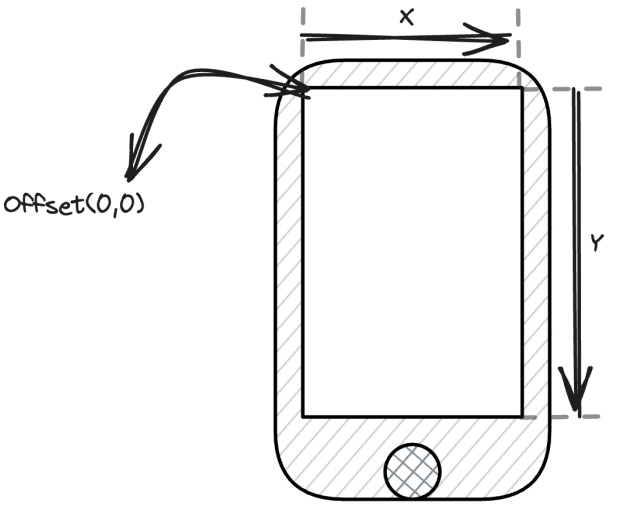

Figure 27.1: 25-1.png

Punto de inicio (0,0) Esquina superior izquierda de la pantalla

El método `paint` itera a través de los puntos, dibujando una línea entre cada punto consecutivo. El objeto `Paint` define la apariencia de las líneas (color, ancho de trazo, anti-aliasing y tapa de trazo); puedes pensar en ello como tu lápiz. Y finalmente, se utiliza el método `drawLine`, que dibuja una línea entre los puntos dados utilizando la pintura dada.

El método `shouldRepaint` devuelve `true`, asegurando que el canvas se repinte siempre que la lista de puntos se actualice, y los métodos `shouldRebuildSemantics` y `hitTest` están relacionados con la accesibilidad y las pruebas de golpeo, respectivamente. Estos funcionan de manera similar a lo que aprendiste en la Parte 1 del libro mientras construías `RenderObject` personalizados.

Paso 2: Define el Widget (`DrawingPage`)

```
class DrawingPage extends StatefulWidget {
  const DrawingPage({super.key});

  @override
```

```
  DrawingPageState createState() => DrawingPageState();
}

class DrawingPageState extends State<DrawingPage> {
  List<Offset> points = <Offset>[];

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('DrawingPage'),
      ),
      body: GestureDetector(
        onPanStart: (details) =>
            points.add(details.localPosition),
        onPanUpdate: (
          DragUpdateDetails details,
        ) {
          points.add(details.localPosition);
          setState(() {});
        },
        onPanEnd: (DragEndDetails details) {
          points.add(Offset.infinite);
        },
        child: SizedBox(
          width: double.infinity,
          height: double.infinity,
          child: CustomPaint(
            painter: DrawingPainter(points),
            child: Container(),
          ),
        ),
      ),
    );
  }
}
```

Esta clase de estado mantiene una lista de puntos `Offset`, actualizada cada vez que el usuario dibuja en la pantalla, que se da por los callbacks `onPan` a través de `GestureDetector`.

```
GestureDetector(
  onPanStart: (details) => points.add(details.localPosition),
  onPanUpdate: (DragUpdateDetails details) {
    points.add(details.localPosition);
    setState(() {});
  },
  onPanEnd: (DragEndDetails details) {
```

```
    points.add(Offset.infinite);
  },
  // ... [CustomPaint]
)
```

El `onPanStart` ****Listener agrega un nuevo punto a la lista cuando el usuario comienza a arrastrar. `onPanUpdate` ****Listener agrega continuamente puntos a la lista a medida que el usuario mueve su dedo y llama a `setState` para desencadenar una reconstrucción. `onPanEnd` ****Listener agrega un punto especial `Offset.infinite`, señalando el final de una trazo continuo.

La lista actual de puntos se pasa a `DrawingPainter`, quien dibuja las líneas en el lienzo.

Figure 27.2: 25-2.png

Simple Drawing App con Flutter

No está sucediendo nada demasiado sofisticado aquí, solo unas pocas líneas básicas de código. ąY voilà! Ahora tienes un lienzo que puede ser utilizado para pintar cualquier cosa. Te estoy mostrando cómo combinar algunos elementos básicos para crear un concepto interesante utilizando una poderosa API en Flutter. Ahora, vamos a explorar más prácticas avanzadas.

27.1.3 Optimizar

En el ejemplo proporcionado, hay varias mejoras clave y mejores prácticas que me gustaría destacar para una implementación más eficiente.

Primero, consideremos la clase `CustomPainter` en Flutter:

```
abstract class CustomPainter extends Listenable {
  const CustomPainter({Listenable? repaint})
      : _repaint = repaint;

  final Listenable? _repaint;

  // ...
}
```

En este fragmento, es notable que `CustomPainter` puede administrar automáticamente la repintura cuando se le proporciona un `Listenable`. Esta característica nos permite simplificar nuestra estructura de widgets al pasar de un `StatefulWidget` a un `StatelessWidget`, utilizando un `ValueNotifier`:

```
class DrawingPage extends StatelessWidget {
  DrawingPage({super.key});

  final pointsListenable = ValueNotifier<List<Offset>>([]);
  // ...
}
```

Dentro de este widget sin estado, los cambios en el dibujo pueden ser fácilmente administrados actualizando el `ValueNotifier` en respuesta a los gestos del usuario:

```
class DrawingPage extends StatelessWidget {
  //...
  body: GestureDetector(
    onPanStart: (details) => pointsListenable.value = [
      ...pointsListenable.value,
      details.localPosition
    ],
    onPanUpdate: (DragUpdateDetails details) {
      pointsListenable.value = [
        ...pointsListenable.value,
        details.localPosition
      ];
    },
    onPanEnd: (DragEndDetails details) {
      pointsListenable.value = [
        ...pointsListenable.value,
        Offset.infinite,
      ];
    },
    //...
  );
  // ... ;
}
```

Además, podemos optimizar el proceso de repintado pasando este notificador al pintor personalizado y envolviendo al pintor con `RepaintBoundary`.

