Building a UI Framework

Ian Hickson (ian@hixie.ch) — 2025 — First Edition — https://software.hixie.ch/ui-frameworks
The comment-free preview version may be easier to read.
~~*To leave comments, use the edit version.*~~

Introduction

This document examines important design decisions for creating a new graphical UI framework.

It was commissioned to answer the question of how to create a UI framework, considering four possible design goals: developer adoption, performance, display effects, and power consumption.

This document has three parts[1]. In the first part, we will cover some background material to set the context for the discussion. In the second part, we will examine each of the stated goals which provide constraints within which to consider how to design a UI framework. In the third part, we will discuss design choices, examine possible implementations, and discuss some non-technical issues within the context of the specified goals.

See also: Table of Contents appendix.

About the author

Ian Hickson's career started with web standards testing and web standards development around 1999. His early work focused on CSS, where he worked as an invited expert to the CSS working group and served as co-editor for CSS 2.1. In 2004, he co-founded the WHATWG standards organisation and served as the editor of the HTML standard for around ten years, introducing many core web technologies such as <video>, <canvas>, and Web Sockets, and providing the first precise definitions of HTML parsing and core DOM Level 0 features on which the web depends. During this time he worked for Netscape, Opera, and Google.

In 2014, he joined the team that would go on to create Flutter, the most popular cross-platform framework. He designed Flutter's novel layout system and various other core components, and fostered Flutter's open source community.

He is now an independent consultant.

[1] It also features many footnotes. To jump to a footnote in Google Docs, double-click the footnote number in the text. To return to the text from the footnote, press the ESC key on your keyboard.

Background

This section introduces basic concepts of application and UI framework design. People familiar with developing applications and frameworks to support them are unlikely to find anything in this section novel or interesting, but readers who find aspects of later sections confusing may find useful clarifications in this section.

Applications

For the purposes of this document, an application is a self-contained software product with which a user interacts. An application runs within a runtime environment, e.g. an operating system or web browser. Interactions happen through input devices (touchscreens, mice, keyboards, etc) and output devices (screens, speakers, vibration devices, etc), and applications have some ability to compute and communicate with other processes and services (e.g. filesystems, networks, clocks).

Users interact with applications via a *User Interface (UI)*. Developers use *Application Programming Interfaces* (APIs) to interact with their software's runtime environment and other libraries.

Application architectures

There are a variety of approaches to building applications that have found favor over the years. They have been given names like "Model - View - Controller" (MVC) and "Model - View - Presenter" (MVP) and "Model - View - ViewModel" (MVVM). In practice there are many, many, *many* variations of these designs, many more than there are names, and over the years the names MVC, MVP, etc, have been applied so loosely to so many variations of these approaches that the names are largely meaningless now[2].

Let us study some of these approaches.

No data, only UI

The simplest application is one that has no state. A "hello world" application, for instance. The application's UI (the "view") is created, and then remains static until the application is terminated.

Such an application is basically a document. As the application consists of only UI, there is nothing to architect.

Mixed data and UI, with no abstraction

An application with state needs some approach for changing that state and having the UI reflect it. The simplest stateful applications do not need a complicated abstraction, and so developers will typically not use one: the data and the UI logic can be mixed together.

For example, consider an application that merely displays a counter that can be incremented. The application can be implemented by having a variable (e.g. a global variable) that holds the current value, and the UI components read that variable to display it on startup, and when the button is

[2] Though that doesn't stop people having many (often contradictory) strong opinions about what these terms mean. This problem is at least twenty years old, with Martin Fowler expressing dismay about it already in 2006.

activated by the user, the button code can directly increment the variable and then update the display to show the new value.

Separate data and UI (Model - View)

When an application's state becomes significant, mixing the data and the UI becomes awkward; for example, showing a piece of data in multiple places of the interface (e.g. document titles) requires interface elements to reference each other; when their lifetimes are not aligned, this requires further coordination logic. At this point, the obvious abstraction is to separate the data model ("model") from the UI ("view").

For example, consider a calculator application. Its model contains the current value and any memorized values, maybe the history. The view consists of the buttons and the display.

When the user activates a button, the button can update the model and then tell the rest of the UI to update itself, so the display can reflect the new value by reading the model.

An advantage of separating the data and the UI, beyond the benefits for building the interface itself, is that the data can be more easily persisted. For example, it provides a single place for data to be stored in a local database, or transmitted to and from a server. Managing this kind of synchronization when the data is mixed with the interface logic is much less convenient.

Model contains logic (separate logic and UI)

One variation of this is to have the model contain both the data and the business logic. When the "add" button is pushed on the UI, instead of the view directly changing the number in the model, the view tells the model "the user clicked add", and the model updates the number accordingly.

This approach has the advantage that the business logic can be defined separately from the UI, and re-used with other UIs. (In extreme cases, the model can be written in a different programming language entirely.) It also means the business logic can be tested without involving UI in the unit tests.

This is the model that [this document recommends] as a default suggestion in a framework's templates and early examples.

Event-based model

Having put the logic into the data model, it may be reasonable to rationalize the API for updating the model into a uniform event-based architecture: every update is an event, and the data model processes the events in some consistent manner. (This is sometimes referred to as the Business Logic Component, or BLoC, pattern.)

This then enables an interesting feature: the events can be stored, and the state of the data model can be defined as being the result of reducing over the events[3]. Undoing an action becomes the trivial operation of removing the last event from the event list[4].

---

[3] That is, iterating over the events, applying transformations for each event that represents that event's actions, and returning the final result. For example, a counter application with increment and decrement buttons would track the sequence of increment and decrement events, and iterating over them, with a starting value, would produce the "current" value to show in the UI.

[4] For performance, this model is usually implemented in reverse, where the current state is cached and the events allow one to walk backwards through past states by applying the opposite of their transformation.

Model notifies listeners (pub/sub)

With the model separated from the display, instead of having display components notify each other when a change needs to be reflected, the model itself can notify listeners (components in the view) when there is a change to the model. With this approach, the button does not need to notify the display that something changed; the button merely notifies the model ("publishing" a change of a particular type), and the model then notifies the display components (which are "subscribed" to particular change types).

Notifications can be fine-grained. For example, if the model tracks both the current number and some memorized values, the calculator's memory count label can listen for a notification regarding changes to the size of the memory, and the display can listen for updates regarding the current number. This reduces the number of redundant UI updates.

Isolated data and UI (Model - View - Controller)

In the Model - View approach, the View still knows about the Model. This can be further abstracted by having a separate object, known as the controller, mediate all interactions between the Model and the View. When the model is updated, it can notify the controller and the controller can then update the UI; when the user interacts with the UI, the UI can notify the controller, and the controller can turn this into commands for the model.

Whether having a separate controller is valuable or not strongly depends on the complexity of the data model and the UI, how similar the data model is to the UI presenting it, and on how much these are expected to change over time. If the data model closely resembles the view, then there is little benefit to a mediating layer. On the other hand, if the structure of the data has little resemblance to the way the data is presented, having a separate component that can pre-process the data before display can greatly simplify the UI layer and help with testability.

View notifies listeners

The view can notify the controller of changes directly, or alternatively, the controller can listen to events from the view. In this latter approach, the Model and the View expose APIs but do not know about each other *or the Controller*, which makes testing the Model and the View even easier. In this architecture, the controller translates notifications from the Model and notifications from the View into updates of the View and updates of the Model respectively, converting between their various conventions.

Data binding (Model - View - View Model/Binder)

A variant of the Controller approach is to have the mediator (in this case called the View Model) expose the data to be displayed by the View, and then having the View bind directly to the View Model (rather than having the Controller push the updates to the View). The mechanism by which the View binds to the View Model is called the Binder.

Two-way binding

A superficially attractive variant on this approach is two-way binding, where the user interacting with the control directly affects the underlying data model in some way (rather than via an API as described above).

This approach is often presented as minimizing the work required of developers, but it has a number of disadvantages: it makes data validation harder, it poses a risk of update loops (where the view updates the data directly, the data is stored in some slightly different form, the model then causes the view to update, but the view displays the data in a slightly different form, triggering a new data update, etc), and it requires the data to be in the exact form that is needed for the view.

Entity - Component - System

An entirely different way to approach the problem of architecturing an application is the Entity - Component - System (ECS) approach, mainly used in games.

In this architecture, every control could be represented as an entity, with components such as position, hover effects, values, etc. A variety of systems, such as positioning, rendering, mouse handling, etc, would regularly iterate over all the entities to apply their behavior.

The big advantage of ECS is that it encourages storing data in a manner that increases data locality for components. The system that checks mouse handling would be able to iterate through all the relevant data for every UI entity without having separate fetches for each entity. This contrasts to most UI approaches, where the data is stored in a tree structure, which tends to lead to a lot of cache misses at the CPU level.

In practice, ECS is not well suited for UI. ECS architectures work best when entities are largely independent; in UIs, there tends to be a close relationship between entities, e.g. for layout purposes. While it is technically possible to represent a tree using an ECS approach, this is not the strength of ECS architectures.

Client - Server

Applications frequently have a remote component and a user-device component. The component on the user's device is generally considered the *client*, and the remote component is generally the *server*. (One server usually services multiple clients simultaneously.) How the two communicate can vary, e.g. involving protocols like HTTP, Web Socket, raw TCP sockets, or UDP, but such details are outside the scope of this document.

Installed applications

The traditional model for application deployment is for the code and supporting assets to be copied to the user's device in some manner, e.g. at the device's factory, or, more commonly, by the user downloading a package from the Internet using another application (e.g. an application store or web browser).

Applications thereafter often download updates automatically, or delegate responsibility for the same to the operating system or supporting libraries (such as the aforementioned application store).

Ephemeral applications

Some applications are designed to run in a runtime environment that fetches the application dynamically on demand. This is, for example, how browsing the web works. A web browser on the user's device fetches the data, code, and assets of a web page (for our purposes, we call that web page the "application"), and then renders and executes the page on the device. In principle, other systems could use a similar approach, though in practice this is exceedingly rare. Android supports

a model called "instant apps" that is similar (though rarely used), and iOS supports App Clips, which are a similar model technically though not intended to be used as a replacement for applications.

Server-side rendering

Because ephemeral apps require the user to download the application before they can interact with it, there is a performance issue: the bigger the application, the longer it will take before the user can see a result. To avoid this, on the web (and in principle any platform that supports ephemeral apps, but right now it's just the web), it is common to design one's application such that the browser receives enough information early in the download process to provide the initial rendering and interactivity of the application, and for the rest of the content to be downloaded in the background (if at all).

One technique for doing this is *server-side rendering*, wherein the application logic is run on the server to create a static form of the application that is screen-ready. This is as opposed to client-side rendering, where the application logic runs on the client to build the UI that is then rendered. Because client-side rendering can require a substantial amount of logic (e.g. a UI framework), while server-side rendering only relies on the client platform's built-in logic (the browser), the latter can be significantly faster to render the first frame. Of course, once the application is entirely locally available, having the client generate the next screen can take mere milliseconds, while obtaining the next screen from the server requires at least a round-trip, meaning dozens if not hundreds of milliseconds. Thus it is common for web frameworks to have a hybrid approach where the developer writes code for client-side rendering, and then the framework uses that code to pre-generate server-side-rendered screens.

Separating style and semantics

In principle, the web is designed so that how the *content* of a web page is *presented* can change dynamically to handle different device characteristics such as varying screen sizes, bit depths, or, more dramatically, media, for example rendering a web page for the screen, for a braille display, for audio using speech synthesis, etc.

To achieve this, HTML is designed so that the semantics (meaning) of the content can be encoded independently of the style (presentation) instructions. Instead of marking a block of text as having a bigger font size, it is labeled as a heading. Sections of the text with headings can be demarked, and these sections can be nested, implicitly defining the relationships between parts of a document. Sections can be labeled as asides, footers, etc. Within paragraphs, spans of text can be marked as emphasis, citations, keyboard input, etc. This allows a separate part of the document written in a format called Cascading Style Sheets (CSS) to specify how the document should render by associating specific styles with specific semantics. A heading could be labeled as blue for screen media, and as loud for audio media. A text span representing keyboard input could be labeled as monospace for screen media, and using a robotic voice family for audio media.

This also allows for quickly switching between different kinds of presentation on the same medium. A navigation section could be presented on the left if the screen is wide enough, or on the top if it is not. The color scheme and font choices of a web page could be changed at a whim by swapping out the style sheet.

In theory, this makes the web uniquely capable as a multi-media, device-independent, application[5] framework[6]. This would further allow computers to semantically analyze web pages for search purposes (or even more).

In practice, the ultimate goal of device independence was not achieved. Content is written for either computer screens or phone screens[7], and accessibility annotations are (if the user is lucky) layered on top to allow accessibility tools to interact with that content[8]. Instead of the web being natively audio-capable, users experience the web as a visual medium that has been adapted for audio. Colors and font choices are frequently baked deeply into the content, rather than being cleanly separated. The semantic elements of HTML are used for the convenience of the developer more than the user, and where that convenience ends, the semantic-free elements of HTML (like `<div>`) are used liberally.

Fundamentally, most humans do not appear to be well adapted to thinking about application development at the level of abstraction required to write code using a strict separation of style and semantics. The status quo, where developers target specific screen sizes and then layer accessibility features on top of that, is therefore the most pragmatic approach in terms of making applications maximally useful[9].

What is a UI framework?

There are many aspects of writing interactive applications that are common to many applications; a UI framework is a library that brings together these common elements so that application developers do not have to reinvent the wheel with each project.

Components of a UI framework

Some typical components of a UI framework include the following.

Widget libraries

UI frameworks typically support a mechanism by which prebuilt components (buttons, check boxes, text fields, scrolling lists, etc) can be referenced. These are also sometimes referred to as controls or elements. Typically, frameworks allow developers to *compose* these components into reusable assets.

---

[5] The web was originally intended for documents, but from very early in the web's development, the line between "document" and "application" was blurred and now there is no meaningful distinction. Very few web pages are strictly static; even the most basic of pages typically has some running code for advertising or analytics.
[6] The styling capability is not unique to the web; similar features exist in other formats, e.g. DSSSL and LaTeX. However, the web is unique in attempting to apply it to multiple media (e.g. print, desktop computers, mobile touchscreen devices, braille, speech).
[7] Or, cynically, for search engines (SEO), but for now let us focus on the user-facing content. SEO is discussed later.
[8] Curiously, the accessibility advocacy community has largely rallied around this approach as well, encouraging developers to use ARIA and other accessibility annotation tools rather than focusing on teaching web developers to use HTML's intrinsic semantics first and adding style second. This is probably the correct pragmatic choice. The final user experience is more viscerally relatable for developers than the abstract concept of semantics.
[9] Furthermore, because code that is not tested typically does not work, and because most developers would only be able to test their applications on certain media (e.g. computer screens), even if developers were able to write in strictly abstract manners, this would not necessarily bring joy to those users who would be using the applications in untested modes (such as with a braille display or with speech).

UI description

The application developer must describe how the widgets come together to form the application. This may be a domain-specific language (DSL), such as HTML, or it could be a series of imperative calls to create a data structure, or it could be a functional-esque hybrid. Some frameworks have tools (user interface builders[10]) that generate the UI description code or data files.

Layout

From a description of the UI, the framework must then determine the positions of the various components. In some frameworks this is trivial (each component being assigned an absolute position and size on a grid), whereas in others the layout phase can be highly involved (enabling *responsive design*, where a single user interface description can work for a wide variety of form factors, e.g. phones, tablets, and computer screens).

How elaborate the layout system of a UI framework needs to be depends on the use cases for which the framework is to be designed. A UI framework designed for embedded systems with fixed size screens and push-button data entry (e.g. kiosks) does not need an elaborate layout system. On the other hand, a UI framework designed for desktop operating systems, with resizable windows, or for mobile devices, where the view can be rotated, necessitates a richer, more dynamic, approach.

Rendering

Once the position of each widget is known, the framework must generate pixels that form the scene with those widgets. This phase can itself have many subparts, such as *painting* (where the widgets are turned into vectors, display lists[11], bitmaps, or other concrete representations), *compositing* (where the various images forming the widgets are combined in some form, e.g. to resolve transparencies, shadows, or blending effects), and *rasterizing* (where the vector images are turned into pixels).

Modern frameworks make extensive use of GPU *shaders* to render UI scenes. For example, rather than computing the pixels of a gradient on the CPU, the GPU would compute them using a shader.

Typography is an important and often large part of the rendering pipeline. Frameworks typically have to support Unicode, fonts, bidirectional text, text shaping (e.g. for Arabic), text measuring, justification, emoji, and numerous other features that are now considered table stakes.

Hit testing

To be interactive in environments with touch screens, mice, or touch pads, frameworks have a mechanism to determine which widgets correspond to which pixel positions in the scene. This is used to implement taps and clicks, hover effects (e.g. tooltips), and drag effects.

Focus and focus traversal

Keyboard support requires that the framework keep track of which widget is currently receiving key input, as well as how the widgets relate to each other to support spatial focus navigation and tab focus navigation.

---

[10] Historically these were called rapid application development (RAD) tools, though that term seems to have fallen into disuse in the modern era.

[11] Lists of graphical commands for later execution. Display lists are essentially the imperative form of a vector graphic.

Accessibility

Permeating the entire framework are decisions and features that are designed to aid users in various ways. For example, hit targets (touch-sensitive areas) must be sized so that users don't struggle to hit specific parts of the application. Text should be rendered at sizes that are consistent with the users' preferences. Controls should be exposed to OS accessibility APIs so that users can use accessibility assistant software to navigate them. Colors should be selected so that they are sufficiently contrasted to be readable even under less than optimal conditions. The list of accessibility features and accommodations that a framework must consider is long.

Operating system integration

Applications run in the context of an operating system (OS, also called the *platform*), whose graphical user interface the application must integrate with. For example, the OS might need to be informed of the application's icon and name, the OS is how the application hears about input events (keyboard, touch, mouse, etc), the OS typically provides an accessibility API which the application must support, and so forth.

The list of integration points for an operating system is huge. Close integration with the OS typically results in a better user experience. For example, an application that seamlessly responds when the OS switches from light mode to dark mode is less jarring than an application that flickers, restarts video playback, and loses pending user input when the mode changes.

Localization and Internationalization

When applications are written for users of more than one *locale*, a number of features are required.

Localization is the term used for APIs that support an application being translated into multiple languages. In addition to dynamically selecting strings, this can also involve combining strings or making other decisions based on the exact messages to display. For example, consider a message containing the string "Found 5 files". If the number of files found is 1, then, in English, the message must instead be "Found 1 file" (singular). In other languages, there may be different strings for when the number is 0, 1, 2, or more.

Internationalization involves supporting differences in rendering and presentation conventions, such as starting the week on a Monday rather than a Sunday, using a 12 hour clock rather than a 24 hour clock, or using different punctuation to represent large numbers or fractions.

Some locale-specific differences can require extensive support in a UI framework. For example, some languages (such as Arabic and Hebrew) are written right-to-left while others (such as English and French) are written left-to-right. This affects the entire orientation of the UI (for example, scroll bars are typically rendered on the far side of the text's writing direction).

It may also be desirable to support other orientations, for example, top-to-bottom writing (vertical text) is sometimes used in Japan. Supporting top-to-bottom text layout is a lot more complicated than merely changing the orientation of the coordinate system[12].

[12] Vertical text is discussed in slightly more detail in the Design choices section.

Common paradigms

Controls

The term *control* is generally used to refer to an element of the user interface that the user recognizes as a distinct component with which they can interact, such as a button, text field, check box, window, or slider. Some controls are *compound controls*, e.g. a spinner is a text field with up and down buttons.

Render objects

We shall use the term *render object* to refer to the basic primitive of a layout tree. This is the node that holds the logic for determining, at a minimum, the geometry of the UI scene, which generally is also used for painting and hit testing. Most UI frameworks have this concept; the alternative being to recompute the potentially expensive layout of the entire interface every frame. There is no standard term in the industry for this primitive; other terms used include *layout nodes*, *render elements*, and *visual objects*.

In many frameworks, these objects are also used to represent controls (e.g. buttons, text fields). In those contexts, the term used may be more generic, such as *view*, *control*, *element* or *component*.

Trees

UI frameworks tend to be awash with trees[13]. UIs are often represented as trees of widgets, layouts as trees of render objects, renderings as trees of display lists. Accessibility information is typically represented as a tree. Nodes in these trees are often implemented using objects in a class hierarchy, another kind of tree. There are also internal data structures in some frameworks, e.g. the virtual DOM of React or the layer tree in Flutter, that are themselves trees.

Children

Nodes in trees can typically have children (those that have no children are, by definition, the leaves of the tree), represented by a list of further nodes.

Some layout primitives don't map perfectly to a *list* of children. For example, a 2D grid or data table. Some frameworks solve this by mapping everything to lists anyway, for example, a table could be a list of rows containing lists of cells (HTML uses this approach). This becomes a little awkward when there are cells that span multiple rows or columns (or both), but it is not insurmountable. Another approach is to have a single list which is then mapped onto the table (e.g. concatenating all the cells of each row into one long list). This is often used internally. Some frameworks are less strongly opinionated about how children are represented and may use different data structures in different situations, as appropriate.

Cross-language interoperation

Cross-platform frameworks frequently deal with multiple languages or runtime environments. (At the very minimum, Android requires interacting with a Java-like environment, while iOS requires interacting with a Swift-like environment, so even if a framework uses the same language as one of those platforms, it cannot avoid needing to work cross-language on the other.)

---

[13] Early members of the Flutter team used to joke that the Flutter team's core competency was transforming trees.

The term *Foreign Function Interface* (FFI) is used to refer specifically to mechanisms that allow synchronously invoking code written in one language or environment from code written in another (either directly, or via a thunk of some form). There are also other mechanisms to achieve language interoperation, e.g. message passing.

Compositing and composition

The term *compositing* refers to how layers of drawing instructions are combined to form a final scene. This is analogous to layering panes of glass.

The term *composition* refers to the ability to combine building blocks together to create more elaborate constructions. This is analogous to using Lego bricks to create houses.

These terms and concepts are unrelated.

Embedded

The term "embedded" is used to refer to the use of software in dedicated hardware, as contrasted with general-purpose computers. For example, a wearable (such as a watch), an appliance (such as a dishwasher), in automotive (such as in a car's dashboard), or in passenger value-added services (such as the in-flight entertainment system of a modern jet). UI frameworks are used in all of these scenarios, but typically these platforms are not supported out of the box and require some porting work by the end developer.

Immediate-mode vs retained-mode APIs

APIs take many forms, and some can be categorized based on their method of operation. An API that exposes a data model that can be modified is a "retained-mode" API. For example, a library could maintain the current state of a picture, and the client (the calling code) could mutate the picture, e.g. moving its components over time or changing its attributes such as color. In contrast, an API that has no internal state but that is used to perform computations on demand is an "immediate-mode" API. For example, a library could have commands to build a picture from scratch, where the client owns the resulting picture object.

Design space

As with any engineering project, it is important to decide up front what one's *core goal* should be. Different goals lead to different design decisions, and different goals are often mutually exclusive. In software engineering, for example, one frequently has to make a trade-off between memory consumption and throughput (the space/time trade-off).

Further constraining these goals are the *requirements*, features that are non-negotiable.

Broadly speaking, requirements can be thought of as features that either exist adequately or not, while goals are typically metrics where success is relative.

Requirements

Graphical user interface

User interfaces come in various forms, e.g. command line interfaces, conversational (e.g. in speech-based assistants), physical hard-wired button panels, and gesture-based (e.g. as used with Microsoft's Kinect or Apple's Vision Pro).

The kind of user interface we shall focus on here is a *graphical user interface* (GUI), as used on flat screens[14]. These are primarily 2D systems[15]. Most applications for desktop computers, mobile devices, lean-back form factors (TVs), and wearables use this kind of interface. Even 3D games tend to use GUIs in their menus and settings screens.

As of the time of writing, the expectations around 2D GUIs have had 57 years to develop[16]. Some features have existed for a long time, such as buttons (shapes that are activated by some sort of pointing device or touch to trigger an action), or text fields (areas where a line of text can be entered or edited). Others have varied a lot over the years, like scroll bars (which, while keeping generally to the theme of having a thumb and a track, have sometimes had arrows on one or both sides of the track, or not had arrows at all; or made the track interactive, or not; or had the thumb be proportional to the scrolling content, or of a fixed height; or played with whether the scroll bar itself is even visible when the content is not scrollable or not being scrolled). The precise set of decisions that a graphical user interface makes is known as a *design language*. Operating systems (or desktop environments) typically adopt a single design language that they expect applications to follow (these are sometimes referred to as "the platform conventions").

In terms of what is expected of a UI framework, generally developers would like their UI framework to follow these platform conventions. In the case of cross-platform frameworks, this can mean implementing multiple design languages, or integrating with the operating system's or desktop environment's own implementation of its design language.

Applications sometimes intentionally depart from these conventions, e.g. in an attempt to form a brand identity of their own. To support this, a UI framework needs to provide the ability to create custom controls with their own bespoke design language.

Support for multiple platforms

For the purposes of this document, we shall primarily consider designs that allow for a single codebase to be used to compile applications for all of:

- iOS (phones and tablets)
- Android (phones and tablets)
- Windows
- macOS

---

[14] Virtual Reality (VR) environments often use 2.5D graphical interfaces (2D interfaces on planes positioned in 3D space). These are considered out of scope for this document.

[15] Over the years, 2D interfaces have sometimes attempted to hint at a third dimension through various means, e.g. careful use of colors or blurs to simulate bevels and shadows. These techniques have come in and out of favor over the years as fashions have shifted. With rare exceptions, these are still pure 2D systems from the perspective of their layout mechanisms, and will be considered as such herein.

[16] 1968 was when Douglas Engelbart first demonstrated a coherent graphic UI (Wikipedia, YouTube). While GUIs have evolved significantly since then, many aspects of UIs today can be traced back to this groundbreaking presentation.

- Desktop Linux (X, Wayland)
- Web browsers (via JS or Wasm)
- Emerging consumer platforms such as OpenHarmony (standard systems)

For practical purposes, these requirements allow us to assume a 64 bit architecture[17], at least hundreds of megabytes of available RAM and storage, and the presence of a GPU. Each of these environments also has some form of hardware abstraction layer; for example, the framework can rely on the host environments[18] to translate hardware inputs into a well-defined structure[19].

The mobile platforms (iOS, Android, and OpenHarmony) also support peripherals such as watches (e.g. Linwear), in-car infotainment systems (e.g. CarPlay), and TVs (e.g. Android TV). In some cases, this takes the form of fully porting the framework and applications (e.g. Android Automotive OS applications are full Android applications that run on the car system directly); in other cases the application runs on a host device (e.g. a phone), and uses an API to drive the peripheral (e.g. software targeting Android Auto does not execute directly on the car hardware[20]). UI frameworks are also used to create the interfaces of "smart" appliances, medical devices, and other Internet-of-Things (IoT) devices.

Supporting these peripherals and IoT devices, together generally known as "embedded devices", is considered out of scope for this document. That said, while the ability to use the framework in embedded scenarios is not a requirement, in practice, a framework that supports the platforms listed above in a modular fashion is likely to be portable to such systems without requiring significant rearchitecting.

Platform-specific conventions

As noted above, different platforms have different conventions. Because of this, a UI framework that supports multiple platforms must implement renderings and behaviors that are specific to some of the platforms they support. Some of these conventions are firmly established and will be called out explicitly in this document. Most, however, are matters of fashion and the conventions change from year to year as operating systems attempt to maintain the appearance of freshness. For this reason, specifics of this nature are generally glossed over in this document; framework developers must examine their target platforms when implementing features to match the contemporary conventions and must update the framework over time as these conventions change.

Inputs and outputs

To be a competitive option for applications on the given platforms, a framework must support features such as:

- Touchscreen input (multitouch), including gestures[21].

---

[17] Technically, as of the time of writing, Wasm is still limited to a 32 bit address space. It does have 64 bit floats and integers, however.
[18] In most cases this is the operating system, but because the web is one of the listed platforms, it could also be a web browser. On Linux the situation is a little complicated also, as the operating system and the windowing system (X Window System or Wayland) are arguably distinct.
[19] Of course, in practice, they all do this differently and usually in a lossy fashion, and for inputs in particular, it would be better if they would just pass the raw HID messages through directly to the applications...
[20] There appears to be a possible exception to this for games, though the Android documentation is extremely unclear on how exactly this works.
[21] Gesture recognition is discussed in detail in the Framework internals section

- Physical keyboard, mouse, trackpad input, and stylus input[22].
- IME (Input Method Editor) input[23].
- Displays with a variety of pixel densities from 96ppi to 500+ppi[24].
- Application spanning multiple monitors[25].
- Multiple windows[26].
- Platform-specific accessibility layers (VoiceOver, TalkBack, JAWS, etc)[27].
- Haptic feedback[28].
- High dynamic range and wide gamut (more colors)[29].
- Embedding other applications into the interface (e.g. embedding an external PDF viewer, or a 3D scene, or an advertisement rendered by a separate library), including passing pointer and key events through to the embedded application[30].

Layout features

Frameworks are expected to support many features, including in no particular order:

- Scrolling (including scroll bars, pinned headers, infinite scrolling, seeking)[31].
- Theming[32].
- Text editing[33].
- Localization and internationalization[34].
- Right-to-left text layout[35].
- Vertical text layout[36].
- Animation[37].

---

[22] Keyboard input and some aspects of mouse input are discussed in slightly more detail in the Framework internals section. Mouse input in general can be treated similarly to other kinds of pointer input such as touch input, as discussed in the section on input.
[23] Input Method Editors (IMEs) are not discussed in detail in this document, but contribute to the requirements on text input and text fields, which are discussed in the Framework internals section.
[24] This topic is broached in the Framework internals section during the discussion of layout, in the context of *logical pixels*.
[25] There is little to say about this topic but it is mentioned in the Framework internals section.
[26] This is not discussed in this document, but is a feature that is important to design for from the beginning. The Flutter team made an early choice to target mobile devices first, rather than following most other frameworks in being designed for the desktop first, so as to provide a better experience on mobile than was typical at the time. This was largely successful, but one oversight in the initial design was that an assumption of there being a single window was baked very deeply into the graphics backend, the framework, and even the API design. As a result, adding multiple window support to Flutter now that Flutter is also targeting desktop devices has unnecessarily become a multiyear project. Had multiple windows been considered in the original design, even if no immediate support had been added, this would have instead been a trivial matter, as it is for most other frameworks.
[27] Accessibility is a topic that recurs throughout this document. There is also a brief discussion regarding its handling in the Framework internals section.
[28] There really is not much to discuss with regard to haptic feedback, and therefore this document does not go into any detail about it.
[29] Color spaces are discussed in slightly more detail in the Design choices section.
[30] This topic is touched on briefly during the discussion on design choices around painting and again in the Framework internals section.
[31] A potential design for implementing scrolling is discussed in slightly more detail in the Framework internals section.
[32] Theming is discussed in the Design choices section.
[33] The list of text input features expected of UI frameworks is discussed in the Framework internals section.
[34] Localization ("l10n") and internationalization ("i18n") are discussed briefly earlier in this document. In practice they are mostly unrelated to other choices one has to make when designing a UI framework, and are reasonably well-understood problems. So long as care is taken not to actively frustrate these requirements, they can be largely handled by a separate library (e.g. ICU).
[35] Bidirectional text is discussed very briefly in the Framework internals section.
[36] Vertical text is discussed in slightly more detail in the Design choices section.
[37] This is briefly touched upon in the Framework internals section.

Goals

As mentioned in the introduction, for the purposes of this discussion, we will primarily consider four possible design goals: **developer adoption**, **performance**, **display effects**, and **power consumption**. Each is examined in its own section below.

UI frameworks can be optimized for other design goals.

One goal that has received a lot of attention in recent years is **compatibility with AI**. Many teams across the industry are experimenting with generative AI creating UIs on demand; a framework can be more or less well suited for this purpose.

Another related goal is **suitability for server-driven dynamic UI**[38]. The web, with its entirely runtime driven dynamic architecture, is well suited for this purpose and many websites are really a form of application entirely built out of server-driven UI. For example, search engines chose UI components in response to user requests; a search results page for the query [SJC to LAX] might feature a form for selecting flights, while the results page for the query [vamp lyrics] might feature a media player. A framework designed for server-driven UI may differ substantially from one designed for showing UIs that are shipped in the application and usable offline.

Frameworks designed with the web in mind may have **search engine optimization** as a goal: the ability for search engines to crawl the application and discover its content, allowing it to be surfaced in search engine results. This is a key feature for social networks (where much engagement is driven by users finding existing conversations relevant to their interests) and store fronts (where sales are driven from users finding relevant products).

Goal: Maximizing developer adoption

One possible goal would be creating a framework that is used in more applications, or adopted by more developers, than existing frameworks.

[38] Server-driven UI is discussed in slightly more detail in the Design choices section.

Determining the competitive landscape

Before being able to design a framework to best the existing ones, one first needs to establish the scale of adoption of existing frameworks.

Determining the overall adoption of existing UI frameworks is non-trivial. We consider two approaches, objectively counting adoption by deployed applications, and developer surveys.

Adoption by deployed applications

Approach

One way to measure adoption is to count how many applications use particular frameworks. This poses some challenges.

How to define "application", and which applications one should consider in scope for such an analysis, is the first problem. Enterprise software forms a significant portion of all software, yet is not publicly accessible. A lot of software is only available for purchase, increasing the potential cost of an analysis. Platforms typically have multiple platform-specific frameworks, it is an open question whether one should consider these to be relevant in a discussion of designing cross-platform frameworks. Furthermore, a lot of software is obsolete or no longer maintained; should one consider the choices made by developers in past decades to be relevant to the discussion today? Similarly, one may want to filter out applications that were created as toys, proofs of concept, or that were abandoned early in development; or one may want to analyze applications separately from games, or one may want to categorize applications by their target audience or by some other metric. When looking specifically at web applications, one must decide whether to count individual pages, hosts, domains, or use some more nuanced approach. For example, is each page on amazon.com a separate application? Is amazon.co.uk the same application as amazon.com? How about read.amazon.com?

A second problem is how to count adoption. Should applications be weighted by number of users? By contemporary installation rates? By released versions? By number of developers? Should one filter an analysis to only include applications with more than a certain number of active users? Or fewer, on the basis that the most popular applications may have different needs and therefore represent a different, much smaller, market?

A third problem is how to define "framework". Should one consider any application that runs in a browser to be using "the web" (meaning HTML, JavaScript, CSS, the DOM, et al), or should such applications be grouped by web framework (e.g. React, vue.js, etc)? Are server-side web UI frameworks relevant? Should any Android application with a Webview be considered to be one using "the web" as a UI framework? Should all applications running on Android be considered to use the JVM? When an application uses multiple frameworks (a common situation, especially in older, larger applications), how should they be counted?

In practice, such an analysis would need to be performed separately for each platform, applications on that platform would need to be identified and collected in some manner, and then each application would need to be analyzed to determine the framework(s) used. Such an analysis is non-trivial for different reasons for each platform. The web has trillions of web pages. iOS and Android have proprietary app stores and do not provide a complete index of their wares. Many

desktop applications are distributed in bespoke or proprietary channels that are not easily discoverable.

Guesses

In the absence of an actual analysis, we substitute some guesses based on experience and marketing claims.

At a very high level, it would appear that "the web" (meaning HTML et al) is the most widely-used UI framework on a per-application basis.

Many, if not most, applications that use "the web" actually use one or (very often) more web-specific frameworks such as React. Which web frameworks are the most popular has varied significantly over time. There does not appear to be a recent publicly-available analysis of web content that would directly answer the question of which framework is most widely used. Such an analysis requires significant crawling infrastructure.

On iOS specifically, according to Google and Apptopia, Flutter, UIKit, and SwiftUI are the three most-used frameworks[39].

On Android specifically, the default framework, Flutter, and React Native are probably[40] the most widely used frameworks (in that order).

Other platforms are even harder to analyze and it is difficult to make any firm claims, but the native frameworks are probably the most widely used, for some definition of "native framework". (This is especially messy on Windows, where Microsoft has recommended more than a dozen frameworks over the years, often simultaneously; and on Linux, where there is no official default framework.)

Conclusion

Approaching the problem from the perspective of applications would require a significant investment to find objective results. On a per-application basis across all platforms, based primarily on guesses derived from the author's experience, the frameworks most likely to be the most widely adopted are the web, Flutter, React, React Native, and the "default" frameworks as a whole.

Developer adoption

Alternatively, rather than looking at adoption in deployed applications, one can look at what developers claim to use in surveys.

Surveys have a number of important limitations. They extrapolate from those developers who respond to the survey; selection bias among respondents is inevitable (e.g. only people who are aware of the survey and have the time to fill out a survey will do so). This risks significant blind spots, for example, if there exists a large community of developers that does not ever come into contact with the survey, and that community has significantly different preferences than those developers who respond to the survey, this will naturally not be captured. Surveys depend on self-reporting. Humans are notoriously poor at accurately reporting their behaviors. These and other issues mean that survey results tend to fluctuate over time to such an extent that assuming an error margin of ±10% is not unreasonable.

---

[39] Flutter "powers nearly 30% of all new [tracked free] iOS apps", according to Apptopia as reported by Google in a December 2024 blog post.
[40] This is an educated guess.

Stack Overflow Developer Survey

One such survey with published data is the Stack Overflow Developer Survey, though interpreting the results through the lens of determining the most popular UI framework is less than entirely satisfying, as the survey splits the relevant results into "language", "web technologies", and "other libraries", rather than, for example, "machine learning libraries" and "user interface libraries", which would be more useful for our purposes. Nevertheless, looking at all three categories and combining the results, we obtain the following adoption numbers for developers responding to this survey[41] in 2024:

Framework	Adoption[42]	Notes	Target
HTML/CSS	52.9%	The web.	Web
React	39.5%	Web library.	Web
.NET	up to 25.2%	Cross-platform[43]. Includes server-side code in addition to UI code.	n/a
Next.js	17.9%	React library (web).	Web
Angular	17.1%	Web library.	Web
Vue.js	15.4%	Web library.	Web
ASP.NET	12.9%	Web library in the .NET ecosystem.	Web
Dart/Flutter	6% to 9.4%[44]	Cross-platform framework.	Many
React Native	8.4%	Mobile framework.	Mobile
Qt	7.3%	Cross-platform framework (primarily desktop).	Many
Svelte	6.5%	Web library.	Web
Electron	6.5%	Desktop framework (uses web technologies).	Desktop
Blazor	4.9%	Web library in the .NET ecosystem.	Web
SwiftUI	4.3%	iOS library.	iOS
Visual Basic	up to 4.2%	.NET library (desktop, also usable for server-side non-UI code).	Desktop
.NET MAUI	3.1%	.NET library. Cross-platform framework.	Many
GTK	2.6%	Desktop framework (primarily Linux; also macOS, Windows, web).	Desktop
Delphi	up to 1.8%	Cross-platform framework. May include server-side code.	Many

[41] Exactly how representative the respondents are of the community at large is unclear. Notably, the survey does not seem to have any representation of frameworks used primarily in China such as WeChat Mini Programs.
[42] These numbers are reported to the same degree of precision as Stack Overflow's report, but as noted earlier, this is almost certainly wildly more precision than is warranted.
[43] This is really a superset of the other .NET frameworks listed in the table, plus some other smaller ones.
[44] It's unclear how Dart (6%) can have less adoption than Flutter (9.4%), as Flutter is a Dart library, but in the author's experience, survey results rarely make sense anyway when examined closely. This does give a lower bound to the error one should project onto these numbers, though.

The table above is an attempt to include a representative sample of the most popular UI frameworks and some other historically notable frameworks that are still used and supported today.

Exactly which frameworks should be included in this list is very debatable.

First, the survey does not always clearly distinguish UI frameworks from non-UI frameworks, as noted earlier. For example, the survey does not distinguish jQuery (not a UI framework) from jQuery UI (a UI framework based on jQuery). Some frameworks blur the line, for example Next.js is used for building UIs but it is really a React library for server-side rendering and UI components are transpiled to HTML/CSS. Blazor (a web UI framework) is a component of, but can be used independent of, ASP.NET Core (not a UI framework), which uses .NET on the server, but is not part of ASP.NET (which contains a UI framework) and is unrelated to the .NET Framework (which contains UI frameworks including ASP.NET), and .NET MAUI (a UI framework which is part of .NET, though not .NET Framework, and used on the client)[45].

Additionally, the concept of "UI framework" is not especially well-defined. Many self-described web UI libraries, for example, build heavily on HTML/CSS and do not specifically introduce UI widgets or layout features, focusing on features such as routing and server-side rendering. Several of the frameworks (especially the ones represented as languages in the survey, e.g. Visual Basic and Delphi), while originally designed for client applications, are also used on the server for business logic.

Other surveys

SlashData and Evans Data Corporation (EDC) have paid reports that cover developer topics such as framework adoption. Their results tend to be broadly in line with the Stack Overflow surveys (within the aforementioned limitations inherent to all survey-based data collection).

More precise information could be obtained, at a cost, by hiring appropriately qualified researchers or pollsters. The specifics of such research are outside the scope of this document.

Analysis

These results are consistent with the earlier claim that the web is the most widely used framework, followed by Flutter and React Native. It suggests that among web frameworks, the most popular are React, Angular, and vue.js; and among cross-platform non-web frameworks, adds Qt as a significant contender.

Examining the popular frameworks

The web and web technologies

HTML, CSS, JavaScript, et al

At its core, the web stack consists of HTML, a document markup language; CSS, a declarative language for customizing the style of HTML elements; JavaScript, a programming language; and an extensive API (sometimes known as the DOM). A mass of other formats and technologies, such as various image formats, font formats, network protocols, and server-side tooling, support this core

[45] Explaining the precise history and naming convention of the .NET family of products is beyond the scope of this document.

stack. Applications written using these technologies are presented by clients known as web browsers. Most web content is transmitted in its uncompiled form[46], and processed on the client as it is rendered.

The web is incredibly successful; more or less everyone with a computer uses the web, meaning that the web's users are a superset of the users of every other platform. This makes it a very attractive platform for developers. EDC claims there are 27 million developers[47]; if over 50% of developers are web developers (as per the Stack Overflow survey results), that means there are nearly 15 million web developers.

In addition to web pages rendering in web browsers, web technologies are also used in other contexts. Electron, described below, is one example, but by far not the only one; web technologies are used in embedded scenarios, enterprise kiosks, and web views in "native" applications. For the purposes of simplicity, we shall refer to any use of a web runtime (i.e. a browser engine) to be a use of "the web". This includes using JavaScript and the DOM, HTML and CSS for rendering, and the wide gamut of web APIs.

In principle, the web is unique in that it is not defined by a particular implementation, but by specifications. How true this is can be debated[48], but the principle must surely have had some part in the web's success. The existence of multiple browsers each implementing these specifications in slightly different ways, however, led to the need for libraries that would paper over the differences. This normalized the idea of using libraries to write for the web, rather than targeting the web's APIs directly.

A potentially bigger contributor to the web's early success was the extremely low barrier to entry for developers. Requiring no SDK, indeed no tooling of any kind beyond a text editor and a web browser, anyone who wished to do so could create a web page. Deployment was also remarkably easy, requiring no more than an internet-connected server, which in the early days of the web would often be provided by one's educational establishment or employer, and later was something easily obtained in exchange for modest sums of money. Over time[49], the web's technologies unfortunately became a mishmash of well-meaning but largely uncoordinated efforts[50]; the lack of a single vision to unify development from the early 1990s to today is clearly evident in the wildly different styles of the web's formats and APIs. Some features are literally the result of

---

[46] At least, from the perspective of the web browser. Most developers use tooling that converts their source code into a more efficient form for transport, a process known as *minification*, in which the original source is transpiled to a form that is borderline impenetrable to humans and yet still not optimal for computers. This is just one of the many ways in which web technologies have evolved over the years, with each small step seemingly logical but the overall arc being questionable at best.

[47] As of 2024. At the time of writing, this statistic is presented graphically on the EDC home page. It's not clear how "developer" is defined. In particular, it's not clear whether the definition used by EDC matches the sample group of the Stack Overflow surveys. The claims by the Qt and Flutter teams discussed later in the document may suggest that the definition used by EDC is rather wider than the sample group reached by Stack Overflow.

[48] In many ways it is almost entirely *untrue*. Originally, the specifications were so vague that browser vendors were forced to reverse-engineer each other rather than rely on the specifications. In time, the specifications were made more precise, but to remain relevant, were written by having the spec writers reverse engineer the browsers instead. (The quote attributed to Ledru-Rollin, "There go my people. I must find out where they are going so I can lead them", seems especially apt here.) Additionally, at several times in the web's lifecycle, including unfortunately today, a single browser has so dominated the install base that its behavior is effectively the canonical specification regardless of anyone's intent.

[49] Not *much* time, really. The web was invented around 1991, and the mess arguably was already well established by 1996.

[50] The author acknowledges his part in this mess.

design-by-compromise[51], others are the result of mistakes whose lessons came too late[52]. This, combined with the increase in expectations around web applications, has raised the complexity of writing for the web, which has provided a fertile ground for further libraries that attempt to simplify web development.

Thus the web, a cross-platform application and UI framework in its own right, became host to further application and UI frameworks frameworks.

React

Probably the most popular such framework is React, a library developed at Facebook (now Meta). React popularized a programming paradigm sometimes referred to as "React-style programming" (not to be confused by "reactive programming"), also known as "immediate-mode GUI", wherein the developer writes functions that describe the UI state based on the application state, which the framework then uses to build and maintain a model of the application[53]. This contrasts with the style that was popular before React, wherein the developer first builds an initial model, then during the application lifetime mutates that model in response to changes to the application state. The immediate-mode GUI style has since been adopted by many other frameworks, such as Flutter, SwiftUI, and Compose.

React was originally programmed using either JavaScript or JSX, a superset of JavaScript (similar to the obsolete E4X) that adds a literal syntax for XML-like data structures, which can be used to describe components instead of imperative code. Today it also supports typed variants of the above, TypeScript and TSX.

React, being a web framework, uses CSS for its layout and styling features, rather than having layout and styling components.

Next.js

This framework adds server-side rendering to React-based applications. According to the Stack Overflow survey, it is used by over a third of React developers.

Angular

Angular is a TypeScript-based web framework with some tooling for server-side rendering.

---

[51] For example, the core DOM events API is just the union of two competing event models that were implemented by competing browsers in the late 1990s. The politics of committee-based standards development optimize for public relations far more than technical beauty.

[52] Once a feature has shipped in a web browser and been used in a (popular) web site, it is too late to remove it, because doing so would break that website, meaning users would refuse to update their browser (or worse, would change to a competing browser). Therefore lessons that are learned only once a technology has been deployed cannot be applied to that technology's design. Unfortunately, one can rarely learn relevant lessons before a technology is used by developers, so one just has to hope that one's original design is good. This is not a good way to design APIs, but it is basically the only way that is available for the web.

[53] This style seems to have originated as a means to reduce memory usage (which is ironic given its typical implementation strategy today). The earliest reference the author was able to find is a 2003 paper by Thierry Excoffier (corresponding GitHub repository). Casey Muratori apparently independently developed the idea in 2002 and brought the concept to more prominence in 2005 (YouTube video). History does not recount whether either of these pioneers influenced React's design directly, or if the idea was once again independently developed at Meta (then Facebook). React's style was definitely an influence on Flutter's design, and SwiftUI and Compose were developed in an environment where React and Flutter were both well established.

Vue.js

Vue is a web framework that is designed to integrate agnostically with other web technologies. At a high level, it provides a component system and extends HTML with data binding. However, Vue also provides many other features that web developers require, such as server-side rendering.

ASP.NET

ASP.NET is a server-side framework for generating web sites. As it is primarily server-based, it is somewhat out of scope for the kinds of frameworks being discussed in this document.

Svelte

Svelte is a web framework that, at a high level, extends HTML with data binding and provides a mechanism for creating reusable components.

Electron

Whereas many cross-platform frameworks have a mechanism whereby applications can be built and deployed for the web, to be accessed using web browsers, Electron takes applications built for the web (or using web technologies), and enables them to be built and deployed as executables on desktop operating systems. This allows development teams who have written elaborate web applications, e.g. Slack or Discord, to repackage their applications as desktop applications without having to rebuild them using the desktop platforms' frameworks.

Electron essentially ships a copy of Chrome with the application's code (HTML, JS, etc).

A similar approach is used by Tauri, though Tauri also supports mobile platforms and reduces the overhead by reusing the platform's built-in web renderer rather than shipping an entire web runtime with the application.

Blazor

Blazor is a client/server web framework based on the .NET ecosystem. It layers over HTML. Client-side code is compiled to Wasm.

Qt

Qt is a cross-platform application SDK, originally designed for desktop applications but now supporting all major operating systems for mobile and desktop, as well as the web, a wide array of embedded platforms, and numerous niche operating systems. Wikipedia describes the supported platforms and some notable Qt applications, and discusses its commercial history and some aspects of the history of its open source community. In summary, major parts of Qt are licensed under the GPL and LGPL licenses for free use, with all of the code and many tools available under a commercial license. The project appears to be a commercial success, with arguably the main benefit being the availability of extensive (paid) technical support.

Marketing materials claim that Qt has over 1.5 million developers[54]. The Stack Overflow numbers multiplied by the above ECD numbers for total developers would estimate 2 million.

---

[54] "Qt Group is a global software company, trusted by industry leaders and over 1.5 million developers worldwide to create applications and smart devices that users love", according to Qt Group in a January 2025 blog post. It's unclear if this is total developers, or developers over a certain time period (e.g. monthly actives or yearly actives).

In addition to supporting a wide array of target platforms, Qt also exposes bindings for most major programming languages (and many niche ones). As a result, it is more or less the only practical choice if one has already chosen a specific programming language and one needs flexibility in terms of the target platform. The only other major cross-platform frameworks either require a specific language (Flutter requires Dart; Electron and React Native require JavaScript) or ecosystem (.NET), or are limited to certain target platforms (e.g. React Native only officially supports Android and iOS; Visual Basic and Electron only officially support desktop). Qt also appears to be the only serious choice for many embedded scenarios[55]. It is likely that its ubiquitous availability is key to its popularity.

Qt controls the rendering of UI controls directly. On platforms that support it, UI controls are rendered using platform affordances to match the native look. On some platforms (e.g. KDE), Qt's own rendering is, by definition, the platform's native rendering.

Flutter

Flutter is a cross-platform application SDK, originally designed for mobile applications but now supporting all major operating systems for mobile and desktop, as well as the web.

Marketing materials claim that Flutter has over 1 million developers[56]. This would put it below Qt's claimed numbers[57], though the Stack Overflow survey has them the other way around. Then again, the Stack Overflow numbers multiplied by the aforementioned ECD numbers for total developers would put the number of Flutter developers at around 1.6 million to 2.5 million, which is much higher than the Flutter team has ever claimed.

Unlike Qt, Flutter requires the use of Dart for its UI elements, and most Flutter applications also use Dart for business logic. Flutter's API design, its use of Dart[58], and its tooling are all designed specifically to make developing Flutter applications pleasant (it is both often easy to write new code and the framework often helps developers find bugs quickly). It is likely that this is the main factor contributing to its popularity. One specific feature that is rarely seen in other developer products, or rarely seen as effectively executed, is Flutter's *stateful hot reload*, which allows a developer to change their application's code and update a running instance of their application without losing any in-memory state. This makes the edit-test cycle for many aspects of application development *extremely* short (sometimes measured in milliseconds).

---

[55] Web technologies are also commonly used in embedded scenarios, for reasons that the author cannot understand.
[56] "Flutter has over 1 million monthly active developers across the globe" according to Google in a December 2024 blog post. These numbers are based on the Flutter tool's metrics. These metrics exclude developers who have opted out. These metrics attempt to exclude bots (e.g. the Flutter tool being used on CI/CD systems).
[57] To be clear, when a team claims a metric is "over" some number, it means it is barely above it. "Over 1 million" is definitely less than "over 1.5 million". A marketing team that *could* claim "over 1.5 million" is extremely unlikely to be so conservative as to only claim "over 1 million".
[58] Dart is a *surprisingly* approachable language. In early Flutter usability tests, participants were given a UI to build with Flutter using Dart, but were not told anything about the language, merely given some examples and Flutter's documentation (with no explicit documentation about Dart itself). In the post-experiment debrief, some of the participants, upon being asked which language they had been using, reported with surprise that despite having successfully written code in Dart for an hour, they had not even thought about what language it was and did not know the answer to the question. While some newer Dart language features have departed a little from this design philosophy, the core language is so unsurprising that programmers who are familiar with other C-like languages are usually able to pick it up with very little or no training. This must surely be a big contributor to Flutter's popularity.

Flutter renders every pixel directly. Controls are reimplemented in Dart[59]. Flutter's main UI library implements Google's design language according to its specification, in some cases more accurately than Android's own implementation[60].

React Native

React Native is a cross-platform application SDK, originally designed for mobile applications, though third-party support exists for a variety of other platforms including macOS, Windows, and the web. React Native is based on React, as the name suggests.

React Native requires the use of JavaScript, which is both a weakness (because JavaScript[61]) and a strength (because it allows the majority of developers to leverage their existing JavaScript skills to create applications for platforms they would otherwise not be able to reach). React Native's use of JavaScript is likely the primary factor driving its popularity.

React Native uses an indirect approach for controlling the UI. The JavaScript code indirectly controls instances of native controls. In principle this enables applications to match the platform's look and feel exactly, but it reduces the level of control developers have over the precise rendering.

React Native is based on React, which uses CSS for layout and styling. Because React Native does not target web browsers, it replaces this with a bespoke styling system based on JavaScript, heavily inspired by CSS. This contrasts React Native with other immediate-mode UI systems which tend to integrate the layout into the component tree.

SwiftUI

SwiftUI is a closed-source framework built on Swift, specifically for use for iOS and macOS. It uses the React-style paradigm but with a lot of magic that is hidden by the closed-source nature of the library.

The main benefit of SwiftUI is its tight integration into the Apple ecosystem: it has direct access to the iOS and macOS APIs without needing a translation layer, it is supported natively by Xcode, important features that Apple wants software to use are supported in SwiftUI typically before any other framework, etc. As a framework it is relatively terse, avoiding some of the boilerplate verbosity of Flutter. For example, it has syntax to create properties that combine a getter and setter into a single closure-like object that syntactically is equivalent to a variable but has the underlying

---

[59] Early in the implementation of iOS style widgets for Flutter, Flutter engineers discovered bugs in Apple's own implementations of those controls. The question of whether to implement those bugs faithfully or whether to be better than Apple convinced the team that, *given sufficient investment*, it was completely possible to faithfully reimplement a UI framework in Flutter's architecture. Unfortunately, having established this, the team then chronically underfunded that development for many years, leading to a general feeling among the community that this approach is not viable.

[60] This leads to interesting existential questions about the nature of correctness. If an application does not match the operating system's implementation of a control, is it wrong even if the operating system is incorrectly implementing the design language according to that design language's own public specifications?

[61] As discussed earlier, the JavaScript ecosystem is a disaster. JavaScript itself is a horrendously quirky language, and its core APIs (including the DOM) are an incoherent mishmash invented by many people who did not have a single vision. The JavaScript ecosystem is successful (63.5% in the Stack Overflow survey) because it enabled interactivity on the web and is very easy to get started with (for example, it does not require an explicit build or compilation step, and has an extremely forgiving set of semantics). However, it is a liability for any non-trivial software project (because it being "forgiving" means fewer ways for the language and APIs themselves to catch bugs). Attempts to fix some of JavaScript's failings range from languages (e.g. TypeScript, which introduces a layer of type-checking in exchange for requiring a compilation phase) to libraries (e.g. jQuery, which attempts to provide an internally consistent API to replace what browsers provide out of the box), but while these have each improved matters somewhat, they are fundamentally unable to completely distance themselves from wider problems in the language semantics and the broader ecosystem.

expressiveness of an object with a getter and setter. Variants of this idea can also be used to automatically mark views as dirty when a property changes, or observe a data model to update views when the underlying model changes.

Unfortunately, as a result of its closed-source nature and its heavy use of Swift's metaprogramming features, much of SwiftUI's behavior is opaque. This means that while the development experience is excellent at first, developers can run into difficulties when they come to the edge of SwiftUI's "golden path". There is no way to step into the framework to learn how things work, to debug strange issues, or to create variants of SwiftUI constructs that work slightly differently. The official documentation for SwiftUI is also rather anemic and uninformative.

Visual Basic

The term *Visual Basic* today refers to a .NET language and associated development environment for creating primarily Windows applications, though in principle one can use .NET tooling to target most other desktop and mobile platforms. Interfaces are built using a visual interface building in VisualStudio. The language is a descendant of BASIC, although, while recognizable as such, it has long evolved beyond it, with types, modules, exception handling, and classes. The compiler is self-hosted and open source[62].

Visual Basic originated in the 1990s. The language is intended to be easy to learn (like BASIC). The primary target audience is enterprise software.

.NET MAUI

.NET Multi-platform App UI (.NET MAUI) is a framework that uses the .NET ecosystem to enable a single codebase to target Android, iOS, macOS, and Windows. The framework is primarily based on declarative markup and imperative code (rather than immediate mode UI code). It supports hot reload.

GTK

The GNOME project's UI framework, GTK, is a C library with bindings for many languages. It supports the major desktop platforms, though the project's priorities tend to position Linux first. It uses Cairo as its graphics backend.

The Vala and Genie programming languages were developed to make using GTK more pleasant.

Delphi

Delphi is one of the oldest frameworks represented in the Stack Overflow survey. Originally an ObjectPascal-based Windows 3.1 developer product created in the 1990s, it has been merged with its C++ sibling product (C++ Builder) and is now sold as a package that allows C++ and ObjectPascal to be used to target Windows, macOS, Linux, iOS, and Android. Much like Qt, the Delphi intellectual property has changed owners several times, and it offers extensive (paid) technical support.

Unlike the other frameworks, development is only supported on Windows machines, with the other targets being supported by way of a proxy tool, allowing software to execute and be debugged on a remote machine from the Windows machine.

---

[62] Part of the Roslyn project.

Delphi's UI builder experience is based around drag and drop, with dialogs being created mainly with absolutely positioned widgets. Widgets can be configured using an interactive object inspector, and event handlers can be easily connected to specific widgets during the UI creation process, with the development environment making it very easy to toggle between the source code and the UI views. This makes the interface for desktop applications in particular very easy to create. This, combined with an approachable language, built-in interactive debugger, and a long list of built-in widgets, probably accounts for its continued steady usage over the past decades.

One of the most notable features of Delphi is the speed of its compiler, which is orders of magnitude better than compilers for most other languages. The compiler and the interactive UI builder that allows developers to see exactly how their interface will appear in production (the UI editor literally runs the actual code for the controls in the IDE), makes the edit-test cycle very short (a few seconds), despite the lack of any hot reload functionality.

Delphi actually has multiple frameworks: the VCL (Visual Component Library) for Windows, and FMX (Firemonkey) for cross-platform development.

Other frameworks mentioned in this document

wxWidgets

This is a C++ library with bindings for numerous languages and support for the major desktop platforms.

Tk

This is a C library primarily intended for use with Tcl, whose popularity peaked at the end of the 20th century. It supports the major desktop platforms.

Clay

Clay is a relatively new minimal C layout library by Nic Barker, with built-in bindings[63] for C++, C#, Odin, Rust, and Zig, and backends for various graphics libraries (e.g. SDL and Cairo), the web (using Wasm), and Win32. While its feature set is limited (e.g. it does not come with text fields built-in), it provides an interesting case study for creating an immediate-mode ("React-style") framework in a non-GC environment[64]. Clay supports dynamic layouts.

NativeScript

At its core, this is a JavaScript runtime for mobile devices and wearables that provides a TypeScript wrapper around iOS and Android APIs, allowing TypeScript or JavaScript code to directly interface with native APIs on those platforms.

---

[63] The project is open source and various community members are working on bindings or ports for other languages too.

[64] Clay uses fixed-size arenas to side-step memory allocation and memory management overheads. This is a very effective mechanism in embedded scenarios where predictable memory usage is much more important than dynamically growing a process' memory usage on demand, but it has the side-effect of putting a hard limit on the complexity of possible interfaces at runtime.

Dear ImGui

A C++ library primarily intended for creating debugging interfaces in games, Dear ImGui uses an imperative API to generate draw calls that describe the interface. Dear ImGui does not support dynamic layouts, as the layout is computed incrementally as it is described.

Some observations of the popular frameworks

Several of the more popular frameworks (including more or less by definition all the web frameworks) are a patchwork of different technologies, either put together and presented as a single package (e.g. .NET MAUI) or intentionally designed to allow the developer to pick their set of technologies and put them together (e.g. Vue). This contrasts to some of the other frameworks, which are designed as a single coherent self-contained package (e.g. Delphi and Flutter).

In general, there is a great benefit to having a self-contained solution, despite it being a significantly greater burden on the framework creator. A single framework can present a coherent and consistent API and tools, rather than requiring developers to navigate different styles of API, tooling that wasn't designed to work as a single solution, multiple sources of documentation, and disparate ecosystems.

This document will primarily focus on creating a single coherent framework, rather than putting together existing technologies.

Factors driving developer adoption

When a developer is considering how to write an application, the default answer is to use the platform-approved framework for their target platform(s). For example, when writing an iOS application, the most obvious choice is to use SwiftUI. That's the framework Apple recommends, and it will solve the problem.

Third-party frameworks do not benefit from this status as "default choice". For this reason, they must be better than the default choice in ways that the developer cares about to be even considered; to be chosen, they must be sufficiently better that the developer believes the risk of choosing a non-default, non-platform-approved framework is smaller than the benefit of choosing that framework.

Merely being a cross-platform framework is not sufficient here. Developers must believe that the costs of walking off the path paved for them by the platform vendors are outweighed by the savings of only having to write the application once. An application with a simple UI and complicated business logic wrapped in a self-contained C library[65] may be so easy to write multiple times that the costs of using a third-party UI framework are still higher than the costs of rewriting the application. This is exacerbated if the UI framework requires extra work on each platform (for example if the memory management conventions are different across platforms).

In this section we shall discuss high-level ways in which a UI framework can lower its costs of adoption and raise its benefits relative to other options, thus becoming an attractive choice to developers.

---

[65] C libraries are the lowest common denominator in terms of code reuse across platforms. Every platform, including the web, has some mechanism to integrate with a self-contained C library with minimal effort.

The ultimate goal for a third-party framework that aspires to large-scale developer adoption should be to be so much better than the alternatives on each platform, that even developers targeting only one platform view it as a superior solution than the "default" solutions. Developers will go out of their way to use frameworks that they love to use. When faced with the decision between a job doing something with a painful framework and a job doing something with a framework they love, they will often pick the latter choice. This drives the salaries of jobs involving painful frameworks up, and lowers the salaries of development jobs involving the more popular frameworks. This also means companies choosing a framework based on the cost of developers will lean towards the frameworks developers love, further driving the framework's popularity.

In summary, creating a framework that people *love* is the cleanest path to making the most popular framework.

Solving real problems

One of the most important features of *any* software project that aspires to have users is to actually solve the real problems of those users. For people building applications today, *their* goal is to get their applications into the hands of their users, and therefore the tools they use to create and deploy those applications must enable that goal. To that end, a UI framework must either integrate with an existing application SDK[66], or must be developed as a part of an application SDK. A UI framework on its own does not solve the problem of how to write an application.

Business logic

In practice, developers do not write a user interface in a vacuum; they create software that happens to have a user interface. If that UI cannot do anything, then the application can be little more than a demo.

To that end, it is important that the UI framework have a clear path for integration with business logic. It should be straightforward to call business logic from UI logic, and vice versa. This could be by having the UI and business logic use the same language and be written as one component, or it could be via a Foreign Function Interface (FFI) or message passing mechanism.

Additionally, developers should be given clear opinionated guidance on how to connect UI and business logic[67]. The aforementioned "model contains logic" approach is the most straightforward to explain while providing sufficient abstraction to be useful for most small to medium applications.

---

[66] This lesson was learned early in Flutter's development. The Flutter team (or Sky team, as it was then known) was firmly in the mindset of creating a UI framework, and expected developers to integrate Flutter's engine and framework into their applications. Early adopters quickly disabused the team of this notion, expressing their need for an integrated build system, built-in integration into basic system APIs, out-of-the-box support for bootstrapping on every supported platform, etc.

[67] This is sometimes referred to as "state management". Realistically, state management is very app-specific, and writing good app-appropriate state management is fundamentally what programming is all about. However, this does not remove the need for early guidance for developers approaching a UI framework. One of the mistakes the Flutter team made during the first few years of Flutter's development was dismissing this need on the basis that "writing state management is what writing an application *is*, we can't do that for you". This led to a vacuum that was filled by a variety of packages and tutorials, all giving contradicting advice. By the time the team realized the importance of providing guidance to early developers, there were so many answers to the question that the official guidance could neither be loud enough to drown out the many other answers, nor opinionated enough to be useful (because any opinionated answer would run the risk of alienating community members). The result is that to this day, one of the first questions new developers to Flutter ask is "how should I manage my state", and they still cannot get a satisfactory answer (today they are overwhelmed by the answers rather than being left hanging, but the result is the same).

Platform integration

Applications typically have some interactions with the platform's APIs. Some of these APIs are common to many platforms, and are most conveniently abstracted directly by the framework itself, such as determining the application's rendering size (e.g. the window's size for desktop applications), dealing with input devices such as keyboards, mice, and touch, reacting to application lifecycle events such as being backgrounded, suspended, or closed, and interfacing with the filesystem.

Every operating system has a long tail of APIs, however, and it is impractical for a framework to try to expose them all[68]. This means some mechanism must exist for developers to directly interface with these APIs. (Ideally, such a mechanism would be packageable so that a developer can wrap a platform API, and distribute that code as a library for use by other developers.)

Platform APIs take various forms. For example, on Windows, the APIs are C-based functions exposed by system DLLs to which an application can link. On Linux they are provided as syscalls[69]. On Android they mostly take the form of Java APIs that must be accessed from Java or Kotlin code.

As with the case of interfacing with business logic written in another language, this could be done using a language-level feature for communicating with the OS APIs (a Foreign Function Interface, or FFI), or via some sort of message-passing mechanism combined with writing code in a language supported by the platform APIs.

The former has the advantage of only requiring developers to learn one language (plus the platform APIs), so for example one does not need to learn the idiosyncrasies of Java on Android, C on Linux, and JS on the web, all just to call platform APIs[70]. On the other hand, it is non-trivial, and may require per-target-language features (one syntax for integrating with Java APIs on Android, one for integrating with JS APIs on the web, another for C APIs on Windows, etc)[71], which may be seen as polluting the purity of the language[72].

The alternative, writing code in the platform's language, is superficially easier to bootstrap and notably requires no support from the language runtime. It also appears to benefit from avoiding having to combine the platform's expected language semantics with the framework's language's semantics. These two benefits, however, pale in comparison to the practical difficulties that developers will encounter. Learning a new language and all its related infrastructure (build systems, etc) is a significant burden to put on a developer, let alone one new language *per platform* (after all, one of the big perceived benefits of a cross-platform framework is removing the need to do just that). In addition to having to write code to interact with the platform APIs, the developer must also marshal information back and forth from this code to the code written using the framework's language. This adds an extra problem that is not the problem the developer was facing.

---

[68] The web tries, with APIs for everything from USB HID to video conferencing to cryptography to preventing screen dimming, and in so doing is the most conclusive evidence that doing so is impractical. NativeScript also tries, by literally providing JavaScript wrappers for every API of the underlying platform.
[69] Linux also has some canonical libraries that wrap the syscalls, e.g. the pthreads library wraps the kernel threading APIs.
[70] That said, those idiosyncrasies typically leak out into the APIs anyway, so it does not completely insulate the developer.
[71] While this may seem to defeat the point of having one language (which is familiar to the developer) to solve the problem across all the platforms, in practice the new material that need be learned here is dramatically less than what one would need to learn to use a separate language entirely.
[72] To which one would say, maintaining language purity is not one of the stated goals.

Developers in general are not motivated to solve problems they perceive the framework is giving them as some form of gratuitous homework.

The problems with marshalling get substantially worse, furthermore, if it requires some form of asynchronous communication. Many platform APIs expect to be used in a synchronous fashion. This is especially the case with APIs involving callbacks from the platform. While it is generally possible to implement all such APIs in a manner that bridges the asynchronous and the synchronous[73], doing so is not easy and often has unfortunate costs (such as increased memory usage).

As a result, in practice, finding some way to expose all supported platforms' APIs in a synchronous manner is the optimal solution for both developers (popularity), despite requiring a higher overall investment from the framework provider.

Ceremony

Many frameworks require developers to include boilerplate code that is identical, or near-identical, for every application. This is sometimes because the framework is unopinionated about some core bootstrapping questions, requiring the application to, for instance, provide a conduit from the operating system to the framework for input events. In other cases it is because the framework or the programming language it uses is highly verbose.

Requiring developers to include this near-identical code in each application is a point of friction that reduces the approachability of the framework.

One way to measure the severity of this problem for a framework is to look at the size of a minimal "Hello World" application[74]. A dozen lines of code is reasonable. Hundreds of lines of code is not.

It is possible to design a framework specifically around making "Hello World" as short as possible. (For example, in HTML, the literal string "Hello World" is a functional minimal application[75].) However, it is useful for the "Hello World" application to have some reference to the API as it gives new developers a starting point to search documentation and tutorials. Indeed, having a canonical short "starter application" that shows some basic usage of the framework's APIs can make getting started fun.

Build

Configuring a build system solves a real problem for only two classes of developers: those tasked specifically with configuring a build system, and those implementing build systems. For everyone else, configuring one's build system(s) is a chore that is, at best, viewed as one step removed from the actual problem being solved. It becomes a special case of "ceremony".

---

[73] For example, consider an OS API where the OS can, at any time, ask the application to synchronously provide its state so that the application can be terminated (this is used to make applications appear to remain open forever on mobile phones). To implement such an API in an asynchronous environment, the API has to be flipped so that any time the state changes, the application updates a cached state, and when the OS asks the application to provide its state, the latest cached state can be provided. This costs extra memory to hold the cache, and extra CPU to update it any time the state changes.

[74] Some frameworks (e.g. Flutter) autogenerate some boilerplate, e.g. build files and platform integration code, that is then the responsibility of the developer despite them not having written that code. Whether to count this as part of the size of a "hello world" is an interesting question. This code tends to not be a blocker for framework adoption, because new developers don't know it's there. However, it is still a cost that developers eventually face.

[75] Technically it's non-conforming HTML for legacy reasons; the shortest confirming HTML "Hello World" requires a magic string prefix, a title, and at least one open tag.

As a result, the ideal framework would, if prioritising developer popularity, reduce the need for developers to deal with build systems to an absolute minimum, ideally none at all. Where a build system is needed, it should be to support the developers' needs outside of the UI framework, e.g. integrating with business logic that needs to be compiled by some other toolchain, or enabling some sort of preprocessing that the developer may wish to perform before the framework's compiler begins its work[76].

Deployment

Arguably the most important aspect of an application SDK is the ability to deploy the code to users. This means building the application in an optimized form (a release build), packaging the application in the platform-vendor-approved manner, and supporting whatever requirements the platform has for distribution.

For example, on iOS, this involves having the application package signed and uploaded to Apple's systems.

Developer productivity

Approachability

Qt, Flutter, and React Native are each popular because they are approachable, but in different ways. React Native is approachable because it leans into an existing widely available skill set (JavaScript). Qt is approachable because its bindings make it language-agnostic, enabling it to be used in many different contexts where there is no other viable choice. Flutter is approachable because it was designed to be extremely easy to learn and use.

Interestingly, these three approaches to approachability are somewhat mutually exclusive. The most popular languages all suffer from problems that make them unsuitable beyond merely being popular (e.g. JavaScript is too forgiving, and JavaScript and Python are not sufficiently strongly typed, all of which reduces how much they can help the developer avoid writing bugs; JavaScript, Java, and C++ are languages that suffer from a long history of technical choices that make them quirky in hard-to-debug ways; C# is tied to the .NET ecosystem). On the other hand, languages that maximize ease of use and developer productivity aren't sufficiently popular (e.g. Dart and ObjectPascal, used by Flutter and Delphi, are both outside the top ten in pretty much all language surveys). For practical purposes, supporting a wide variety of language bindings more or less forces one to pick C as the core language — but C is neither wildly popular *nor* easy to use.

Of the three approaches, however, one has more potential than the others. Using a popular but quirky language unsuitable for language bindings will never lead to the language being productive and suitable for bindings. Using a language suitable for bindings won't fundamentally make that language more popular over time (the whole point of that approach is that it does not need to). Using a niche language, or creating a new language, that is designed to be productive to use, however, could, in principle, in the long term, lead to a situation where that language *becomes* the popular language.

This argues for developing a framework using (or creating) a language that is optimized for developer productivity, both in terms of being easy to pick up, and in the sense that the language

[76] Though frankly, discouraging this kind of code generation is best, as it interferes with the abilities of IDEs to provide instant feedback.

and tooling directly helps developers catch bugs early and steers developers away from making mistakes. Ideally, it would be possible to modify the language in response to usability testing done with the framework, meaning the language team should work closely with, and in enthusiastic consort with, the framework team.

Minimizing languages

As briefly discussed above in the *solving real problems* section, developers will have a better experience if they are able to stick to a single language throughout their development experience, rather than having to switch between different general purpose languages (e.g. switching between Kotlin and Swift) or having to switch between different DSLs[77] (e.g. switching between HTML, CSS, and JavaScript).

The platform integration solution used (as discussed above) can have a big impact on the ability to minimize the number of languages used. Flutter's architecture, for example, provides developers with platform-specific code (Kotlin for the Android build, Swift for the iOS build, C++ for the Windows build, etc), which the developer may need to deal with in order to solve some problems (e.g. to talk directly with platform APIs). This naturally places a higher burden on developers than if, somehow, the developers could achieve all those same results with only one language.

Usability testing

The gold standard for evaluating decisions that affect the factors driving developer adoption is usability studies. To create a product that is truly *loved*, one must continually look at how developers experience one's product, and one must fix what developers don't like.

This is harder emotionally than it is technically. It is normal for framework creators to become enamored with their creation and believe it to be good. One must be willing to objectively watch developers using one's product and change one's creation even when doing so means changing something one previously thought was good.

Usability studies are best started early, often, and continually throughout the lifetime of the product. We will discuss this in more detail later.

Growing the total set of developers

Instead of competing with other UI frameworks for developers, another approach for creating the most popular UI framework is to create one that is usable by people who, today, are not served by any existing UI framework. If the unaddressed market is larger than the current market, then potentially a new framework could target that market and be the most popular.

The market that the frameworks discussed above largely ignore is the "non-developer" market, that is, people who wish to create software but do not have the skills to program. Targeting this market is not a new idea; it is known as "zero code developement". In practice, however, the market does not seem to be especially large; apparently most users do not in fact desire to write software. Additionally, even with the current generation of code-writing LLMs, the technology is not yet at the

[77] What is considered a separate language vs a part of a language is an interesting semantic question. For example, are regular expressions a separate DSL or part of the host language? Is React's TSX a separate DSL, or is it part of TypeScript? The developer's experience is what drives the practical answer here. Usability testing is key to determining which DSLs are experienced as just part of the host language, and which are experienced as conceptually separate.

point where computers can generate non-trivial software that works adequately without the assistance of humans knowledgeable in programming.

The remaining approach therefore is to grow the number of humans knowledgeable in programming, by providing a very usable and approachable framework with a very usable and approachable language. This relies on adapting or creating a new language, a topic we shall examine in a later section.

Potential

Fully funded

There is remarkably little competition in this space, all things considered, and all the existing solutions have some significant limitations. A team dedicated to creating a truly compelling UI framework (or rather, application SDK), willing to make brave[78] technical and project choices driven by customer needs (as demonstrated by usability studies), with the resources to support a decade of steady investment before expecting any significant return, would be well positioned to create a product that would eclipse all existing cross-platform frameworks[79].

Modestly funded

With a more modest investment, less time, or with more constraints on the technical choices (e.g. if one is forced to build the framework in a specific language regardless of whether that is the best fit for maximizing popularity), one is still likely to be able to create a framework on par with the existing solutions, though one is unlikely to then achieve the goal of being the *most* popular. (For contrast, all current frameworks fall into the category the author would describe as "modestly funded" at best, with many being chronically under-funded.)

There is also a risk in creating Yet Another Framework. Developers are faced with a multitude of "good enough" solutions that each have their strengths and weaknesses. Adding more options for them to consider does not make their life easier. Today, the community reaction to announcements of work starting on new frameworks and framework-adjacent libraries is often mixed, not because of any aspect of the new product's design, but merely because of its existence.

Evaluating success

The leading indicator of popularity is usage, which can be measured without too much difficulty if using the framework requires the use of a tool that can report analytics[80]. For the purposes of comparing numbers with industry standards, the usual metric is *monthly actives* (really *28-day actives* so that daily trends don't add noise to the monthly trends), meaning unique instances

---

[78] "Brave" choices include those that sacrifice short-term gain over long-term success. For example, releasing all source code under a free license such as BSD or MIT rather than using a GPL/commercial dual licensing scheme. Another example would be being willing to increase one's development costs to maintain high code quality, even when the quality increase may appear more theoretical or less immediate, while the corresponding costs are high. Some of the choices that the author views as notable examples of "brave" choices are labeled as such in this document and can be found by searching for that word.

[79] Probably excluding the core web itself, which at this point is less a framework and more a platform. To create a *framework* more popular than the web would require also creating a *platform* more popular than the web, which is a topic for another document.

[80] Analytics are discussed in more detail in the Framework internals section.

detected over each reporting period. Care should be taken to identify tool instances that correspond to human users rather than tooling, otherwise CI/CD usage of the tool will quickly make the numbers useless (not to mention dramatically increase the load on the reporting infrastructure).

The trailing indicator is the number of deployed applications, which is much harder to measure, as discussed above.

Goal: Maximizing performance

A different goal one could consider is *performance*. In the context of a graphical UI framework, performance is generally a measure of how quickly the framework can generate the pixels for the screen. In practice, this ends up being two problems: how quickly the *first* frame can be rendered, and how quickly subsequent frames can be rendered (e.g. during an animation).

In computing, performance is often seen as a challenging problem where one must find tricks to do more in less time. For example, using multiple cores to parallelize workloads, sequencing loads so that the hardware is never blocked, or creating efficient data structures that make optimal use of cache lines. While these techniques are all very valuable and important, the single most effective technique for driving faster code is simply to *do less*. In general, fewer instructions take less time to run than more instructions, regardless of how clever the instructions are. Fewer abstraction layers means less code is running, so better performance. Fewer features means less code is running, so better performance.

This means that it is critical, to maintain performance, to minimize the impact on the core of the framework. Any code that has to run for every entity, whether that entity uses the feature that the code is supporting or not, is slowing down *everything*.

It's also worth considering that while premature optimization typically leads to decreased maintainability, this should *not* be used as an excuse to add more code than is necessary. Writing less code is not *optimization*, it's merely writing fast code. Once one gets into the habit of arguing "well it's only one more instruction", one quickly slides down a slippery slope where one has added millions of "one more instructions", with the inevitable effect being a reduction in the overall performance.

Time to first frame

Application startup time, or time to first frame, is dominated by one-time costs. For example, loading the application from storage, initializing data structures in memory, preparing the hardware, and computing the initial layout. There are various techniques that can minimize these costs but they all boil down to "do less".

Time to first paint

One approach used by many web frameworks is to precompute the data structures representing the application's UI, and serialize it in a form that can be quickly turned into an image (in the case of web frameworks, this is typically HTML, which can be parsed into the DOM structures faster than JavaScript code can generate those DOM structures). This approach often results in the initial UI being non-interactive (because the serialized form lacks the bindings to the UI logic, which must then be backfilled as soon as possible after the first frame is rendered), which can be frustrating for users.

Time to interactive

The gold standard measure for startup time is therefore "time to interactive", i.e. the time from when the user requested the application to the moment where the user can both see *and interact* with the application.

Ultimately, achieving a low time-to-interactive time is a matter of doing absolutely nothing that does not directly support rendering that first frame and setting up the input pipeline to handle user interaction. On most platforms, the basic act of starting an application, initializing the graphics drivers appropriately, computing the application's initial layout, rasterizing it, and rendering that frame can be done in less than 500ms[81], and this should be considered the target. Achieving this requires the framework developers to remain disciplined in deferring any work that can be deferred.

Application vs framework costs

Performance in general is one of the features of a framework that can be most easily undermined by application developers. If an application developer sends a message to the network and awaits a response (e.g. to perform login authentication) before allowing the framework to render the first frame, then any work the framework developers have expended in trying to optimize startup time will be entirely moot, as a single round-trip on the network can take seconds.

A framework can be designed to reduce the temptation for developers to delay the first frame, but little can be done to stop a determined engineer who is not focused on optimizing startup time.

In the same vein, there is little a developer can do to mitigate costs introduced by the framework. A framework that always takes one second to start up is never going to be used to make an application that takes 100ms to start up, regardless of the skill level of the developer.

---

[81] Some might argue that the target should be even smaller. For example, the time-to-interactive for Clay examples on Linux desktops is easily less than 200ms. However, this is unrealistic for some platforms; e.g. the Android app launching animation (which happens while the app is starting up) is several hundred milliseconds and an app can't be interactive in less time than that animation.