Te sugiero que revises la documentación oficial y el código fuente para aprender sobre `RepaintBoundary`. Explicar los detalles requeriría muchas páginas. Sin embargo, puedo decirte que para mostrar un borde de color alrededor de cada widget; necesitas establecer la propiedad `debugRepaintRainbowEnabled` en `true`. Estos bordes cambiarán de color a medida que el usuario se desplace por la aplicación. Para establecer esta bandera, agrega `debugRepaintRainbowEnabled = true;` como una propiedad de nivel superior en tu aplicación. Si ves que los widgets estáticos rotan a través de colores después de establecer esta bandera, considera agregar límites de repintado a esas áreas.

```
main() {
  ****debugRepaintRainbowEnabled = true;//<---
  runApp(MyApp());
}
```

`RepaintBoundary` asegura que el repintado esté contenido y no afecte otras partes del árbol de widgets de Flutter innecesariamente:

```
child: RepaintBoundary(
          child: CustomPaint(
            painter: DrawingPainter(pointsListenable),
            child: const SizedBox.expand(),
          ),
        ),
```

Además, la implementación del `CustomPainter` en sí es crucial. Así es cómo se puede adaptar para aprovechar el `ValueNotifier`:

```
class DrawingPainter se extiende de CustomPainter {
  ValueNotifier<List<Offset>> puntos;

  DrawingPainter(this.puntos) : super(repaint: puntos);
  // ...
  @override
  void paint(Canvas canvas, Size size) {
    for (int i = 0; i < puntos.value.length - 1; i++) {
      canvas.drawLine(
        puntos.value[i],
        puntos.value[i + 1],
        pencil,
      );
    }
  }

  @override
```

```
  bool shouldRepaint(DrawingPainter oldDelegate) =>
      oldDelegate.puntos != puntos;
  // ...
}
```

Esta modificación mejora enormemente el rendimiento, especialmente en aplicaciones de mayor escala. Es una técnica de optimización importante para una experiencia de desarrollo de Flutter más eficiente y efectiva.

Este es el momento perfecto para presentarte la siguiente sección sobre las mejores prácticas.

27.1.4 Mejores prácticas

CustomPainter ofrece una inmensa flexibilidad para crear visuales personalizados en Flutter, pero dominar su eficiencia y mantenibilidad requiere práctica reflexiva. Profundicemos en las mejores prácticas esenciales con ejemplos de código para una comprensión más profunda:

Minimizar reconstrucciones con `shouldRepaint`:

Imagina que tu CustomPainter dibuja un gráfico dinámico. Cada ajuste de datos desencadena una re-dibujado completo, impactando el rendimiento. ą`shouldRepaint` al rescate!

```
class ChartPainter extends CustomPainter {
  final List<double> data;

  ChartPainter(this.data);

  @override
  bool shouldRepaint(ChartPainter oldDelegate) {
    // Compare only data changes
    return !listEquals(
      data,
      oldDelegate.data,
    );
  }

  @override
  void paint(Canvas canvas, Size size) {
    // ... draw chart based on data
  }
}
```

Con este código, el gráfico se repinta solo cuando los datos cambian, no por pequeños ajustes de UI, mejorando el rendimiento.

Cache con `PictureRecorder` para animaciones suaves:

Piensa en una escena animada compleja con elementos de fondo estáticos. Repintarlos en cada fotograma es redundante. Entra `PictureRecorder`:

```
class AnimatedPainter extends CustomPainter {
  PictureRecorder recorder = PictureRecorder();
  Picture? picture;

  // ... otros métodos

  @override
  void paint(Canvas canvas, Size size) {
    if (picture == null) {
      return;
    }
    canvas.drawPicture(picture!);

    // ... dibuja elementos de animación dinámicos en la parte superior
  }

  @override
  bool shouldRepaint(covariant CustomPainter oldDelegate) {
    // ...
  }
}
```

Aquí, el fondo estático se graba una vez y se reutiliza repetidamente, haciendo que las animaciones sean más suaves y menos intensivas en recursos.

Separa la Lógica con Mixins para Reutilizar Código:

A veces, tu lógica de pintura se vuelve compleja. Mezclarla con la gestión de widgets puede ser complicado. ąLos mixins al rescate!

```
mixin ShapePainterMixin on CustomPainter {
  @override
  void paint(Canvas canvas, Size size) {
    // ... lógica de pintura común para dibujar formas
  }
}

class MyPainter extends CustomPainter
    with ShapePainterMixin {
  // ... otras propiedades y métodos
}
```

Este mixin encapsula la lógica de dibujo de formas reutilizable, manteniendo tu clase `MyPainter` enfocada en detalles específicos y facilitando la reutilización de código en otros pintores.

Modulariza con Pintores Reutilizables para Grandes Composiciones:

Las visuales grandes y complejas se benefician de un enfoque de dividir y conquistar. Ingresan los pintores modulares:

```
class ChartPainter extends CustomPainter {
  // ... pinta el gráfico
}

class GaugePainter extends CustomPainter {
  // ... pinta los medidores
}

class LabelPainter extends CustomPainter {
  // ... pinta las etiquetas de texto
}

class DashboardPainter extends CustomPainter {
  @override
  void paint(Canvas canvas, Size size) {
    ChartPainter().paint(canvas, size);
    GaugePainter().paint(canvas, size);
    LabelPainter().paint(canvas, size);
  }
}
```

Aquí, los pintores individuales manejan elementos específicos; el `DashboardPainter` los combina sin problemas para obtener una imagen completa. Esto promueve la modularidad y simplifica el mantenimiento.