Interframe performance

The time it takes to generate a single frame is a direct contribution to the frame rate, which is visible as the smoothness of animations and responsiveness to input.

In practice the absolute minimum acceptable frame rate is about 60Hz. However, modern devices are typically capable of much faster frame rates. High end mobile devices typically support 120Hz, and high end desktop monitors can go up to 240Hz, with niche displays supporting refresh rates as high as 540Hz.

At 540Hz, an application has less than 2ms to update its rendering, and that's without considering that the operating system and hardware will also need some time to complete their work (such as compositing the application into the screen's scene or handling user input), and that the application's business logic may have to do work as well, to compute the data required to update the display in the first place. This puts a great deal of pressure on the framework's design.

Overhead

One way to do less while rendering a frame is to offload computation to a later stage when the application is idle. For example, using a bump allocator with a garbage collector dramatically reduces the overhead of *allocating* memory, but requires more careful handling of memory blocks later (to distinguish longer-lived memory blocks from blocks that are no longer needed, moving memory around to coalesce free blocks, updating pointers to those moved regions, etc). This allows one to reduce the time for generating a single frame, in exchange for paying a higher cost later when (hopefully) the system is idle.

As with pipelining (discussed below), this cost is not always free. In particular, if the application is never idle, then eventually the overhead must be paid while the application is busy, which tends to be noticeable as *jank* (missed frames).

Layout

Layout typically involves walking a tree to compute geometry. We shall discuss, in a later section, various approaches to this problem. For the purposes of maximizing performance, however, there are certain features that one would want of the overall layout algorithm, all of which boil down to "reduce work".

Walking the tree

The framework should, if possible, avoid algorithms that walk the same tree multiple times. Such algorithms are common; for example, it is not unusual for a layout algorithm to first walk the tree to compute the *sizes* of elements, and then walk the tree to *position* the elements. Avoiding the bulk of multi-pass algorithms is relatively easy, but for some layouts (e.g. tables with shrink-to-fit columns), requires making a simplicity/performance trade-off.

Avoiding such algorithms reduces paying for the overhead of walking the tree multiple times, but even more importantly, it avoids the possibility of accidentally introducing $O(N^2)$ algorithms into the layout. This could happen if, for instance, each piece of geometry first requires its entire subtree to perform a hypothetical layout (e.g. to find its intrinsic, or "preferred", size), then uses the

information gathered from that process to configure the subtree for a "real" layout. If every node does this, then the algorithm is effectively quadratic.

Reducing the scope of text layout

Measuring text is one of the most expensive operations in a UI framework. To the extent that a UI framework can avoid providing features that require measuring text, or can reduce the cost of measuring text (e.g. by limiting which fonts are available, or not supporting features such as ligatures or text shaping), this cost can be reduced. The trade-off, naturally, is expressiveness and support for locales that require those features. For example, skipping support for bidirectional text and text shaping makes support for Arabic essentially impossible; this is a high cost given that Arabic is the third most-used script in the world[82].

Simple nodes

In the web stack, each node in the layout tree supports the full set of CSS layout primitives. Every node must know how it can be padded, how it can have a background color, how it can have scrollbars, and so forth. This means each node in the layout tree has a complicated set of semantics, and there are many complicated rules for how these semantics should interact.

A simpler approach, for instance the one used by Flutter's framework, is to have specialized layout nodes that support a single feature and coordinate with a simple protocol. For example, padding in Flutter is a feature implemented by the RenderPadding layout node, a layout node that is unaware of the semantics of background colors or scrollbars. The nodes coordinate using a relatively straightforward protocol that is agnostic to the concepts of scrolling, padding, etc. (A suggestion for such a protocol is described later.)

From a performance perspective, the simpler approach of having dedicated specialized nodes and a simple mechanism for nodes to coordinate allows changes to the tree to only trigger the parts of the algorithm that are necessary to handle the changes. This reduces the workload across frames.

Rendering

In a similar way, rendering using a retained-mode approach where parts of the scene that have not changed from one frame to the next are reused verbatim avoids a lot of interframe work. For example, if a part of a scene is fading out, the actual contents of the scene need not be repainted each frame; the whole frame's scene graph or rendered bitmap (depending on the framework's architecture) can be reused unmodified, and just composited with a different opacity.

Finding equivalent ways to solve the same problem

Sometimes, there are multiple effects that would result in the same pixels, but one is expensive to compute and the other is cheap. For example, clipping complicated geometry to a path is non-trivial; however, masking one image using another is comparatively cheap. Similarly, it is less work to apply a translation to a cartesian point by adding two 2-dimensional vectors than it is to express the translation as a 4×4 matrix and then multiplying the vector by the matrix.

When building primitives, noticing cases where significant work can be avoided without changing the resulting effect can be a useful way to improve performance. One should always benchmark

---

[82] After Latin and Chinese, and used by over 800 million people.

these optimizations, however, to avoid cases where the optimization is itself more expensive than the supposedly "expensive" full solution[83].

Just-In-Time optimizations

To reduce interframe costs, the framework can spend cycles preparing efficient solutions at runtime. Frameworks that use runtime-interpreted code, for example, can apply just-in-time compilation to hot code paths to reduce the cost of running those code paths. Similarly, graphical backends can prepare dedicated shaders optimized specifically for the precise scenes encountered at runtime — rather than having a single shader for drawing all circles, for example, a framework could notice that certain shapes are commonly found in the UI, and generate a shader optimized for, say, 24 pixel radius blue circles.

This approach has a cost: the actual compilation step itself is expensive, and can delay frames. This means that this approach directly hurts the time to first frame. If an application must be compiled on startup, it cannot do anything until it has been compiled. If shaders must be compiled before the first frame is rendered, then the application cannot show anything until they are compiled.

Wherever possible, therefore, computation should be shifted to the build step (i.e. during development, not at runtime). Unfortunately, this means that the compilation lacks some information about what to optimize for; but on the other hand, because orders of magnitude more time and computation can be expended on the optimization step, the resulting code may just be faster in all ways than the just-in-time compiled code.

Application vs framework costs

As with startup costs, application and framework developers must both write performant code for the application to have good interframe performance. Even the leanest of frameworks will miss frames if the application developer performs expensive computations on the same thread as the framework, and even the most efficient of developers will struggle to make a silky smooth application if the framework is unable to render anything at 60Hz.

Providing clear affordances for developers to hook into for expensive computations (for example, trivial one-line APIs for offloading work to another thread) allows the framework to steer developers in the right direction.

Pipeline latency

One approach for reducing interframe performance is to split the work and run it in parallel. A typical approach is to split layout computation and rasterization (this is convenient because layout is often designed to be single-threaded, easily executed on a single CPU core, while rasterization today can often be expressed in terms of pixel shaders, naturally fitting the GPU model). In principle, one can maintain a 120Hz refresh rate while still taking 8ms for layout and 8ms for rasterization, by running the layout for frame N+1 on the CPU while the rasterization for frame N is

[83] An early, but formative, experience when developing Flutter's framework involved a similar situation. The framework cached a certain computation. The caching logic was quite complicated and bug-prone, and so this document's author decided to rewrite it. As the first step of this rewrite, the optimization was first removed, to establish a baseline performance against which to evaluate the final solution. It was at this point that it was discovered that the existing caching optimization, in addition to being bug prone, was also in fact slower than not having it at all. In the end, the cache was removed and not replaced, thus fixing the bugs and improving the performance by merely deleting code.

being executed on the GPU. Each frame takes 16ms to render, but there is still one frame being rendered every 8ms.

This leads to a different problem, however: that of a *high latency*. This is noticeable by the user primarily when they are interacting with the application. A user dragging a UI component across the screen will notice that the latency is 16ms even if the actual refresh rate is 8ms[84]. If a UI control is designed so that interacting with the control immediately changes its rendering (e.g. right-clicking a UI element to show its context menu, or click a text field to position the cursor), the user expectation is that the change will happen on the frame that immediately follows the interaction, not one or more frames later.

As a result, pipelining or parallelizing the work of the UI framework, while better than nothing for dealing with high loads, is not a solution that a framework can reasonably rely on in normal operation.

Designing for performance

Limiting scope

Higher performance can often be achieved by sacrificing flexibility (contrast this with the next section, which discusses maximizing the range of possible display effects). A framework that exposes a finite set of layout or rendering primitives can be optimized for those specific features at the expense of all else. This approach remains true even when the range of supported features is large.

For example, the web stack has a finite (though now quite large) set of layout primitives (mainly centered around the 'display' property in CSS). An alternative approach could have been to offer a JavaScript API via which web pages could achieve *any* layout, by putting the developer's JavaScript code in the layout loop. However, this would have made overall layout on the web much slower, because switching from the browser's C++ internals to JavaScript and back for each element's layout would be much more computationally expensive than staying entirely within the C++ internals with a finite set of layout primitives[85].

Another example of this approach is Apple's CoreAnimation, which provides primitives from which animations can be built. The provided animations can be executed very efficiently, but if a developer desires an animation that is not provided, they must instead build it themselves using less efficient primitives.

A design such as Flutter's instead trades peak performance for maximum flexibility. For example, all Flutter animations use the same basic primitives, whether they are animations shipped with the framework or bespoke animations created by the developer. While these primitives are designed to be high-performance, in principle a framework that only supported a subset of these animations could achieve even better performance.

---

[84] Real latency is higher (50-100ms) due to other factors such as the time it takes for physical touches to be recognized and transmitted to the framework. Microsoft Research argued in 2012 that there is a user experience cliff around 1ms where the latency stops being noticeable. Current hardware is still far from being able to achieve that level of responsiveness.

[85] Also, HTML layout predates CSS, and CSS predates JavaScript, so there would have been additional chronological issues with that design.

Tooling

Tools can help developers optimize their applications. Without tooling, it can be very difficult to even diagnose performance issues (much more so than logic issues, for which basic "print" debugging can be sufficient[86]). Two tools in particular are key. In practice all frameworks need these tools, but they are even more critical if the framework's core goal is to enable high-performance applications. They are also the primary tools that framework developers will use to troubleshoot performance issues in the framework.

Profiler

Many toolchains have a profiler. A profiler tracks program execution (either by literally counting every instruction executed, or by sampling execution over time), and reports which lines of code are most invoked or take the most time, as well as the call graph leading to those lines of code. This allows the developer to focus their attention on the parts of the program that are taking the most time.

This is primarily useful for tracking down the reason computationally intensive algorithms are slow. For example, if a developer expects an algorithm to visit every node of a large tree once, but they have accidentally used an $O(N^3)$ algorithm instead, then the profiler output will make this clear because the salient functions will be shown to have been called clearly many more times than expected.

Timeline inspector

To obtain a broader situational awareness of the performance of an application, a timeline inspector is more suitable. A timeline is created by instrumenting the code to report when certain phases of logic are starting and ending; a timeline inspector then allows the developer to examine the output of that instrumentation to determine what the application is doing.

For example, by instrumenting the start and end of the "layout" phase of a framework, and then separately the start and end of the "paint" phase of the framework, the developer can quickly determine which phase is taking the most time. Similarly, if each widget's logic is individually measured, the developer can look at the timeline and determine if a particular widget is taking more time than expected.

This is useful, in contrast with a profiler, when comparing different configurations of calls to the same API. For example, every widget likely uses the same core parts of the framework, so a profiler would say those core parts are the most expensive, without necessarily making it easy to determine that one particular widget always takes longer to run those same core parts than any other. The timeline, on the other hand, would clearly show the relative times for each widget.

---

[86] Because of I/O itself being slow, logging tends to be an inadequate tool to track down performance problems: the act of observing itself changes the performance characteristics too much.

Potential

Making an extremely fast UI framework is simplest if the feature set is kept small. The most performant UI frameworks are probably not widely known: they are likely to be small bespoke frameworks created for a single purpose, maybe for the interface of some embedded hardware[87].

Making the fastest UI framework while still supporting the kinds of interfaces that users and developers expect is a much harder problem. That said, should one truly want to create a framework that minimizes time to interactive and maximizes the potential refresh rate, in the context of a well-defined set of required features, and in an environment where developer convenience is not a priority, it should be possible to create a competitive framework that achieves these goals.

Evaluating success

In many ways, this is the simplest goal to evaluate. One must create benchmarks, then run them in controlled conditions on test hardware, and chart the results over time.

In practice there are several difficulties.

Deciding which benchmarks to write and run

Running benchmarks is costly: it takes literal time (to get valid results, benchmarks must run long enough to avoid edge effects in hardware performance, which usually means multiple seconds per test) and literal space (to get valid results, benchmarks must run on literal hardware equivalent to what users will experience, which usually means at least 8000 cubic centimeters per device once thermal needs are taken into account).

Because of this, one must be much more judicious in one's choice of benchmarks than one would be when it comes to writing unit tests (where a single test might run in less than one millisecond and where, therefore, thousands of bad tests only waste a second or two of time per test run).

The best tests are those that measure the performance of the framework rendering interfaces that are actually experienced by users, in somewhat realistic conditions, while also being sensitive to the parts of the codebase most commonly experiencing regressions.

Noise

Tests that are realistic and yet provide stable numbers (meaning that running the same test on the same hardware returns the same data) can be difficult to create. Obviously, other software running on the test hardware (notably background processes of the operating system) must be kept to an absolute minimum, but this is not the only source of noise. Tests can be sensitive to the slightest of differences. A test that happens to include the date in its rendering and triggers a line of text to wrap on "Wednesday" but not on the shorter "Friday" might have detectably worse performance when the text-wrapping code is triggered, but this is unlikely to be noticed as anything other than noise in the results. Therefore, discipline in the creation of performance tests is critical.

---

[87] This is not to say that all such dedicated frameworks are fast, quite the contrary. Writing a UI framework is hard work, so most of them are mediocre on all axes!

Maintaining the lab

The other big difficulty is that hardware is unreliable, and running any useful quantity of devices will require full-time maintenance. The kinds of problems encountered when dealing with hundreds of physical phones are surprising, sometimes hard to debug, and often require direct physical manipulation[88].

Longitudinal stability

Ideally, benchmark numbers would be comparable over years. In practice this is a difficult target to achieve. Individual devices have different performance characteristics even when purchased in a batch, so numbers may only be strictly comparable when run on the same device. Devices have different performance characteristics over time, so what appears to be a regression may merely be hardware degradation. Device performance varies based on the operating system running. Performance can vary based on atmospheric conditions (e.g. temperature, humidity, pressure).

Backfilling data can mitigate some of these issues (running tests from older versions of the framework interleaved with running the latest code), but is usually impractical due to the cost (generally, getting enough capacity to keep up with development velocity is already a challenge).

[88] For example, the Flutter team once noticed a test had appeared to regress, only to find it was because the phone was showing a Wireless Emergency Alert and application performance drops significantly on Android when such an alert is showing. Another notable case involved a device that worked 99% of the time but very occasionally failed in obscure ways; it was eventually discovered that the cause was a very slightly damaged USB cable. One possibly apocryphal case where performance on one test device seemed erratic was claimed to have been traced to an insect occasionally resting on the device's touchscreen, causing the CPU to switch to a higher speed (phones do this to appear more responsive when the user is interacting with the device, while keeping power consumption at more reasonable levels the rest of the time). Another definitely *not* apocryphal maintenance item occurred when it was discovered that a whole batch of devices had all had their batteries physically fail simultaneously (swelling), which in addition to causing physical problems in the device stands, was also a significant fire risk! See also the discussion of testing infrastructure later in the document.

Goal: Maximizing the range of possible display effects

Graphical user interface libraries are used for four kinds of applications: serious software for getting things done, software whose goal is to encourage humans to do things differently (gamified tools), games, and demos to show how awesome the library is. Three of these categories benefit significantly from being able to show pretty pictures, animations, and other cool effects.

One can design a UI framework to provide a catalog of off-the-shelf effects. This technique is especially useful for frameworks that are designed in such a way that the main manner in which it is used does not lend itself to high-bandwidth or low-latency computation. For example, while modern hardware has enabled it to be used for some impressive effects, JavaScript is not fundamentally well suited for implementing complicated graphical effects and animations. This has encouraged the designers of web standards to provide a broad range of graphical effects in the declarative CSS layer.

This approach, however, is fundamentally limiting. Once one goes outside the catalog of pre-canned effects, anything else must be implemented using that "main manner", so effectively one is reduced to two categories of effects: those that are built into the framework, and those that are achievable even in a low-bandwidth or high-latency environment.

To maximize the range of achievable effects, therefore, one must provide a high-bandwidth, low-latency environment in which to implement these effects. If one architects the framework such that the mechanism used for this is the very same mechanism that the framework uses to implement its own out-of-the-box effects, there are two important benefits that result: one is that any performance improvements made to the framework's architecture will also improve effects written by users of the framework; the other is that when a limitation is found that prevents an effect from being implemented in the framework, fixing that limitation will also unblock users of the framework in the same way.

Types of effects

Layout

Some of the more practical display effects are actually layout. Centering something in the middle of the display, distributing controls along a row or column, pinning a header while scrolling a list, rendering controls in a grid, showing a tooltip or menu overlapping other controls but positioned relative to a specific control, rendering controls in a toolbar with an overflow button so that controls that don't fit in the toolbar instead go into a menu... all these effects require control over the layout.

Typography

Text rendering is an area filled with potential effects. For example, choosing a font, changing the text color, outlining the text in a different color, or animating the text along OpenType Font Variations axes. These effects must also be combined with basic text rendering requirements such

as implementation of the Unicode bidirectional algorithm, rendering color emoji, text justification, and text shaping, as discussed in the background section.

2D effects

Common effects for 2D user interfaces include rounded edges, semi-transparency, translucency (frosted glass effects), and shadows. It's also common for elements to be transformed in some way, e.g. rotated or scaled. Implementing these effects typically involves some form of antialiasing, non-linear clipping, and blurring. Other effects of a similar nature include applying Porter-Duff compositing operators and other blend modes, and gradients (linear, conical, radial).

While traditionally these effects were rendered on the CPU, in practice, with modern hardware, most if not all of these effects can be most efficiently achieved directly on a GPU using shaders. By providing an easy to use affordance for applying shaders to parts of the scene, one can enable an absurdly wide range of effects to be applied.

3D effects

Finally there are 3D effects. These come in two flavors. The first is applying 3D transformations to 2D surfaces, for example, being able to flip a UI card around. This is merely an extension of rotation and scaling, in the sense that 2D transformations can be expressed as 3x3 matrices while 3D transformations can be expressed as 4x4 matrices.

The second category of 3D effects useful in user interface frameworks is the ability to insert 3D assets into the 2D scene, for example 3D avatars, or 3D models of products that the user can examine from different angles.

In principle there is a third category, which is embedding a full 3D scene, as in a game or geospatial application, into the interface. These interfaces are considered out of scope for this document.

Potential

A framework that allows developers to build layout and rendering primitives in the same manner as the framework's built-in features, and that exposes affordances for shaders, should be able to achieve more or less any 2D display effect possible with modern hardware. With such a design, the main limiting factor should be ease of use of the framework, rather than performance or features.

Evaluating success

Evaluating how successful one has been in maximizing the range of possible display effects is more difficult than measuring the success of the other goals discussed. One cannot create a benchmark or run a usability study to quantify the number of possible effects. One cannot even run a competition to see what effects people can create, because demoscene developers thrive on creating incredible effects under absurdly overconstrained conditions, and what a demoscene developer can do tells you little about what a normal developer would find practical.

One metric could be obtained by examining reported issues and feature requests, but even this can be easily misleading, as more popular projects get more bug reports, which means that in practice

the quantity of issues can be correlated with both success (more users means more reports) and failure (more problems means more reports).

Goal: Minimizing power consumption

In many ways power consumption is equivalent to performance, in that doing less work uses less energy and is faster, while doing more work uses more energy and is slower. However, the goal of minimizing power consumption is even less forgiving than performance. Tricks like pipelining or deferring work until after the frame may help performance in some cases but they are no help in reducing power consumption (and may indeed make things worse overall, if there is more net work to do in the deferred regime). Techniques like using multiple cores may similarly make power consumption worse overall.

Using power also results in heat (the energy used in computation must go somewhere). On small devices (e.g. laptops, phones), increased temperature can result in thermal throttling, where the device forcibly reduces its own power consumption by reducing its speed (e.g. stepping down CPU cores). Avoiding excesses of power consumption therefore correlates with maximizing performance in multiple ways.

Power consumption is worsened by any display effects that increase the workload; animations, notably, but also graphical effects that require more computation, or require more hardware. For example, a screen with only text and buttons can be rendered in software using only a CPU with any graphical acceleration hardware powered off, whereas a scene that involves computing 3D geometry on a GPU typically requires both the CPU and GPU to be powered. (On some hardware, even the choice of UI color can affect power usage[89].)

In general, many ways in which display effects are improved directly consume more power. For example, increasing an animation's frame rate by running the device at 120Hz instead of 60Hz will require twice the computation.

[89] For example, research by Purdue University in 2021 showed that OLED screens use less power to render black than white.

In many ways, this goal is therefore quite misaligned with some of the others discussed above. In trying to minimize power consumption, one is pushed to decisions that are likely to minimize the range of possible display effects, reduce the amount of work that can be done per frame, and avoid developer conveniences like garbage collection.

Design

Extreme

Creating a UI framework with no constraint other than minimizing power consumption is relatively straightforward; one need only minimize the power-consuming features. This is essentially any feature that requires additional CPU (or GPU) cycles, for example:

- Animations, or any logic that requires painting more than one frame per user interaction.
- Language features like garbage collection that do work at runtime that the developer could do during development instead.
- Immediate-mode UI APIs, which typically require more overhead than retained-mode APIs, especially in an environment without animations.
- Graphical features requiring compositing (which involves copying large chunks of memory).
- Graphical features that require extensive computation, such as blurs (including shadows and glass effects), curved edges, non-axis-aligned clips, or anything requiring antialiasing.

Similarly, keeping features simple avoids CPU cycles. Supporting only one font means that only one font need be decoded. Avoiding images means images don't need to be decoded.

Taken to an extreme, one could restrict oneself to console (text-mode) support only.

Moderate

If we assume that the UI framework must at least enable fashionable interfaces while maintaining low power consumption, we may need to compromise on these restrictions. For example, supporting multiple fonts, images, curved shapes, and effects that use blurs or effects that use shaders and compositing. Designing such a framework requires a careful inventory of the specific features that are desired; those limitations then drive everything in the design. For example, if specific animations are to be supported, then the framework can be designed around those animations while avoiding the overhead of a general-purpose design that can animate anything.

Potential

Existing frameworks do not, on the whole, prioritize minimizing power consumption over other design goals. As a result, there is a huge potential for creating a modern UI framework that does make this a priority (in the sense that it would be relatively easy to become the *de facto* leading framework with that goal).

However, the market for such a product is probably relatively limited. While it could be useful for developers creating small embedded battery-powered systems, in most other environments minimizing the power consumption of the UI framework is not seen as a goal worth pursuing if it compromises on performance, modern display effects, or even developer productivity.

Evaluating success

To track progress towards a power consumption goal, the only effective solution is to measure actual power usage on real hardware, and chart it over time[90]. This requires creating benchmarks as features are created, and either maintaining a physical laboratory with test devices with power probes (or contracting a third party to perform this task). This is essentially the same work as is needed for evaluating the performance goal as discussed earlier (with the same complications), but with the additional complexity of physically probing the hardware to measure the power consumption.

Design choices

There are a number of choices that impact the core design of a UI framework. Some of these choices involve objective trade-offs, e.g. deciding between performance and ease of use. Some of these are more subjective. In this section, we will cover some of these choices, in no particular order.

Everything is a designable surface[91]. Implicit decisions are still decisions, and every decision impacts the developer experience. Some of the most impactful can be those that nobody has ever considered questioning before. It is therefore important to not fall into the pattern of doing things because that's how they have always been done. Every other UI framework has a build system, but that does not mean that a build system is necessary. Similarly, it is important not to assume that different is better. Every UI framework has a build system, but that does not mean that a build system is bad. Conscientious and deliberate choices are the key to a high-quality product.

As the reader considers the trade-offs discussed here, it is important to bear in mind that every trade-off is a choice between two groups of users, to decide which group is happy and which group is sad. The implications of this are discussed later, in the section on team culture.

[90] See the discussion of testing infrastructure later in the document.
[91] Credit to immersive experience designer and media analyst Johanna Koljonen for popularizing this concept.

Usability testing

It is tempting to think of API and language design as an art, and indeed in many ways historically that has been how both have been approached. However, that approach discards the single most valuable tool that an engineer can apply to design: testing. Specifically, usability testing.

Every decision can be evaluated. Techniques for doing so cover a wide range, with different methodologies being appropriate for different kinds of decisions. The value of this kind of testing (combined with an API or language designer who has great foresight and taste, and most importantly, a willingness to be repeatedly proved wrong by testing) cannot be overstated.

Usability testing is relatively cheap; a single usability engineer working full time[92] can support the needs of a dozen engineers[93]. However, it requires sincere buy-in from the development team and leadership. Usability testing is pointless if it is not taken seriously.

Investing heavily in usability testing for API and language design is an example of a "brave" choice: one that, in the author's opinion, would have a high pay-off, but that will face a lot of resistance and even mockery from the wider industry community. What is clear, however, is that having more information, more data, can only help designers make better decisions. Even a single usability test with a single participant moves the needle from *zero* data to *some* data.

Usability testing methodologies

Moderated, in-person, hands-on use of a live prototype

A researcher sits with a participant and introduces them to the prototype software. They provide them with a task to achieve, and then carefully watch the participant as they attempt the task with the prototype, prompting them to speak their thoughts and experiences as they do so. Meanwhile, the engineering team watches remotely (or from behind one-way glass)[94], taking notes on stumbling blocks, matters of confusion, or bugs that the participant experiences. If necessary, the engineering team can privately communicate with the researcher in real time to suggest different

---

[92] They could be members of the team, or they could be independent third parties. It is important, however, that they be objective, detached from any particular outcome of their research, and feel no pressure from the team to provide a particular result.

[93] Usability research participants are typically paid a stipend, but this is a pretty minimal cost compared to the other capital costs of building a UI framework (such as building and maintaining a test lab), let alone the cost of employing the team. The opportunity cost of stopping parts of the engineering team to watch a usability study seems like it would be expensive, but the value generated by these studies vastly outweighs any lost productivity. As the old adage goes, if you can't afford to do it right, how can you afford to do it twice?

[94] Such sessions are commonly recorded, and such recordings can be useful in the context of a well-funded usability research team (e.g. they can re-examine studies to find common themes over a longer time period, and can use snippets of recordings to support their points in presentations). In principle, recordings also allow engineers to time-shift their experience of a particular study. However, the value of watching such recordings pales in comparison to having engineers watch such sessions in real time. It is too easy to dismiss content that can be paused and fast-forwarded. There is no substitute for being forced to sit through a user's painful experience of one's product. Separating the engineering team from the participant is important, however, for psychological reasons. The participant must feel that the researcher is a neutral and objective observer (it is the researcher's responsibility to engender this relationship at the start of the session), and they must feel absolutely free to criticize the product and stumble without embarrassment (they must feel that the *product* is being tested, not them, and they must not feel anxious that they might offend the engineering team by making mistakes or raising concerns). It would be very difficult for most participants to feel comfortable if they could see the engineering team watching their every move. In practice, even if they are told that the session is being recorded and watched remotely in real time, a good researcher can make most participants feel comfortable and open to providing unfiltered feedback if they are physically alone in the research lab.

questions or areas to explore. A post-experiment interview can be used to collect final thoughts and to probe further into areas of interest.

This is the most visceral form of user feedback possible, and such studies should be performed continually throughout the development process of any software, especially development tools[95]. It is very difficult to lie to oneself about the quality of one's product when one is literally watching a user fail to understand what one has created.

A single test protocol can be used with a dozen participants over the course of a month, collecting actionable issues that can take multiple quarters for the engineering team to resolve. (In practice, most common issues will be evident within the first 5 or so participants using any particular test protocol[96].)

Because engineers need to commit time to watching a usability study in real time to get the most out of a session, it can be disruptive if the team culture does not prioritize testing, or if the tests happen too frequently. The ideal setup is to have a regular time slot for studies, at most weekly, with the engineering team paying attention to the study together. Whenever the team feels that a particular test protocol has run its course (meaning that each run is bringing diminishing returns in terms of new insights), the team should move on to a new test protocol. It is critical to ensure that everyone is finding value in these tests, otherwise one is wasting both the opportunity and the good will of the team.

Remote use of a prototype

Similar to in-person hands-on use of a prototype, a user can participate remotely, either with a researcher acting as facilitator (with the participant sharing their screen, which is then recorded), or working independently (recording their screen and local video). This is a practical alternative to in-person interviews for getting a more diverse set of participants.

Observation of developers creating real applications

Once the UI framework is barely usable, having developers use it for real production software is the ultimate test. Having members of the *framework's* development team embed into that *application's* development team can be a very good way to give the framework's team insights into the real needs of their audience. This is especially valuable when the applications and the framework can evolve together, so that framework developers embedded in the application's team can change both together and experience their new APIs essentially in real time (or even better, e.g. by running the application against their development version of the framework on their workstation).

Usability researchers can also provide valuable insights by observing developers using the product "in the wild" and reporting their aggregate findings to the team.

---

[95] This is very rarely done in practice, which is a terrible loss to our industry. Usability testing was a huge factor in Flutter's early design, and several key decisions can be directly attributed to insights obtained through usability testing.

[96] Assuming relatively homogeneous participants, and with the caveat that this is a very broad generalization. Testing with a diverse set of participants will find more problems and is valuable. The number 5 is derived from research in 1993 by Jakob Nielsen, *et al*, more recently reiterated in this blog post. In practice, testing with *anyone* is the first hurdle, as having any data is infinitely more informative than zero data. Each subsequent participant reduces the risk that the participants so far were outliers (if your only participant is not representative of your general target audience, there is a danger of making decisions that take the product in the wrong direction, just like basing decisions on pure intuition).

Migrating framework customers

Once a framework is being used by many developers, it becomes intractable to make some API changes, even if they have been shown to be massively beneficial, because the cost of each developer migrating is so high that one risks losing users[97] and establishing a reputation for being "unstable", which will further discourage new users.

This cost can be mitigated by having the framework team be responsible for making these migrations. This is viable when the number of customers is small, and the framework team can manually migrate each customer directly.

It may also be possible in some situations where the migration is mechanical (e.g. renaming an API). However, it is easy to forget that even apparently trivially-automatable changes will run into difficulties. For example, maybe the API in question is used by a third-party dependency that hasn't migrated, or whose migration is not compatible with some other third-party dependency. Maybe the API in question is in code that is being generated by some other tooling. Maybe there are some situations where the migration does not apply cleanly (e.g. the new API uses a name that conflicts with code in one application).

Developer surveys (quantitative and qualitative)

A survey is developed to study a specific question, and a large number of potential users are canvassed to obtain results. This data can then be analyzed to determine answers to specific questions, and to track metrics over time (e.g. general satisfaction with the product). Market research can also be performed in this way.

It is possible to test language design features using surveys[98], though doing so is very tricky. The difficulty is designing test protocols that are not susceptible to confirmation bias. As language designers, we usually have our own preferred answer to any design question. The art is to find ways of asking questions where one is confident that different answers would result in different conclusions.

For example, suppose one were trying to decide whether to use abbreviations for keywords or to fully spell them out (e.g. `fn` vs `function`). One could ask users of an existing language that uses short keywords whether short keywords make them more productive. The problem with such a question is that regardless of the answer, it can be interpreted either way. If 75% of users say yes, then either "clearly they are biased by their experience and 25% of users know the real answer is no" or "we should clearly abbreviate since 75% of users are more productive that way". If 75% of users say no, then "clearly they don't recognize their own productivity gains, the fact that they continue to use that language means the real answer is yes" or "we should clearly avoid abbreviating since 75% of users are less productive with abbreviations".

To avoid these problems, first, one should establish how one would respond to different results *before* distributing the survey. Second, one should use questions that are less subjective and more practical. For example, one could provide different code samples using various possible syntaxes,

---

[97] "I'm fed up with having to spend all my time fixing my code for this framework every time I upgrade, I'm going to use this other framework instead!"

[98] For example, this study examines an API design for Dart where the team administered an unmoderated code comprehension test using web-based survey software, with the results being used to inform the design of Dart's new string manipulation API. (The present document's author was tangentially involved in this study design.)

and ask questions of the survey participants. Combining the precision (how many participants correctly interpret the code) and the performance (how quickly they answer these questions) can provide real data on whether each style is misleading (many wrong answers), clear (mostly correct answers), confusing (answers take a long time to come in), etc.

In general, being genuinely open to making decisions that differ from one's own biases is very difficult, but it is critical if one is to create a truly good product. Actively finding questions that have possible answers that would *actually* change one's mind is critical.

Paper prototypes

Similar to online surveys but with a hands-on approach, this is a technique useful especially in early design phases. When applied to programming languages it can take the form of putting code snippets together as if it were magnetic poetry, or examining code snippets and answering interview questions (but where the goal is to test the code, not the interviewee). As with surveys, good test design is critical to avoiding confirmation bias. This form of testing can be useful to take into the field, e.g. at a conference.

Market research

This is not usability testing, but can complement it: asking users directly for their opinions, e.g. regarding what features they personally feel is most lacking, asking them to prioritize paint points, or learning about the demographics of users and potential users. This kind of research can then help the usability research team focus their efforts, e.g. to evaluate possible approaches to addressing user requests.

Asking for user opinions in this way requires carefully crafting questions and proportionally sampling the population. Doing this rigorously is important to get an accurate picture. (For the same reason, merely asking usability research participants for their opinions is *not* an appropriate way to collect this information; usability testing is for testing usability, not market research.)

Risks

The danger of overfitting

When designing developer products based on usability studies, there is a risk of over-interpreting individual events and losing sight of the overall picture. For example, if one test subject is especially experienced with Lisp and resonates well with parts of the framework design that reminds them of Lisp, it is worth bringing in a few more study participants before pivoting the entire framework around a Lisp-based system.

Always consider how an API or feature fits into the overall architecture, and what the overall experience of developers using the framework will be. A framework can introduce new patterns that developers will learn which will help them understand the framework better than relying on existing patterns; studies should be designed to take into account the potential for an initial learning curve and find ways to tease out the difference between complexity that helps developers be more productive, and complexity to which merely developers adapt.

The danger of uncritical application

It is important to thoughtfully interpret the results. When a participant acts in an unexpected way, or uses an API in a surprising manner, the team should spend time attempting to understand *why* they may have done so. Post-study conferences where the usability researchers and framework leads and relevant team members discuss the results and attempt to reach plausible explanations of the participants' behaviors (based both on their self-reported thinking, the observations, and other data collected from other studies) are important before designing improvements.

This is the main source of value from these studies, and it is much more important than summary metrics such as how fast participants are achieving the given tasks, the overall success rates, and aggregate reported satisfaction scores. A team should avoid falling into rote patterns of executing usability tests, summarising the data, and reporting them as quarterly results, ignoring the human element and losing the main purpose of doing such research in the first place.

The danger of confirmation bias

When designing an API, one often has personal preferences that guide the design. Often, these preferences are based on intuition and past experience for what is effective and what kinds of complexities one should avoid. When *testing* these APIs, it is important to find ways to capture whether these preferences are correct or not. It is very easy to see a test participant struggle with one feature and find a second feature easy, and to conclude that a design feature common to both is good because the participant had no problem with the second feature.

Looking explicitly for ways to disprove one's preferences helps one make best use of the invaluable resource that is a usability research team.

The danger of complacency

It is very easy for a team to begin to believe that they understand the needs of their customers better than their customers (meaning, anyone using the framework), or that they are sufficiently aware of their needs that they no longer need to work closely with those customers.

Especially as a team grows, it can become easy for team members to feel like cogs in a machine[99], who do not need to ever use the product themselves to work on it. Similarly, leaders of a developer product can begin to feel as if their intuition is infallible.

Dogfooding the product (i.e. writing one's own applications) should be standard practice for everyone working on a developer product[100]. Usability studies should be standard practice at the start of a project and still be standard practice when the project has created a firmly established product.

Fostering a culture of testing designs

It is easy for team leads, who are nominally paid for their experience and opinions, to see usability testing as a threat; after all, it could show that their ideas were wrong.

---

[99] This feeling may even be accurate. In a large team, unless the leadership makes a concerted effort to focus the team culture towards empowering all team members, an engineer may *de facto* have no real ability to affect the kind of change that would be necessary to improve the product holistically.

[100] Or really any product, but that's out of scope for this document. See also the discussion on dogfooding later in this document.

It is therefore critically important to foster a culture wherein being wrong is accepted, where there is no shame in having one's experience and opinions questioned, and where changing one's mind in the face of new data is not only accepted but normalized. The earlier this culture can be established (if possible, from day 1), the more value can be pulled from the research.

This must include insulating the usability research team from consequences of the results they report, so that they never consciously or unconsciously bias their results towards the stated opinions of the engineering leads. It must also include a sincere and genuine desire and commitment from the team, and especially the team leads, to use the results of research objectively, even when they lead in an unexpected direction.

Composition

UI frameworks provide primitives that developers use to create UIs. For example, Qt has a QPushButton class, Delphi has a TButton class, HTML has a <button> element. These all represent the basic concept of a clickable button control.

Some UI frameworks additionally provide a *composition* mechanism, wherein one combines such primitives into reusable components that themselves act as primitives. HTML has Custom Elements, Flutter has Stateless and Stateful Widgets, React has Components. For Flutter and React in particular, these are fundamental to the way the frameworks are used (the developer must create, at the very minimum, a widget or component to represent the application itself).

The superficial benefit of composition is that it allows for the simple creation of reusable components, thus enabling easier code reuse. Interestingly, however, if composition is implemented well, it enables the primitives themselves to be dramatically simplified.

For example, instead of having a button primitive, Flutter's buttons are themselves composed from simpler primitives: a primitive that handles gesture handling (to handle taps on the button), a set of primitives to handle the rendering of the button, a primitive that handles focus management, and so forth.

When considering composition, therefore, there are two important questions: first, whether to offer composition as a feature at all, and second, whether to lean into this mechanism, reducing the primitives to very simple components and having common widgets themselves be built using composition.

The main advantage of having simple primitives is that compartmentalizing features reduces complexity, which reduces the likelihood of bugs. However, all else being equal, composition does tend to use more memory than having dedicated high-complexity primitives.

Having simple primitives and then building a library of controls using those primitives also helps ensure that the primitives are high-quality and enables developers to make new kinds of controls more easily than having to build them from scratch.

On the whole, unless memory consumption is the highest priority, the author's recommendation would be to embrace composition and use it heavily in creating the controls that ship with the framework. The benefits of flexibility and expressiveness, and the tendency for such systems to have fewer, more manageable bugs than the alternative "heavy primitives" approach, plays into making the framework more approachable.

Type-agnosticism in composition

It can be tempting to create components that are specialized around specific child components. For example, a button might have an "icon" child, and might expect specifically that the type of that child be that of the icon component. Similarly, a dialog box might have a list of buttons, and might specifically require that they have that type.

This is a mistake. The framework, and each component in the framework, will be infinitely more flexible and powerful if *any* component can be used in *any* place where a child is expected. For example, this allows the button to show an icon... or show an animated progress indicator, or an image, or a Unicode symbol, or any number of other possible components that would have been artificially excluded if the component had expected a specific type of child.

Similarly, any component that would usually expect a string can be made more powerful by expecting any arbitrary component type as a child, and recommending the use of a common text component used specifically for rendering text. Essentially, other than the specific components that are exclusively for rendering text, no other component should ever have a parameter that only allows text where allowing any component would work.

Similarly, components should never check for the type of a child widget. As a rule, it should always be possible to substitute a component with one that does nothing but wrap that component.

Immediate-mode UI vs retained-mode UI

(Not to be confused with immediate-mode vs retained-mode graphics, which will be discussed later.)

Pure retained-mode UI

The traditional model for building an application UI is for the application to create a tree of controls, configured to represent the initial state of the interface, and then for the application to mutate the configuration of this tree to represent changes in the underlying data as the user manipulates it.

There are many approaches to this. HTML has a declarative markup language that describes the initial tree and its configuration, and a scripting language and API (JS and the DOM) to manipulate it. Alternatively, one can create the tree entirely in code. Manipulating the state can be done via an object-oriented API, or via "patches" that are applied to the tree dynamically.

The key, however, is that in this model there are two completely separate paths for specifying the UI: an initial *creation*, and subsequent *modifications*.

In general, this approach minimizes the amount of work that is required to update the UI, especially in terms of working memory. No data structures need be created to update the interface, the existing data structures can be manipulated in-place. It also makes it relatively easy to implement animations and other features that are *stateful*. A text field with a text caret showing the current location of data entry can trivially blink its caret by updating its internal state on a timer. A progress bar can be directly manipulated to show the state of some computation.