Mejora la Accesibilidad con Semántica:

La accesibilidad garantiza que todos puedan usar tu aplicación. La semántica ayuda a los lectores de pantalla a entender tus visuales personalizados:

```
class ChartPainter extends CustomPainter {
  // ...

  @override
  void paint(Canvas canvas, Size size) {
    // ... dibuja elementos del gráfico

    for (int i = 0; i < data.length; i++) {
      final rect = Rect.fromLTWH(
        i * 10,
        0,
        10,
        data[i] * 50,
      ); // Ejemplo de rect para punto de datos
```

```
      CustomPainterSemantics( //<---
        rect: rect,
        properties: SemanticsProperties( //<---
          label: 'Data point ${i + 1}',
          value: '${data[i]}',
        ),
      ).paint(canvas, size);
    }
  }
}
```

Con la semántica, los lectores de pantalla pueden interpretar puntos de datos y valores, haciendo que tus gráficos personalizados sean accesibles para todos.

Reutiliza Objetos Paint:

Instancia los objetos `Paint` fuera del método `paint` y reutilízalos. Esta práctica conserva recursos ya que crear una nueva instancia de `Paint` en cada repintado puede ser costoso.

```
final paint = Paint()
  ..color = Colors.blue
  ..style = PaintingStyle.stroke;

void paint(Canvas canvas, Size size) {
  // usa el objeto paint existente
  canvas.drawLine(
    Offset.zero,
    Offset(size.width, size.height),
    paint,
  );
}
```

Manejo de Pantallas de Alta Resolución (High-DPI):

Asegúrate de que tu código de pintura personalizada se escala correctamente en pantallas de alta resolución para una experiencia visual consistente en todos los dispositivos.

```
final pixelRatio = MediaQuery.of(context).devicePixelRatio;

void paint(Canvas canvas, Size size) {
  // Escala tu dibujo basado en pixelRatio
}
```

Perfilado y Optimización:

Perfila regularmente tu código de pintor personalizado, especialmente cuando se trata de dibujos o animaciones complejas, para identificar y optimizar los cuellos de botella de rendimiento.

Utilizando RepaintBoundary para la Optimización del Rendimiento:

`RepaintBoundary` es un widget que aísla a su hijo del resto del árbol de widgets en términos de pintura. Esto puede mejorar significativamente el rendimiento, especialmente para los widgets que son costosos de pintar y solo cambian a veces. Al usar `RepaintBoundary`, le dices a Flutter que maneje la pintura de este widget por separado, reduciendo el costo total de repintado.

```
RepaintBoundary(
  child: CustomPaint(
    painter: ExpensivePainter(),
    isComplex: true,
    willChange: false,
  ),
)
```

Este código envuelve a `ExpensivePainter **()**` en un `RepaintBoundary`, optimizando su rendimiento de renderizado. Las propiedades `isComplex` y `willChange` informan aún más al sistema de renderizado sobre la naturaleza de la operación de pintura, permitiendo un manejo más eficiente.

Vértices - Renderizado avanzado de formas con alto rendimiento:

En la programación de gráficos, los vértices son la piedra angular del renderizado de formas y modelos. Un vértice es un punto en el espacio 2D o 3D, definido por coordenadas, y sirve como la unidad fundamental para construir formas geométricas más complejas. Los vértices son cruciales en la definición de los contornos de los polígonos, particularmente los triángulos, que son los bloques de construcción básicos para la mayoría de los objetos gráficos en entornos bidimensionales y tridimensionales. Estos puntos también pueden llevar atributos adicionales como color, coordenadas de textura y normales, esenciales para crear gráficos detallados y visualmente ricos.

En Flutter, la clase `Vertices` es crucial para el renderizado personalizado de formas, especialmente cuando se trata de diseños intrincados o la necesidad de gráficos de alto rendimiento. Esta clase permite a los desarrolladores definir una serie de puntos y cómo deben estar conectados y coloreados. Cuando se usa con el método `Canvas.drawVertices`, permite el dibujo de formas complejas que no se pueden lograr con los widgets estándar.

Las formas dibujadas usando `Vertices` se basan principalmente en triángulos, el polígono más simple en la programación de gráficos, y se pueden combinar para formar cualquier forma compleja. La forma en que se forman y se renderizan estos triángulos depende del modo especificado:

1. **Triángulos:** Cada conjunto de tres vértices forma un triángulo independiente.
2. **Tira de triángulos (Triangle Strip):** Los vértices están conectados en una forma de tira, donde cada nuevo vértice forma un triángulo con los dos anteriores.

3. **Ventilador de triángulos (Triangle Fan):** Todos los triángulos comparten el primer vértice, abanicándose desde este punto común.