The difficulty with this approach, however, is that it is bug-prone. It is very easy, when modifying code written in this form, to update one path and forget to update the other, thus leading to

situations where interfaces display different information based on whether some data was present when initially creating the UI vs that data being added when the UI is already visible. It is also easy to introduce bugs where the UI is updated in response to some user actions but not others[101].

Pure immediate-mode UI

To remove the danger of having two code paths to creating the UI, one can instead just have one code path, the *creation* code path. Each time the underlying data changes in some way that requires the interface to update, the entire UI is recreated from scratch to represent the new data.

This trivially removes any risk that different ways of interacting with the UI will have different results. If there is only one way to describe the UI, it will always be consistent.

This approach is especially appropriate for non-interactive user interfaces, where no UI state needs to be maintained across frames. It is more difficult when the UI must be interactive, e.g. if the framework must maintain some idea of the layout for handling pointer-based or touch-based input (the geometry being necessary for hit testing). It is also less than ideal, from a performance perspective, if some small part of the UI is expected to animate while the rest of the UI stays static. For example, users generally expect the total overhead from blinking a text caret in a text field to be *de minimis*[102].

Immediate-mode UI API with a retained-mode layout backend

A hybrid approach is appropriate for interactive UIs. In this design, the API for describing the UI continues to be immediate-mode, meaning that there is a single mechanism for describing the UI which is used both on initial creation and subsequently for updates. The framework, however, creates and maintains a model of the interface. Conceptually, the model described using the immediate-mode API is compared by the framework to the retained model, and the retained model is updated dynamically to match.

This design has the advantages of a pure immediate-mode UI, while maintaining the ability for the interface to have long-lived internal state.

One disadvantage is that an immediate-mode API is memory intensive compared to a pure retained-mode model. This can be somewhat mitigated using a memory allocator optimized for cheaper allocation (in exchange for greater fragmentation, or needing a garbage collector that can move longer-lived allocations to defragment the memory), or using an arena specifically for the elements used in the API, but will never be as cheap as doing nothing.

The other potential disadvantage is that walking the entire tree anytime anything is updated is suboptimal. This can be mitigated by designing the framework to allow fragments to be updated in isolation.

When combined with an architecture that has a first-class composition mechanism, each composite control can be described using the immediate-mode API. This, with a mechanism to allow fragments to be updated in isolation, allows animations (such as blinking a text caret) to be performed with relatively little overhead compared to a full pure immediate-mode UI.

---

[101] This is especially relevant when the same conceptual interface command can be given using a menu, a button, and a keyboard shortcut, for example.
[102] See also the discussion on this topic later in this document.

This model is sometimes referred to as the "React-style paradigm", as it was first popularised by React. Popular modern UI frameworks such as Flutter, React Native, SwiftUI, and Compose, as well as many smaller newer libraries such as Clay, and many web frameworks, all use some version of this model.

There are several variants of this approach.

Separate approaches for layout and UI

The approach taken by React and React Native is to separate the layout logic from the UI logic. The UI is immediate-mode, but the layout itself is described using declarative styling (CSS or CSS-like).

The author would not recommend this strategy as it increases the number of technologies that the developer must learn to use the system. It also tends to require a render object system with more elaborate primitives, which is more complicated and thus more bug-prone.

Uniform approach for layout and UI, layered over a separate render object backend

In the simplest version of this model, the same mechanism is used in the UI layer to describe layout components as high-level UI controls. This works very well with a composition-first strategy and with simple render objects[103].

Overall, when optimizing for popularity (and thus the developer experience), this approach is the state of the art. Some more detail on a possible implementation approach is discussed later.

Merged immediate-mode UI and retained-mode layout backend

The disadvantage of separating the layout and immediate-mode UI layers is that creating layout-dependant UI choices can only be done during layout (for example, a toolbar with buttons where buttons that don't fit overflow into a drop-down menu requires knowing the size of the buttons and the toolbar when building the UI).

If the layers are separated, this can be solved by having some render objects defined at the UI layer that are aware of the UI system and can thus interleave the immediate-mode logic and the layout logic[104].

An alternative would be to merge the two layers, so that the layout and "build" (immediate-mode logic) happen as a single coordinated walk. This would have the advantage of exposing geometry more naturally to the immediate-mode logic, at the cost of a slightly more complicated overall algorithm (caused by having less separation of concerns).

This approach deserves further study.

Trees

As discussed earlier, trees are a common data structure in a UI framework. There are various ways to represent trees, with different trade-offs.

---

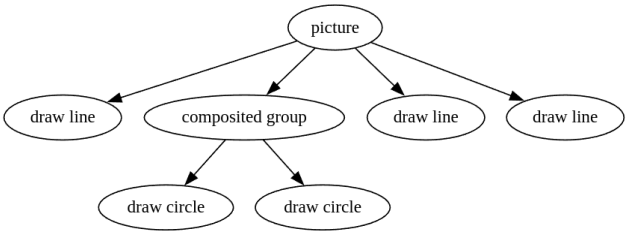
[103] See the discussion later in this section.
[104] This is the approach taken by Flutter for widgets like LayoutBuilder or SliverList.

Modeling trees

The most common approach is with pointers or references[105], where instances of nodes in the tree point to other nodes in the tree.

There are other models. Trees representing graphical operations often use a style similar to reverse Polish notation; this is especially useful when the backend uses a stack-based approach to rendering:

1. Begin picture.
2. Draw line.
3. Start group.
4. Draw circle.
5. Draw circle.
6. Composite and draw group.
7. Draw line.
8. Draw line.
9. End picture.



For the purposes of this document we will focus on trees stored as nodes with pointers to other nodes.

Managing children

A common approach is for each node to be an instance of a data type with a list of children of that type. In the author's experience, this is insufficiently flexible. Many trees have nodes that differ in their approach to children. For example, it is very common, when building a tree of heterogeneous simple nodes, to have many node types that have exactly one child (sometimes termed "proxy nodes"). It is similarly common for there to be several kinds of nodes for the leaves, which by definition have zero children. This leaves a few types of nodes that have a variable number of children, but how those are organized can vary, for example, it could be a list, a two-dimensional grid, or an ordered map. There are also more advanced cases where the children may need to be generated on demand.

Examples of leaf nodes

The prototypical example of a leaf node is an image in a layout tree or UI description. Images are typically considered atomic opaque data in the context of their parent tree; e.g. the data might be raw pixel data. Sometimes the data for an image, in particular a vector image, is itself represented in the form of a tree of a different type, e.g. a display list (commands for the GPU backend). Such a tree also has leaf nodes, e.g. drawing a line or circle.

Examples of proxy nodes

Simple layout primitives often have a single child. Padding is a good example; the purpose of a padding node in a layout tree is to take the child's geometry, add some spacing on each side, and

[105] For the purposes of this discussion, references are essentially just pointers. The precise semantics of how such links are made vary based on the language.

otherwise pass on all responsibilities to the child. Flutter's core framework has upwards of 75 different layout nodes that use this style, mostly nodes that apply a single feature, like changing the opacity of a subtree, or sizing a subtree, or drawing decorations around a subtree.

Examples of nodes with child lists

Whether positioning nodes horizontally, vertically, or atop each other, a list is often the simplest way to store children. This could be a linked list or some form of array, depending on the performance and memory needs; benchmarking is advised when making such decisions.

Examples of nodes with named slots

A very common pattern is for a component to have a specific set of children. For example, an application typically has a toolbar, a left sidebar, a right sidebar, a bottom navigation bar, an overlay for transient messages, a navigation button, and the main application content. This can be represented by having named slots, rather than a list, where a slot can be empty or have a single child.

Examples of nodes with child grids

We touched on the different data structures for storing children as grids earlier. While the data model for storing children could be specialized for grid primitives to optimize them for performance, in practice, the performance characteristics of child storage solutions are typically a small part of the costs of implementing grid layout, as we will discuss below. For this reason, grids often just use lists, or lists of lists, and reuse the infrastructure in the framework that is designed around such child lists, instead of introducing potentially complicated new mechanisms.

Examples of nodes with on-demand child generation

Sometimes, the children are too numerous to instantiate. The most obvious example is a literally infinitely scrolling list. As there is, by definition, not enough time nor memory to instantiate all the children, they must forcibly be instantiated on demand. Once this ability is available, it can be used for finite but large sets of children as well.

In general, this approach should be used to avoid doing any work for off-screen content (e.g. not even instantiating objects to represent that content, and certainly not doing any work to lay out or paint that content).

Examples of even more esoteric child arrangements

Rarely, another form of storage presents itself as the logical choice. A quadtree, maybe, or a map where the children are keyed by some semantically relevant value because they are often accessed by that value rather than by position. Having the ability to organize children arbitrarily enables these options to be kept open. That said, the benefits of these more esoteric storage mechanisms are often outweighed by the necessary bespoke infrastructure they require; often, the convenience of being able to use one of the more common storage methods, and the shared tooling that a framework will have available for managing those, is more valuable.

Storing child-specific data

It is very common, in a UI framework, for a node to need to store per-child data, e.g. position information, animation status, or the status of some gesture recognition logic. This kind of data could be stored in a parallel array, but if the child list is managed by some general helper code, keeping the two in sync can be non-trivial. One could also use a dictionary to store a mapping of child to per-child data, but this has some not insignificant overhead.

Instead, a pattern that has been found to work quite well is to reserve space in every child node for use by the parent. This is variably called "parent data" (data owned by the parent; this naming scheme makes sense from the child's perspective) or "child data" (data about the child; this naming scheme makes sense from the parent's perspective). The overhead is that of a pointer per node, plus the cost of the data. The access cost is O(1); a few pointer dereferences and casts. This pattern does not typically lend itself well to locality (because the data is heap-allocated and therefore not contiguous with the child's own fields), but frankly, tree structures in general suffer from a lack of locality already so that ship has likely already sailed.

In the event that the child list is stored as a linked list instead of an array, the parent data is also a useful place to store the linked list pointers.

Managing parents

It is very tempting, when creating tree data structures, to omit a reverse link from the child to the parent. This would save a pointer per node, and would remove the need to maintain the parent pointer.

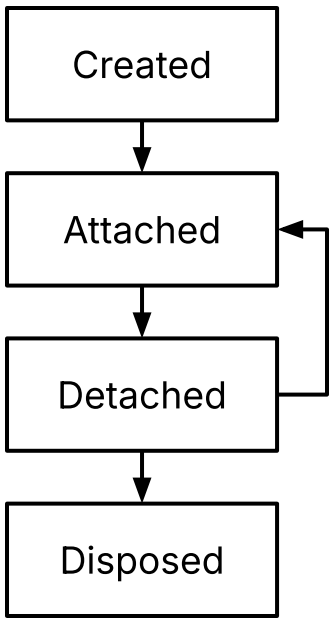
This is a fool's errand. It is reliably the case for trees in UI frameworks that sooner or later, some algorithm will need to be able to walk *up* a tree, more or less regardless of the type of tree. For this reason, a parent pointer should be included in tree nodes as a matter of course.

When a node is added to another, it must be notified so that it can update its parent pointer. Similarly, when a node is detached, it must be notified so that its parent pointer can be cleared.

Node lifetimes

Any tree that contains mutable state must have a defined lifetime and clearly defined lifecycle.

Typically, the lifecycle of nodes in UI framework trees is as follows:



When a node is created, it will typically allocate memory-intensive resources (e.g. loading image assets) that are needed for its normal operation. When a node is attached, CPU-intensive resources (e.g. timers) will be activated, e.g. to run animations. A node can be detached temporarily, briefly suspending the CPU-intensive resources, e.g. when a UI component is moved from one location in the UI tree to another, or when a UI control is obscured by another window. Finally, when the node is no longer needed, it will be disposed, and memory-intensive resources will be freed.

It is tempting to do away with the "disposed" state, especially in GC environments where one assumes that the garbage collector will handle memory-intensive needs. However, in practice this is insufficient, because garbage collectors do not have sufficient information. For example, the "memory-intensive resource" may merely be a handle to a manually-reference-counted image resource in a cache (the image itself being managed by code unrelated to the garbage collector); there needs to be some mechanism by which the reference count is decreased when the node is no longer in use. (Changing the reference count during the attach/detach phase changes is inappropriate because it risks the image itself being freed by the cache when the node is only temporarily detached.)

Dirty tracking and depth sorting

Nodes in UI framework trees will often need to mark themselves as dirty so that a later pass can (for instance) update the rendering.

Walking the entire tree is more expensive than ideal (these trees can be large, and the count of dirty nodes very small). It is therefore useful to be able to keep track of which nodes are dirty, and only visit those. This can be done by having the nodes track if they are dirty with an internal flag, and when they are marked dirty, if their parent is not already dirty, adding themselves to a tree-wide dirty set. Then, when a node is to be "cleaned" (processed due to being on the dirty list), it can reset its flag, then recurse to any child with the flag set.

To do this efficiently, the dirty set must be sorted into a dirty *list*. The needed order is the partial ordering wherein any node is guaranteed to be later in the tree than all of its ancestors. (There is typically no need to maintain an order among siblings, for the purposes of the dirty list.)

An efficient way to implement the comparator for this sort is to have every node track its relative depth. Let's say that a field named `depth`, initialized to zero, tracks this value. Whenever a node is attached, if its `depth` was not already greater than its parent's, it should set its `depth` to one more than its parent's, then notify the child that it has been attached. (If the depth is already greater than its parent's, then nothing need happen.) The comparator can then just return the difference between two nodes' `depth` fields.

This has the advantage that in the common case, maintaining the `depth` is O(1), because trees are usually built top-down. On occasion, when a subtree is moved to a deeper part of the tree, an O(N) operation must update all the nodes' `depth` fields (where N is the number of nodes in the subtree being regrafted). Subsequent moves of the subtree between its new location and its old location are O(1), as the depth of the node that is the root of the subtree is already greater than the parents at both locations.

Tree context

Some state is tree-specific. For example, the dirty list.

Having this state on a root node, and requiring each node to walk up the parent chain to obtain the tree-specific state, is memory-efficient but CPU-expensive. A practical solution is to store a pointer to the root[106] on each node. This is highly redundant and uses more memory[107], but is CPU efficient.

A mechanism that could be used that is both memory-efficient and CPU-efficient is to use a page-oriented arena[108] to allocate a tree's nodes, and to have tree-wide data (e.g. a pointer to the root node) stored at the top of each page. A node can then access the root node by merely masking its own pointer[109]. This costs a pointer per page, rather than a pointer per node. This kind of mechanism is only practical if the programming language offers the ability to provide custom memory allocators, however.

How to match platform conventions

Platform conventions for UI controls fall into two categories: the look, and the feel. The look is the precise pixels drawn to the screen, and the feel is the combination of gesture and keyboard handling, animation curves, haptic feedback, and other interactive aspects.

There are, broadly speaking, three approaches to matching platforms when it comes to drawing pixels.

---

[106] Or some other object that holds tree-wide data.

[107] The parent pointer, the parent data pointer, the root pointer, the depth, and the dirty flag: in just the last few sections, we've already made every node of every tree hold at least 28 bytes (probably more like 40 bytes). Assuming five trees with a thousand nodes each, then this upwards of 200KB of overhead just for the features mentioned in this section! For embedded scenarios with very constrained memory limits, this may point to the need for a different design. For a UI framework aimed at modern mobile devices and desktop computers, however, this is well within today's user expectations, where Chrome tabs for average web pages routinely use hundreds of megabytes, as do typical Android applications.

[108] Meaning an arena that allocates one memory page at a time.

[109] Concretely, suppose an arena using 4KB aligned pages. Any pointer to a location in this page can have its low twelve bits set to zero to get a pointer that points to the top of the page.

Rendering using system APIs

The first is to leverage the platform's code and literally have the platform draw the widgets. This is only possible on some platforms. For example, Windows has several APIs (e.g. Win32's UXTheme[110]) for drawing controls. This has the advantage that if the platform allows for the theme to be customized by the user (or if the controls render differently on different versions of the OS), the controls will fit with the platform's conventions. It has the disadvantage of requiring that the framework's rendering code be able to directly interact with OS level APIs, interleaving drawing and compositing at the framework level with drawing by the OS. Doing this in a cross-platform manner is non-trivial.

This approach also leaves the *feel* unsolved, so the framework must still implement per-control interaction logic.

This approach is used by HTML/CSS, Delphi, and Qt.

Using the actual OS controls

To match the *feel* exactly as well, one can literally use the platform's controls. This requires that the UI framework essentially be a layer over the platforms' frameworks. While this does result in a very precise match for the UI controls, it makes it very difficult for users to customize the controls used beyond what the platforms already support (and this can vary wildly from platform to platform). In general this forces frameworks to a "least common denominator" approach where anything that isn't supported on all platforms becomes either unavailable or impractical. Display effects that the platform doesn't support with its controls also become impractical (e.g. applying a perspective transform to a control is not practical if the platform does not support having its controls rendered into an off-screen buffer and its input events transformed by the application).

This approach is also even more architecturally constraining than just using the platform APIs to render pixels. The framework must be able to interact with the OS APIs at a very fine-grained level from the core of the framework.

This approach is used by some web frameworks (those that provide a layer on top of the HTML controls). React Native and NativeScript use this approach to allow code running in a JavaScript runtime to use controls defined by the mobile platforms.

Faking it

The third approach is to just entirely reimplement the platform controls. This approach requires the most maintenance work (e.g. each time a platform changes its conventions, the library must be updated; this is not always a predictable event and so planning for it can be awkward and can affect other timelines). It also has the least platform fidelity. "Minor" bugs in the reimplementation will be obvious to some users, and may be subconsciously distracting to others. Applications running on a different version of an OS than what the controls were written to emulate will look out of place. User preferences (on platforms that allow the theme to be replaced entirely) will not be honored.

This approach is not as intractable as it may first seem. Controls are, by and large, not *that* difficult to reimplement. Careful study of the controls being emulated is needed, but this is doable. It does,

---

[110] Ironically, this API does not support the latest Windows UI design language (Fluent), so using it does not really solve the problem.

however, require a significant and ongoing investment, and for some controls (in particular text fields) there is a long tail of features that users expect the controls to implement that will take a long time to implement. There are also some difficulties with this approach, because some operating systems do not completely expose the required APIs to fully implement every control[111]. This approach will also attract a baseline level of hatred from some developers for whom not using the platform widgets directly is immediate grounds for dismissal.

In practice, the main disadvantage of this approach, namely that it won't match the platform perfectly, is one that our industry has largely made moot. Most operating systems today, out of the box, have mismatched UI conventions within their own UIs[112]. Application UIs vary wildly beyond that. In many ways, applications having a bespoke "branded" UI is seen as a differentiator; a UI framework that can reimplement platform controls will naturally be well-suited to implement custom controls also.

The main advantage of this approach is the complete control that it affords the framework and developers using the framework. Every pixel is ultimately under the application's control.

This approach is used by Flutter. It's used by any framework that is attempting to match the look and feel of a different platform than the application's host platform (e.g. any web framework that implements Google's Material Design look and feel, or Microsoft's Fluent UI design language). It's also the approach used by all UI frameworks that have their own bespoke design language.

Conclusion

On the whole, the most practical approach when considering the goals of maximizing performance and display effects, is to reimplement the platforms' controls.

If combined with an architecture that has a first-class composition mechanism, the best approach is to create primitives that can underpin platform-specific controls for many platforms, as well as a set of controls that are easily customized for branded UI. Care should be taken to avoid baking features that are specific to platform-specific controls into the core primitives in such a way that they cannot be also reused in creating branded UI[113].

Theming

While a strict separation of semantics and style is unpopular, the ability to quickly change themes is nonetheless a common request. That is to say, rather than having the developer express "this is a menu, menus are red" (separate semantics and style), and rather than merely saying "this is red"

---

[111] Again, text fields are the main problem here. For example, iOS does not expose the required accessibility APIs to fully implement text fields (the iOS text field uses secret private APIs for some of its features), so frameworks must play tricks with hidden iOS text fields to achieve the results users expect. Similarly, on the web, text fields have features that have no exposed API, so hidden HTML text fields must be used as part of implementing the platform backend. This is not entirely satisfying. The author would recommend approaching other UI framework developers to coordinate putting pressure on platform vendors to provide first-class support for implementing controls with high fidelity.

[112] For example, macOS scrollbars in the settings app behave inconsistently (this bug is still present as of macOS 14.6.1). UI controls in the built-in YouTube app on Android do not match the UI controls in the rest of the OS. The Windows 11 settings UI is a mixture of Fluent UI controls, Windows XP-era controls, and Windows 95-era controls.

[113] Text fields are, once again, a great test case here. It would be easier to create a primitive text field building block that can only render iOS-style and Android-style text magnifiers. This would, however, preclude a branded UI library from creating branded magnifier UI for its own text fields. Careful construction of the text field primitive to allow platform-specific magnifiers to be configured as easily as custom magnifiers would avoid this problem.

(purely presentation), developers like the ability to say "this uses my primary color, my primary color is red" (theming).

This problem is fundamentally unbounded. Consider a button:

Here are some aspects of the button that developers might want to control from a theming feature:

- The color of the button ("make it blue"). This needs to change both the background color, the outline color, the text color, the icon color, plus the colors used for the focus, tap, active, focused, and disabled modes appropriately.
- Each of those colors individually, in each state (focus, tap, etc).
- The size of the button ("make it bigger"). This needs to change the text size, the padding, the size of the icon, the border radii, and the spread of the shadow; and they all need to be updated appropriately in each of the aforementioned states.
- Each of those sizes individually, in each state.
- The font ("make it bold"). This needs to update the font appropriately in each state.
- Each of the fonts for each state individually, including options like text decoration (underlining), font features and variations, etc.
- The timing, duration, and curve of animations when transitioning between each state.
- The elevation of the button ("make it higher"), updating the shadow settings (color, spread, etc) appropriately for each state.
- The individual shadow settings for each state separately.
- The capitalisation of the text ("make it lowercase").
- The icon family ("make the icons more rounded").
- The shape of the button ("make it pointy!").
- The shape of the button in each state, with appropriate animations ("when I tap it I want it to become pointy!").
- The haptic feedback of the button when the user interacts with it. Or audio feedback!
- The alignment of the text, and its position relative to the icon ("put the icon on the right... but not in the Arabic version").
- The shader that animates the texture of the button when it is activated.
- The settings for that shader.
- The style of the border around the button ("give it a dashed outline"), both as a single setting that affects all the states appropriately, and the individual settings of the outline (e.g. dash length, dash pattern, border width) for each state independently.
- The overflow behavior when the label would not fit in the available space (ellipsis, wrapping, clipping, ...).

- The background of the button (as repeating or scaled images, nine-patch images, various kinds of gradient).
- The mouse cursor (as a whole and individually for each state).
- All these settings (color, font, shape, etc) for a tooltip shown for the button, if any.

This list is almost certainly missing some key feature that some developer will desperately need. And this list is just for a single button, the simplest interactive UI element that exists. The equivalent list for, say, a tree view widget, would be an order of magnitude bigger. Plus, this list is for a specific imagining of a button — a button could be implemented with an entirely different approach and therefore need a different set of themable properties (e.g. a "button" could be a primarily voice-activated command that shows its label in a virtual keyboard strip when available, or could be only an icon, or could be a hyperlink in a text paragraph).

Evidently, therefore, providing all these features in a theming system is completely impractical.

It must also be emphasised that, as noted in the list above, the most obvious theming axes, e.g. "color", are actually quite nuanced. A "green" button is typically gray (or maybe dimmed green) when disabled, and some sort of brighter green when activated in some way. The text, icon, and border colors of the button above would all need to change if the color was changed to "yellow". However, developers typically do not want to have to think of these details. The button above is just "green"; developers typically want the other colors automatically derived... except in some cases, where extra control is needed!

There are several approaches to this problem (though in the author's opinion, none are perfect, so this is an area where innovation would be particularly interesting).

Define a subset of properties as themeable

Property bags

Rather than attempt to make everything configurable, each widget can define the various properties that it will support, and provide some API (e.g. a bag-of-properties object) that can be set at some high level so that all instances of that widget will use those properties.

A developer could define multiple variations and then select them on a per-widget basis (similar to how HTML+CSS developers use the "class" attribute). Alternatively, or additionally, the variations could be configurable on a per-subtree basis, so that different parts of the UI are configured differently.

This approach creates an API cliff; developers who want to configure something not yet supported will either be disappointed, or have to expend greatly more effort to change that feature. Scope creep will be inevitable, as more and more properties get added to each widget's list of properties; the later additions will be harder to implement as they were not considered in the original design.

Styling language

A variant on this approach is to have the bag of properties be the graphics primitives, use those to generate the look of the widget, and then have a language that allows the look to be overridden dynamically.

This is essentially the approach taken by CSS-based web frameworks (and those they influence, such as React Native). The widget will consist of a few generic elements (e.g. s) controlled by JavaScript, and the styles will be applied using style rules in CSS. Developers can override the default style rules with their own, which *cascade* and thus override the default look.

This approach also suffers from an API cliff; if the structure of the generic elements is inadequate to the ambitions of the developer, the developer will be unable to achieve their goals. Similarly, configuring the *feel* of the widget is limited by what the JavaScript can do; CSS does not allow it to be overridden[114].

Composition

Developers could be told to create their own widgets, either from a set of building blocks, or by literally copying and pasting some template widgets and tweaking them.

This approach is very powerful, but inconvenient for the developers. It also suffers from the problem that bugs in the templates or introduced by the developers when combining building blocks will linger and so applications written with a framework that uses this approach will be inconsistently buggy. Making simple widgets like buttons can be tractable — though developers will forget to take into account, or test, less commonly used features like accessibility and focus — but asking developers to create their own variants of more complicated widgets like text fields is likely well outside the realm of reasonable.

In practice this approach is not widely appreciated[115].

Extension

An alternative to composition is extension. A button widget can be created, and developers can extend it, hooking into various provided extension points to change the look and feel.

Depending on the level of extension points provided, this can vary from being limited in how flexible it is, to being overly burdensome. For example, if a button such as the example above provides a single extension point giving a single color, then it is easy to change the color of the button, but impossible to change its shape. On the other hand if the extension point is a canvas API, the developer must not only select the color, but also every derived color, and the shape, and the text settings, etc.

Finding the right level is a balancing act and fundamentally can never address every developer's needs.

---

[114] In the late 1990s and early 2000s, attempts were made to develop a technology known as XBL to solve this problem. An early version was used in Mozilla's UI. Microsoft also developed a similar technology known as HTCs at the same time. Both allowed scripting to be bound to an HTML element via CSS. Unfortunately, HTC suffered from a rather incoherent design and XBL was an intractably complicated mess and neither survived. (The author was one of the individuals who unsuccessfully attempted to standardize XBL.) Later, Web Components took some of the better ideas of XBL and HTC but dropped the ability to bind from CSS, which makes it irrelevant in the context of theming.

[115] Flutter supports composition of this form as a first-class feature; all the out-of-the-box widgets are implemented in this way, using the exact same mechanism made available to developers. This mechanism is literally how applications are written in Flutter; the "hello world" is a widget built using composition, different screens and dialogs in an application are all made in this way, all custom widgets use this mechanism. Nonetheless, developers have consistently resisted making use of this approach for the purpose of theming, instead preferring more or less any other option even when they seem inferior in almost every way.

Mixed approach

The state of the art is to provide all three options. This is definitely not an ideal option, but in the absence of a better one it does address most developers' needs. Basic interactive widgets can be provided that have various extension points, allowing developers to create custom variants of those widgets relatively easily without having to remember to implement accessibility, focus, keyboard control, etc. High-level widgets can be provided that have a defined bag of properties, and those bags of properties can have convenience constructors to, e.g., convert a single color into all the necessary derived colors. Finally, those high-level widgets can be built using the same mechanisms made available to developers, so that in the extreme case developers can fork them and implement entirely new widgets composed out of the same low-level primitives as the built-in high-level widgets, but with their own specific look and feel.

Platform integration

As discussed earlier, if prioritising developer popularity of the UI framework, the author recommends exposing platform APIs in a manner that uses a language feature for binding to those APIs (rather than requiring that developers write code in the platforms' various "native" languages) in a synchronous fashion (and not, for example, requiring a thread hop). The impact on the language design is discussed below.

This enables developers to write platform-specific code without learning platform APIs *and* platform-specific languages.

Out-of-the-box support

Providing mechanisms to directly communicate with the platform APIs is the bare minimum needed to expose platform features, but the developer experience is much more pleasant if the framework exposes idiomatic APIs for those features itself. For example, while not generally considered part of a UI framework, camera access is intrinsic to many applications today. If the UI framework has excellent (cross-platform) support for camera access out of the box, the experience for developers who need camera access will be much more pleasant than if they need to learn APIs on iOS, Android, web, OpenHarmony, etc, and write separate code for each.

This leads to the question of how to expose these APIs when different platforms have different features. There are two common approaches: the least-common denominator, and the superset. In the least-common denominator approach, features only present on one platform are ignored, and the API exposed to the developer supports only those features that can work everywhere. A slightly less aggressive version of this approach may include features that have no effect on certain platforms, without losing key functionality. The superset approach, on the other hand, exposes an API that appears to support every feature that exists on any platform, relying on the developer to know that certain features will not function on all platforms (or even on most platforms). Such an API could throw an exception, for example, when used on a platform where the feature is unsupported.

This is one decision where the choice differs depending on whether one is attempting to create "the best cross-platform UI framework", or "the best UI framework for every platform". The best cross-platform UI framework is one where the developer can confidently write an application once,

and trust that it reliably works exactly the same everywhere. The best UI framework for a specific platform, on the other hand, is one where the developer can make optimal use of that platform's features to provide the best experience for the user on that platform.

In general, the best cross-platform UI framework is a less ambitious goal than the best UI framework for every platform. The former is fundamentally a less capable system than the latter (as it has a subset of the features, by definition).

In practice, the ideal design is one that makes it clear which features are reliably supported everywhere, while also supporting features that are only available on a subset of the target platforms. Designing APIs to be coherent under such constraints can be non-trivial, but when executed competently, this provides developers with both the ability to create a core product that will reliably function on all platforms, and the ability to create bespoke code on a per-platform basis to further improve the user experience.

Targeting the web

Of the various platforms a cross-platform UI framework can target, the web has the least in common with the others. The web therefore requires additional non-obvious work to support. Much of the work around compilation (and hot reload support[116]), rendering, language support for binding to platform APIs, and the platform bindings themselves are different when targeting the web compared to other platforms.

In terms of the basic approach, there are two main axes to consider: the target language for compilation, and the rendering architecture. These two decisions then provide a clear direction for the rest of the work.

The target language choice is between JavaScript and Wasm. The common wisdom is that JS is just-in-time compiled and Wasm is ahead-of-time compiled and that therefore one picks Wasm for more consistent high performance, but none of these statements are strictly true. In practice, JavaScript engines are so highly optimized that code compiled to JavaScript (which stays within JavaScript fast paths and avoids known performance pitfalls) can run approximately as fast as code compiled to Wasm, possibly with faster startup[117] and less overhead when communicating with the platform APIs. Furthermore, the debugging experience when targeting JS can be better (as it has had much longer to mature), though that is less of a concern for a project looking to the future.

The reason to choose Wasm is more a matter of practicality. The target is designed for being compiled to, unlike JS[118]. It is more straightforward to cleanly maintain language semantics when

---

[116] As the Flutter team discovered, developers do not appreciate being told that hot reload is available on all the other platforms and therefore they should develop on one of those and only compile to the web later. Developers, it turns out, want to develop on their target platform the entire time.

[117] Though not necessarily. JS compilation is a significant cost on cold load, and Wasm moves the compilation step to the development cycle. JavaScript-based applications therefore need a different architecture to be competitive on startup (one that defers compilation for the bulk of the code until after the application is interactive).

[118] It also enables 64 bit integers with less overhead than would be required to implement them in JavaScript (which out of the box only supports 64 bit floating point numbers and bitwise manipulation of 32 bit values). This sounds like a minor point until one writes a parser for a binary data format using a language like Dart where 64 bit integers appear to be supported unless one compiles the code to JavaScript and suddenly one loses up to 11 bits of every value because the integers get treated as binary64 floating point numbers (doubles) instead of true 64 bit integers.

compiling to Wasm than compiling to JavaScript. It also represents a more likely future direction for the web as a whole, so targeting Wasm positions the UI framework more solidly for the future[119].

The second axis to consider is the rendering target for the backend. There are essentially three possible choices:

- Using, to the extent possible, straight HTML and CSS. This is the most difficult option in terms of fidelity (HTML and CSS were not designed for faithfully rendering arbitrary graphics commands). It seems attractive at first because one can imagine this approach making it easier to achieve good accessibility characteristics, but in practice that does not pan out either.
- Using the web's 2D Canvas features. As most UIs are essentially 2D (as discussed earlier), this is often a good match. However, it has a number of limitations in its current state. Most obviously, it would preclude any 3D content. Less obviously, the current 2D Canvas features of HTML don't actually support multiline text, which can make rendering text tricky. It also does not support the full range of font features that the UI framework might support.
- Using the web's 3D graphics API (WebGPU[120]). This is the closest the web comes to exposing the same kind of API as one would target on other platforms, which maximizes the potential for code reuse between the web platform and other platforms.

Search engine optimization (SEO)

A requirement unique to the web is that search engines be able to crawl the application sufficiently to enable the URLs that represent the application to be exposed in search results. For example, an application that hosts discussions must expose those discussions to search engines so that people searching for topics discussed in the application will result in links to the application being provided by search engines.

While search engines do, to some extent, run JavaScript, this is insufficient to index an application that is entirely rendered by code at runtime. A purely client-side application framework therefore will struggle to provide what is necessary here. Instead, the framework must support some form of server-side rendering. Web-specific frameworks commonly support this as a performance optimization as well: the server generates content that is then sent to the browser and rendered before the JavaScript loads, at which point the JavaScript can add interactivity, and subsequent content loads are done dynamically by the application.

Having the mechanism be so elaborate is not strictly necessary for SEO. Instead, the server can generate static HTML that is used by the search engines, and the application can be an entirely separate code path. Making this transparent for developers is not trivial.

Programming languages

There are three decisions to be made regarding the programming language(s) involved in a UI framework, and they are which language(s) to use for each of:

- the users of the framework.
- the framework itself.

---

[119] Or so the author hopes, anyway.
[120] There is also an older API, WebGL, but it is essentially deprecated.

- the framework's support environment.

For some frameworks, the answer is the same across the board. For example, Delphi applications are entirely written in ObjectPascal[121], the framework itself is written in ObjectPascal, and the IDE is written in ObjectPascal[122].

Other frameworks have more complicated answers. Developers targeting the web[123] use HTML and JavaScript, but the framework itself (i.e. the browser's implementation of HTML, CSS, and the DOM), and the JavaScript runtimes, are typically written in C++. Flutter apps are written in Dart, as is the bulk of the framework, but the runtime itself is written in C++.

From the perspective of developers using the framework, the important language is the one that they will use to write their cross-platform code. (This is also briefly discussed in the section on how approachability affects developer productivity and is a factor in driving developer adoption.) The choice of language here drives the choice of language for the framework itself and for the framework's environment.

In terms of selecting a target language, there are essentially three options: provide a framework optimized for a popular language; provide a framework that can support many languages, thus side-stepping the question; or create or adapt a language optimized for a new framework.

Leveraging an existing popular language

The three most popular languages, according to most sources[124], are Python, JavaScript, and Java. This has been relatively stable for many years. C# and C++ are also relatively popular.

Building a UI framework to target one of these languages provides an immediate benefit to the many developers familiar with that language. This has been a major driver in React Native's popularity, for example.

JavaScript

JavaScript has a number of very popular UI frameworks already, including the web itself, React, and React Native. It isn't clear what a new framework would bring that the existing frameworks don't already provide[125].

Python

There are many, *many* UI frameworks that can be used from Python via bindings, such as Qt[126], wxWidgets, and Tk (via Tcl), but there does not seem to be a widely used framework that was specifically *designed* for use with Python, and none of the popular Python-enabled frameworks seem to support iOS, Android, or other platforms other than the major desktop environments.

---

[121] Delphi and C++Builder could be argued to be one product, in which case technically developers can use C++ for Delphi-based programs also.
[122] According to communications from Embarcadero, the compiler is also "primarily written in" ObjectPascal.
[123] Meaning the traditional HTML+CSS environment; things are more complicated when considering Wasm.
[124] There are various ways to determine the "most popular language", c.f. TIOBE (current), IEEE Spectrum (2024), or the aforementioned Stack Overflow survey (2024). They all put Python in the top three, JavaScript in the top six, Java in the top seven, C# in the top eight, and C++ in the top nine, and beyond that disagree somewhat wildly. (They also all put SQL pretty high, but that's not a general purpose programming language and therefore is out of the scope of this discussion.)
[125] Not that this stops anyone, new JS frameworks appear seemingly every day.
[126] Indeed it has multiple bindings for Qt, e.g. PyQt and PySide.

Python itself is not especially suited for high-performance computation (e.g. UI layout) or low-level access to I/O devices (such as drawing to the GPU or receiving touch input). It also is not directly usable on mobile and embedded devices (to run Python on such devices, one is expected to embed the Python runtime into an application written in some other language, e.g. Kotlin or Swift). A Python-first framework, therefore, would need to be written in some other environment, which then embeds Python. This is somewhat similar to the approach used by React Native for JavaScript and Flutter for Dart (though in Dart's case, the source code is compiled to machine code during deployment, not on the user's device).

Open questions that could probably only be answered by attempting to create a framework for Python include:

- What performance could one achieve when computing UI layout in Python? Is it sufficiently performant to enable Python-driven animations? If not, layout and other animation computations would have to be offloaded to the backend. At some level, the less is actually implemented in Python, the more this is just another UI framework with Python bindings.
- What are the memory characteristics of an immediate-mode framework in Python? (e.g. does it introduce jank due to garbage collection, does it require too much memory to be reasonably viable for typical interfaces, is there any risk of memory thrashing hurting performance during complex animations?)
- Which Python backend is best suited for this task? (CPython or PyPy, probably, though there are other options up to and including creating an entirely new backend)
- How viable is building graphical scenes in Python and then transferring them to the backend? Is it more efficient to build a list of graphics commands in Python, then pass it to the backend, or is there less overhead calling through to the backend on each call?
- Which mechanism is best suited for integrating the backend code with Python? (e.g. CFFI, Python C extensions)

In the opinion of the author, creating a Python-first cross-platform UI framework is a high-risk premise (in that it *may* not be possible to create a compelling product), but a high-quality execution would be well positioned to reach a large number of developers. An easy-to-use, adequately performant UI framework based on idiomatic Python would create new opportunities for many developers who are currently not well-positioned to create applications, especially mobile applications.

Java

As with JavaScript and Python, there are many UI frameworks for Java. Indeed, the second most popular[127] UI framework in the world, the default Android UI framework[128], is a Java UI framework (albeit not a cross-platform one). However, in terms of cross-platform frameworks, there is no clear contender in terms of popularity in the application creation space. The existing cross-platform Java frameworks generally suffer from a lack of modernity, either in the API styles (e.g. using a markup language or other retained-mode API rather than a React-style immediate-mode API), rendering styles (e.g. having controls that use the aesthetics of 1980s or 1990s GUIs), or both.

---

[127] In the sense that web browsers are the most deployed platform, and Android is the second-most deployed platform.
[128] It's not clear if this framework has a name. It's sometimes referred to as "Views" because controls inherit from the View class.

The Java ecosystem is somewhat tainted by the litigious nature of its primary maintainer (Oracle)[129]. The author cannot recommend that anyone create a UI framework for Java, however popular the language might be, because one would always have to assume that if the product was in any way successful, Oracle would attempt to extract some sort of compensation from it. This also extends to any language that is Java-adjacent, e.g. Kotlin.

C#

There exists a plethora of C#-based UI frameworks (e.g. ASP.NET for the Web, .NET MAUI for cross-platform applications). As with JavaScript, it's unclear what a new framework could bring that has not already been attempted.

That said, of the top languages, this is the most promising. Tooling for the .NET ecosystem is extensive, and the primary caretaker of the ecosystem, Microsoft, would likely welcome a popular UI framework using their development environment[130].

C++

While C++ is a proven language for developing cross-platform programs, and indeed is used by pretty much every platform for much of the core platform code, it is less than ideal for GUI work. The most obvious problem is the complexity of manually managing memory for trees of UI widgets and supporting objects.

C++ would be a reasonable choice for a language that is used to expose a framework to other languages via bindings, though, on par with C.

Dart

While not one of the top languages currently, many parts of Dart are very approachable, and Dart has the advantage of having already been adapted to the needs of a UI framework, which makes it particularly suitable here. It has limitations that could be avoided by developing a new language, but they may be sufficiently minor that the benefit of being able to reuse an existing language would offset them.