La clase `Vertices` se puede instanciar usando su constructor predeterminado o el constructor `Vertices.raw` para un control más directo sobre los datos. Aquí hay una descripción general básica de cómo usar cada uno:

```
Vertices(
  VertexMode mode,
  List<Offset> positions, {
    List<Color>? colors,
    List<Offset>? textureCoordinates,
    List<int>? indices,
})
```

Este constructor es más sencillo y utiliza listas de objetos `Offset` y `Color`, lo que lo hace fácil de usar pero ligeramente menos eficiente.

```
Vertices.raw(
  VertexMode mode,
  Float32List positions, {
    Int32List? colors,
    Float32List? textureCoordinates,
    Uint16List? indices,
})
```

`Vertices.raw` es un constructor más orientado al rendimiento que utiliza matrices de datos tipados como `Float32List` y `Uint16List`. Este enfoque está más cerca del formato de datos de bajo nivel que utiliza el motor de renderizado. Es ideal para aplicaciones críticas para el rendimiento o cuando se trabaja con datos ya en un formato bruto.

Un ejemplo en el código fuente proporcionado en este libro te permite navegar a `/lib/custompainters/snowfall.dart`. Una vez que ejecutes la aplicación, puedes activar y desactivar la implementación de `Vertices.raw` y volver al dibujo estándar de canvas para observar la diferencia en el rendimiento.

27.2 Explorando Shaders

Los Shaders son programas pequeños pero potentes que se ejecutan en la unidad de procesamiento gráfico (GPU), responsables de renderizar los detalles intrincados de luz y color que dan vida a las imágenes digitales.

Los Shaders operan dentro de la pipeline gráfica, una secuencia de pasos que un sistema de gráficos por computadora usa para renderizar objetos 3D en una pantalla 2D. Esta pipeline incluye etapas como el procesamiento de vértices, la rasterización y el procesamiento de fragmentos, con los Shaders desempeñando un

papel vital en varios puntos. Hay varios tipos de Shaders, cada uno con su función única:

1. **Vertex Shaders**: Estos procesan cada vértice de un modelo 3D, transformando las coordenadas 3D en coordenadas de pantalla 2D y pasando datos por vértice a lo largo de la pipeline. Son esenciales para manipular posiciones de vértices y crear efectos como animaciones.
2. **Fragment (Pixel) Shaders**: Estos calculan el color final de cada píxel, teniendo en cuenta luces, sombras y texturas. Son clave para lograr efectos de superficie detallados y renderizado realista de materiales.
3. **Geometry Shaders**: Operando entre los Vertex y Fragment Shaders, pueden generar nuevos vértices y formas, agregando complejidad y detalle a los objetos.
4. **Tessellation Shaders**: Utilizados para ajustar el nivel de detalle de los modelos 3D, estos Shaders mejoran la eficiencia y la calidad visual al adaptarse a la distancia de la cámara.
5. **Compute Shaders**: Estos manejan tareas de computación de propósito general dentro de la GPU, como simulaciones de física o efectos de post-procesamiento, separados del proceso directo de renderizado de imágenes.

La pipeline gráfica es un marco conceptual que describe los pasos para renderizar un objeto 3D en una pantalla 2D. Comienza con el procesamiento de las coordenadas 3D (procesamiento de vértices), luego convierte el objeto en píxeles (rasterización) y finalmente colorea estos píxeles (procesamiento de fragmentos). Los Shaders son integrales para este proceso, proporcionando la flexibilidad para crear efectos visuales complejos.

27.2.1 Entendiendo el lenguaje Shader (GLSL)

OpenGL Shading Language (GLSL) es un lenguaje de sombreado de alto nivel ampliamente utilizado en gráficos por computadora para escribir Shaders personalizados. Los Shaders son pequeños programas que dictan cómo renderizar gráficamente cada píxel, vértice o geometría en la pantalla. Se ejecutan directamente en la unidad de procesamiento gráfico (GPU), lo que los hace increíblemente eficientes para los cálculos gráficos.

GLSL es una parte integral de la API de gráficos OpenGL, una especificación estándar que define una API cruzada de lenguaje, plataforma cruzada para renderizar gráficos vectoriales 2D y 3D. El lenguaje se asemeja mucho a C y, en esencia, a la sintaxis similar a Dart, lo que lo hace familiar para los desarrolladores de Flutter.

Los tipos principales de Shaders en GLSL incluyen **vertex shaders**, que procesan datos de vértices, y **fragment shaders**, que determinan el color y otros atributos de cada píxel. Otros tipos, como los Shaders de geometría y teselación, ofrecen un control adicional sobre el renderizado. Me centraré en los Fragment Shaders ya que Flutter solo los admite.

Un Fragment Shader típico, que generalmente tiene extensión `.frag` o `.glsl` en GLSL, incluye:

Declaración de Versión: Especificando la versión de GLSL. Esto es opcional.

Variable de Salida: Una variable para almacenar la salida de color para el píxel.

```
out vec4 fragColor;
```

`vec4` es el tipo de variable de salida más comúnmente utilizado en los shaders de fragmentos. Representa un vector de cuatro componentes correspondiente a los componentes RGBA (Rojo, Verde, Azul, Alfa) del color. A veces, puedes encontrar `vec3` si el componente alfa no es necesario o se maneja por separado. Representa un vector de tres componentes para los componentes RGB del color. En casos donde solo se necesita un canal de color o una salida en escala de grises, se puede usar un `float`.

Función Principal: Donde ocurre el cálculo del color.

```
void main() {
    fragColor = vec4(1.0, 0.0, 0.0, 1.0);  // Color rojo
}
```

Aquí, `fragColor` se establece en un color rojo estático para cada píxel.