One downside of relying on Dart for a new framework is that its development is tightly controlled by Google, and it is not clear how welcoming the team would be to new contributors who had a separate agenda. Another downside is that there is already a Dart-based UI framework, and it's not immediately clear what benefit one would bring to the table by creating another.

Use a language that allows the framework to be language-agnostic via bindings

If the goal is to leverage people's familiarity with an existing language, what better solution than to simultaneously target multiple languages? This is the approach used by UI frameworks like Qt, GTK, and wxWidgets.

Generally, this approach involves using C, as for most languages, C is the first (if not only) supported extension language, and therefore using C presents the least friction to creating bindings. Other

---

[129] See, e.g., how Oracle sued Google over the aforementioned most-popular Java-based UI framework and associated operating system, in *Google LLC v. Oracle America, Inc.*

[130] Having good relations with those with influence with relevant decision makers is important for a project that aims to become popular. See also the section on politics.

languages that create object files or linkable libraries that are indistinguishable from C libraries can be good choices too (e.g. to some extent, C++, Rust, or ObjectPascal).

The low-effort approach to creating language bindings is to expose the C API directly in the target language. This is usually the most expedient way to create bindings, but it leads to very non-idiomatic APIs. What is common in C is not common in, say, Python or Java. The conventions for even the most basic aspect of an API, such as naming, vary from language to language. Where a function in C might be named using `lowercase_snake_case`, C# would use `UpperCamelCase`, and JavaScript would use `lowerCamelCase`. It is jarring for an API to be inconsistent with the rest of the language and its standard library.

An approach that leads to a more pleasant experience for developers is to create bespoke bindings for each supported language. This requires deep knowledge of both the framework's implementation and the target language's conventions, idioms, ecosystem, and community. (Typically this means each binding needs to be maintained by a separate expert or team of experts.)

There are a number of disadvantages to this approach:

- For target languages whose idioms don't match the underlying framework and language well, either the binding must introduce an expensive abstraction layer, or the resulting API will be non-idiomatic, which reduces the likelihood that it will be popular with developers. Common ways in which this impedance mismatch might manifest include memory management (e.g. the framework is written in C and requires manual memory management, while the target language is garbage-collected and it is non-idiomatic to manually release resources), threading (how are threads exposed and managed in the two languages and their runtimes), programming paradigms (e.g. the framework uses an object-oriented approach while the target language typically does not, or vice versa), and collection types (e.g. whether to use the language's native array types, or wrap the underlying C arrays).
- Even when the language is a good match, there is often a non-zero overhead to switching contexts (switching from the runtime environment of one language to the other).
- Sharing documentation between bindings is tricky, which results in either dramatically increasing the cost of producing documentation, or poor documentation (if an insufficient effort is attempted), or confusing documentation (if the binding just points to the main framework's documentation, that documentation lacking language-specific information).
- Bindings can easily get out of sync with the underlying framework. There might be lag between when a feature is introduced to the core and the feature being supported by the target languages. This becomes especially problematic when the new feature is one of those with a high impedance mismatch, requiring disproportional (and possibly unscheduled) effort on the part of the team maintaining the binding.
- Having two languages involved (or three, in the case the platform language is different than either of the other two[131]) increases the likelihood for issues. For example, if the languages each have their own preferred build systems or dependency management systems, there are now two (or three) of those, further increasing the chance that the developer will face a confusing problem.

Despite these disadvantages, given Qt's success, this approach is clearly viable.

---

[131] In addition to the bindings for the target language, one must also create completely separate bindings for the target platforms. (This is not unique to this approach.)

Create or adapt a language highly suited for the task

The third and most expensive choice is to either create a new programming language, or adapt an existing programming language, specifically for the purpose of the UI framework. This is probably the approach with the highest risk, but the greatest potential.

Five of the UI frameworks discussed in earlier sections have used this approach. The web's programming language, JavaScript, was designed specifically for the web, and is now, by many metrics, the most popular programming language, powering the most popular UI framework. Visual Basic adapted BASIC for use with its UI framework; Delphi adapted Pascal for use with its UI framework. The .NET ecosystem, while no longer specific to one UI framework, essentially started in the same manner. Finally, Flutter adapted Dart for use with its UI framework[132]. In addition, the Vala and Genie programming languages were developed specifically for GTK (though GTK itself uses C and supports many other languages via bindings).

Every error message, every piece of boilerplate, every comment in every example — everything works together to affect the developer's experience. Being able to fully control the stack — the compiler, the build system, the standard libraries, language conventions, package management, IDE tooling, the framework, documentation, etc — maximizes the number of design choices that are available, which provides the most opportunities to create a product that is popular with developers.

We will discuss the topic of creating or adapting a language for a UI framework in more detail below.

Selecting the environment's implementation language

From a practical perspective, the core of the system must use a language that can be compiled directly to the target platforms' lowest level native instruction sets[133], that gives developers low-level control over memory management, and that has a significant active community with a mature standard library. This leaves few serious options[134], mainly C/C++ and Rust. Alternatively, if the framework is written in a new language that meets these needs, that language could be used directly, self-hosting the runtime (this is the approach used by Delphi, as noted earlier, and of course by any C/C++ UI framework, such as Qt, GTK, or Clay, since the C/C++ runtimes are themselves written in C/C++).

The build system

The build system is one of the most critical parts of the experience of developing an application. As discussed earlier, build systems, from the perspective of the developer, do not solve problems, they *are* problems. In an ideal world (as might be achieved when creating a bespoke language), an

---

[132] More accurately, the Dart team adapted Dart for Flutter.

[133] Almost by definition; if it does not, that means it builds on top of *another* environment which itself must target the platform. For example, a UI framework with a custom language whose language runtime is written in Python must then itself be hosted by the Python runtime, which is written in C, which compiles to machine code. Because adding such layers has few advantages and some major performance costs, such an architecture is not seriously considered in this document (except for the special case of targeting the web).

[134] Languages such as Carbon, D, Nim, ObjectPascal, and Zig could also be considered, though their communities are, as of the writing of this document, somewhat smaller.

explicit build system is unnecessary, with the source files fully expressing both the dependencies on other packages and the interdependencies within the current package[135].

In practice, design choices around the build system are primarily driven by the choice of language (which, if anything, may be a sufficient argument for creating a new language).

The purpose of a build system is to configure the compiler and linker to find the appropriate source files and combine them appropriately. This may extend to identifying dependencies and thus configuring a package manager. It may also be used to trigger pre-processing steps (e.g. re-encoding image assets, filtering fonts to remove unused glyphs, or running code generators), and post-processing steps (e.g. package an application for deployment).

Should one design a build system, here are some of the choices one may have to make.

Format

How should dependencies and other metadata be expressed? Common choices include a binary file (the approach used by Xcode[136]; essentially impossible to code review or edit by hand), an auto-generated text file (also sometimes used by Xcode; difficult to review or edit by hand), a declarative file (as used by Dart; limited in expressiveness), an imperative script in some other language (as used by Gradle[137]; difficult to review, difficult to maintain as few people are experts in the relevant language), and a hybrid of declarative and imperative (as used by Make; has all the disadvantages of both declarative and imperative solutions and none of their advantages).

The need for an explicit build system is reduced each time this out-of-band metadata is moved inline into the source file.

For example, languages like C require that the compiler be informed of each source file, and the linker be informed of each object file to combine them together. Languages like Dart avoid this by having the source files import each other, allowing the compiler to discover all the necessary files and link them without any metadata. Similarly, C compilers typically require explicit command line arguments to configure various features (e.g. enabling or disabling warnings); with some languages, e.g. Pascal, these kinds of settings are configured inline using compiler pragmas, removing the need for explicit command line arguments in a build file[138].

We will revisit this when considering the needs of a programming language.

Scope

A UI framework's build system could be limited to just compiling and packaging the final application, but a more expansive scope might include the ability to compile business logic (e.g. if it is in a different language), or integrate with the build system for third-party libraries.

While an ideal design would reduce the need for a build system to zero, *some* mechanism to trigger the build systems for dependencies (like business logic written in another language), or for preprocessing assets in project-specific ways, does seem necessary.

---

[135] Here the word "package" is being used loosely to refer to a group of source files that could be packaged separately by a developer. For example, a reusable library or an application.
[136] The Apple development toolchain, used for macOS and iOS programs.
[137] The build system used by, among others, Android.
[138] In practice, Pascal compilers still require some command line configuration, unfortunately.

Cross-compiling

A common limitation of UI frameworks is that they can only build for certain platforms from certain platforms. For example, compiling for iOS typically requires a macOS host, because Apple requires the use of certain tools provided with Xcode to package iOS code. Similarly, compiling for Windows often requires a Windows host.

The ability to build for other platforms, however, is very popular. Solo developers often don't have all the hardware to compile to every platform they could target. Being required to own build environments on three machines is a much higher burden than one machine for someone who is just getting started in the industry.

Error messages

Usefulness

Throughout the creation of a developer product, one will need to inform the developer of errors. These may be warnings presented in an IDE, errors presented by a compiler, runtime warnings and errors caught by the UI framework itself, problems when sending applications to test devices, etc. Every single warning and error message is an opportunity to delight the developer[139].

Developers are not surprised to be dealing with errors. Most of programming is fixing bugs, many of them are bugs caught by the tooling and reported to the developer. In many ways, error messages are the primary way that a developer experiences a UI framework. Each such error has the potential to send the developer either into spiralling frustration, or onto a quick path to resolution. The latter is preferable, and more importantly, it is not that difficult to achieve.

The key is that any time one is writing code that would report a warning or error message, one must stop and think about exactly how this will be experienced by the developer. What information will they need to resolve the problem? What context will they have, and what context will they need?

Furthermore, one should stop and think about the need for the error in the first place. Could the API be redesigned to avoid that error at all? Could the error be caught in some more useful fashion?

As an artificial example, consider some runtime code in the framework backend that examines a string to decide on some graphics behavior. The string specifies one of a finite set of options. If the string is not one of the valid options, the framework throws an exception. What should the message be? "Invalid opcode" is probably not ideal; what could a developer intuit from such a message? What about "The graphics opcode 'foo' is not recognized"? That would let them at least search for the word "foo" in their code. Even better would be to specify what *would* be recognized, e.g. "The graphics opcode 'foo' is not recognized; valid options are 'bar', 'baz', and 'quux'". But how should the developer fix this? They're not calling this backend code, they're probably specifying this string in some unrelated (to them) frontend API. Unless the string luckily happens to be unique and passed verbatim from their code to the backend, it might be difficult to pin down. What if instead the frontend code did the validation? That way, the error message could specify which API was being incorrectly invoked. "The graphics opcode passed to the Blender widget, 'foo', is not recognized.

---

[139] Research supports the (possibly uncontroversial) premise that the quality of error messages affects the developer experience, which raises the question of why more effort is not made to improve them: https://www.brettbecker.com/wp-content/uploads/2016/10/becker2016effectiveAM.pdf

Valid options are 'bar', 'baz', and 'quux'". With some effort, one could even keep track, at runtime, of the line of code that called the Blender widget constructor, and report that in the error message[140].

Even better, though, would be for this to be a compile time error instead of a runtime error. For example, instead of using a string, could an enum be used instead? That way, the error would be obvious as soon as the developer typed "foo", long before the runtime was invoked. As a side-effect, this would likely also minutely[141] improve memory usage and performance.

Target audience

Obviously the target audience for error messages is the developer using the framework. That said, many developers today use AI tools that interpret error messages and attempt to apply fixes automatically. In principle this does not change any of the advice; error messages should be actionable, clear, supported by documentation, and so forth, but it is worth also testing error messages against LLMs, in addition to the testing done with humans.

Openness[142]

Source availability

Software products vary in how their source code can be obtained. At one end of the spectrum, code can be kept a trade secret, never distributed. On the other end of the spectrum, source code can be freely distributed using a permissive open source license. There are many other options in between.

For UI frameworks in particular, there is additionally the question of which *parts* of the product's source code are made available, if any. The tooling (compilers, IDEs, build system), the backend code (graphics, platform integration, language runtime), and the frontend code (the APIs, UI controls, layout logic) can each be licensed or distributed according to separate terms.

The availability of source code greatly helps developers debug their applications. The frontend code, which is the code that would be directly linked to and called by the developer's own code, is the most useful to have available. Being able to step through that code in a debugger, or examine exactly how an API is implemented, can save a developer lots of time. The backend code and tooling code are less critical, but still useful on occasion.

Most of the UI frameworks discussed earlier in this document are open source, and their entire stack is freely available. The commercial offerings vary somewhat. Qt has non-free components. Delphi's frontend is available under license, the backend and tooling are kept proprietary. SwiftUI is entirely closed[143].

---

[140] Some of the most useful error messages are those that use introspection to tell the developer exactly how to fix the bug. Great care must be taken not to confidently state information in such an error message that is not actually confidently known, however. A detailed error message is not useful if it's a lie; gaslighting the developer is no better, and may in fact be worse, than giving them no useful information at all.

[141] The impact of one such change is probably not even noticeable, it would be in the noise. However, the difference between a zillion such choices being done the slow way and being done the fast way does add up. This is one of the most difficult things to optimize in a framework after the fact, because changing a zillion trivial API choices is tedious, breaking, and each individual change is hard to justify. Doing it right from the start, on the other hand, is cheap and easy.

[142] For a more detailed discussion, consider the author's "Spectrum of Openness" blog post from 2023.

[143] Well, ironically, since Swift and LLVM themselves are open source one could say that some of the *tooling* is open source, but Xcode and SwiftUI's frontend and backend libraries are not.

In practice, commercial products will leverage access to the source code as an income stream, and open source products will make the source available as a matter of course. In some cases, the use of a restrictive open source license (e.g. the GPL) is combined with a commercial license under different terms so as to attempt to get the best of both worlds (from the perspective of the framework vendor, anyway).

Development processes

A more subtle question is how to manage the project. This again is a spectrum, with one end being your typical proprietary development with a company doing all development behind closed doors, and the other end being a fully independent, democratically accountable, open governance model, with all product and engineering decisions made in public.

The benefits of closed development are:

- It is easy to have a clear vision and execute on that vision without being distracted by team members who have completely different priorities and objectives.
- It is easy to maintain the product team's quality. Team members can be provided money to encourage them to continue working on the project, which helps long-term engineering continuity.
- Keeping the rights to everything means that customers can be charged for permission to use the product.
- Keeping development secret means that embarrassing developments can be kept secret also. Failed attempts can be buried and customers never need to know about them.

The benefits of open development are:

- Transparency is appreciated by the community, and gives customers confidence.
- As the project gains popularity, people are motivated to contribute to it, potentially growing the team far beyond the size it would reach as a purely commercial concern.
- The project can survive a catastrophic failure of the primary organization.
- The project can be used by more people, as money does not gate usage.

It's sometimes claimed that keeping plans under wraps (as in closed development) means that new features can be announced to great fanfare, but in practice, open source projects can do this too even when everything was discussed and developed publicly. The reality is that the general public and the media do not follow the inner politics of open source projects.

A pragmatic middle ground is to develop the UI framework as a public open source project with an unelected but independent core team. This allows the clear vision of a single leader (or leadership group) to drive the project, something which is very important for a developer product[144], without sacrificing the transparency of open source, and while still (eventually, potentially) benefiting from a broad active contributor community.

[144] Developer products without a clear vision tend to feel incoherent, with a mishmash of API styles and seemingly incomplete features. This is not a recipe for popularity.

Out of the box experience

The first experience a developer will have with a framework is usually installing it and running their first demo program. Making this process as simple and quick as possible, potentially deferring unavoidable complexity until later, will improve the out-of-box experience and thus help predispose the developer towards liking the framework.

It's worth noting that the competition here requires literally no effort whatsoever to get started. Every computer has a text editor already, every computer has a browser already, and this means that the barrier to creating new HTML web applications is literally zero. Someone can type "Hello World" into a file, save it as `myfirstapp.html`, load it in their browser, and they've created a Hello World using the web. Every piece of friction required by another framework is taking it further from this ideal.

Concretely, this requires several features, which we shall discuss presently.

Installation

Installing the framework should require nothing more than unzipping an archive or pulling a git repository. In addition, each platform has an idiomatic way to distribute software; providing the framework via that mechanism as well will help users who are more comfortable with using that path than fetching the framework themselves. (Thus an installation program for Windows, a `.dmg` for macOS, a `.deb` file for Ubuntu, etc.)

This requires the framework to be able to bootstrap itself once installed. For example, the entrypoint into the framework's tooling can notice that it is on a new computer and fetch dependencies itself, prompt the user regarding analytics opt-out, compile itself, whatever is necessary.

Archives of the framework should include all dependencies, so that downloading that package is all that is needed to get started before a long trip where connectivity may be unavailable: it is not unusual for developers to try new things when going on vacation or to conferences, where their Internet access may be unreliable or non-existent, even today. Thus, the package should contain any dependencies that the tooling can then install appropriately without requiring additional connectivity.

Dependencies

Minimizing third-party dependencies helps reduce the complexity of getting started. In particular, dependencies whose installation can't be entirely abstracted away by the tooling should be avoided.

This means that to the extent possible, iOS support should avoid needing Xcode, Android support should avoid requiring the Android NDK and the Java SDK, and Windows support should avoid requiring Visual Studio. (How to avoid these is unclear, but doing so would be ideal.)

Any requirement that the user install third-party dependencies manually will immediately and dramatically worsen the developer's experience, and they *will* blame the framework regardless.

Deferment

To reduce the up-front load on the user before they can experience the framework first hand, the use of dependencies can be deferred until the point where those dependencies are unavoidable. For example, on Android, the Java SDK is typically only required to compile an application's bootstrapping code and code written in Java; it is not needed to connect to an Android device and upload a binary or run a program on a device. Thus, if a shell can be precompiled and used whenever compiling an application that does not have any additional Java code, the need for the Java SDK can be deferred until the point where the developer wants to link their application with some Java code (e.g. a third-party Android library).

Deferring the need for third-party tooling to be installed is generally successful at transferring the blame for any installation complexity to that third-party tooling as well, because psychologically the developer has been using the framework successfully, and is then presented with a requirement that comes from the platform, not from the framework.

Templates

Having fully-functional demo applications available immediately upon initial installation with no additional work allows the developer to see the framework in use immediately. Such code should be commented thoroughly (but usefully — usability testing is key here), to allow the developer to quickly understand basic features and to give the developer something to play with that will generate immediate gratifying visual effects. Minimizing the time from installation to dopamine hit is key for retention.

First target platform

Whether the framework should steer users towards their first experience being a desktop application (which probably requires no dependencies), a simulator (which requires some dependencies), or a device (which requires slightly different dependencies plus additional hardware) is a question that usability studies[145] can best answer[146]. There is a tradeoff to each option. A physical device gives a more visceral experience, but minimizing the need for dependencies is also valuable, as noted above. The simulator has the advantages and disadvantages of both the other options.

State-of-the-art features

These are features that require extra effort, but that are much harder to add retroactively, due to their impact on the very core of the framework. Not every one of these features is necessarily a good idea. Generally, for each one, an early decision should be made about whether the feature will be supported, and then the project should live with that decision thereafter.

---

[145] And analytics. Comparing the retention numbers of users categorized by their first test platform may be particularly informative.

[146] One example of how usability research helped the Flutter team was an analysis of installation pain points. By observing where research participants showed the least patience when following Flutter's convoluted multistep installation process, the team was able to learn more about the potential sources for early attrition, an area into which the team otherwise had no visibility (because it happens before the product can ask the user to agree to analytics).

Stateful hot reload

Flutter popularized *stateful hot reload*, a feature that some other projects have attempted to implement with various levels of success. Hot reload is extremely valuable for developing user interfaces.

Much of UI development consists of making small adjustments and seeing how they look. How quickly one can iterate is a huge factor in determining how pretty one's UIs can be. There are basically four approaches that UI frameworks have used to solve this problem:

- A "what you see is what you get" (WYSIWYG) interactive visual interface editor, also known as a visual builder, visual UI designer, visual design tools, graphical user interface builder (GUI builder), or sometimes a RAD IDE. The interface is presented during development as a static display that matches the default display of the application, and the editor allows the UI to be adjusted directly (e.g. using drag and drop). This is effective for interfaces that are largely static, but becomes difficult when the UI is expected to change dynamically at runtime (e.g. with animations, scrolling effects, screen transitions, etc), and does not lend itself to rapid testing with actual user data.
- Having the compiler be very fast (sometimes called "hot restart"). This reduces the iteration time, and is sufficient for testing the application's start screen, but becomes cumbersome when the UI being tested is deeper in the application, as the developer must return to the UI under test each time (or change the application to automatically jump to that UI, which is not always practical). This becomes especially awkward when the UI under test relies on some long-lived state, e.g. an active network connection.
- Visual previews in the IDE. Similar to a visual UI builder, but rather than being interactively editable, the preview is a live interface. The previews are configured by code that is only used during development, for the previews. This relies on the compiler being very fast as well. One advantage of this approach is that it allows multiple previews to be rendered at once, for different modes (e.g. testing dark mode and light mode variants of a widget). The disadvantages are that testing against real data is impractical. This is essentially the same as the previous option but with the UI rendered in the IDE instead of as a live application.
- Stateful hot reload. The application is launched once, and then when the code is edited, the running application is patched so that the new code runs instead of the old code. This allows updates to the user interface to be seen without restarting the application. This means that even live network connections are maintained, which enables very rapid iteration even in very deep parts of the UI.

Hot reload is clearly the most challenging to implement of the various options, as it requires the ability to literally patch running code. To be useful it must also be very fast (measured in hundreds of milliseconds at most[147]), which requires a fast compiler.

The ability to reload code has inherent limitations. For example, code that only runs once will have already run, so hot reloading that code is not useful. This means that retained-mode APIs are not

[147] A common user experience here is to change some code, hit save, look at the device or simulator or second screen where the application is running, then look back at the code to continue tweaking it. If it takes more than a few hundred milliseconds to update the running code, then the user will look back at their code and wonder what they did wrong, change it again, then look back at their application, see the *original* change, and get frustrated and confused. (Obvious solutions like quickly throwing a progress indicator over the application are also bad, because they prevent the user from easily comparing the old and new renderings.)

practical with hot reload: the creation code path and the mutation code path must be coordinated but in a hot reload scenario the application reaches a state where the creation code path was using an old version of the code while the mutation code path is using a new version, which means they are no longer coordinated, and the application under test (if it does not crash or otherwise fail catastrophically) is at best running a scenario which no user will ever encounter.

Even immediate-mode APIs have retained-mode APIs hidden inside them. For example, initializing a variable from a constant is "retained-mode". If the constant's value is changed, variables that were assigned that constant will keep their old value. This can be mitigated in cases that the UI framework controls (e.g. by rerunning all constructors after a hot reload), but it is very easy for developers using such a system to fall victim to traps like this.

There are more subtle complications with hot reload. Patching code that is actively running is impractical. For example, suppose three statements assigning values to local variables are replaced by five different statements assigning different values to different local variables, and the application is hot-reloaded while the instruction pointer is on the second of the original three statements. How should the code be patched? There is no correct answer to this question (and most incorrect answers are likely to trigger catastrophic failures or at least completely incoherent application state).

One solution would be a design where the code patching waits for the application to reach a known idle state (e.g. awaiting an event in the framework's event loop) before replacing any code. Another would be for the code patching to introduce any changed methods as new methods, changing all pointers to the old methods to point to the new methods, and leaving the old code around until the instruction pointer returns from that code. Both of these solutions suffer from an annoying flaw: if the application ever reaches an infinite loop during development, the application is no longer hot-reloadable and must be restarted.

Some language features are especially tricky to manage. For example, structs (or records, or objects) that have a size defined by their members are typically allocated using a certain fixed amount of memory. If the size of the data structure changes, all existing instances of that data structure must be reallocated during the hot reload. This is only tractable if the language uses a managed memory model.

Another language feature that requires careful consideration is lambdas (anonymous functions). Because they are anonymous, there is no reliable way for the hot reload logic to recognize that the developer changing the code in a lambda has attempted to change the code in previously-created instances of that lambda. This means that previously-registered callbacks may continue to exist and be called.

Implementing hot reload requires close integration with the language runtime (and is a feature that is typically only possible when creating or adapting a language specifically for the UI framework, which is one reason why it is so rarely provided, and even more rarely provided in a compelling fashion[148]). One result of this is that hot reload can be difficult to integrate with platform-specific code. For example, if the framework supports linking with C libraries, and the developer adds a new

[148] Flutter (through Dart) has an excellent implementation of Hot Reload, but the feature is also claimed to be supported by several other languages and frameworks. Unfortunately, in many cases, those implementations are lacking. For example, the language support might be so limited that many changes cannot be applied, or some substantial portion of the state is not maintained during reloads.

C library dependency and attempts a hot reload, the developer may expect the new C library to be fully initialized and ready for calls from their code. This expectation is likely one that would be difficult to meet in practice.

Adding hot reload later

Frameworks that try to retrofit hot reload later tend to disappoint their developers, due to the inherent limitations that earlier architectural decisions force upon them. For example, trying to retrofit hot reload onto a framework based on a retained-mode API will probably be quite unsatisfying.

Over-the-air updates

The web changed the expectations around application updates: any time you visit a web application, you are getting the very latest version. This is a feature that many developers seek in other UI frameworks.

For applications deployed to the application stores (e.g. Apple's iOS App Store), this responsibility is generally delegated to the store, but this is considered insufficient by many developers, and of course this option does not exist for applications installed directly, especially, e.g., in desktop or embedded environments.

There are policy and security complications around this feature.

The policy complications are the most controversial. In some environments, especially the Apple and Google app stores, the terms of service either strongly discourage or outright disallow out-of-band updates. Therefore, using such a feature is risky: one's application might be banned outright from the store[149]. Furthermore, if a UI framework provides this feature, even if it is not enabled by default, there is a risk that the UI framework, and thus any applications using it, will be banned[150]. What has happened with some existing frameworks is that the UI framework vendor has avoided adding such a feature, and some other prominent community contributor has provided a third-party solution[151]. This provides the feature to developers who want to take the risk, without risking the entire ecosystem.

The security complications are comparatively straightforward, if still serious. The ability to remotely update code on a user's device is a huge target for any malware developer attempting to deploy their code. Therefore any update mechanism must be secured against such attacks. (This is nominally why the stores attempt to discourage it.) Doing this securely is not an intractable problem, but it is one that requires great care. A failure that is abused by an attacker to distribute malware code to a wide array of applications all using the same UI framework can fatally affect the framework's reputation and reduce every other effort put into it to irrelevance.

Approaches

There are generally two approaches to server-driven updates: replacing the binary wholesale, and having a patch mechanism. Wholesale replacement is trivial, but may be impossible (or contravene

---

[149] This is not hypothetical; there are examples of this happening.
[150] There is reason to believe that this has come close to happening also, though this is mostly gossip shared among UI framework developers.
[151] React Native and Flutter are examples of ecosystems that have settled on this approach.

vendor policies) on some platforms. Patching, on the other hand, requires deep integration with the language compiler and runtime.

Adding over-the-air updates later

Adding a wholesale binary replacement should be relatively straightforward; adding a patching mechanism likely would require somewhat invasive changes to the compiler and runtime. It is therefore something that would take much less effort to implement from the start.

Server-driven UI

A feature related to over-the-air updates is the ability to display UIs configured dynamically over the network. The web, fundamentally, is a UI framework entirely built around this concept. Designing a UI framework that supports this feature as deeply as the web is outside the scope of this document (as briefly discussed earlier). However, implementing this as a layer on top of a layout and control library should be relatively straightforward, so long as the framework supports programmatically building a UI dynamically. (One could certainly design a UI framework where this is not possible; for example, if all interfaces had to be statically declared at compile time, with the compiler pre-building data structures to use at runtime.)

Adding server-driven UI later

Because the model discussed here is merely a layer over the UI framework, it can be trivially implemented at any time, so long as the framework supports basic dynamic UI features.

Color spaces and varying color depths

A color space describes the range of colors that are addressable. There are various color spaces, for example sRGB (more or less the lowest common denominator for hardware from the turn of the century), Display P3 (Apple's color space for its modern hardware), or Rec 2020 (the color space used in "UHD" 4K video content).

The color *depth* (in bits) describes how many colors within that range are addressable. It can be expressed as a per-channel value (e.g. 8 bits per channel) or as a total number of bits for all channels, which may or may not include an alpha channel.

Many simple UIs can be described using only 24 bit sRGB (32 bits with an alpha channel), but developers are coming to expect UI frameworks to support broader color spaces and greater color depths, especially when rendering images or video.

Supporting multiple color spaces and color depths from the start is relatively straightforward, though it may be expensive (in terms of memory and CPU, at least compared to only supporting one color space and color depth), and may require some effort to avoid having awkward APIs.

Adding support for different color spaces and color depths later

Retrofitting arbitrary precision into APIs that have assumed a 32 bit integer can express all colors can be non-trivial. Being disciplined in using an opaque type consistently when referencing colors will be a significant aid, but it is difficult to prevent assumptions regarding color depths from seeping into APIs.

Even if one extends such a type, libraries and applications in the ecosystem that have been written with the assumption that there is only one color space and color depth will also need migrating, which may force the overall migration of the ecosystem to take significantly longer than ideal.

Vertical text

As briefly discussed earlier, in some languages, notably Chinese, Japanese, Korean, and Vietnamese, text can be written vertically. Generally, all these languages are now also commonly written horizontally, but some regions continue to use vertical writing as well. (A complete discussion of the history of typographic conventions is beyond the scope of this document, but let us add only that vertical writing was also historically used with a number of other scripts, which may be relevant if a UI framework is to be optimized for use by applications of a scholarly nature.)

Supporting vertical text exclusively is not viable, because even in locales that still use vertical text, horizontal text is still used, e.g. when embedding text from other languages (e.g. English). Supporting horizontal text exclusively, on the other hand, is possible.

Vertical text support includes support for embedding horizontal text inside vertical text runs, both straight and rotated; this becomes especially tricky in situations involving bidirectional text, text shaping, or both (e.g. with Arabic). It also requires deep support in various controls; for example, arrow keys in vertical text fields behave logically rotated from arrow keys in horizontal text fields (because they operate in the absolute physical space), drop-down widgets should drop horizontally rather than vertically, progress bars should be vertical, etc. Vertical text also affects pure-layout features like tables (where rows and columns are logically flipped in the vertical text mode, much like how the growth direction is flipped in right-to-left contexts).

The web today supports vertical text quite comprehensively. Most other frameworks do not.

The ethical dimension

Not supporting vertical text excludes a category of applications from a subset of humanity from being possible with the UI framework in question. How to balance this against the cost of implementing vertical text is left as an exercise to the reader.

Adding support for vertical text later

While vertical text is unlikely to be a feature added in an early version of a new UI framework, deciding at the outset whether the framework will *ever* support vertical text can save significant effort, either by permanently removing an entire category of features from consideration, or by allowing extension points to be inserted at appropriate points so that the eventual addition of these features can be done cleanly without extensive rearchitecting

While there is in principle no reason why vertical text could not be added to a framework that made no attempt to prepare for it (as happened with the web, for instance), doing so may require painful reengineering of text-related aspects of the framework to remove accidental assumptions. Certain layout designs may make this particularly difficult; for example, if the framework assumes a width-in-height-out algorithm, vertical text layout will be unable to automatically size to fit the size of the screen then scroll horizontally.

Concurrency

Separating some work to run in parallel allows for greater throughput and thus a higher framerate. For example, running the layout logic in one thread while the previous frame's compositing and rasterization runs in another potentially doubles the available budget, albeit without improving the overall latency. This level of multithreading is quite coarse.

To the author's knowledge, no widely successful UI framework has ever successfully made extensive use of the parallelism available in modern CPUs and GPUs. The overhead of switching to another core to complete work is often substantial even in systems with very lightweight concurrency, and the cost of any specific part of rendering a frame tends to be extremely small; the cost is only visible in the aggregate. It is therefore not obvious that aggressive concurrency helps.

This is an area where further research could reveal some interesting gains. For example, a render object tree could be walked to determine the general size of various branches of the tree, and a node with multiple children where there are multiple branches that are each sufficiently large could fork its computation so that each branch does its logic in parallel, with the main thread blocking until the subthreads return. Tuning this system to be effective is a challenge; the overhead of coordinating these threads and determining when it is worthwhile must be much less than the time gained for the extra complexity to be worth it.

Instead, concurrency is generally used for work that will definitely take longer than one frame and where therefore the overhead will not negatively affect frame times. For example, loading assets, managing network resources, and other I/O; decoding big data files (e.g. parsing large JSON files); or running expensive application business logic.

Graphics backend

There are a variety of existing graphics backends suitable for UI frameworks, such as Skia (used by Android and Chrome, among others) and Cairo (used by GTK). Some UI frameworks target OpenGL, using libraries like ANGLE to map the OpenGL API to Vulcan, DirectX, Metal, et al. Some frameworks target platform-specific graphics APIs directly, with their own graphics backend (e.g. Flutter[152]).

The choice of backend is one that can be somewhat deferred, so long as the lowest-level APIs are sufficiently backend-agnostic (for example, one should avoid baking in specifics about the backend into the lowest-level API of the UI framework, because that will make migrating to a different backend harder). If the UI framework begins with merely some basic 2D graphics features, more or less any backend will do. As more elaborate features are added, such as full 4x4 transformation matrices, custom shaders, video, 3D models, or embedding textures from other libraries or applications[153], the demands on the backend will grow, and eventually there may be a point where it is cost-effective to build a bespoke graphics backend that is specialized to the needs of the framework.

---

[152] See the *Performance priorities* section immediately below.

[153] A typical use case for this latter feature is ads libraries. To reduce fraud, some vendors require that the library maintains some level of control over the rendering of advertisements and the handling of corresponding user input. This can require deferring compositing of the library's graphics and the application's UI to the OS. Another similar situation occurs with DRM video, where the graphics backend will be denied access to the pixels entirely and will need to coordinate with the platform to have the video composited into the application's UI.

Backend technologies

Immediate-mode vs retained-mode graphics

It is common to attempt to categorize graphics APIs as "immediate-mode" vs "retained-mode". Conceptually, in an immediate-mode API, the client (the UI framework) retains the scene description, and the graphics backend generates the pixels in direct response to calls into the API. As a trivial example, the UI framework might make calls to clear a rectangle, draw text on the rectangle, draw a box, then draw more text. If in a subsequent frame the text has changed, the client must send all the commands again, but with different values.

In supposed contrast, with a retained-mode API, the API itself holds a description of the scene, and the client instead mutates the description, e.g. changing the text in a scene graph owned by the API. The client subsequently tells the graphics backend to render its scene.

In practice graphics libraries use a mixture of these styles, and therefore this is somewhat a false dichotomy. Pure immediate-mode graphics is not reasonable given current hardware architectures; it would be impractical to upload to the GPU the entire geometry, all textures (images, font data), any shader programs, etc, each frame. As the graphics backend manages the GPU, if there is any data cached on the GPU, the graphics backend must have some form of retained-mode API to manage that cached state. Similarly, regardless of how "retained" the API is, there will typically be some form of canvas API wherein the framework can make draw calls to describe parts of the scene ("pictures"), and it is often impractical to have these pictures mutated each frame rather than just redrawing the entire picture when it changes.

Low-level graphics APIs (Vulkan, Metal, Direct3D, OpenGL, etc)

If designing a bespoke backend for a UI framework, one may consider whether to target Vulkan, OpenGL, or some other low-level API. In practice, each platform (software and hardware) has a preferred answer, and the right answer for the framework's backend is to target the platform's preferred answer. For example, today, on iOS, the framework will need to target Metal, and on Android, the framework will need to target Vulkan, falling back to OpenGL for older devices. The specifics vary over time and from platform to platform and therefore it is not useful to give concrete recommendations here.

Fonts

Text rendering will generally speaking require the use of Behdad Esfahbod's HarfBuzz library (combined with ICU for bidirectional text formatting); there are no serious cross-platform alternatives. The only other approach is to use platform-specific APIs on iOS, macOS, and Windows, and HarfBuzz on all other platforms, but the benefits are minimal.

Performance priorities

For several years, Flutter used Skia, largely out of convenience (the code from which Flutter was derived, Chromium, already used Skia). Skia is designed for optimal performance, by compiling custom shaders on demand and caching them for subsequent uses. This approach works well for Chrome, where the graphics backend is long-lived and shaders are shared among many web pages, so compiling a new custom shader is only rarely needed.

This approach has a drawback that was not originally fully appreciated by the Flutter team: on first startup, when no shaders are yet cached, drawing the very first frame of the application will take an appreciable amount of time, because all the relevant shaders must be compiled before the scene can be rendered. Often, applications start with an animation; if any frame of that animation requires new shaders as well, all those shaders will also have to be compiled before that frame of the animation can be shown.

Compiling shaders can take hundreds of milliseconds (for context, ideally an application would render frames every 8 milliseconds or so). This means that the first experience users had with many Flutter applications was a stuttering animation.

This problem does not occur for Chrome, because, frankly, users are accustomed to web pages having bad startup performance, and because the shaders are shared between web pages[154].

The first solution the Flutter team attempted was to allow developers to create a scene that would use all the necessary shaders, and then have Flutter render this scene first (off-screen), to trigger all the required shader compilations. This was later supplemented by some automation to aid developers in this process[155]. Neither solution was very satisfactory as a developer experience, and neither solved the fundamental problem; any shaders missed in this warm-up scene would still cause jank, and the application startup was still stalled by doing this compilation.

This problem was recognized by the team for a long time, but no solution could be found short of replacing Skia entirely, which is a massive undertaking. This was, ultimately, the path taken; the new library is called Impeller. As predicted, it has been a significant multi-year effort that is, as of the writing of this document, not yet complete for all platforms (iOS and Android are largely supported).

Impeller could, in principle, be used by other projects as well; it is not deeply tied to the rest of Flutter's engine.

Vector graphics

The state of the art for exchanging vector graphics is an undefined subset of SVG. (SVG itself is effectively a rarely used UI framework substitute for HTML, implemented by web browsers in parallel with HTML.)

SVG is not well suited for use as a vector graphics format for UI frameworks, largely for performance reasons. There isn't currently a really suitable standard alternative, unfortunately[156]. Instead, the common solution is to implement a custom binary format and associated renderer with good performance characteristics in the context of the UI framework, and convert SVG images to that format.

---

[154] It is unclear to the author why this problem is not significantly impacting Android applications either. Shaders are not shared between Android applications (doing so could theoretically open a privacy-leaking side channel), yet reports of performance issues on first launch of Flutter applications always implied this was a Flutter-specific issue and not common to all Android applications.

[155] Applications could be run in a mode where all shaders encountered were cached. This cache would then be shipped with the application, and the shaders in the cache would be compiled on first run.

[156] An earlier document by the author, *A vector format for Flutter (and beyond)* (2019), explores the requirements that would guide the creation of such a new format. The conclusion at the time was that to be sufficiently useful for the long term as to justify the effort of creating an entirely new industry standard, a format would need to be optimized for implementation entirely in shaders, and that compute shaders in particular would need to be more ubiquitously deployed in consumer hardware (especially low-end phones) before this was a viable approach.

An approach that could be investigated is compiling vector images themselves directly to bespoke shaders[157], rendering them to cached textures on the user's device at the appropriate size. This would move more of the rendering cost from the CPU to the GPU.

3D

A rarely used feature today that will likely gain more popularity in the near future is that of embedding 3D assets into 2D scenes. For example, a 3D game might have a 2D interface for creating a character, but want to show the user's 3D avatar in that interface, e.g. to preview the character's look during the game. Similarly a car sales application may allow the user to configure their vehicle, and may benefit from an inline 3D preview of the car showing the current configuration.

This requires surprisingly little flexibility in terms of 3D features. Being able to position an asset, or several assets relative to each other in a scene, with some basic lighting configuration, is likely sufficient, if the asset can be dynamically rotated and updated (e.g. changing the character's hat or the car's body texture and shader).

Assets

Applications typically have supporting images, fonts, sound effects, and other files[158]. These are known as *assets*. The precise mechanism for packaging assets differs from platform to platform, so some cross-platform API for defining which assets to package, and for obtaining them at runtime, should be exposed.

Typically the mechanism for specifying assets to ship is part of the build system. In the event that the UI framework is being built with a new bespoke programming language, or if the language used is able to be adapted for this purpose, a possible solution that would side-step some issues in existing systems would be to integrate the asset specification with the source files, maybe as part of a wider rethink of the build system.

An important feature to consider is the ability to preprocess source assets before they are included in the application. (In the same way that the source code is compiled before being included in the application.) For example, SVG files might be converted to a more efficient format, or font files might be processed to remove glyphs that are not used by the application.

Fonts

There is little in the way of design decisions when it comes to font support; developers expect the same features as they can get on every other platform. A new API can make some decisions that take into account some more recent developments[159], e.g. treating boldness as a numeric value (a 1000-point scale) rather than a boolean or ten-point scale, but other than that there will be little difference between what a new framework can support and what other frameworks already support.

One core feature that the graphics backend will need to provide for the UI framework, and one of the few features that must be designed for efficient two-way synchronous use, is text measuring. UI

---

[157] A proof of concept for this approach was created in 2018 by Daniel Zduniak.
[158] For example, a file containing the licenses of all the third-party packages used by the application whose licenses require attribution. The specifics of how to honor the license terms are out of scope for this document; seeking expert legal advice is highly recommended.
[159] As in, from the last few decades.

layout frequently requires fitting to text, which requires measuring the text, which requires doing almost all the work of rendering the text. Additionally, given the cost of these operations, *caching* these results is critical for performance. Designing the framework API to encourage treating text as an expensive resource helps here. For example, rather than having a canvas "`drawText`" call that takes a string, Flutter's Canvas API has a "`drawParagraph`" method that takes an object that must itself be pre-configured with text and laid out explicitly (which measures the text). This has the effect of reifying the cost text in the API, and thus encourages developers to cache the object for use across multiple frames.

Leveraging modern GPUs

Traditional UI frameworks and graphics backends could not reliably assume the existence of GPU compute shaders, or indeed GPU shaders of any kind. There has recently been work investigating the possible benefits of leveraging modern GPUs more extensively in graphics backends[160], by rethinking how graphics are rendered. To some extent, this can be considered an implementation detail of the framework, but in practice different approaches like this have different trade-offs. For example, some of these experiments work well for fills but are less efficient for strokes; being aware of these performance characteristics is important for designing a UI framework because developers should generally be guided towards fast paths.

Render objects

Most UI frameworks cache their layout computations using a tree of objects representing the layout geometry.

This is not necessarily required. A UI framework with a very fast (and likely simplistic) layout system can recompute the layout whenever necessary. For example, this is the approach taken by Dear ImGui. However, this approach is naturally constraining. Animations become quite expensive (as the entire application must be rebuilt and layout recomputed for each frame). The upper limit to how complicated the interface can be is much lower than with a system that caches layout.

We will discuss a high-level description of a way to implement render objects in the framework internals section below.

Scope

Frameworks that *do* have render objects generally fall into two camps in terms of the complexity and power of individual render objects in their library. This was briefly discussed earlier in the context of performance. There are other trade-offs to consider too, however.

The traditional approach, used (for example) by HTML and CSS, is to have a small number of extremely powerful big primitives. A node in a CSS render tree supports a large range of features. Take the `display: block` primitive: it supports sizing, padding, margins (including collapsing margins between adjacent blocks), borders (with individual control of each side's styles and thickness and the radius of each corner), backgrounds (multiple layers of backgrounds, which can be colors, gradients, or images, and each of which can have a different scrolling regime, blend mode, clip mode including being used as the color of any descendant text, alignment and origin,

[160] For example, vello.

repetition control, and resizing control), multicolumn layout with column gaps and column rules, internal scrolling of children, etc.

This approach is quite powerful. Especially after decades of features being added to such a system, it provides developers with a staggering amount of control over the result. It also works well as the basis for a system that uses a visual interface builder, because each node can have the same set of properties applied, which provides the developer with a consistent experience.

The opposite approach is to keep these features separated, with independent components that use a common protocol (such as the one described later in this document). Thus, for instance, one might have a render object that sizes its subtree to a specific width and height, another that sizes its subtree so as to enforce a specific aspect ratio, and another that sizes its subtree to a fraction of its parent's size.

This has several advantages of its own. It forces the framework to consider how to enable such features to be built. This means that developers can create their own primitives, which is ultimately even more powerful than the big primitive approach, albeit at the cost of requiring that developers learn how to build the primitives themselves. Such primitives can also be distributed in libraries, which enables the community to build an ecosystem of layout features that the framework maintainers never considered.

The other big difference between the big-primitive approach and the small-primitive approach is that with big primitives, the framework developer must consider the interactions between each of their features. As the number of features increases, this becomes exponentially more difficult. Efforts can be made to attempt to mitigate this by making sweeping statements about how the features work, but there is a limit to how far this can help. For example, CSS must define how the drawing of background images is affected by the padding, border, and margins of a box, the internal scrolling feature affects both padding (is it inside or outside the scrolling region) and background (does the background scroll with the contents or with the outer box), and so on.

In the small-primitive approach, there is nothing to define except how each primitive interacts with a very well-defined protocol. Background images don't need to define how they interact with padding if the padding primitive and the background image primitive are separate. Instead, the developer can control how they interact by controlling how they are composed.

Coordinate space

Most UIs operate within a very limited range of coordinate space. In the simplest case, the coordinate space is literally the screen (or window on the screen). In more elaborate cases, the range may be several multiples of this, e.g. in a social media application where the user is scrolling for hours. Some animations may play with a few orders of magnitude around this, e.g. a zoom animation when transitioning between screens.

All of these ranges are easily handled by floating point math[161]. Position information (e.g. for painting or hit testing) can be determined by accumulating offsets and applying transforms while walking down the tree.

---

[161] Or even fixed-point integer math, and it may be tempting to do everything in integer math, either for performance reasons or correctness reasons. Experience suggests the performance difference is negligible and the correctness argument is based more on a fear of floating point based on misunderstandings than on actual errors. That said, there is no harm in investigating this and testing it, as with everything else.

There are rare cases where this is insufficient. For example, if the application pans to positions more than 10^{15} pixels from the origin, even a 64 bit floating point number will not have enough precision to represent a single pixel, and the user experience will feel jittery. (This is a situation that can arise in applications that naïvely try to draw space scenes covering a lightyear or more, for example[162].) There are two approaches to addressing this use case: the most obvious is using higher-precision math[163] (e.g. 128 bit floats); the less obvious but more practical solution is the technique usually called *floating origin*. This is widely used in games with large world spaces, and consists of ensuring that the parts of the UI that are visible are positioned near the origin (where there is enough precision), with the origin shifting as needed when the scene is panned (as opposed to shifting the "camera"). Supporting this in the context of the UI framework based on a tree of render objects is tricky, as it requires starting the computations in the middle of the tree and working both up and down.

Realistically speaking, it is unlikely that supporting such scenes are of high enough importance to justify either technique.

Painting and compositing

Painting, as discussed earlier, is the action of turning render objects into concrete drawing instructions. Compositing is the process of combining images described by painting, so as to form the visual image that is the output of the UI framework. For example, there might be a background image, and a scrolling component with text inside it. The text would be scrolled to the appropriate position, clipped to the dimensions of the scrolling component, then *composited* onto the background.

It is generally cheaper to recomposite a set of pictures obtained from painting, than it would be to repaint those pictures. Thus, for instance, in the aforementioned scrolling example, it is likely more efficient to paint more of the scrolling content than fits on the screen, and then recomposite it multiple times as it moves (applying the clip and combining the background image), rather than repainting the content each frame. (In the case of scrolling specifically, there is also a trade-off to be made between minimizing the cost of painting content that is not visible, and painting enough content that compositing can be used to achieve the scroll effect without repainting every frame.)

Some render objects are atomic units in the compositing process. For example, a render object that represents a video will need to insert a node into the compositing tree that represents the video. In order to draw both under and over such nodes, ancestors of such render objects would need to be able to add nodes to the compositing tree that are both before and after (and between) their children. The video itself may come from a different library or even different hardware; for example, video encrypted for the purposes of digital rights management (DRM) may be decrypted on dedicated hardware, and the output may be available only in the GPU. In that kind of situation, compositing is required because the framework can never have access to the underlying pixels to paint them.

---

[162] If a single pixel represents a meter, then the precision of a 64 bit float allows for addressing individual pixels up to a distance of about 2^{53} meters from the origin, which is just under one light year. Of course if the scene includes details smaller than a meter, the range is smaller; 2^{53} nanometers is barely 10,000km.

[163] This is not really a practical solution, because there is little if any hardware support in current consumer devices for this, and doing the math in software would be crippling to the framework's performance even in the vastly more common situations where it isn't needed.

A similar case where compositing is needed is combining the content of multiple applications or independent libraries in a single application. For example, a game may have a 3D scene created in Unity, overlaid with a 2D UI scene created using the UI framework. A compositor is required to combine the two scenes into the final output. This process would also be used to combine the map from a library like Google Maps, or a web view displaying HTML content, with content from the framework.

Compositing may in some cases be delegated to the operating system or windowing system. This is especially likely in cases where the content in question cannot be accessed by the application, as in the case of DRM video mentioned earlier, or when the application with a scene from another application in a different security domain[164].

Integration with the composition or immediate-mode UI layers

As noted earlier, a potential novel design would be to merge the render object layout walk with the immediate-mode UI walk. While this approach is not the one described in the *Framework internals* section below, it may be worth investigating.

IDE

Some UI frameworks tightly integrate with a dedicated IDE, e.g. to enable a graphical UI designer to be source-aware. For example, the Delphi IDE allows the developer to position a buttons in a form (an application window), then double-click the button to generate a callback that is invoked when the button is pressed, with the IDE taking care of registering the button and its position, and setting up the callback.

A simpler solution is to create an IDE plugin for an existing IDE. For example, Flutter has a plugin for VS Code, which allows developers to inspect the widget tree at runtime, to better understand how their application lays out.

In general, *requiring* a specific IDE risks alienating developers who have already heavily invested in a particular solution which they feel enables them to be highly productive. It is important therefore to ensure that a UI framework's tooling can work independent of IDEs, e.g. entirely as command line applications.

On the other hand, not having an IDE solution at all requires new developers to make yet another decision when choosing to use the framework, and they may very well decide to choose another "simpler" one instead (meaning one with fewer up-front decisions).

For this reason, recommending a specific IDE, ideally with framework-specific support, should be considered a prerequisite for a complete solution. Supporting other IDEs (without reducing the recommended path to merely being one option among many) is naturally even better, but it must be acknowledged that each IDE supported represents a significant additional investment.

---

[164] The latter is a rare situation, but could arise e.g. if the application is a window manager, or if the operating system allows applications to be composed (e.g. a search application embedding the user's calendar application and a mapping application), a scenario that is likely to become increasingly common if typical user interactions move towards AI-mediated dynamically generated UIs.

Ideally, code should be shared between the command line tooling and IDE tooling. This can be done by having the IDE tooling use the command line tooling directly, or by having both work as different front ends to a shared library.

API design

Specific API proposals are described later, but there are some general choices one can make when designing APIs to improve the usability of the API. See also the principles for designing a programming language, which cover other API design principles that apply especially to a standard library.

Removing errors

It can be useful, when considering an API's design, to examine the total space of possible inputs, including "invalid" inputs in particular.

A detailed example

For example, consider this API that gives access to a list of strings in a file:

```
int openFile(String name);
String readKeyFromFile(int file, int index);
void closeFile(int file);
```

This is a simple API. The consumer calls `openFile()` to obtain a file handle, then calls `readKeyFromFile()` to get a value from the file, giving the file handle and the index of the desired string (zero to N-1, for some unspecified value of N specific to the file), and finally when the consumer is finished with that file, they call `closeFile()`.

When enumerating the possible inputs to this API, there are a number of cases that are possible but invalid: `openFile()` calls with invalid, non-existent, or inaccessible file names (quite likely in a situation where code written for one platform is compiled for another), `readKeyFromFile()` with invalid file handles or invalid indices (invalid file handles being quite likely as confusing the order of the arguments would be very easy and not caught by the type checker), and `closeFile()` with invalid file handles.

This API could be redesigned to avoid some of these failure modes:

```
KeyFile openFile(File source);
class KeyFile {
  String readKeyFromFile(uint index);
  void closeFile();
}
```

In this API design, the `openFile()` function takes a `File` object (which we assume here is some object from the standard library) rather than a filename, decreasing the chance of giving invalid file names, and moving the risk of giving inaccessible files to another API (which may be designed to reduce that possibility also). The `readKeyFromFile()` function is now a method on the value returned from the `openFile()` function, which means it cannot be called with an invalid integer file handle, and it has only one argument, which means the arguments can't be passed out of order. It also now takes an unsigned integer, which prevents negative values (which are guaranteed to be

invalid) being passed as indices. Finally `closeFile()` is similarly a method and therefore cannot be given an invalid file handle.

There are still error cases, however. The `File` object could represent a bad resource, the `readKeyFromFile()` index could be higher than the maximum valid value, and the two methods could be called on a `KeyFile` object that has already been closed. Avoiding the latter problem probably requires a language that supports enforcing object lifetimes statically (similar to Rust). Whether this would be an improvement or not depends on how convenient such a mechanism is compared to the danger of the developer calling `closeFile()` prematurely. The invalid `File` object could be partially mitigated by careful design of the `File` API, though no design can be foolproof (the file system can change dynamically while the program is running, in ways that are undetectable until an I/O call is attempted) and the convenience of being able to provide a hard-coded file name may outweigh the risk of that file name being wrong.

The last remaining risky input is the index. This could be mitigated as well. For example, the index could be interpreted modulo the number of entries, so e.g. specifying index 4 when there are 3 strings could return the same string as string 1. This reduces the number of error cases in the API, but in practice would not be a great choice as it does so at the cost of hiding programming errors and instead exposing them to the user as bad data.

Instead, the API could be changed so that it has an affordance to express that the index is out of range. For instance, it could return a string when given a valid index, and null when given an invalid index. In a language with null safety, this would require that the caller verify whether the returned value was valid (or explicitly require the caller to indicate that they are taking responsibility for the error if it is not).

Other examples

Leveraging types is a common way of reducing the potential error space. Instead of a defined finite set of integer constants, an enum can be used. Instead of encoding structured data in a string, a record type or other data structure can be used. Languages with particularly powerful type systems can go as far as encoding units in types, e.g. so that a time is distinct from a quantity, and dividing a quantity by a time gives a rate, and multiplying a rate by a time gives a quantity, and dividing a quantity by a rate gives a time (with no implicit conversions allowed).

Conclusion

In general, APIs should be designed to reduce the surface area of invalid inputs. Sometimes it can be difficult to find convenient, ergonomic ways to do so. Other times, however, attempting to remove invalid inputs can dramatically simplify the API. Looking at an API with the lens of trying to see how it could be accidentally misused is a great way to tend towards more usable APIs.

Keep concepts orthogonal

Where possible, keeping concepts orthogonal makes them easier to reason about. It is often tempting, when designing APIs, to try to solve all the problems at once; the result is often more complicated than the equivalent would have been if orthogonal concerns were split apart.

An extreme example is the CSS rendering model. Boxes have margins and padding and borders. A box can be block or inline. When a box is a block its margins act differently than when a box is

inline. An inline box can be an image (a "replaced element"). When it is an image, its padding is ignored. The width and height of a box control the space between the padding, but that can be changed so it is the space between the borders. This has no effect for replaced elements, where there is no padding. But for inline boxes, the width and height has no effect at all, those sizes being determined from the content. And so on. If you start pulling on a thread related to one property in the CSS model, you eventually find it is connected to everything else.

On the opposite end of the spectrum, one could design a rendering model like Flutter's (e.g. the suggestion below). There, padding and borders are separated into unrelated features that speak the same protocol but are unaware of each other. The concept of sizing is separated into another unrelated concept. A box cannot change from block to inline. Inline boxes and images are separate concepts. The general approach is to define a protocol (an API used to connect many disparate features), and then create lots of primitives based on that protocol.

This allows developers to learn about concepts in isolation. When they learn the protocol, they can then see how each concept is implemented, and they can build their own variants.

Concrete examples

Contrast, for instance, the way that padding layout is implemented for HTML, with the way padding is implemented in Flutter.

In Flutter, the padding layout code is about 200 lines of simple code in one file, only a few dozen lines of which aren't boilerplate or documentation. The code's correctness can almost be determined purely by inspection.

In contrast, in Firefox, the padding code is spread across several files implementing the CSS layout model, e.g. nsContainerFrame.cpp, nsBlockFrame.cpp, and nsInlineFrame.cpp all contain code referencing "padding" (these are not the only relevant files, but they're a start[165]). Looking through the code reveals any number of seemingly unrelated concepts very nearby; for example, code relating to the legends of <fieldset> elements, the sizing of tables, and shrink-wrapping are all intrinsically linked to how padding works.

Lest this seem like a criticism of Firefox, Chromium's implementation is similar, q.v. layout_box.cc, or block_layout_algorithm.cc.

Consistent naming conventions

One of the simplest ways to make an API easy to use is to be rigorously consistent with naming conventions.

This can be especially difficult in large teams, and is one reason to be deliberately slow in growing a team, to have a detailed onboarding process, and to use care in granting the privilege to review and land code into the codebase.

Some suggested naming conventions:

[165] The links are to files as of the time of writing in August 2025, in case the code is refactored, but the general approach hasn't changed since this code was written in the late 1990s.

- Subclass names ending with the name of the superclass (e.g. a FooWidget is a Widget subclass), and more importantly, avoiding the opposite (so never having a FooWidget that is not a Widget subclass). There are corollaries and exceptions to this:
    - When a class is likely to be used a lot and its class hierarchy is either obvious in context or not important, then having a concise name is worth more than having a name that gives a detailed inheritance tree. For example, widgets are actually a case where it's probably better to just have them be clearly named (Button, CheckBox, Row, Stack) rather than duplicating the word "Widget" everywhere.
    - A class name that is likely to be used in a lot of other class names is a dangerous choice, because it'll make a lot of classes appear to be a subclass. For example, avoid having a Controller class, because lots of unrelated classes are likely to be named ThisController or ThatController.
    - This pattern does not work well for deep hierarchies.
    - This pattern does not work well when the important base class is not the immediate superclass.
- Distinguish callback properties, handlers (callback implementations), and callback types. A common convention for callbacks is onFoo for the property, with handleFoo as the function, and FooHandler or FooCallback as the type. If that convention is in use, never have a non-callback property called onSomething, and never have a callback property that doesn't follow that pattern.

These conventions can become especially difficult to maintain when two namespaces clash. For example, when the framework is required to expose a platform's system API.

Follow other frameworks

Where it is reasonable to do so, re-use existing terminology from popular frameworks rather than inventing new terminology. This can help bootstrap new developers by leveraging their existing knowledge. (However, avoid doing this if the concepts do not align very closely, as that would mislead developers and hurt their ability to pick up the framework.)

Avoid advertising in APIs

It is tempting, when designing an API, to name it after the reason it was created, i.e. naming the API after the advantage of the API. This can often be misleading. Instead, either name APIs objectively, or explicitly after their disadvantages.

For example, if there is a `foo()` API that "foos" a list, and there is a need for a variant that is less precise but significantly faster, it may be tempting to name the new API `fastFoo()`. This would be a mistake, because it would suggest to developers that the trade-off is between doing it fast or not doing it fast, in which case, why even have the non-fast API? Instead, a name like `approximateFoo()` would be clearer. Documentation for `foo()` could point to `approximateFoo()` as an option when speed is an issue, and developers reading the code would immediately be able to tell what trade-off is being made.

If renaming the original API is also an option, `pairing approximateFoo()` with `slowFoo()` might make sense.

Consider having an explicit lightweight API review process

It is very common for new team members to begin writing code before they have become experts in the UI framework. In a UI framework, where almost all code is an API surface, this can lead to accidentally introducing inconsistent naming. If the code reviewers themselves are also not deeply familiar with the naming conventions, such mistakes can slip through undetected and by the time they are noticed it may be impractical to change the code.

This can be easily avoided by requiring all changes that involve any API surface whatsoever to have a quick API review by a senior team member who is deeply familiar with the naming conventions. This should not be a complete code review (code review should still be done, but separate from the API review), and it should not be an in-depth review of the API itself (the purpose is to keep the API conventions, not to ensure the quality is high; the quality issue should be part of the normal code review process). It is critical to keep this process lightweight, both to reduce the burden on the senior team members who would be completing these reviews, and to avoid any unintentional incentives with negative consequences (e.g. encouraging developers to avoid making API changes so that they can avoid the API review).

The main audience for an API is not the expert

While the people who will be *using* the API the most are those who have extensive experience with it, they are also the people who have the least trouble understanding it. On the other hand, new developers have no experience with the API and the easier it is to get started, the more likely they are to become experts and continue using the API.

This extends even to relatively low-level or otherwise "advanced" APIs. It is not uncommon for a developer to be an expert, write code using low-level or advanced APIs, then not look at the code for months or years, only to return to it and be expected to be proficient at it with minimal ramp-up time. The simpler the API can be, regardless of how low-level or "advanced" it is considered to be, the more supportive of the developer the API is, and thus the more the developer will appreciate the API and even come to love it, which, as previously discussed, is the best way to become popular with developers.

Make performance obvious

The performance characteristics of APIs should not be surprising. Developers should generally be able to guess the performance of an API from its name or how it is used. (This is sometimes achieved via naming, as in the `slowFoo()` suggestion from the previous section.)

Indicators of performance include[166]:

- APIs that use indexing (e.g. `foo[x]`) are expected to be O(1).
- Property getters (which look like accessing a variable) are expected to be fast (and O(1)). When something is expensive, it should be a function call, not a property getter.
- Functions that apply to N objects are generally expected to be O(N) or better (e.g. mapping over a list).
- Sorting (in the absence of specifics, e.g. when named "sort") is expected to be O(N log N) or better.

[166] These assertions should be tested with usability testing.

- APIs that use algorithms with known complexities should be named according to that algorithm. For example, cocktail sort is a sorting algorithm that is O(N) on sorted input but $O(N^2)$ on random input. A function that sorts using cocktail sort should be named `cocktailSort()`, not `sort()` or `quickSort()`, to make it immediately clear to the informed reader what is happening, and to make it clear to the uninformed reader that something specific is happening that can be determined from the documentation.

This is important, especially in the context of a UI framework, because so many operations are themselves O(N) that it is very easy to accidentally introduce $O(N^2)$ logic (or worse), which can potentially destroy performance in deployed scenarios (where there is more data than the developer expected).

Programming language

The previous section discussed three approaches to selecting a programming language for a UI framework: using a popular language, using bindings to target many languages, and creating or adapting a custom language. In this section, we explore the characteristics that would make a custom language particularly well-suited for the purposes of building a UI framework[167].

Some of the principles that apply to language design for a UI framework are the same as those that apply to the framework itself. Where relevant, this section will defer to the earlier section rather than repeating the same concepts.

Cost

Creating a new programming language is an absurdly expensive proposition. It requires designing a language, creating compilers for each supported platform, creating developer tooling for those platforms, creating a standard library, writing extensive documentation, creating tutorials, and fostering an ecosystem that allows developers to focus on their application instead of having to

[167] Anyone creating a new language should, of course, study the literature extensively. For example, Chapter 4 of Bjarne Stroustrup's 1994 book *Design and Evolution of C++* is still relevant today.

reinvent the wheel every time they need to use a data structure or algorithm that is not built into the language. All of this takes years and a big team.

The rest of this section largely glosses over this cost, but anyone seriously considering taking this approach would be well served to keep it in mind. Creating a new programming language is something people do as a hobby. Creating a new successful programming language *ecosystem* is something that happens comparatively rarely.

High-level goals

The premise is to create or adapt a language such that it is able to reach high popularity among a wide audience, specifically for the purpose of building GUI applications using an associated framework.

Audience

Initially, the target audience will largely consist of engineers with prior experience with existing languages, including notably one or more of[168] JavaScript, Python, Java, C#, and C++.

In the long term, success would mean that the audience may in time shift to including new developers as well, who have no significant background in engineering (in the same way that Python and JavaScript are often the first language experienced by developers today).

Purpose

The primary purpose of the language is to support a UI framework. However, this comes with two caveats.

First, a UI is of no use without business logic, so it is important that it also be useful for writing the business end of the application (the alternative, having people write the business end in some other language and then combine the two, is more or less universally unpopular as an approach and has, to the author's knowledge, never been successfully applied as a strategy for a UI framework that strives for widespread popularity).

Second, languages have a longer potential life than UI frameworks. Some care must therefore be taken to not overspecialize the language for the UI framework. Indeed, making the language attractive for folks who are not specifically looking to solve problems the UI framework solves, for example making it appropriate for writing network servers, command line tools, or games, can make the UI framework itself more attractive by making the adoption of the new language less of a risk for developers.

Overall approach

Rigorous, evidence-based, validated design

The single most valuable tool available to a language designer is usability testing. Everything about the language design can and should be tested. Many decisions in language design are permanent decisions that are impractical to change later; the cost of a mistake early in the design process can

[168] See the earlier discussion regarding the most popular languages today.

therefore be the difference between a language that is 30th in the rankings, and a language that is 1st in the rankings. Catching such problems early is therefore absolutely critical if the goal is popularity.

This is controversial. Most language designers have a strong idea of the language they want to create, and bristle at the implication that language design is something that can be approached in a systematic and scientific way, with ideas rigorously being tested.

It's also important to pair usability testing with a language designer who is able to keep the entire design coherent. It would be easy to fall prey to the trap of making each decision based on isolated tests, without considering the overall design holistically.

Familiarity

Designing a language that feels familiar to developers is key to attracting existing developers. This pushes the design towards either a C-like language (*à la* Java, C++, C#, Dart) or a Python-like language (*à la* Python, Nim, Mojo). Other styles, such as those based on Lisp (e.g. Clojure), ALGOL (e.g. Delphi's ObjectPascal), or array languages (such as APL), are much less commonly used today and are less likely to be seen as familiar to most developers, although languages like Swift and Ruby (with small but still robust communities and styles that merely take *inspiration* from C or Python) somewhat extend the envelope of what can be considered familiar.

What feels *familiar* need not be *identical* to existing languages. The core of Dart, for example, manages to feel familiar to Java developers without having adopted Java's verbosity. The key here is to *test*. Presenting prototype code and asking developers to describe what it does (measuring both how correct the answers are and how quickly correct answers are obtained), as described above, as well as presenting developers with sample code and asking them to write new code to achieve specific tasks using the same language (and measuring again the correctness and speed of answers), is key to determining what is familiar.

Approachability

While existing developers migrating to a new language benefit from being able to reuse their knowledge from previous languages, new developers learning to program for the first time cannot rely on such knowledge and must learn to write code afresh.

Designing the language to be approachable when one has no prior experience is non-trivial but once more, testing is key. Participants with an interest in programming but no significant experience could be found in first year CS courses, as well as in roles adjacent to software engineering (e.g. UX designers, technical writers, or quality assurance engineers[169]).

Principles

This section describes best practices in terms for designing a language and, especially, its standard library. How to apply these principles can be difficult to determine, and concrete proposals should be tested as discussed earlier, as there may be exceptions to all of these rules[170]. (Many of these

[169] Not that *all* UX designers, technical writers, and quality assurance engineers will have no programming experience; but *some* won't and would be excellent candidates for testing a language's approachability.
[170] Or they may be entirely wrong.

principles apply to the design of the UI framework APIs as well, and will therefore be revisited later.) These are listed in no particular order.

Solve real problems

It is very easy to get distracted by hypotheticals and theoretical conundrums that, in reality, never or rarely affect developers writing applications for real users. Always bringing things back to concrete problems (e.g. specific applications that actual developers using the framework are trying to create) will keep the project grounded. Usability testing is also a good way to focus on real problems. If something never comes up during a usability test or in feedback surveys, that suggests it is less important than what did come up, and should be accorded a proportional amount of time and effort.

Corollary: Avoid implementing features "for completeness"

While there is value in consistency (see below), this should not be used as an excuse to add significant features that exist only to serve some aesthetic need for completeness. If providing one special case is simpler than providing a general solution, and if the only problem to be solved in the relevant space is solved by the special case, then that may be sufficient.

The good path should be easy

By and large engineers are fundamentally lazy, and will take the easy path. Therefore if there are two ways to do something, and one is considered the best practice, it should be the easier one. Live usability testing can be a good way to find places where developers consider the good path harder than some other less optimal path (whether consciously or not).

Corollary: The bad path should be harder, but not necessarily impossible

Having made the good path easy, one may be tempted to make the bad path artificially hard or even impossible. This would be misguided. There may be valid use cases that one has not considered for which the "bad" path is reasonable.

Corollary: The bad path should still be documented

There is a tendency among some programmers, upon learning of some undocumented feature, to be compelled to use it, like a toddler who has been forbidden some secret candy[171]. Hiding a feature thus tends to be less effective at avoiding usage than documenting with clear warnings. In general, one should trust developers to generally do the right thing when fully informed.

Corollary: Avoid APIs that encourage bad patterns

There are many kinds of "bad" APIs, but some patterns are particularly common and can be avoided.

One such is O(N) algorithms in functions that are very likely to be called while iterating over the same data structure. For example, an algorithm that processes a list that is likely to be called while processing that list. Such APIs encourage accidentally creating $O(N^2)$ algorithms.[172] One sometimes

---

[171] This document's author must admit to having fallen prey to this kind of thinking on occasion.
[172] This is also a concern in the framework's API design, as previously discussed.

sees this with string manipulation APIs, e.g. a "length" API that must walk the string to determine the length, which is then used while walking the string to parse it.

Another common situation is designing APIs that use strings to describe parameterized operations. For example, using strings containing SQL to describe queries, or strings containing JSON to describe configuration, or strings containing file names plus some extra characters to describe how to open the file. Such APIs are very likely to be misused in ways that lead to code injection vulnerabilities (or similar).

Expensive operations should look expensive

Developers should be able to get an intuition about how many CPU cycles or how much memory some code will use by merely looking at it.

One example of applying this principle[173] is to make O(1) operations look like variables (e.g. property getters), and O(N) operations look like function calls. For example, `foo.length` should be O(1), while `bar.length()` can be O(N).

This kind of thing can be effectively tested using surveys, by presenting examples of proposed APIs and asking developers to predict their performance characteristics. Where developers are consistently predicting incorrectly, the API is misleading. Where developers are not providing consistent answers, the API is confusing or unclear.

Language features should be reasonable to use

Sometimes languages have features that are so expensive (or otherwise problematic) that the community decides that the best practice is to avoid those features. Ideally, one would avoid introducing any such features, and never require someone to write a style guide that discourages the use of any core language feature.

This is one case where usability testing is not the best solution. Instead, regular review by experts without a close tie to the language is more effective. (This can take the form of private requests, or commissioned reviews; but it can also take the form of public roasts by experts affiliated with competing languages. It is useful to seek feedback from any source, regardless of the form.[174])

Be consistent

Developers are fickle creatures, and an easy way to lose their favor is to be unpredictable or inconsistent[175]. There are many ways to be consistent: having a reliable naming pattern for APIs, having syntactic patterns that apply uniformly, having limitations that apply uniformly; the entire design space is rife with opportunities to apply consistent rules. Noticing patterns early and documenting them in a style guide can be a great help here. Usability testing, the reader will be unsurprised to read, is an invaluable tool for maintaining consistency. In particular, watching

---

[173] This also applies to API design, as discussed earlier.

[174] Sometimes wading through caustic biased feedback from a hostile commenter with an axe to grind will reveal nuggets of valuable insights. (Such feedback should not be accepted within the confines of the community, as discussed later, but can often be found in other fora, like blogs or YouTube comments.)

[175] Consider, for example, how quickly boringcactus dismisses Dart based on a syntactic inconsistency: "Flutter is a Google framework for cross-platform UI development, but you use it from Dart. Dart has switch statements and switch expressions with completely different syntax. Dart sucks." There are reasons for this design choice, but developers don't care, they just want things to work reliably and predictably.

developers using a product, with an eye specifically for spotting where their assumptions do not match the current design, is key for tracking down unexpected inconsistencies.

(That said, as noted above, one should avoid the temptation to implement features *only* for consistency and completeness.)

Avoid interleaving concepts

Where possible, keeping concepts orthogonal makes them easier to reason about[176]. For example, if the language has both classes and generics, it is likely to be simpler if the generics apply uniformly to all typed declarations (e.g. stand-alone functions, methods, classes, etc), rather than making them a feature specific to classes, with class-specific syntax. Another example would be requiring that exceptions be instances of classes, rather than allowing any value to be thrown.

(That said, if tying two concepts together makes the overall design simpler, then it is worth considering.)

Avoid API cliffs and API oceans

A programming language's standard library typically has to expose wrappers for platform APIs, e.g. file system and network APIs.

When wrapping such APIs, there are two failure modes to avoid, which could be seen as opposite ends of a spectrum.

The first, which is termed an API cliff, occurs when the underlying API is incompletely wrapped. In this situation, an API developer may find themselves suddenly unable to complete their task using the provided API, and forced to fall back to some lower-level API (e.g. directly communicating with OS APIs). The standard library should be sufficiently complete that developers rarely find themselves unable to complete their tasks.

The opposite failure mode, sometimes called an "API ocean", occurs when the provided APIs are so extensive that developers find themselves at a loss to determine which API to use. This often happens when the API is designed with insufficient layering. For example, a standard library that exposes basic arithmetic and trigonometry functions, complex numbers, statistical analysis functions, matrix operations, numerical integration, fourier analysis, and a high-level human-language query evaluation system, all as one API, would be overwhelming. On the other hand, such a library could be split into layers, built on each other. The dependencies between these libraries would then itself provide documentation for how developers could think of the APIs. (This would also possibly point to a way that some of the higher-level functions could be moved into an independent package separate from the core language and its standard library, without removing any capabilities from the developer.)

Avoid performance cliffs

A developer should be able to guess correctly whether a particular change to code might cause a dramatic change in performance. Specifically, a developer should never be confident that a change will *not* cause a significant performance regression when it actually will.

---

[176] This principle applies even more strongly to API design, as discussed earlier.

This kind of expectation might be violated when a compiler has an arbitrary limit after which it changes its code generation approach. For example, a compiler that uses a radically different structure for a function with N arguments vs a function with N+1 arguments (for some large value of N) might violate this principle.

A more likely trigger for a violation of this principle is when the compiler detects a pattern that it can optimize to functionally equivalent but logically different code. For example, noticing that four numeric variables are being manipulated in the same way, and converting that code to use SIMD instructions. Slight changes to the code might invalidate the pattern. If the developer is unaware that the code is being optimized in this way, or could be optimized in this way with minor changes, they may make those changes without fully understanding the implications.

In general this can be difficult to guarantee in all cases (assuming the compiler is always attempting to obtain best performance). For example, a performance cliff that is introduced when a function goes from being able to be compiled using one register per variable to requiring some variables to be spilled onto the stack is unavoidable.

Compilers could help provide developers with a better intuition and better situational awareness by having a mode that flags code that is being optimized in this way. For example, IDEs could subtly highlight areas of code that are being rewritten in this manner, and provide a brief explanation of the function of the equivalent code. Such a feature would allow developers to notice if their changes remove that annotation. It would also give them a feeling for when they might expect to see such an annotation, which would let them investigate cases where they do not.

This principle is closely related to the next one.

Avoid magic

Code should do what it looks like it's doing. When the compiler converts code significantly away from the developer's expectations, debugging becomes very difficult. Predictability is reassuring. The most cutting criticism of a programming language is that it is exciting.

Corollary: If the magic is good enough, nobody will mind

In practice, CPUs today execute code out of order and speculatively execute both sides of a branch, compilers convert linear code into vectorized instructions, operating systems preempt execution and reposition code in memory... computers in general violate the "avoid magic" principle all the time! Mostly, this is fine, because the abstractions are sufficiently air-tight that nobody notices and users get better performance. It is when the abstractions leak that developers become frustrated.

Corollary: Explain the built-in features using the same mechanisms developers use for their features

Trying to minimize the number of core features that are strictly necessary to explain the rest of the language leads to a simpler language. For example, a language could have a variety of integer types: signed and unsigned, of different bit widths, e.g. int8, uint8, int16, uint16, etc. The language could define these as built-in types. Alternatively, it could define a single mechanism for defining integer types as a range of numbers like 0..255, -128..127, 0..65535, -32768..32767, etc. In the latter case, there is only one primitive feature to explain: how to declare an integer type as a range (and how the compiler decides how to represent that data type). The actual built-in types are then just cases of

the standard library using this one built-in feature, instead of each of them being a separate built-in feature.

When selecting which features should be the core features, the key is to find the complexity minimum. The core feature that explains other features needs to be fundamentally simpler than what it is used to explain. For example, one could explain the language syntax as being merely a manifestation of a built-in programmable parser system. However, that would *increase* the complexity, not decrease it, because now instead of a simple syntax, one has to understand how to program a whole metasyntax[177].

Corollary: When the code is ambiguous to the reader, it should be an error

It is always possible to design rules to disambiguate confusing code. For example, one could allow multiple variables with the same name to be in scope simultaneously, and one could define rules for disambiguating which variable is being referenced when that name is used. However, if that disambiguation is not obvious to the reader, then some readers will assume it is disambiguated a different way, and that will lead to bugs. Therefore it is simpler all around if the compiler refuses to disambiguate such confusing situations and reports an error instead. (Determining what is truly confusing and what is actually uniformly understood even though it's arguably ambiguous is a task well suited for usability testing.)

Corollary: C-style "undefined behavior" is dangerous

The C standard allows compilers to make certain assumptions (e.g. that dereferenced pointers are not null), and allows compilers to optimize code based on these assumptions. In extreme cases, this can lead to the compiler emitting code that the developer in no way ever intended (there are many reports of significant vulnerabilities being introduced in C and C++ code because of this).

A trivial example of this is the following program:

```
#include <stdio.h>

int main(int argc, char** argv) {
  char* a = NULL;
  printf("argc=%d\n", argc);
  if (argc == 2) {
    printf("argc=%d\n", argc);
    return argc;
  }
  *a = 0; // UB; cannot dereference null pointer
}
```

As compiled by clang 13.0 and later[178], this program, when called with no arguments, prints the following and returns 2:

```
argc=1
argc=2
```

This is because the final line would result in a dereferencing a null pointer, which is considered undefined behavior; the compiler therefore assumes that the code cannot reach this point, and

[177] Also such a feature would encourage developers to invent their own syntax and every program could end up using its own private dialect that nobody else understands. That way lies madness.
[178] Tested with versions of clang through 20.0 and with trunk as of the writing of this section.

determines that this must mean that the earlier `return` must always be reached, which implies that `argc == 2` is true.

This behavior is *very confusing*. It has value when the programmer can guarantee that their code is correct[179], as it allows for very aggressive optimizations, but it is quite misleading and is likely to trip up developers unexpectedly. In practice, optimizations this aggressive are only useful in very performance-sensitive code, such as codecs, or the "hot loop" of a UI framework. A better solution for enabling these kinds of optimizations would be for the language to have explicit features to enable them. This could take the form of a variant on the "assert" statement that allows the compiler to assume the assert will not fail, or special types that encode specific assumptions (e.g. an integer type that is defined to not support overflow).

The "assert" approach has the advantage of being self-documenting. Combined with aggressive warnings from the compiler to point out situations where the compiler cannot prove that the code is correct (e.g. in the example above, pointing out that the code appears to have dereference null), or cases where the compiler can prove that the assertion is wrong, allows the compiler and the programmer to work together to improve the quality of the code while simultaneously enabling optimizations.

Avoid global state

This applies primarily to the standard library, rather than the language proper: APIs should not have global state, but should instead operate from instance data. For example, rather than a global "random" function that uses a globally-set seed, a data structure can be created on demand to represent the current random state and that data structure can be queried to obtain a random number. This approach makes testing code significantly easier; for example, tests that use random numbers can now be initialized with a specific random state, rather than interfering with each other's global random state.

Other examples include getting the current time (which can instead be abstracted out to a clock API, some variant of which can be driven from the real time, but which can be substituted for a fixed-time variant for tests), accessing the file system, and accessing network APIs.

Introduce concepts with keywords

For example, have a keyword such as `function` to declare a function, rather than implying its declaration from the syntax (as in the C style).

This is a more controversial principle; the fashion among newer languages is to eschew keywords in the interest of brevity even more than C does. However, there are several benefits to explicit keywords:

- It makes searching the codebase easier. You can more easily find concepts if there is an explicit keyword that introduces it.
- It is easier to introduce a concept when teaching a language if the concept has a concrete keyword.
- Keywords are harder to obfuscate. Whether through malicious (or cheeky) intent, or through accidental misformatting of the source, code that relies on punctuation and context alone can

[179] They cannot.

mislead the reader more easily than if there are concrete unavoidable keywords serving as signposts.

Corollary: Avoid unintuitive punctuation

A lot of programming languages make heavy use of punctuation in ways that are not immediately obvious. While usability testing should be used to determine what precisely is intuitive in practice, in general, if one is attempting to create a language that optimizes for familiarity and approachability, this tendency should also be avoided.

One way to determine if a particular kind of punctuation is intuitive is to create a small program that uses it, and then quiz developers (e.g. using a survey) about the specific meaning of each segment of punctuation. Another is to ask developers to write code in the same style, solving a slightly different problem, to see if they are able to intuit when particular punctuation is needed and what it means.