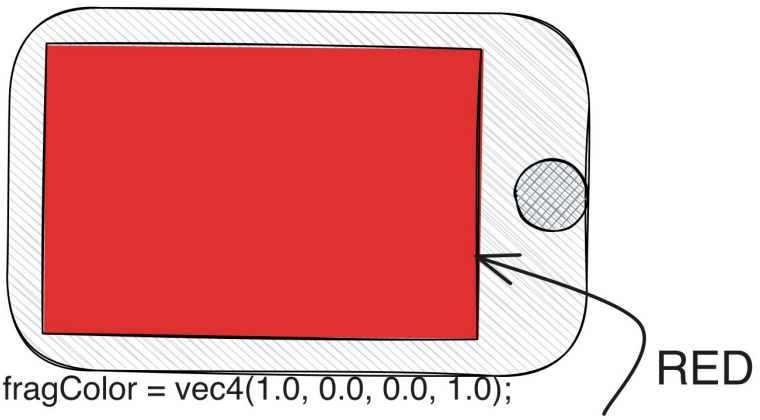

Figure 27.3: 25-3.svg

Demostrar Rojo para Cada Píxel

Incorporar Variables Uniformes: En GLSL, las variables uniformes son un tipo de variable que puedes usar para pasar datos desde tu aplicación principal (que se ejecuta en la CPU) a tu programa de shader (que se ejecuta en la GPU). Los uniformes son globales y permanecen constantes para todos los vértices y fragmentos procesados durante una sola llamada de dibujo. Son una forma clave

de hacer que tus shaders sean dinámicos y respondan a lo que está sucediendo en tu aplicación.

Por ejemplo, puedes pasar el tiempo transcurrido para crear animaciones.

```
uniform float u_time;
void main() {
        // La función **sin** crea un patrón similar a una onda,
        // y **abs** asegura que el valor sea positivo.
    float red = abs(sin(u_time));
    fragColor = vec4(red, 0.0, 0.0, 1.0);
}
```

El shader de fragmentos producirá un color rojo pulsante en el objeto renderizado, con la intensidad del rojo cambiando con el tiempo en un patrón sinusoidal[1].

Manipular Coordenadas: Utilice las coordenadas de cada pixel (`gl_FragCoord`) para crear gradientes o patrones. `gl_FragCoord` proporciona las coordenadas del fragmento actual (o pixel).

En el siguiente ejemplo, estas coordenadas se dividen por `u_resolution`, una variable uniforme pasada al shader que representa la resolución de la ventana de renderizado o textura. El resultado es una coordenada normalizada `st` (con ambos valores x e y que varían de 0 a 1) a través de la superficie renderizada.

```
uniform vec2 u_resolution;
out vec4 fragColor;

void main()
{
    vec2 st = gl_FragCoord.xy / u_resolution;
    fragColor = vec4(st.x, st.y, 0.0, 1.0);
}
```

Este shader producirá un efecto de gradiente[2] a través de la superficie renderizada, haciendo una transición suave en color basado en la posición del pixel. El gradiente se mezclará desde el negro en la esquina inferior izquierda hasta el rojo y verde en la esquina superior derecha.

Manipular Coordenadas para Crear Efecto de Gradiente

Efectos Complejos de Iluminación y Color: Aquí, estamos calculando el componente difuso de la iluminación basado en una dirección de luz y una normal de superficie.

[1] https://www.shadertoy.com/view/XclSWr
[2] https://shadertoy.com/view/4csSWr

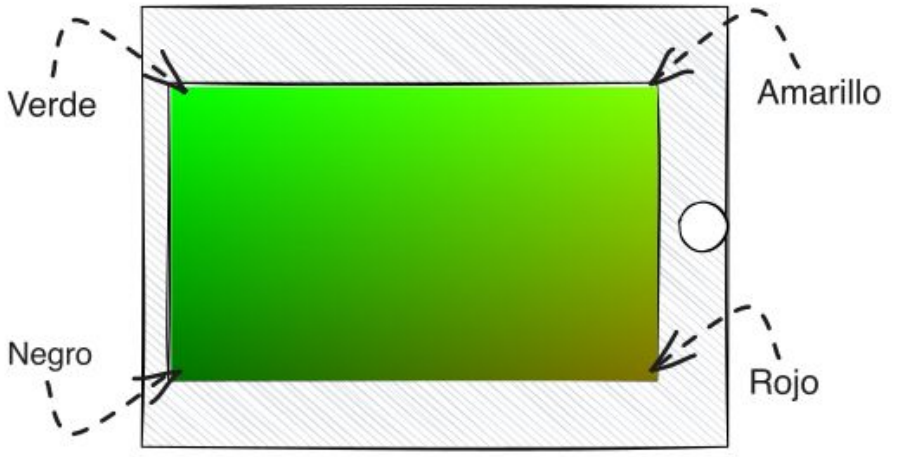

Figure 27.4: 25-4.svg

```
uniform vec3 u_lightDirection;
uniform vec3 u_normal;
void main() {
    float diff = max(dot(u_normal, u_lightDirection), 0.0);
    // Efecto de iluminación difusa
    vec3 diffuse = diff * vec3(1.0, 0.5, 0.3);
    fragColor = vec4(diffuse, 1.0);
}
```

Entonces, como puedes ver, también hay funciones de ayuda disponibles. También puedes crear una función para definir variables `const`.

Muestreo de texturas: En los shaders, las texturas son imágenes que agregan detalles de superficie como color, patrones o protuberancias a los modelos 3D.

En el ejemplo, un uniforme `sampler2D` en un shader se utiliza para pasar una textura 2D desde su aplicación al shader. El término "sampler" se refiere a la funcionalidad en un shader que le permite leer o muestrear datos de una textura. Luego, el shader muestrea colores de esta textura para aplicarlos a la superficie renderizada.

```
uniform sampler2D u_texture;
void main() {
    vec4 texColor = texture(u_texture, gl_FragCoord.xy);
    fragColor = texColor;
}
```

También existen otros tipos de sampler para diferentes tipos de texturas, como `sampler3D` para texturas 3D y `samplerCube` para texturas de mapa de cubo, cada uno utilizado para técnicas de renderizado específicas.