Avoid double negatives

It is sometimes easy to think of values as changing the default, so, for example, one might need a flag to "disable" some feature that is normally enabled. The problem with such APIs is that they lead to double negatives, as when one needs to specify that the feature is enabled, one must write code along the lines of `disabled=false`. Double negatives are harder for humans to understand[180]. For this reason, it is best to avoid negated identifiers like "disabled".

Strings are sequences of extended grapheme clusters[181]

The naïve implementation of the core string data type is to expose raw bytes, or raw Unicode scalar values[182]. A data type that represents raw bytes is useful but is not a text string, it's binary data. Exposing Unicode scalar values, however, is useful and reasonable.

As the underlying encodings for Unicode data (usually UTF-8, sometimes UTF-16) are variable-width encodings, indexing into a string is a potentially expensive operation (worst case O(N) in the length of the string for indexing to the last Unicode scalar value). To reflect this and make the good path easy, strings should be iterable but not indexable.

Unicode scalar values, however, do not represent what users think of as *characters*. That is better represented by extended grapheme clusters. These are themselves varying substrings of Unicode scalar values, and thus, strings should also be iterable as sequences of extended grapheme clusters.

Make tooling output actionable

As discussed earlier, every error message is an opportunity. Another aspect of this is that all log messages should represent a problem that developers will want to fix. "Informational" messages, or warnings that developers can safely ignore but cannot silence, are noise that reduce the developer's willingness to read the output at all. The more noise a developer sees, the less they will notice the really important messages.

---

[180] And may be confusing; in some languages, double negatives are used for emphasis and have the opposite meaning as the equivalent in English.
[181] Ok this one is weirdly specific but strings are really fundamental to languages, get introduced very early, and very few languages get this right, so it's important to mention it prominently.
[182] The *common* implementation of strings is to expose raw UTF-16 codepoints, which stretches naïveté to the breaking point.

Everything should be loggable

As codebases grow, the ability to meaningfully step through code in a debugger reduces, and the value of logging data while debugging increases. To support the ability to log data, every value in the language should be usefully printable. A useful stringification is one that fits in around 80 characters or less (usually a lot less), and has the salient facts about the value clearly stated, including if necessary the type itself. For example, a `Point` structure with x and y coordinates might stringify as `Point(10.0, 20.0)`.

Support small programs

New users will begin by making very small applications (e.g. "Hello World", or a counter application, or a to-do list). These should be possible without excessive ceremony. Ideally, a single file should be all that is necessary to represent a simple application, and it should compile from anywhere on the file system.

Support large programs

If programmers adopt the language, some will eventually write very large applications, split across many files, developed by many people with little communication. Some form of project-internal packaging to manage code and reuse it in different applications will be necessary. Extensive unit testing, integration with CI/CD systems, source control repositories, and ways to tie in custom source code analysis will be expected. There should be a clear path from "small program" to "large program" that developers can smoothly follow.

Minimize overhead

Avoid features that would require extra memory to be allocated for every instance just in case it happens to interact with that feature. Costs should only be paid when a feature is actively used, and only to the extent that the feature is used.

For example, one could implement a locking mechanism (for thread safety) by adding a pointer in every allocated instance, just in case that object is ever locked[183]. Alternatively, one could have a dedicated lock object that is only allocated when one requires locking. In the former case, an object-heavy single-threaded program pays a huge amount of memory (and some CPU cycles due to the data being further apart and thus caching less efficiently) for a feature it never uses. This kind of design should be avoided.

Similarly, avoid features or patterns that require extra CPU cycles even when the feature is not used.

Documentation is critical

We will discuss the principles behind good documentation in more detail later, in the context of documenting a UI framework. Those principles apply equally well to the language and its standard library. In short, documentation should be *useful*, and should take as its basic premise that if the reader is reading the documentation, it is because the language is not sufficiently clear, and therefore the documentation must make up for this failing. Documentation should therefore explain even what the language designer thinks is obvious (because by definition, if the reader is reading it,

[183] This is how TMonitor is implemented in Delphi.

it is not obvious), and should explain how to use features (with sample code), how they work, potential pitfalls, etc. Empowering the reader should be the goal of every line of documentation.

A language reference is also critical

In addition to teaching folks how to use the language, a language should also have a formal reference that answers questions that would otherwise require studying the compiler's source code. For example, documenting the exact syntax of the language, the semantics of every feature[184], and as much detail about observable internals as possible (e.g. private APIs used by the standard library).

This kind of documentation is necessary to grow a high-quality ecosystem. For example, tooling that is based on parsing source code needs to know precisely what the syntax is. Substitutes for the standard library for use in obscure embedding scenarios need to know exactly what contract the standard library must implement to work with the compiler. A reference also serves to definitively answer questions that the more approachable documentation may gloss over, but which may matter in edge cases that more experienced developers find themselves dealing with.

Compilation of development code must be fast

As Hot Reload is a goal (which requires sub-second latency for the entire save-update loop to be useful), compilation must be extremely fast.

There are some design decisions that can help with this. One is to design the language specifically to enable one-pass compilation. Whether this is practical given the goals of familiarity and approachability is an interesting question that deserves investigation by usability testing, but it is worth keeping in mind as a principle where possible. Avoiding features that require deep analysis or lots of context can also help; for example, type inference should not require running a constraint solver.

All of this is specifically in the context of compilation during development. Here, the compiler must produce correct code fast, but does not necessarily need to produce especially fast or memory-efficient code.

Keeping the language fast as new features are added is important. Some language features are difficult to implement fast[185]. It is easy to fall into the trap of thinking new features are so important that they are worth sacrificing performance, but doing so risks making the entire development experience worse, even for developers who do not care about the new feature (because even if the feature is only slow when used, that feature might be used by one of the developer's dependencies). This discipline is especially hard to maintain because the most vocal members of one's community

---

[184] Some would argue that some semantics should be left open; C and C++ have many semantics that are intentionally defined as "undefined behavior", meaning that the compiler can assume that they will never happen (and can optimize around that assumption). This leads either to very confusing behavior for developers (because the compiler is able to make assumptions that the developer did not intend), or inconsistencies in behavior across platforms (because different compilers or architectures may operate differently in the face of undefined behavior). Confusion is bad for popularity, and inconsistency is bad for cross-platform application development, so neither is recommended here. Instead, one should strictly define the semantics of the language and implement it identically across all target platforms. See also the earlier section, *Corollary: C-style "undefined behavior" is dangerous*.

[185] For example, the Dart team aborted their attempt to add high-level macros in part because of the performance impact on compile times (and thus hot reload).

will be those who believe they would be happy to make the trade-off of losing performance in exchange for the new features![186]

Compilation of *production quality* code can be much slower, as the production flow is much more rarely used by developers. Thus, whereas the development mode compiler might skip applying most optimizations, the production mode compiler can perform very expensive whole-program optimizations, finding common code patterns to deduplicate, analyzing the optimal places to inline function calls, replacing virtual calls with direct calls where types can be proved to never require the virtual calls, etc.

Dogfood the language

As the premise here is that the language is being designed specifically for a UI framework, this should be simple, but to be explicit: the designers of the language should *use* that language on a day to day basis. (This is one reason to recommend self-hosting a language, if possible; by definition, this forces the developers of the language to use the language.)

This is another way to *test* the language, and serves a similar purpose as live usability testing. It cannot serve as a replacement for live testing, as the language developers are typically not representative of the wider audience[187], but it still serves as a complementary sanity check, and tends to catch really trivial issues that may be missed in usability testing and may never be reported by developers using the language (for example, typos in comments in the standard library).

Memory management

A decision should be made regarding whether the language is memory-managed or not. This affects a number of language features. A memory-managed language cannot[188]:

- ...allow arbitrary bitwise manipulation of its managed data structures, or blind-casting between statically incompatible managed types, as that would risk compromising data being held for the purpose of memory management.
- ...expose custom allocators for managed objects, as memory management is typically closely tied to specifically allocation schemes.
- ...have raw pointers to managed objects, or features like pointer arithmetic on managed data, as managed data may need to be arbitrarily moved in memory, invalidating all pointers.
- ...give developers control over decisions like stack allocation vs heap allocation.

Lacking these features means that a memory-managed language is probably never going to be the optimal language for highly performance-sensitive code, or for code that needs to closely integrate with hardware or low-level OS APIs.

On the other hand, language-mediated memory management frees the developer to focus on much higher-level concerns without risking invalid pointer dereferences and memory leaks[189]. React-style

---

[186] They may also be the ones who most vocally complain about the performance regression later, not understanding the relationship between their two requests.
[187] The language's designers are likely to know the language too well to feel the frictions that "normal" (less experienced) developers would encounter.
[188] The use of the word "cannot" in this section is not meant as a challenge, but as good practice. It is of course possible to have a memory-managed language that has custom allocators, allows arbitrary bitwise manipulation of memory, and allows pointer arithmetic; it's just that such a language would be extremely bug prone and would largely defeat the advantages of having the language manage the memory in the first place.
[189] Ideally. Some situations, especially involving captured variables in lambdas, can still effectively leak memory.

expressions building giant arbitrarily-sized data structures would involve a lot more verbose ceremony if the developer was required to manually free each such expression when it was no longer required[190].

Impact of hot reload on the memory management decision

Designing the language to enable hot reload, also discussed earlier, is critical if the framework is to support it.

This almost certainly requires creating a memory-managed language, as stateful hot reload requires the ability to resize live objects, and thus update pointers, during program execution. This requires that the developer be unable to "hide" pointers from the runtime by, e.g., casting them to integers or worse[191].

Mixed mode

One option is to provide a hybrid approach, with manually managed raw memory, and a separate managed world. In a strongly typed language, the language can enforce that pointers can't be taken to managed objects, thus safeguarding the garbage-collected heap and allowing the compiler to manage objects on the stack, while still allowing applications to directly interface with platform APIs like `mmap` where raw memory access is required.

This could follow a pattern similar to that used by Rust, which distinguishes between a "safe" regime and an "unsafe" regime, or Dart, whose FFI API allows access to raw pointers, without exposing pointers to the garbage collected heap. Another approach is discussed in a little more detail below, in the section on novel features.

Minimize dependency on an external build system

Reflecting the need of the UI framework to avoid a complicated build system, the language itself should fundamentally not require a build system to be compiled. This can be achieved by inlining features that would normally exist in separate files into the syntax and making the compiler responsible for the build process.

However, to ensure that the language can also fit into projects where the use of this language is not driving the entire project, the compiler should still be amenable to being driven from existing build systems, e.g. with command line arguments that direct it to output intermediate build products.

Openness

The benefits of keeping the entire toolchain open are discussed earlier, and apply equally well to the language and its tooling as to the framework as a whole.

---

[190] There are also many patterns, especially those seen in Flutter code, that would be borderline impossible if the developer was required to track the precise lifetime of each widget instance. For example, Flutter widgets commonly cache subtrees to avoid the overhead of rebuilding them each frame when they are known not to have changed, but such caches would need to coordinate with the subtree's original instantiators to ensure the memory was not freed too early or too late. Additionally, traditional allocation is much slower than a bump allocator (or equivalent) as used by memory-managed languages used with the React style. This means that it is not a given that such a lower-level language would even be faster than the equivalent code in a memory-managed language. As discussed earlier, the C layout library Clay avoids this problem by using fixed-size arenas to store the widget-equivalent data structures, but this has its own limitations: it prevents meaningful caching across frames, and puts a hard upper limit on the number of widgets in the application (and requires pre-allocating that memory even if it is never used). This is untenable for many applications.

[191] For example, xoring two pointers together.

One aspect of openness that could be considered is standardising the language with a body such as ISO or ANSI. In practice, this does not seem to have much benefit unless there is a strong desire to have multiple independent implementations of the compiler. The driving use cases discussed in this document do not require multiple implementations, so standardisation does not seem useful.

Tooling

The language should include libraries that handle common tasks such as parsing the language to an AST, serializing an AST back to code, making common kinds of mutations and introspections of ASTs (e.g. renaming members, reordering parameters, etc).

First, this encourages the proliferation of tooling in the language's ecosystem, because it makes writing such tools easier.

Second, it makes such tools more reliable, because they are all going to have a consistent interpretation of the language. It also allows tooling to evolve quickly with the language, because a new syntactic feature can be implemented in the language's library and all the tools can be recompiled with the new version, rather than having to wait for each developer to reimplement the syntax in each tool.

These libraries should be used to implement the compiler itself and any other first-party tooling for the language (including the formatter[192]), to ensure that all such tooling is consistent also.

Multiplatform

The ideal compiler would support building for each supported (target) platform from each supported (host) platform. This would be a required step towards having the framework tooling support building an application simultaneously for every target platform from a single host. The ability to do this is often requested, especially from small teams who do not have dedicated build engineers or dedicated build hardware[193]. For some platforms, notably iOS, this is technically difficult due to limitations imposed by the platform vendor.

Web

Looking forward, targeting the web is probably most conveniently and efficiently implemented by targeting Wasm. As with other platforms, this requires some mechanism to bind to the platform APIs (in this case, the DOM APIs exposed in the JavaScript runtime, with which the Wasm runtime can communicate).

See also the earlier section regarding the UI framework targeting the web, where the compilation target is discussed.

---

[192] The formatter is called out explicitly here because formatters have notoriously annoying requirements on such libraries. For example, they may require the AST to include insignificant whitespace and comments, not to mention the ability to parse code inside comments recursively, handle (and round-trip!) syntax errors gracefully, and be functional even when dependencies are missing. If the language's library can be used to build the formatter, it will probably handle most other use cases.

[193] The ability to cross-compile is also rather useful if the compiler is self-hosted, as it reduces the bootstrapping problem to just the first platform.

Security

While undervalued by the industry in general, a new language ideally would use best practices for supporting supply chain integrity systems[194]. For example, builds should be bit-for-bit reproducible, ideally independent of the host device (e.g. building a Windows build of an application from source should give the exact same bits whether compiled from a Macbook Pro, a Linux workstation, or a Windows machine, assuming the same compiler version and source code is used in each case). The compiler should be capable of generating signed provenance attestations.

Programming paradigms

Strong static typing

Types are an invaluable tool for helping developers catch errors in their code. Having strong static type checking would differentiate the language from JavaScript and Python, and would help developers be more productive[195].

Strong typing here is in contrast with weak typing. In a weakly-typed language, using a value of one type where another is expected will silently convert the value to the expected type before performing the operation. For example, arithmetic on a string may first parse the string to an integer, or silently treat the string as zero, or similar. In a strongly typed language, it is an error to use a value of one type where another type is expected. Static typing means that such errors are caught by the compiler (or earlier, e.g. by a linter running in the IDE).

Implicit types

One area where testing will be necessary is determining the right balance between requiring type declarations, and avoiding type declarations where types would be redundant. Developers express dissatisfaction with redundant type declarations, but may not always agree with a language regarding which type should be inferred. Inferring a wider type than expected will miss errors; inferring a narrower type than expected will force the developer to be more verbose than they may desire.

Situations requiring ambiguous types

It can be difficult, in languages that only have strong static typing, to create friendly APIs for parsing mixed-type data formats like JSON, CSV, YAML, etc. Creating a library that implements parsing of arbitrary JSON files and exposes a clean API, using the language and early in the language's development, forces the needs of such APIs to the forefront.

Options include one or more of:

- A compile-time-untyped, runtime-type-checked type that can stand in for other types, with some form of dynamic dispatch.
- Some sort of variant record or tagged union type with accessors that perform the runtime type checking.
- Multiple dispatch (function overloading).

---

[194] See, for instance, SLSA.
[195] Research supports the argument that strong typing has a positive impact on developer productivity: https://www.inf.unibz.it/~rrobbes/p/ICSE2014-docstypes.pdf

- A mechanism to define type coercions between primitive types and custom types.

These all have advantages and disadvantages, and the general direction of the language will influence which options are most applicable.

Soundness

Some languages have type systems that do not strictly guarantee correctness. For example, Kotlin generics are unsound due to JVM type erasure[196]. TypeScript is similarly unsound. This leads to potentially difficult to understand runtime behavior (and crashes), and furthermore limits the compiler's ability to reason about types in the language, preventing several categories of optimizations.

When designing a type system from scratch, it therefore behooves the language designer to keep the type system sound.

Class-based object-oriented programming

GUIs are especially well served by object-oriented programming (OOP). While the fashion in our industry is to move towards interface-based polymorphism (e.g. traits in Rust or protocols in Swift; discussed below), GUIs specifically benefit greatly from being able to incrementally build an implementation alongside the interface. For example, defining the logic for rendering a tree, independent of the coordinate system, and then layering in Cartesian assumptions.

The safe option is to go with class-based object-oriented programming (as used by many of the existing popular frameworks discussed herein). This is an area where some experimentation may reveal a new model that is even better suited for UI. However, there is a risk in creating something too novel. As noted above, familiarity is one of the best ways to attract developers. Novelty will slow down adoption.

Visibility

At a minimum a language should support public members (visible to all), private members (visible only to code from the class itself), and protected members (visible only to code from the class and its subclasses).

Other design decisions may impact the precise semantics of visibility specifiers, and may suggest the need for more fine-grained control. For example, can other code in the same file see private members? Can code running for one instance of a class read private fields of another instance of the same class? Only the precise shape of the language, and lessons learned from testing it, can answer these questions with certainty.

A possible innovation here would be defining visibilities that control the provenance of code that can use a member. For example, a protected function that can be overridden, but only called by code in another (explicitly named) class hierarchy, or some mechanism to flag references to an instance as being allowed to use a particular API or another via the type system. This would enable classes to be described not just in terms of "public" and "private" (or "protected") but in terms of the identity of API consumers and API implementers, which might reduce a category of bugs caused by misunderstanding (or intentionally misusing) APIs.

---

[196] See, e.g., https://arxiv.org/pdf/2408.10804

Simple metaclasses[197]

Most languages that support classes do not expose them as first class values within the language itself. For example, there is typically no way to pass a *class* as an argument, allowing the receiver to then construct an instance of that class. This feature is especially useful for serializing and deserializing trees of objects, which is itself something that is often useful for a UI framework to support. For example, it may enable a simpler mechanism for dynamic server-driven UI[198].

Essentially this requires only three features in addition to the more commonly supported class features: class reference types, virtual constructors, and virtual static methods. Class references are values that identify actual types at runtime, virtual constructors are constructors which can be called on class references to construct instances of those classes, and virtual static methods are methods that can be invoked on class references in a similar way. They are *virtual* because, like regular virtual methods, the compiler uses dynamic dispatch to select the actual code executed at runtime.

A language that supports class-based object-oriented programming and aims to be especially useful for UI frameworks should probably add support for these features.

Multiple inheritance

Some mechanism for sharing code between otherwise unrelated classes is useful for UI frameworks.

Multiple inheritance in the C++ style is rarely what developers need when reusing code (in particular, the duplication of superclass fields is unlikely to be what is desired in the context of code reuse in a UI framework). The Python approach, while *less* confusing, is still rather bug prone. Some form of mixin feature that presents a more immediately clear class hierarchy is likely to be less confusing. (This is an area where early usability testing using paper prototypes would be extremely valuable, allowing study participants to describe their understanding of code given various different syntaxes.)

Composition

We assume, for this discussion, that the UI framework will use a modern React-style (immediate mode) programming paradigm, as that is the state of the art in terms of API design for UI. This implies the use of composition, which in practice requires some form of polymorphic data structure and references thereto, for example objects in an object-oriented language.

Imperative programming

All the most popular programming languages are imperative, so any attempt to make a language that feels familiar to existing developers will benefit from also supporting that paradigm.

---

[197] Simple, as opposed to elaborate metaprogramming or macro features that are sometimes referred to as "metaclasses".
[198] Delphi uses this mechanism to serialize forms (UIs) created with its IDE to disk in a format that can then be decoded at runtime using runtime type information (RTTI). This is central to Delphi's design. RTTI as implemented in Delphi and FreePascal (the leading open source ObjectPascal compiler) is significantly more detailed than the equivalent features in standard C++, more akin to the kind of type introspection found in many managed languages.

Declarative and functional programming

React-style immediate-mode programming typically involves creating expressions to represent each component in the interface (sometimes rather large expressions). The language and standard library can greatly affect the convenience of this style. For the language, this revolves mainly around syntax (discussed below); for the standard library, this is mostly a matter of providing efficient functional-programming features, such as ways to manipulate lists and dictionaries (aka maps, hash tables).

Features for child lists

For example, it is common in UI frameworks to manipulate lists of child nodes. Business data in the form of a list may need to be filtered (removing certain values), sorted, then mapped to create a list of controls (applying some function to each item in the list). Having to run imperative code for each of these steps can be awkward if the final result is to be inserted somewhere deep in some nested expression. A functional API around the list primitive can provide a more pleasant developer experience.

Callbacks (function and method references)

A React-style framework would be difficult to use without some mechanism to trigger code in response to events. The most basic form of this is the ability to reference some function, and pass that reference to other parts of the codebase. For convenience, such references must be able to have attached state, e.g. an object instance (a method pointer).

Closures (anonymous function references with their scope)

A particularly convenient way to implement callbacks in a UI framework context is to use closures (anonymous functions whose references include a handle to the state that was in scope when the closure was evaluated).

Closures in loops

A seemingly minor detail, but one that is especially important in the context of a UI framework, is that a closure inside a loop construct with a loop variable should close over the value of the loop variable *during the iteration where the closure was evaluated*, as if the variable's scope was the loop iteration itself and not the whole parent function. For example, a loop over an array using some variable *foo* that creates a button for each value in the array, with that button have a callback closure that uses the value of *foo*, let's say to print it to the console, should act on the value of *foo* for the specific button that was created, not the last value for all buttons.

Consider for example this pre-2015 JavaScript:

```
const callbacks = [];
for (var i = 0; i < 10; i++) { // after 2015, "let" solves this problem
  callbacks.push(function() {
    console.log(i);
  });
}
callbacks.forEach(function(c) {
  c();
});
```

This prints the number "10" ten times. The equivalent using `let` instead of `var` prints the numbers 0 to 9. The latter is what developers expect[199].

Event-driven programming

GUI applications uniformly use an event loop. It would be difficult to develop a practical programming language that did not support creating an event loop, but for completeness, we must note that supporting one is a requirement of a UI framework on its language.

Generics

The ability to parameterize types and functions is expected by developers using typed languages at this point[200].

Language features

As noted earlier, the need to feel familiar to Python, JavaScript, Java, C++, and/or C# developers pushes the language towards either C-style or Python-style. Many C-style languages exist. This is a relatively safe choice[201]. Python-style languages are rarer; Nim and Mojo may be the only recent examples of self-contained languages that resemble Python. The syntax of GDScript, the scripting language used by the Godot game engine, is also quite similar to Python.

Both options are valid, as demonstrated by the popularity of Python and JavaScript. The main disadvantage of Python-style significant whitespace is that some environments do not preserve whitespace, which makes copying code around more awkward. Clearly, however, this is not a fatal flaw, given Python's success.

Table stakes

There are some syntactic choices that most major languages, whether Python-like or C-like, share in common; deviating from these norms will cost significantly in terms of familiarity for existing developers[202] and should be avoided unless it brings some benefit that truly outweighs this cost (ideally proven with usability testing).

Character encoding

Regarding the character encodings supported for the source files written in a programming language, there is only one reasonable choice today, which is to require the use of UTF-8 and support no other encodings. (Naturally the standard library itself should support a wide range of character encodings, this is only a question of the source code itself.)

---

[199] This claim is based on experience fielding bug reports in the pre-2015 JavaScript world and the Dart world (Dart has the suggested semantics). Naturally, as with everything, this should be tested.
[200] For example, as Wikipedia puts it, "the lack of support for generic programming in initial versions of Go drew considerable criticism". Go famously did not add generics for many years, and it was the biggest or nearly biggest request from developers for many years.
[201] Research suggests the C style can be improved; q.v. https://www.vidarholen.net/~vidar/An_Empirical_Investigation_into_Programming_Language_Syntax.pdf
[202] The precise operators listed here should be tested to double-check the accuracy of the claims made herein before developing a language based on these claims. The norms vary over time, the author may be biased by their own preferences, and some of the more rarely used operators may be unfamiliar to the majority of developers despite being defined identically in all popular languages.

Comments

C-style languages use `/* ... */` for block comments and `//` for line comments. Python uses # for line comments. Which one is used should be based on whether the language as a whole uses the C style or the Python style.

Assignment using =

The vast majority of programming languages use a single equals sign to specify value assignment.

There are a (very) few exceptions in the most popular languages, but by and large, initialization of variables, fields, and constants, as well as declaring default values for optional parameters, are all done with a single equals sign.

In some languages, assignment is itself an expression. Testing may demonstrate otherwise, but in general it seems that treating assignment to be a *statement* is more likely to catch unexpected bugs (such as using = when the similar comparison operator, ==, was intended).

Comparison operators ==, !=, <, >, <=, >=

These operators are again pretty consistently used among the currently popular languages.

Arithmetic operators +, -, *, /, %, **

The unary and binary operators + and `-`, and the binary operators `*`, `/`, and %, are widely used in popular programming languages.

The division operator sometimes has a variant (e.g. Python has floor division `//`, Dart has integer division `~/`, Pascal has integer division `div`). Whether this is necessary or not depends on the precise semantics around numeric types in the language.

The modulus operator (%) has different semantics in different languages. In general, for a UI framework, truncated division (the most commonly used option, and also the option implemented in Intel hardware) is not ideal, as it leads to discontinuities when used for animations where the divisor changes sign. Floored division is probably the most flexible (and matches Python), though Euclidian is also an option (one used by Dart). Providing some mechanism (e.g. in the standard library) that lowers to the hardware definition is recommended for those developers looking for performance and for whom truncated division is sufficient.

The exponentiation `**` operator is not critical. Python and JavaScript support it, and it is convenient in the context of a UI framework (especially for curves and other animation-related expressions). If an exponentiation operator is included, `**` is preferred over ^, which usually means xor.

Logical operators !, &&, ||

The `!` (not) operator is unary, the `&&` (and) and `||` (or) operators are binary. These operators are widely used; and more or less universal among the most popular languages (Python being the main exception; it uses keywords like "`and`" and "`or`"). The binary operators are expected to be short-circuiting (evaluating the left hand side first, then only evaluating the right hand side if the logical result is not a foregone conclusion).

Bitwise operators `&`, `|`, `~`, `^`, `<<`, `>>`

These are similarly common. The ~ (bitwise not) operator is unary, the others — `&` (bitwise and), `|` (bitwise or), `^` (bitwise xor), `<<` (left shift), and `>>` (arithmetic right shift[203]) — are binary.

C# and Python[204] also have logical variants of the `&` and `|` operators, which are not short-circuiting (unlike `&&` and `||`). This seems like a potential source of bugs (accidentally omitting a single character, which is not obvious when reading the code, can significantly change the semantics of the code), and so may be worth skipping.

Additionally supporting `^` for logical xor (operating on booleans rather than integers) is a reasonable and probably harmless extension[205] (supported by C# and Python).

Some languages (e.g. Java, JavaScript) add a `>>>` operator for logical right shift[206].

Parentheses for expression grouping

Languages universally support wrapping expressions in parentheses to allow the developer to override operator precedence.

Member access (`.`) and indexing (`[]`)

Languages have, by and large, converged on using a single period (`.`) to indicate member access on a compound data structure, e.g. fields or methods on an object, fields on a record or struct, etc.

Similarly, many languages use square brackets as a unary suffix operator for indexing into a data structure such as array or map.

Some languages (e.g. JavaScript) more or less treat these two operators as equivalent. This seems unnecessary; keeping these operations distinct allows one to separate *code* access (compile-time verifiable, and possibly even statically resolvable) from *data* access (happening at runtime).

We will discuss the topic of operator overloading below, but regardless of how much support the language has for arbitrary operator overloading, it makes sense for developer-defined types to be defined to support an indexing operator. This also allows the language to explain how built-in data types implement that operator, possibly even allowing such types to be implemented in a standard library rather than in the compiler.

Operators for handling null values (e.g. `??`, `?.`, `!`)

C#, Swift, and Dart have converged on a common set of operators for handling null values. The `??` operator is a binary operator similar to `&&`, which evaluates to the left hand side unless it is null, in which case it evaluates to the right hand side. The `?.` operator is a variant of the member access operator that evaluates to null if the left hand side is null, and otherwise acts like the member

---

[203] Meaning sign-extending, as opposed to zero-extending (the leftmost bit is set to its previous value after shifting).
[204] It's not documented as such, but the effect is the same.
[205] One exception might be if the language supports pointers and uses Pascal-style pointer syntax (using `^` and `@`), which is, in the author's opinion, more intuitive than C-style pointer syntax (using `*` and `&`), though naturally this would have to be tested with developers.
[206] Meaning zero-extending (the leftmost bit is set to zero after shifting).

access operator[207]. The `!` suffix operator evaluates to its operand, and has as its return type a non-nullable version of its operand's type; but if the operand is null, the expression throws an exception.

Kotlin has similar-enough operators that these will likely also feel familiar to Kotlin developers (`?:`, `?.`, and `!!` respectively).

Assignment operators

Languages commonly support the non-comparison binary operators from the previous sections combined with the assignment operator, e.g. `+=`, `-=`, `>>=`, `&&=`, `??=`, etc. In general if this is supported, all binary operators other than comparison operators should have assignment operator equivalents, for consistency.

Ternary operator (`... ? ... : ...`)

The existence of some sort of conditional expression is important, especially for a React-style UI framework where subparts of large expressions need to vary based on state. For example, the fill color for a button might vary based on whether the button is being tapped or not.

If punctuation is used for this, the widely supported three part `?:` ternary operator (as in, `condition ? trueValue : falseValue`) is likely to be the most familiar. However, this operator is not without its problems, and another syntax, e.g. based around the `if` and `else` keywords, may be worth considering. This would be especially worth studying in the context of nested conditions.

Conditional expressions may also be worth coalescing into some form of a switch expression syntax. There is much less consistency around this form between popular languages, which affords some flexibility to the language designer.

Type operators (`is`[208], `as`)

This one is more controversial, but the industry does seem to be converting on "is" for checking the type of a variable, and "as" for a type-checking cast. C#, Dart, Swift, Kotlin, and ObjectPascal, use "is". Those languages as well as a few more (like TypeScript) use "as".

That said, there is a risk in terms of the familiarity goal here. The keyword "is" is a form of equality operator in Python; Python uses "isinstance" for type checking. Java and JavaScript use "instanceof". C++ does not really have a direct equivalent; the nearest idiom is to test if the return value of `dynamic_cast` is null or not.

This should naturally be tested, but this is a good example of an area where it is important to test the language design holistically — the specific operator "`instanceof`" maybe more familiar to many developers than "is", but would the language as a whole be more usable using "`is`", or "`instanceof`"? Keeping the overall goals of the language in mind when performing testing, and

---

[207] Technically in Swift the operator is just `?` followed by the regular member access operator. This means it also works with other suffix operators like indexing (`[]`). C# and Dart support `?[]` as well, but as an explicit variant of their normal indexing operator.

[208] The use of type checking of this form is generally a red flag, as it violates polymorphism and (in the context of UI frameworks especially) can lead to breaking composition. For example, if a widget checks the type of a child widget and acts differently for one specific kind of child than other children, then wrapping the child in a no-op widget (one that just immediately defers to its child in every way, i.e. the simplest possible widget) will break the functionality of the parent widget. This is counter to the premise of composition in a UI framework.

designing the tests to address the overall goals and design of the language, is what is important. It is easy to lose track of the big picture and instead collect data on very narrow questions that do not generalize.

Negated "is"

Dart has an "`is!`" operator (pronounced "is not"), Kotlin has a "`!is`" operator. These avoid a problem present in most other languages, which is that testing for an object *not* being of some type requires extra parentheses due to operator precedence rules (without the parentheses, the negation would apply to the value itself rather than the result of the type check):

```
if (!(value instanceof Foo)) { ... }
```

Python has an "`is not`" operator, which is the reverse of its identity operator "`is`". This is therefore not strictly speaking precedent for an "`is not`" operator meaning "is not an instance of the type", but nonetheless that form could be considered as well.

Consistency with a unary *prefix* `!` operator for logical negation suggests the Kotlin-style "`!is`" is the most idiomatically consistent syntax[209], even if it does not read directly as "is not".

Metaclasses and "is"

In a language where types are themselves values, it is worth considering supporting the case of the right hand side of an "is" operator being a variable whose type is itself a type. (In a language with generics, this can also be written as a generic function with one type parameter and one normal value parameter of arbitrary type, whose return value is the result of checking if the type of the value "is" the type parameter.)

Type inspection

The "is" operator checks if the value on the left hand side is of the type *or a subtype of the type* on the right side. Sometimes, it is important to check if the type is *exactly* equal to another; this in particular occurs in the reconciliation phase of React-style frameworks. Some mechanism to make such tests is therefore also important (as is its performance).

Operator precedence

In general, operator precedence should be implemented so as to match the most popular languages (meaning Python, Java, JavaScript, C#, C++[210]), which are all more or less aligned.

Identifiers

There is greater variety in conventions around identifiers in the most popular languages' ecosystems than there is in many other aspects of syntax.

The majority consensus seems to be that identifiers should be case-sensitive, that classes should use UpperCamelCase, and that methods, fields, and variables should use lowerCamelCase. There is

---

[209] This may even be a case where the difference is so minor that the results from testing would just be in the noise. However, if a language design team has taken the earlier recommendation to heavily invest in testing to heart, then this would be an interesting test case for the testing itself, to see whether objective data can be teased out of even the most minor differences.

[210] And notably not Pascal, which has confusingly different operator precedence rules for its logical operators (e.g. and).

little consensus on constants, though lowerCamelCase has the advantage that it reduces the cost of changing an identifier from being a constant to being a getter or variable, or vice versa, which happens surprisingly often when writing code for UI[211].

Instances of classes are often named the same as their class modulo case, in languages where identifiers are case-sensitive and types and variables have different conventions, e.g. `Game game` introducing a variable "game" of type "Game".

One inconsistency that is common in the popular languages is that the names of user types (e.g. classes) use a different style than the names of built-in types (e.g. the types for integers and strings)[212]. This would be an area where creating a clean and consistent style may improve approachability despite being inconsistent with every existing language; this may be worth testing.

While some languages still use underscores (e.g. Python, Dart), this seems to be less universally popular, and might be worth intentionally avoiding. Some languages or frameworks use naming conventions with prefixes (e.g. starting types with T); as with underscores, this is not widely used and may be worth avoiding.

Scope

Some languages have multiple namespaces, e.g. one for types and one for variables (for example, C/C++ and TypeScript). (This is separate from the concept of having multiple name scopes, such as those introduced by the `namespace` keyword in C++.)

Humans tend to think in terms of a single namespace[213]. When a developer sees "int" in a language where "int" is the name of an integer type. they think "that's the integer type". The possibility that it might be a variable name, parameter name, a function, or something else is unlikely to occur to them. Similarly, developers are unlikely to intentionally use the same name (with the same case) for their variables as for their types. Languages that make it easy to do this therefore risk making bugs easier.

This suggests that having a single namespace for all identifiers within any particular scope will lead to fewer bugs. It also has some other advantages: it avoids the need for C++'s distinction between `::` and `.` member lookup operators, and it allows types to be used as values (e.g. as suggested above for metaclasses).

Statements vs expressions

Some languages have only expressions, and allow an expression to discard its result when used as a statement.

---

[211] For example, a color might start as a constant, then later be made configurable.

[212] Seriously, this is so common it's absurd. Java and Dart have "`int`" but "`Object`". C# has special keywords, like "`int`", that map to the "real" types that use the other form, e.g. `System.Int32`. C++ has decades of accumulated synonyms for basic types, some of which have multiple keyword names (e.g. "`unsigned long int`"), some of which end with `_t` (e.g. `std::nullptr_t`), but classes are UpperCamelCase. Delphi uses a `T` prefix for types, like `TObject`, but the basic types (for which there are again decades of accumulated synonyms) are not prefixed with a `T`, e.g. `Integer`. Python types use a variety of styles for types, e.g. `int`, `fractions.Fraction`, or `NotImplemented`. Surely if one designs a new language, one could do better? Rust, which has a pretty clean set of basic type names, still manages to be inconsistent here, e.g. `u8` vs `NonZeroU8`, or `f32x4` vs `Simd<f32, 4>`.

[213] This claim is provided without evidence and is an example of something that should be double-checked with usability testing before being relied upon.

If, instead, the compiler flags any use of an expression with a discarded value, or if the language separates statements from expressions (possibly by having a keyword to convert an expression into a statement that ignores its value, like in Nim), the developer is much more likely to notice bugs where a typo turned a statement into a useless expression instead.

For example:

```
foo == bar; // oops, meant to assign bar to foo (using = rather than ==)
```

Control flow keywords `if`, `for`, `while`, `break`, `continue`, `return`

The popular languages all use the keywords `if`, `for`, `while`, `break`, `continue`, and `return`, with broadly similar semantics.

For loops

The "for" keyword's semantics in particular are the least consistent; it is used for two kinds of loops, the C-style initializer/condition/increment form and the for-in iterator form. The latter form has been added to all the popular languages over time. Both forms are useful.

Given the lack of consistency on exact semantics, this may be an area where some small amount of innovation would be welcome without harming the familiarity goal. For example, one common operation that happens in the context of UI frameworks and that is not directly supported in popular languages is looping over parallel arrays (which requires either an indexing variable or multiple item variables). Currently, this is generally implemented by using the for loop over a range, using the range of one of the arrays, after having asserted the arrays are identical in length.

Another possible area of innovation is finding a way to implement the iterator form, which loops over a collection object like an array, more efficiently. Most implementations today involve multiple function calls and allocation. In contrast, iterating over a range requires only a few instructions (increment, compare, jump).

Variable declarations

In the discussion on language principles, we discussed the benefits of introducing concepts with keywords. One place where this principle could be put into use is variable declarations.

There are four components to a variable declaration:

- A keyword to introduce the declaration (if the language requires one; possibly optional). This could be used to set variable semantics (e.g. mutability).
- The name of the variable.
- The initial value of the variable (possibly optional).
- The type of the variable (possibly optional).

Optional types

Developers are likely to request that the type be made optional. There is some tension between having the code contain explicit type information in order to increase the chance of catching bugs, having the code be self-documenting, having the code be easy to write, and having the code avoid redundant information.

For example, developers are generally likely to be unhappy with code such as:

```
var Map<int, String> table = Map<int, String>();
```

...which repeats a lengthy type twice with no immediate benefit. Omitting the type entirely enables a new category of bugs, for example, the compiler might not notice that in the following case, `loadTable` returns an integer rather than the expected `Map`:

```
var table = loadTable(); // what does this actually return? what is the
type of table?
```

On the other hand, saying that the type is (only) optional when the initial value makes it obvious leads to inconsistent code:

```
var table = Map<int, String>(); // initializer has obvious type
var Map<int, String> table = loadTable(); // initializer does not make
type obvious
```

This kind of inconsistency can lead to code that is harder to scan. Suppose integer literals are considered to be "obviously ints", but that expressions are not considered to have obvious types:

```
var a = 1;
var b = 1;
var int c = a + b;
var d = 1;
var e = 1;
```

Is that more or less readable than:

```
var int a = 1;
var int b = 1;
var int c = a + b;
var int d = 1;
var int e = 1;
```

This is the kind of question where usability testing (measuring people's actual ability to interpret code correctly, and how long people take to interpret the code) can really answer the question of what is better more accurately than intuition or asking developers for their opinion.

Immutable variables by default

It's surprising how often variables don't vary — that is, how often a variable is given a value once, and keeps that value for the remainder of the scope. Indeed, this pattern is probably more common than having a variable that changes value multiple times within a single scope, at least in the context of applications written with UI frameworks.

For this reason, it's often the case that setting a variable to a new value is actually indicative of a bug.

It's therefore probably the case that variables should be immutable by default, or should require their mutability to be explicitly set.

Function declarations

As with variable declarations, function declarations can be broken down into components:

- A keyword to introduce the declaration (if the language requires one).
- A function name (possibly optional, for anonymous closures).
- A parameter list, where each parameter has:
    - A name.
    - A type (possibly optional).
    - A default value (possibly optional).
- A return type (possibly optional).
- A body (possibly optional, in the case of declaring abstract methods).
- Zero or more annotations, depending on the needs of the language.

Optional types

As with variables, developers may request that types be optional. There is again a tension between having code be self-documenting, giving the compiler more information to catch bugs, and having the code be less verbose.

Named arguments

Most languages historically used exclusively positional arguments, as in, a function had a certain number of parameters and callers had to provide arguments in the same order. This works well for certain functions, e.g. a function that takes a single argument, or a mathematical function that has two obvious operands. In a UI framework, there are often functions (or constructors) with an inane number of arguments (e.g. at the time of writing, the Flutter `TextField` widget has 68 arguments).

The solution is named arguments: allowing the caller to specify the arguments by name instead of by position.