Esto debería darte una visión general de cómo leer el código GLSL del shader. Sin embargo, apenas hemos tocado la superficie, y lo mejor es leer algunos ejemplos de shaders, particularmente fragment shaders, para familiarizarte más con el concepto.

27.3 Uso de Shaders en Flutter

Ahora que entiendes qué son los shaders y cómo juegan un papel en la programación gráfica de computadoras, exploremos cómo Flutter los aprovecha. Flutter incorpora Fragment shaders para efectos visuales mejorados. Como ingeniero de Flutter, entender cómo integrar y usar shaders puede elevar significativamente el atractivo visual de tus aplicaciones.

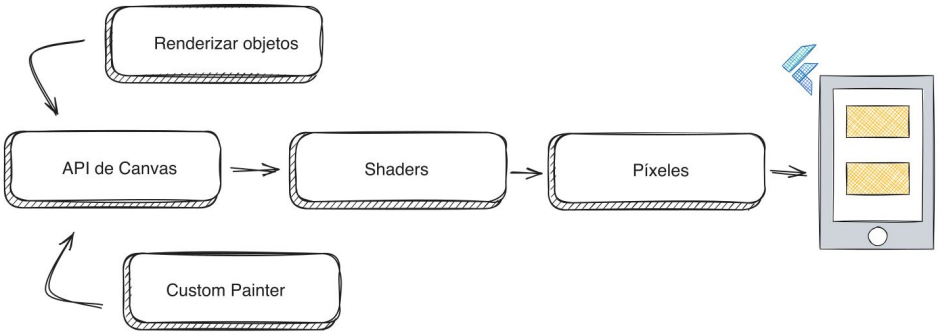

Figure 27.5: 25-5.svg

Widgets to Pixels Simple Pipeline in Flutter

27.3.1 Agregar un Fragment Shader a Flutter

En Flutter, principalmente trabajas con fragment shaders. Aquí te mostramos cómo puedes agregar un archivo de shader `.frag` a tu proyecto de Flutter:

Creando el archivo Shader (`simple.frag`): Crea un archivo de fragment shader con tu código GLSL. Aquí tienes un shader de ejemplo que crea un gradiente utilizando los colores de marca de Flutter:

```
#version 460 core

#include <flutter/runtime_effect.glsl>

uniform vec2 u_surfaceSize;
```

```
out vec4 fragColor;

vec3 flutterBlue = vec3(5, 83, 177) / 255;
vec3 flutterNavy = vec3(4, 43, 89) / 255;
vec3 flutterSky = vec3(2, 125, 253) / 255;

void main() {
  vec2 st = FlutterFragCoord().xy / u_surfaceSize.xy;

  vec3 color = vec3(0.0);
  vec3 percent = vec3((st.x + st.y) / 2);

    color = mix(
    mix(flutterSky, flutterBlue, percent * 2),
    mix(flutterBlue, flutterNavy, percent * 2 - 1),
    step(0.5, percent));

  fragColor = vec4(color, 1);
}
```

- Cualquier versión de GLSL desde 460 hasta 100 es compatible con Flutter, aunque algunas características disponibles están restringidas.
- `#include <flutter/runtime_effect.glsl>`: Esta línea incluye declaraciones para usar características específicas de Flutter en tu shader.
- `FlutterFragCoord()`: Una función específica de Flutter que proporciona las coordenadas del fragmento. A diferencia de `gl_FragCoord` en GLSL tradicional, `FlutterFragCoord` se ajusta para el sistema de coordenadas de Flutter.
- Este Shader necesita dos `floats` que definen el tamaño de la superficie, que podemos pasar desde Flutter.

Añadiendo Shader a `pubspec.yaml`: Incluye tu archivo de shader en el archivo `pubspec.yaml` de tu proyecto Flutter:

```
flutter:
  shaders:
    - shaders/simple.frag
```

Cargando Shaders en tiempo de ejecución

Una forma de usar shaders en Flutter es cargándolos en tiempo de ejecución:

```
void loadMyShader() async {
  final program = await FragmentProgram.fromAsset(
    'shaders/snow.glsl',
  );
```

```
  final program2 = await FragmentProgram.fromAsset(
    'shaders/simple.frag',
  );
}
```

- `FragmentProgram.fromAsset`: Carga el shader desde los activos.
- Puedes ver tanto las extensiones `.frag` como `.glsl` para los shaders de fragmento.

Usando Fragment Shaders con APIs de Canvas

En Flutter, los shaders de fragmento se pueden usar con las APIs de Canvas estableciendo la propiedad `Paint.shader`:

```
void paint(Canvas canvas, Size size) {
  shader.setFloat(0, size.width);
  shader.setFloat(1, size.height);
 **** final pencil = Paint()..shader = shader;//<---
****
    canvas.drawRect(
    Rect.fromLTWH(0, 0, size.width, size.height),
    paint,
  );
}
```

- En este caso, simplemente estamos concatenando el shader con el `fragment shader` que debe pasarse a la clase de pintor personalizado.
- Los shaders se pueden aplicar a varias formas y rutas dibujadas en el lienzo, ofreciendo posibilidades versátiles para gráficos personalizados. ąLos shaders, en combinación con los modos de mezcla, también pueden tener efectos!
- Por último, pero no menos importante, dado que tenemos que definir el tamaño de la superficie a flotar, ahora podemos definirlo como el máximo con altura: el `setFloat` establece el uniforme flotante en [index] a [value].

Ahora, podemos usar todo junto en nuestra aplicación Flutter.

```
glsl
void main() async {
  // 1
  final fragmentProgram = await FragmentProgram.fromAsset(
    'shaders/simple.frag',
  );
  // 2
  runApp(MyApp(shader: fragmentProgram.fragmentShader()));
}
```

```
class MyApp extends StatelessWidget {
  const MyApp({super.key, required this.shader});

  // 3
  final FragmentShader shader;

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Demo Simple de Shader',
      theme: ThemeData(
        colorSchemeSeed: Colors.blue,
        useMaterial3: true,
      ),
      home: MyHomePage(shader: shader),
    );
  }
}

class MyHomePage extends StatelessWidget {
  const MyHomePage({
    super.key,
    required this.shader,
  });

  final FragmentShader shader;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Demo Simple de Shader'),
      ),
      // 4
      body: CustomPaint(
        size: MediaQuery.sizeOf(context),
        // 5
        painter: MyShaderPainter(shader: shader),
      ),
    );
  }
}

// 5
class MyShaderPainter extends CustomPainter {
  MyShaderPainter({required this.shader});
  // 6
```

```
  final FragmentShader shader;

  @override
  void paint(Canvas canvas, Size size) {
    // 7
    shader.setFloat(0, size.width);
    shader.setFloat(1, size.height);

    // 8
    final paint = Paint()..shader = shader;

    // 9
    canvas.drawRect(
      Rect.fromLTWH(0, 0, size.width, size.height),
      paint,
    );
  }

  @override
    bool shouldRepaint(
    covariant CustomPainter oldDelegate,
  ) =>
      false;
}

Desglosemos lo que hace cada sección numerada en el código:

1. **Inicialización de Shader**:
    - Carga el fragment shader del activo `'shaders/simple.frag'`.
    - `FragmentProgram.fromAsset` carga de forma asíncrona el programa shader.
    - `fragmentProgram.fragmentShader()` crea una instancia de `FragmentShader`
2. **Inicio de la aplicación**:
    - `runApp` inicializa y ejecuta la aplicación Flutter.
    - Se crea el widget `MyApp` con el `shader` pasado como parámetro.
3. **Propagación de Shader en MyApp**:
    - La clase `MyApp` toma una instancia de `FragmentShader` como parámetro.
    - Este shader se pasa a `MyHomePage`.
4. **Widget CustomPaint en MyHomePage**:
    - Se utiliza el widget `CustomPaint` para proporcionar un lienzo en el que
    - `MediaQuery.sizeOf(context)` obtiene el tamaño actual de los medios (pant
5. **Pintor para CustomPaint**:
    - `ShaderPainter` se establece como el pintor para `CustomPaint`.
    - Este pintor personalizado utiliza el shader proporcionado para dibujar.
6. **Shader en ShaderPainter**:
    - `ShaderPainter` toma una instancia de `FragmentShader`.
    - Este shader se utilizará para pintar.
7. **Estableciendo Uniformes de Shader**:
    - Se establecen los uniformes del shader, que en este caso son el ancho y l
```

```
    - Estos valores se proporcionan al shader para controlar su comportamiento en func
8. **Creando Paint con Shader**:
    - Se crea un objeto `Paint`, y su shader se establece en el `FragmentShader` propo
    - Esta pintura se utilizará para dibujar con los efectos del shader.
9. **Dibujando en Canvas**:
    - Se dibuja un rectángulo que cubre todo el lienzo.
    - Se utiliza el `Paint` con el shader aplicado, por lo que el efecto del shader se

Este proceso puede ser mucho más simplificado utilizando `flutter_shaders`. Vamos a re
```

```
glsl
void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      home: MyHomePage(),
    );
  }
}

class MyHomePage extends StatelessWidget {
  const MyHomePage({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      // 1
      body: ShaderBuilder( //<---
        assetKey: 'shaders/simple.frag', //<---
        (context, shader, child) {
          return CustomPaint( //<---
            size: MediaQuery.sizeOf(context),
            // 2
            painter: ShaderPainter(shader: shader),
          );
        },
      ),
    );
  }
}
```

Ahora hemos eliminado algunos pasos y simplemente estamos usando el widget constructor `ShaderBuilder` para cargar el shader para nosotros. Puedes usar

otras características del paquete, como la extensión `SetUniforms` (mira en flutter_shaders/lib/src/set_uniforms.dart), donde técnicamente, puedes configurar tus uniformes mucho más fácil y como por arte de magia.

27.3.2 Convirtiendo desde ShaderToy

Convertir los shaders de ShaderToy a Flutter implica varios pasos para adaptar el código al entorno de Flutter. Por ejemplo, consideremos convertir un simple shader de efecto láser de ShaderToy, que se puede encontrar en este enlace de ShaderToy[3].

```
//convertir HSV a RGB
vec3 hsv2rgb(vec3 c)
{
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(
      K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y
    );
}

void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 fragPos = fragCoord / iResolution.xy;
    fragPos.y -= 0.5f;

    vec3 color = hsv2rgb(
      vec3(fragPos.x * 0.5 - iTime * 3.0, 1.0, 1.0)
    );
    color *= (0.015 / abs(fragPos.y));

    color += dot(color, vec3(0.299, 0.587, 0.114));

    fragColor = vec4(color, 1.0);
}
```

Los shaders de ShaderToy utilizan una función `mainImage`, que difiere de la función `main` estándar utilizada en los shaders de Flutter. Además, ShaderToy proporciona automáticamente ciertos uniformes como `iResolution` e `iTime`, que debes declarar manualmente en Flutter.

Diferencias clave y ajustes

1. **Punto de entrada**:

[3] https://www.shadertoy.com/view/4f2GRR

- ShaderToy utiliza la función `mainImage`, mientras que Flutter utiliza la función `main` estándar.

2. **Coordenadas de fragmento**:
   - Los shaders de ShaderToy utilizan `fragCoord`, pero en Flutter, debes usar `FlutterFragCoord()`.
3. **Uniformes**:
   - ShaderToy proporciona los uniformes `iResolution` e `iTime` automáticamente. En Flutter, estos deben ser declarados explícitamente y pasados desde el código Dart.
4. **Inclusión de Shader**:
   - La directiva `#include <flutter/runtime_effect.glsl>` es específica de Flutter y necesaria para usar ciertas características específicas de Flutter.
5. **Variable de salida**:
   - El color de salida se almacena en `fragColor` tanto en ShaderToy como en Flutter.

```
#include <flutter/runtime_effect.glsl>

uniform vec2 iResolution;
uniform float iTime;
out vec4 fragColor;

vec3 hsv2rgb(vec3 c) {
    // ...
}

void main() {
    vec2 fragPos = FlutterFragCoord().xy / iResolution.xy;
    // ...
}
```

27.3.3 Configuración de uniformes en Flutter

En tu aplicación Flutter, configura los uniformes para el shader usando `shader.setFloat`:

```
// En tu clase de pintor personalizado
void paint(Canvas canvas, Size size) {
  shader.setFloat(0, size.width);
  shader.setFloat(1, size.height);
  shader.setFloat(2, time);
```

```
  final paint = Paint()..shader = shader;

  canvas.drawRect(
    Rect.fromLTWH(0, 0, size.width, size.height),
    paint,
  );
}
```

Aquí, `size.width` y `size.height` corresponden a `iResolution`, y `time` corresponde a `iTime`. Estos valores deben pasarse al shader para imitar el comportamiento de los uniformes incorporados de ShaderToy.

Este ejemplo ilustra una conversión de shader sencilla. Sin embargo, hay escenarios más complejos donde se requieren pasos adicionales. Por ejemplo, al tratar con muestras de textura 2D, deberías usar un `AnimatedSampler` del paquete `flutter_shaders` en Flutter y llamar a `setImageSampler()` en el objeto `shader`.

```
// En Flutter
shader.setImageSample(0, image); // pasa dart:ui Image al shader
// en GLSL
uniform sampler2D uTexture;
```

Para aquellos ansiosos por profundizar en los shaders y el lenguaje GLSL, un gran recurso es The Book of Shaders[4]. Este libro en línea ofrece una experiencia de aprendizaje extensa e interactiva, perfecta para cualquiera que busque expandir sus conocimientos.

27.4 Conclusión

Explorar pinturas personalizadas y shaders en Flutter abre un fascinante dominio donde la creatividad se encuentra con la tecnología. Estas potentes APIs mejoran el atractivo visual de tus aplicaciones y ofrecen un patio de recreo atractivo y adictivo para los desarrolladores. La capacidad de transformar el código en visuales cautivadores no es solo una habilidad técnica sino una expresión artística, haciendo que el proceso de desarrollo con Flutter sea un viaje emocionante y gratificante.

Te animo a aprovechar esta oportunidad para experimentar e innovar. Cada incursión en la pintura personalizada y los shaders es un paso hacia el dominio de estas herramientas, empujando los límites de lo que es posible en el diseño de aplicaciones. Así que, sumérgete, juega y deja que tu creatividad florezca. Cuanto

[4] http://thebookofshaders.com/

más explores y crees, más aprenderás, lo que llevará a aplicaciones que no solo son funcionales sino también visualmente atractivas obras maestras.

Palabras Finales

Entonces, has llegado al final de este libro. Al pasar la última página, no es solo la conclusión de un capítulo en tu viaje de aprendizaje sino una puerta a nuevas posibilidades. A lo largo de este libro, hemos navegado a través de conceptos complejos, temas, código detallado y soluciones. Estas páginas han enriquecido tu comprensión y han encendido una chispa de curiosidad y entusiasmo por el mundo en constante evolución de Flutter.

Flutter es un viaje continuo de aprendizaje y descubrimiento. Los conceptos y técnicas que has encontrado aquí son herramientas que pueden empoderarte para construir, crear e innovar. Ya seas un estudiante, un desarrollador profesional o un aficionado, el conocimiento que has adquirido es una base sólida sobre la cual puedes construir cosas increíbles.

Al salir de este libro, recuerda que cada línea de código que escribes refleja tu creatividad y habilidades para resolver problemas. Los desafíos que enfrentarás en el mundo real pueden ser complejos, pero las habilidades que has perfeccionado aquí serán tus aliados. Sigue experimentando, sigue aprendiendo y, lo más importante, sigue codificando. La práctica hace la perfección.

Espero que este libro sirva tanto de guía como de paso hacia tu éxito con Flutter. Que tu futuro código sea eficiente y elegante y refleje tu potencial.

Gracias por elegir este libro como tu compañero en tu viaje con Flutter. Aquí tienes muchas más líneas de código, avances e innovaciones en tus futuros esfuerzos.

No dudes en contactarme a través de las redes sociales, correo electrónico y mi boletín de noticias flutterengineering.io[5]. Estoy en una misión para escribir y lanzar más contenido. Apreciaría tus pensamientos, comentarios y éxitos en tu carrera y proyectos.

Recuerda: EL DESARROLLO DE FLUTTER ES DIVERTIDO.

Con los mejores deseos para tu éxito continuo.

Majid Hajian

[5] http://flutterengineering.io/