Infix notation

Some languages (e.g. Objective C, Kotlin) use some form of infix notation in addition to, or instead of, named arguments. This does not appear to have been widely adopted and seems likely to be a source of complexity that outweighs its advantages, but may warrant some testing if a particularly compelling approach is apparent.

Closures

Ideally[214], closures would just use the same syntax as function declarations (probably without requiring a name).

Being able to (or being required to) specify the return type may help catch bugs, but may be too verbose.

Null safety

Some types represent references to instances of data structures. Historically, these types could also represent the "null" reference, i.e. the absence of an instance. More recently, languages have moved towards having types that cannot represent "null", and having a separate type for the case of a reference that can be null.

Having this feature in the type system helps developers by removing a whole category of bugs.

[214] Well, probably. Usability testing could always prove otherwise.

Exceptions

Developers expect some error-handling mechanism. While there are various options, the most commonly implemented and therefore most familiar for developers is traditional `try-catch` exception handling.

Shared references

The nature of GUIs is that many different parts of the system need to communicate with many other parts of the system. For example, a layout component's size might be affected by the window dimensions, the dimensions of the component's children, the current font size, the position of the mouse, or some system accessibility setting. That same component may need to be notified by the mouse system when the user clicks its area of the interface, by the focus system when the user presses "tab", by the keyboard system when the user types while the component is focused, and by a system observer system when the user changes the system's color scheme.

All of this means that the language must support references to objects being shared by multiple other objects, and that mutations could be triggered via many codepaths.

Convenient trees

UI frameworks tend to operate heavily in the world of graphs, or, more specifically, trees, as discussed earlier. The language must therefore support creating data structures that represent graphs without great ceremony or boilerplate.

Some examples of syntaxes that represent trees include XML-like syntax, as seen in HTML and React's JSX:

```
<dialog>
  <heading>Trees</heading>
  <label> <checkbox value={treesEnabled}/> Enable trees </label>
</dialog>
```

...as well as C-style expressions, as used in Flutter:

```
Dialog(
  Heading('Trees'),
  Label(CheckBox(value: treesEnabled), 'Enable trees'),
)
```

These syntaxes have the advantage of being syntactically verified at compile-time. A mismatched closing tag, or an inconsistent number of close parentheses, is a compile time error that the developer will therefore immediately notice.

A typical syntax used to represent trees that is *not* convenient, and is indeed quite bug-prone, is the push/pop style sometimes seen in C APIs:

```
OpenDialog();
  OpenHeading(); Text("Trees"); CloseHeader();
  OpenLabel(); Checkbox(treesEnabled); Text("Enable trees"); CloseLabel();
CloseDialog();
```

Despite being apparently equivalent to the earlier examples, this style suffers from a lack of compiler support. An omitted or mismatched "close" will not necessarily be noticed during development, and may result in wildly confusing behavior at runtime, ranging from vague diagnostics (if the library attempts to detect such mistakes), to very broken UI, to hangs or crashes in the worst case[215].

Modules (packages, libraries, units, partitions)

There are two levels of code separation that are commonly desired.

The first is splitting code into multiple files for the purposes of code organization.

The second is splitting groups of such files into reusable modules that can be shared with other developers or used in multiple programs.

In both cases, there is the question of how to distinguish the exposed interface from the internal implementation.

In many languages today, there is either no formal mechanism, or there is only the concept of file-private members and public (exported) members. Some languages support or require the explicit separation of the interface from the implementation.

Separating the two (e.g. as required by ObjectPascal and as supported by C++) makes the distinction between what is exposed and what is not quite clear. However, this distinction may come at too high a cost; the code that implements an API is now separated from the code that declares the API, which requires both some duplication in the code (so that the implementation can identify what interface it is implementing), and makes following the code more difficult (because e.g. fields of a class are defined far from the code that uses them). This is an area that would benefit greatly from usability testing to drive the design.

Language server for IDEs

Every modern language must have an LSP language server for use from IDEs, for syntax highlighting, autocomplete, navigation, etc. (This should be implemented using the common libraries suggested above, as a way to ensure that those libraries solve real problems.)

Novel features

Creating a new language brings with it the opportunity to attempt to solve some problems in entirely novel ways. In this section, we discuss features that either don't exist in popular languages, or that do not yet have well-established patterns.

Compiler directives and a build system replacement

As discussed earlier with regard to build systems, one way to reduce the need for an explicit build system with build configuration would be to move all build and compiler configuration inline into source files.

Several languages support a small version of this. C++ compilers use a syntax derived from the C preprocessor, e.g. `#pragma pack(4)` to control the compiler's packing alignment for data structures. C# uses a similar syntax. Pascal uses comments prefixed with a $ sign, e.g.

---

[215] Did the reader notice the intentional error in the example?

`{$PACKRECORDS 4}`[216]. This kind of feature goes back a long way, e.g. Fortran has `!DIR$ ATTRIBUTES ALIGN: 4 :: myvar` to control alignment of variables. Other languages also support ways to control the compiler, e.g. Perl's "`use`" and "`no`" keywords can change compiler configuration dynamically (e.g. "use integer" to force integer math instead of floating point math). Python's "`from __future__ import`" is a similar concept.

The precise syntax to use is unclear, as existing languages that have any version of this vary wildly in their syntax. Some experimentation (combined with usability testing) would be warranted.

Configuration that could be moved from an external build file, or from the command line, to inline using compiler directives include:

- Everything to do with dependencies: specifying required modules, specific versions, where to obtain them, etc.
- The target platform(s): what platforms and architectures to compile for.
- Inclusion of debug information: whether to include symbols, whether to generate them inline or as a separate file, what information to include, what format to use, etc.
- Errors and warnings: what errors to flag, whether they should be fatal.
- Code generation: whether to include or omit runtime error checking, whether to enable various optimizations, etc.
- Asset processing: commands to run to generate source files (e.g. to convert a CSV to a source file that is then imported), files to attach to the resulting executable (e.g. images to ship with the application).
- Packaging: how the executable and assets should be packaged.

The more these features can be integrated into the language itself (rather than having two languages in the same source file), the more the developer will avoid running into the problem of having to learn a separate language to configure the build system.

Imports

A component of this is how to specify dependencies on local files or files in other (maybe third-party) modules. Avoiding an explicit dependency on the local file system path syntax is wise, as file systems conventions vary, but the syntax should be intuitive and support importing files relative to the current file, and files in other modules (so long as they have been declared as a dependency by some other file in the module).

Conditional compilation

To support multiple platforms, there typically needs to be some mechanism to decide which code to compile when targeting a particular platform.

Many languages have some form of compiler preprocessor directive to enable or disable code at the syntax level. The disadvantage of this approach is that it makes static code analysis impractical (any analysis must assume a particular configuration to see the code in that configuration, and may need to compute the complete set of possible configurations and attempt to compile the code for every single option to determine correctness in all cases).

---

[216] Pascal block comments use `{...}`.

An alternative option would be to have compile-time code evaluation, with compile-time constants that specify the target platform. Analysis can then analyze all branches, just as with branches that are evaluated at runtime. This, or some other mechanism whereby the compiler can *efficiently* verify that code provided for every configuration is correct, would be a very compelling feature, especially in the context of building cross-platform applications.

Inline assembler

Another feature that would be useful to build platform-specific code would be the ability to inline assembler in the source code, similar to `asm` blocks in C or Pascal. This is particularly useful when building the lowest levels of the standard library (e.g. threading), and for certain forms of optimization where particular instructions of the hardware architecture are known to be optimal, but that the compiler would not normally emit.

Pattern matching

Some modern languages have pattern matching syntaxes, allowing developers to branch to multiple code paths in one statement (or expression) based on some relatively elaborate conditions. In the opinion of the author, none of the solutions so far developed are compelling, and this is an area where more innovation is needed. Maybe the range of allowed patterns should be reduced, or new syntax could be found that is clearer.

Expressive expressions

Often, using a React-style UI framework involves creating large expressions to convert a (potentially elaborate) data structure representing the user data into another (potentially even more elaborate) data structure representing the user data.

Such expressions typically involve creating collections (lists, sets, maps), calling constructors with many arguments, and conditionally setting values based on the input data.

It can be extremely useful, therefore, if expressions have the ability to:

- conditionally include one or more items in a list or map literal.
- apply some function(s) to a list and insert the result into a list or map literal.
- loop over some list and create new values to insert into another list or map literal.
- conditionally include one or more named arguments in some function call or constructor call.

Having expression forms of control flow statements (`if`, `for`) that can be used inside collection literals, being able to group arguments and include them conditionally (`if` in argument lists), being able to "spread" a collection into a collection literal, etc, greatly help with this.

Dart's collection literal `if` and `for` syntaxes, and the various languages with spread operators (e.g. Dart's `...` or Python's `**`) could be used as inspiration here.

Transferring ownership of lists

If lists are mutable but not observable, then there is a risk that when passing a list (e.g. a list of children) to a new owner object (e.g. to a widget) that the list will subsequently be mutated without

the awareness of the new owner. This will likely lead to bugs and confusion[217]. Being able to freeze a collection or otherwise indicate that the list's ownership has changed would enable developers to catch such errors early. A copy-on-write mechanism for data structures would also allow such bugs to be avoided, albeit at a higher runtime cost[218].

Integration with other languages (FFI)

Some language support for cross-language communication is required to enable that support in the UI framework.

The form this takes necessarily depends on the target supported languages, but it basically boils down to:

- Have a way to declare the API to which the code wishes to bridge, specifically in a form that gives the compiler all the information that is necessary to implement the ABI correctly. This could take the form of redundantly converting the API to some new form, or it could take the form of importing the target code (e.g. the relevant C++ header files, if targeting C++).
- Define how the language's semantics map to the target language's semantics. For most languages, this should be relatively straightforward, e.g. calling a function maps to calling a function. When the semantics don't match well, this is trickier. For example, mapping to JavaScript is necessary when targeting the web (even if compiling to Wasm, currently, as the platform APIs are all JavaScript APIs), and a JavaScript object is really more of a dictionary than an instance of a class, JavaScript only supports doubles (and not, say, integers), and how to pass references to and from JavaScript from Wasm may not be obvious in all cases.
- Implementing that mapping for each supported runtime.

For some languages, this may require interfacing with the target runtime's garbage collection mechanisms. For example, on Android, directly interfacing with Java APIs requires participating in the Dalvik VM's garbage collection. One way to implement this would be for the compiler to output Dalvik bytecode instead of ARM machine code.

For other targets, it may require providing mechanisms for allocating and manipulating memory directly, not managed by the language runtime.

Garbage collection in the standard library

Traditionally, memory-managed languages implement the memory management in the compiler and in code emitted by the compiler that is not under the control of the developer.

One possibility would be to expose hooks that the compiler calls, and which can be configured from source (and which would be configured by the standard library). This could be done in a manner similar to how ObjectPascal enables the memory allocator to be replaced[219], or it could use a

---

[217] This is a common problem experienced by new users to Flutter. Lists are widely used, but the Flutter framework assumes that any list that is passed to a widget will never change. Developers see that they are passing a list to a widget, and assume that changing the list is enough to update the rendering. The widget sees that the list's identity has not changed, and assumes that means that nothing needs updating. If widgets could freeze any list they are given in their constructor, then this error would be immediately caught when the developer tries to mutate the list, thus saving them a lot of time.

[218] Typically, the old copy of the data structure won't be used after the new version is created, so a copy-on-write mechanism would frequently be redundantly copying data just before that data is discarded.

[219] The standard library exposes a set of function pointers that can be replaced; the compiler invokes these pointers when memory functions are called.

compiler directive system as described above to have a lexically-scoped mechanism (there would need to be some logic to handle the existence of multiple parallel memory management schemes simultaneously).

In general, having the runtime be implemented in the standard library using the language itself would be a very powerful and flexible mechanism (for example it would open up options for more efficient management of tree-wide data, as discussed earlier). While it would require careful design, it would not necessarily require significantly more effort than the usual mechanism of having the runtime be an opaque wrapper around code running in the language (in a manner reminiscent of a VM, though typically not literally implemented as a VM).

Raw memory features

Implementing garbage collection in the language itself would require also exposing some raw memory allocation features (so that the managed memory logic can put its managed data somewhere!). If the language has distinct types for reference-to-managed-object and pointer-to-raw-memory, and does not allow casting from one to the other (except, e.g., for the purpose of placing a managed object in memory), the safety of garbage collection and managed memory can be maintained while still having raw pointers and pointer arithmetic.

It would be useful if raw memory could be interpreted in a structured fashion (e.g. casting a raw pointer to a pointer to a struct, and then dereferencing it, in a language like C++). This may require having a set of types that represent raw data, and may mean that types like integers, floats, and booleans are distinct from managed instances of objects even at the level of the type system.

List-walking language primitives

It is common, in UI frameworks, to iterate over a list, performing an operation on each item of the list, where that operation has a high probability of attempting to mutate the list. For example, calling a list of subscribed callbacks when some event happens, resulting in the callbacks unregistering themselves.

The common idiom to avoid this problem is to copy the list and iterate over the copy, which works fine for small lists but can get expensive with bigger lists. Some idiomatic mechanism to avoid this cost (e.g. copy-on-write for lists) could improve both the clarity of the code and potentially its performance.

Tree-walking language primitives

Adding entirely novel features is risky for a language intended to be familiar to existing programmers, but given the quantity of tree manipulation in a UI framework, a built-in construct akin to `for` and `while` loops, but specifically designed for walking trees, might be interesting[220].

This could be expected to bring two main benefits.

[220] This idea appears to have been first suggested by Tyler Glaiel, though that description seems to miss the potential performance benefits of being able to avoid any additional allocation or function call overhead for the walk algorithm itself.

The first is syntactic. Much as a `for` loop can be more clearly understood by some programmers than equivalent functional APIs like `map` and `reduce`[221], a tree-based looping construct might be clearer than the equivalent APIs. Notably, breaking out of such a construct would probably be much clearer than the common pattern of returning a boolean value from the walking function to indicate if the loop should be terminated or continued.

All this could be moot, however, with good API design. This would need to be tested with usability studies.

The other potential benefit is performance. A compiler-supported tree walking construct could avoid allocating any iterators or other data structures (e.g. zippers), instead storing all state on the stack, and would be able to avoid any function calls that are purely supporting walking the tree. Finally, ending the walk early could be as efficient as a single jump (plus trivial stack cleanup). How much benefit is possible here could be determined with benchmarks.

An alternative is to have the compiler recognize common tree walk patterns (especially those used by the UI framework for which it is being primarily created), and optimize those equivalently. This is more brittle, however, as developers may make seemingly trivial changes to the code that mitigate such pattern detection, thus making apparently trivial changes have disproportionate effects on performance.

Ecosystem

A core part of the experience of using a language is its ecosystem. There are several components to this. Success for the language will depend on all of these parts of the ecosystem thriving. This can often be the most difficult part of creating a new language.

Standard library

The standard library is the set of APIs that are not strictly speaking part of the programming *language* but are an inherent part of using the language. The line between the *language* and the *standard library* can be a little blurred at times. For example, while array literals are typically a language feature, the APIs for interacting with arrays are typically part of the standard library.

This section has largely assumed that creating a programming language goes hand-in-hand with creating its standard library. A good standard library integrates tightly with the language such that the developer does not need to distinguish them.

Developer tooling

In addition to having a compiler, a language should integrate with IDEs (typically by implementing a *language server* as discussed above) and with debuggers (e.g. by having the compiler emit debugging information in a widely-supported format such as DWARF). Ideally, the language should provide basic libraries that people can use to create additional tools for the language.

---

[221] This depends of course on the kinds of patterns the developer is familiar with. Someone who learned functional programming first is likely to be more comfortable with functional APIs, and someone who learned imperative techniques first is likely to be more comfortable with explicit loops. Which is more common in the target audience is a perfect candidate for usability research.

Algorithms and data structures

There are many data structures and algorithms that have been invented over the years, and if the community shares their implementations, it enables other people using that language to quickly use those data structures and algorithms. As languages mature, the number of libraries of this kind available to developers grows; a very new language naturally has very few such libraries. Finding ways to either grow this aspect of the ecosystem quickly, e.g. by seeding it with implementations created by the language team, or by lowering the friction for sharing code in this way (e.g. by providing a package manager) can help a new language's adoption.

Third-party libraries

There are many libraries that are considered staples; ensuring that developers using a new language can still use these libraries is important. For example, it would be vastly better if the language allowed developers to directly use the OpenSSL or BoringSSL libraries rather than requiring developers to create a new cryptographic library[222]. Similarly, making it possible for developers to directly use libraries such as SQLite, cURL, or HarfBuzz would help bootstrap the language ecosystem.

This mostly requires having some form of integration with C (i.e. C FFI).

Platform integration

By definition, a new language is not the language that any existing platform uses, and therefore there is immediately a need for libraries that expose platform APIs. The simpler this process, the more useful the language can be.

This probably is best done by having some form of integration with the platform's language (i.e. platform FFI).

Services integration

Another important set of API surfaces that a new language will need to expose is those of online services. OAuth2, Google, Facebook, and Apple authentication APIs. Payment processors. AWS and other cloud providers.

Bootstrapping this part of the ecosystem can be done using money (e.g. paying the providers to support the new language, or contracting a developer to implement the API wrappers), or time (i.e. having language team members implement the wrappers).

UI framework

Finally, given the context of this new programming language, we must also consider the need for libraries that directly relate to the UI framework itself. For example, new design languages, specialized controls not included in the core framework, new animations, and so forth. While fostering this part of the ecosystem is naturally a problem for the UI framework itself, it is nonetheless a part of what will decide the success of the language as well.

---

[222] Nobody should ever create a new cryptographic library.

Non-features

The hardest part of language design is not doing it.

Adding a feature to a language is not always a win. Some features make the language more complicated in a way that is a net loss to the language: the added cognitive load or performance costs may outweigh any benefits the feature has brought.

Unfortunately, every feature always has an advocate. Advocates for not adding a feature have psychologically less motivation than those advocating for adding one. As more and more features get added, the advocates for the net-negative features will eventually run down those who advocate for keeping those features out, and the feature will get added.

There are ways to attempt to mitigate this. None are especially reliable. One can make very clear statements regarding what features will *not* be added, with detailed reasoning. This can help subsequent generations of the language's maintainers stay consistent and understand the original vision. Another option is to design the language specifically with the net-negative features in mind, carefully crafting the design so that those features would be disproportionally hard to add. This can work in some cases, but usually requires too many other compromises to be worth it. The only other mitigation that comes to mind is to have a single language designer who never dies and forever stays in ultimate control of the language, though historically even that has not been successful (in extreme cases, the language just gets forked by the community, leaving the original designer out of the decision making process).

That said, let us examine some features that could be candidates for "this should never be added"[223].

Special dispatching

Operator overloading

Languages often allow developers to define how operators work on custom types. The canonical example of why this is a good idea is complex numbers; it is reasonable for a language to not have a built-in complex number type, but it is also reasonable for a developer to expect to be able to add two complex numbers using the + operator, and for complex numbers and real numbers (using a built-in type) to seamlessly work together.

The argument against operator overloading is that in almost every other case, developers will not be able to guess what code using overloaded operators does.

There may be some viable middle ground, where for example the indexing operator ([]) is overloadable so that user-defined collection types can work like built-in or standard library collection types, or where all the comparison operators are defined by a single comparator function, thus guaranteeing that they are consistent with each other.

The likelihood of operators being overloaded in confusing ways is not low; in many cases, even people who frankly should know better do not even recognize the problem. For example, C++ overloads the bit shift operators to do I/O, which has some redeeming value but is still, ultimately, more confusing than just having methods. Flutter overloads the `&` operator between `Offset` and

[223] This section is particularly biased by the author's experiences and the claims herein should absolutely be tested rather than believed unconditionally.

`Size` as a convenient way to make a `Rect`, which seemed like a good idea at the time and is used quite a lot within the framework itself, but is not sufficiently commonly used overall to be better than an explicit method or constructor[224].

Even the "reasonable" overloads in Flutter are dubious; for example, an `Offset` object can be compared to a `Size` object using the < operator, but whether this is always intuitive is unclear (it is, in particular, a partial ordering, so it is possible for two values to be neither smaller than, larger than, nor equal to each other).

Function overloads

Java, C++, and C# all implement function overloading, where the precise method selected depends on the types of the arguments.

Similarly to operator overloading, there are times where this is convenient. For example, a serialization API that supports many different types can have a single method with a version for each type, so that the caller can just call that method without worrying about types.

However, in practice, this leads to even more confusion than operator overloading, and opens up the developer to more bugs, because overloading increases the space of valid programs. For example, consider a function that accepts both an integer and a custom type. The developer tries to call the function with an instance of that custom type, by calling another function that obtains such an instance. Unfortunately, they accidentally call the wrong function and instead call one that returns an integer (e.g. giving an error code). Because the first method is overloaded, the compiler has no way to notice that the wrong methods are being called. So, hypothetically:

```
auth.authorizeUser(message.arguments.parseUserData());
// Oops, authorizeUser accepts either a User or an
// integer giving the user ID.
// The parseUserData method returns an integer representing
// how many fields were parsed, the actual User object is
// put in the message.arguments.user field.
// We just authorized some random user.
```

On the whole, function overloading is almost certainly a mistake.

Function overloading can also, depending on the design, lead to expensive compilation times.

Implicit coercion

Avoiding magic as much as possible (as discussed earlier) increases the predictability of the language, which makes it more approachable and easier to debug. One example of magic is coercing values of one type to another, e.g. converting doubles to integers or strings to booleans, etc. Some languages go all-in with this, others try to be more restrained.

An alternative to coercion that can help minimize the need dramatically is to do function overloading on operator overloads for base types, for example, having both + operators for int-int, int-double, double-int, and double-double. Whether this is more intuitive for developers than requiring explicit int-to-double and double-to-int conversions (or whether having implicit

[224] This is the author's fault, ultimately, and the main reason Flutter only has one such operator is that Flutter co-founder Adam Barth set forth the premise that "every project can only have one crazy operator overload" while reluctantly approving the PR adding &.

conversions is better than either other option) is an interesting question that deserves usability studies.

There are two places where *some* form of coercion is almost certainly required regardless, however.

The first is with numeric literals being used in integer and floating point contexts. These statements are both expected to work by developers[225]:

```
var double x = 0;
var int count = 0;
```

If the language has no implicit coercion, this could be achieved by changing the type of the literal based on the context[226].

The other is if the language supports multiple integer types, e.g. unsigned and signed, with various widths, or with arbitrary ranges. Developers will expect to be able to add unsigned integers to signed integers[227], will expect integer literals to work with all integers, etc. Similarly, developers will expect to be able to compare a 64 bit floating point value to a 32 bit floating point value, if the language supports varying widths of floating point numbers.

Implicit coercion can also, depending on the design, lead to expensive compilation times.

Adding members to an existing type

Some languages have the ability to add members to existing types. This can be *extremely* confusing when reading code, especially without the help of an IDE (e.g. when browsing source code, or watching videos). When code uses the member access operator, going to the definition of the type of the operand on the left hand side should be sufficient to find the member named on the right hand side. If the language has inheritance, one may need to follow the inheritance chain, but that should be all that is necessary.

When arbitrary code can add fields to a type, whether elsewhere in the file using that member, or later in the file declaring the type, or, even worse, in an entirely different file, the developer has little chance of being able to find what code is running. (Combining this with function overloading can be actively hostile to developers, who may in fact believe they have found the function in question, despite it not being the one the compiler will choose!)

Traits, protocols, interfaces

These features are more or less the same: various (potentially unrelated) concrete types can be given the same groups of members, and code can interact with all of these types equivalently by using only members from that group.

---

[225] The version of Dart used in early Flutter builds treated "0" as an integer literal, which was not type-compatible with double; this caused developers a great deal of confusion.
[226] This would be a limited form of type inference, which we will discuss shortly.
[227] Unless the types are explicitly marked as distinct types, which is a useful feature some languages have. Having distinct (type-incompatible) numeric types allows values to have associated units, which can catch bugs: for example it would prevent developers from accidentally mixing logical pixels and hardware pixels. This is one of those features that trades short-term pain for long-term benefit, and designing usability tests to objectively evaluate that trade-off would need careful consideration.

Such a feature can be very useful, when used in moderation. When used extensively, however, it can become very difficult to follow the code, as objects no longer have a clear "primary" identity, and instead are always merely the sum of their various traits. It becomes difficult for a developer first learning a codebase to determine if any particular combination of traits is reasonable or absurd. This contrasts with a class-based approach where the class hierarchy provides a "primary" identity for each object, and the interfaces implemented by each class are clearly "secondary" identities.

Type inference

Continuing the trend of dismissing features on the basis that that lead to bugs and confusion, one should avoid introducing any non-trivial form of type inference to a language.

Developers need to understand what their code does. This means that they need to have a clear intuition for how the compiler is going to interpret their code.

Complicated algorithms, or worse, heuristics, are the antithesis of easy predictability.

Thus, for example, the following hypothetical examples are probably reasonable cases of type inference:

```
var food = Apple(); // food is an Apple
var foods = <Fruit>[]; // foods is a List<Fruit>
```

The following are examples where the developer might not come to the same conclusion as the compiler, and thus these kinds of cases should be considered errors rather than the compiler attempting to infer types:

```
var food = vegetarian ? Potato() : Burger();
var foods = [Apple(), Potato(), Burger()];
```

The worst harms of type inference are when code that is just wrong *and would be caught with explicit type annotations* compiles because the compiler inferred a very broad type (e.g. the root of the type hierarchy). Consider for example if `Apple` refers to the company rather than the fruit, in a situation where the compiler infers for a `List` literal the narrowest type that is wide enough to encompass all given members:

```
var foods = [Apple(), Potato(), Burger()]; // error will be hard to spot
var foods = <Food>[Apple(), Potato(), Burger()]; // error caught by
compiler
```

Type inference can also, depending on the design, lead to expensive compilation times[228].

Operators that mutate values in expressions

The unary prefix and suffix ++ and `--` operators in various languages increment or decrement a variable and return its value (before or after the mutation, depending on the position of the operator).

In general this is somewhat bug prone. It is clearer to write code without side-effects; these operators are entirely intended to cause side-effects. Additionally, extra confusion can occur when a

[228] Notably, Swift suffers from this problem.

variable is used multiple times in an expression or statement and mutated in one or more of those places, as the precise order in which the expression is evaluated is rarely obvious to the reader.

Encouraging developers to use explicit statements (e.g. using += and `-=`) removes the side-effect issue and thus clarifies the code.

Reflection

A powerful feature that some languages support is reflection.

Type introspection, or runtime type information, is the ability for code to reason about its own data structures. To some extent, this is necessary to implement features like "`is`" and safe type casts, and is likely required to enable a simple implementation of an immediate-mode UI framework's reconciliation phase. Simple metaclasses as mentioned several times so far in this document would also benefit from the language supporting some form of runtime type information.

When constrained to these use cases, these are valuable features. It is tempting to take this further, however. For example, enabling code to *modify* its own types, or create new ones. These features, which are generally labeled *reflection*, do not have a good cost-benefit ratio. They are expensive, preventing many possible optimizations, and typically requiring a lot of indirection and data annotation (growing both CPU and memory costs). While sometimes useful for prototyping and research, they do not solve production problems that cannot be solved more efficiently in other ways.

Long term strategy

Long-term backwards compatibility (or not)

Once the language is popular, problems will be discovered that can only be completely fixed by making a change to the syntax or semantics of the language such that previously-valid code is now either invalid or has different semantics.

There are two valid approaches.

One is to state a compatibility policy wherein such changes will be made, but only after clear announcements, with compiler warnings for versions over a sufficient period of time (e.g. a year), ideally with some form of automatic migration tooling.

The other is to never make such changes.

The advantage of being willing to make changes is that the language can evolve over time and keep improving. Dart, for example, has used this approach to make radical changes to its semantics, going from a weakly typed language to a strongly typed language, and then introducing null safety.

The disadvantage is that such an approach costs developers significant effort.

If the effort required to migrate is lower than the productivity gains from the changes, then one could argue that overall, the changes are valuable. On the other hand, developers with millions of lines of code and minimal funding will find almost any such migration practically speaking impossible. When one considers that there are codebases with lifetimes measured in decades, with

intervals between maintenance cycles measured in years, it becomes quickly apparent that no breaking change can ever be acceptable to all developers.

A language should clearly state which approach it is going to take, and then follow that approach rigidly.

Use in other spaces

Creating frameworks for using the language in completely unrelated spaces, e.g. as a server language, for driving AI models, or for statistical modelling, would help the language distinguish itself from the UI framework and would thus reinforce the language's value (which would subsequently make it even more attractive when used for UI, thus helping the popularity of the UI framework).

Use in education

In the long term, to support making the UI framework popular, the language itself must be popular. The popular languages all have one thing in common today: they are taught in universities and schools. Displacing Java and Python in favour of a new language would be very difficult, but not impossible with sufficient investment. A language specifically designed to be approachable and useful for creating UIs would be extremely well-suited for the role of education language.

It would require the creation of curriculums and other teaching materials, and significant advocacy, none of which would really be possible until the language and framework were well established, but it would enable a second wave of popularity.

One thing to avoid is having the language become associated specifically with being "The Education Language". This was a mistake made for Pascal in its early days, which probably contributed to "serious" developers not using it, allowing C++ to take the space that ObjectPascal could have taken.

Framework internals

In this section, we will discuss some specific designs for various parts of UI frameworks. This section is intended to provide some insights about how various components could be designed[229], and is not intended to be a guide for creating a complete framework. The designs presented in this section are informed by the author's close experience with Flutter and the web, and by research into other UI frameworks. They are, however, not tested in the real world.

Developing a UI framework is easy until a real application is built using that framework, at which point suddenly reality descends firmly on the project and asserts itself. The designs in this section should be taken as inspiration and not as the last word on UI framework design. Where reality, as represented by the experiences of real programmers writing real applications using the framework, contradicts a suggestion in this section, reality is correct.

As noted in other sections, one of the best ways to explore the needs of real application developers when developing a UI framework with the goals described earlier is the deep involvement of usability research and frequent usability testing.

Layout

The purpose of the layout component of a UI framework is to efficiently position components on the screen, both when they are first created, and subsequently as the interface shifts due to animations, scrolling, the window being resized or the device being rotated, etc.

In this design, each layout component should perform a single simple task, rather than combining features into mega-components. (This was one of the design options mentioned earlier.)

Background

Logical pixels

Output surfaces — e.g. screens and printers — vary in their pixel density, or resolution. This makes it difficult for developers to achieve a consistent user experience when describing their UIs in pixels; a 32 pixel square image on a 120 dpi screen will look very different than the same 32 pixel square image printed on a 1200 dpi laser printer (dpi being the unit of "dots per inch"). This problem became even more acute when screens, which previously all held to a more or less uniform convention of approximately 100 dots per inch, started being offered in much higher resolutions such as 300 or 500 dots per inch. In addition, with phones and TVs being used more widely as output devices for software, the same pixel density may have very different results merely due to the distance from the user to the screen.

This leads developers to seek *resolution independence*. As a concept, resolution independence has been around for many decades, but CSS crystalized the concept of the "logical pixel" in a way that is now firmly established (it is sometimes also called a "device independent pixel"). A logical pixel is 96th of an inch as seen at about arm's length. It corresponds to one physical pixel on displays contemporary with CSS' introduction in 1996. The ratio of physical pixels to logical pixels for a

[229] *The Mahogany Staircase - Flutter's Layered Design* (2016)

particular device is its *device pixel ratio* (or *device pixel density*). For example, the first iPhone with a Retina display had a device pixel ratio of 2.0, meaning that every logical pixel of width would correspond to two physical pixels of width[230].

User interfaces are now commonly described in terms of *logical pixels*, rather than physical pixels. Even bitmap images are often described in terms of their *logical* dimensions plus a scale (also known as the pixel density or resolution), rather than their literal number of image pixels. This allows an image to be swapped out for a higher-resolution image as necessary. For example, a 32×32 icon (in logical pixels) rendered on a 96 dpi screen will use a 32×32 bitmap, but when used on a screen with a 2.0 device pixel ratio will use a 64×64 bitmap, which might be described as a 32×32 bitmap with scale 2.0.

Multiple monitors

An application can span multiple monitors, and monitors can have different device pixel ratios. Windows spanning multiple monitors with different device pixel rations is a sufficiently rare edge case that UI frameworks generally do not attempt to handle this in a sophisticated manner. Instead, it's generally deemed sufficient to pick one device pixel ratio for the entire window — for example, that of the monitor on which the top left corner of the window is displayed, or that of the monitor with the highest device pixel ratio — and scale the rendering accordingly.

When there are multiple windows, however, each one should independently determine its device pixel ratio.

Width-in-height-out

A common approach to layout is to specify the width of a box, lay out its children using that width, measure the resulting height, and use that height for the height of the box. This is called width-in-height-out.

This approach works very well for situations where the context is primarily text[231], and it is being rendered to an arbitrarily tall output medium with a fixed width, such as paper, or a window that may scroll vertically. Text can flow to an arbitrary width before being wrapped, and that will determine the height to which it will grow.

A strictly width-in-height-out layout model can be implemented using a single pass, walking down the tree depth-first. The leaves of such a model are special, as they must be measured to determine the height, as opposed to all other nodes, which receive a height from their child(ren), and use it to directly compute their own height.

While this approach can get quite far, in practice, a strictly width-in-height-out model is insufficient for most UI needs. It is most appropriate for simple documents such as letters, or for systems where functionality is more important than aesthetics (e.g. enterprise forms software).

---

[230] Because pixels are two-dimensional, the total number of physical pixels per logical pixel is a square of the device pixel ratio. This is a pedantic point that rarely factors into UI development, where linear dimensions are used far more than measures of area.

[231] Meaning horizontal text. Vertical text would prefer a height-in-width-out algorithm, and this is one reason why knowing well in advance whether vertical text will ever be implemented is important. A system that has a deeply ingrained sense of width-in-height-out is difficult to adapt to a world where some content is height-in-width-out.

Intrinsics

During layout, it can be useful to have a way to establish the "natural" dimensions of content, rather than having to specify it. The intrinsic size is sometimes known as the preferred size. It represents the size the content would naturally take on if left to its own devices.

For example, bitmap images have an intrinsic dimension determined from their pixel width and height and their resolution (the number of bitmap pixels per logical pixel). When an image is being laid out, the UI may be designed to size itself to the image. This requires finding the image's intrinsic dimensions during layout. Similarly, if a heading is to have a box drawn around it, the dimensions of the text must be determined so as to draw the box.

There are generally two cases where intrinsic dimensions matter.

One is when measuring a leaf so as to size its parent, as in the width-in-height-out algorithm described above. This is a normal operation and determining the dimension is a natural result of laying out the element.

The other is when the dimensions of one subtree affect the dimensions of another. For example, wanting to size a column[232] to be as wide as the widest element in the column, and then having all the elements in that column be sized with that width, so that they can position content across the entire cell:

Fruit	Apple
This cell intentionally left blank.	
Color	Green

Querying the size of subtrees in this way is expensive because it requires double the work: the subtree must be effectively laid out twice, once to determine the intrinsic size, and once to get the actual layout. (For example, in the case above, the "Fruit Apple" cell needs to be measured to find its intrinsic size, which is then discarded when the second cell's intrinsic size is found to be bigger; the "Fruit Apple" cell must then be laid out again, given the size of the second cell.)

In most cases, it is possible to lay out the interface in a specific order so that only one layout is required. For example, when there are two children and the first is to be positioned under the second, using the size of the second child, instead of first computing the intrinsic size of the second child, then laying out the first, then laying out the second, a more efficient solution is to lay out the second child first, then lay out the first child using the sizes obtained from the second. There is not always such an option; e.g. the column example above is a case where it is not possible.

Kinds of intrinsic sizes

There are several intrinsic sizes that are useful in different contexts.

Minimum intrinsic width: The smallest width beyond which making the element smaller would cause layout issues, e.g. overflowing text. For example, this is the width of the widest word in a paragraph where the text can be wrapped. For an image, this is zero, because an image can be shrunk arbitrarily small.

---

[232] See the next section for details on grid sizing.

Maximum intrinsic width: The narrowest width the element can be given without increasing the height. This makes sense for width-in-height-out layout elements. For a paragraph, it represents the width of the text when it has not been wrapped at all. For an image, it is again zero, because decreasing the width does not increase the height.

Preferred intrinsic width: The "natural" width of the content. For an image, this is its pixel width. For text, it is usually the same as the maximum intrinsic width.

The same kinds of dimensions can be defined for height.

Intrinsic aspect ratio: If some element has a naturally preferred height for any given width, or vice versa, this is their ratio. This usually applies to images and not text, though in principle text where the minimum and maximum intrinsic widths are the same (e.g. a single word, or text that's styled so as to never wrap) could have an intrinsic aspect ratio.

Two dimensional layouts

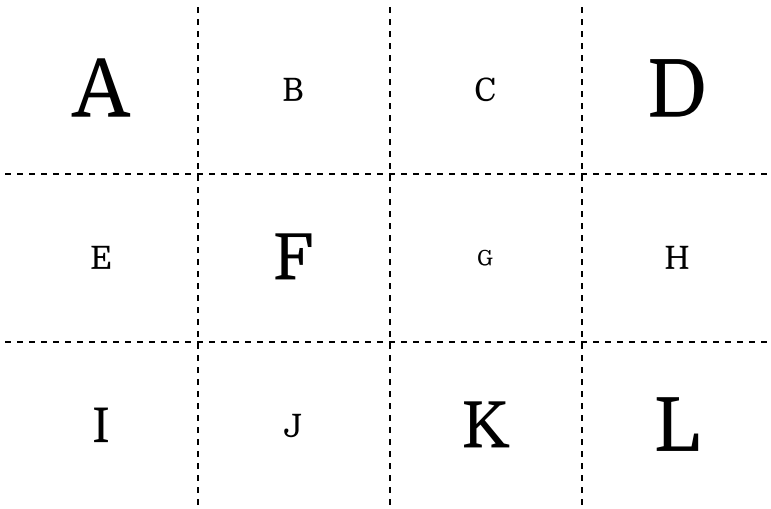
Most layout primitives operate in a single direction at a time — for example, flexible rows, or flexible columns, scrollable list views. Even primitives that wrap, e.g. paragraphs of text, or wrapping rows of elements, operate in only one direction at a time: content is first laid out horizontally until more room is needed, then a new vertical position is determined, then content is laid out horizontally, etc.

When the layout requires both dimensions to be considered simultaneously, it is often called a "grid". The term "grid" is used for various different layout concepts. In CSS, it is a mechanism by which children can be positioned relative to each other along grid lines. In Flutter, it is one of the scrolling primitives that allows children to be positioned along grid lines while scrolling. In Windows, it is a type of control that resembles a spreadsheet.

The most common type of grid-like layout is the table.

Table layout is probably the trickiest of the common layout primitives that developers expect.

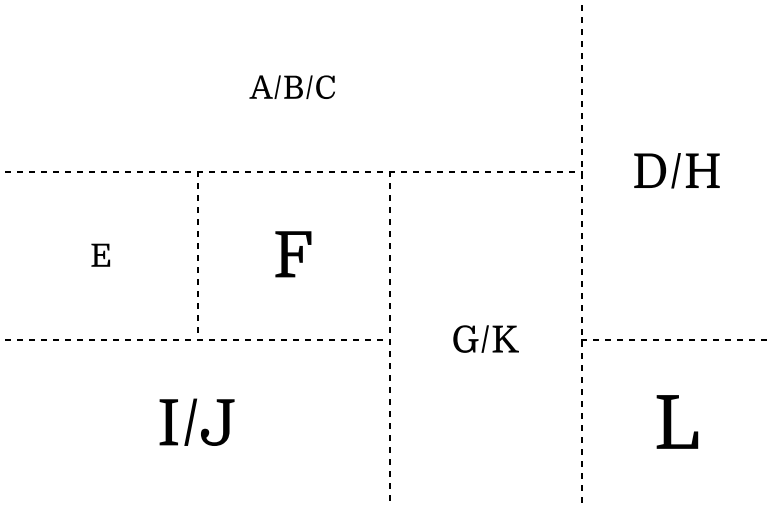
The defining feature of a table layout is that the children are positioned along grid lines, regardless of their size:



Typically, rows are sized so as to fit their contents. Columns vary; they may be given fixed sizes, or they may be sized to fit their contents, or they may be sized relative to each other so as to fit the available width (flexible layout). Where columns are sized to fit the contents, the concept of intrinsic dimensions, as discussed in the previous section, is used; this leads to the layout being expensive.

Each cell's contents must then be positioned within its cell, vertically and horizontally.

The difficult part of table layout is that it is common to desire that some cells span rows or columns:



There is no definitive answer for how table layout with merged cells should work, especially when either rows or columns or both are being sized to fit their contents. A good place to begin is to consider the contribution of a merged cell as being split across the columns (or rows, or both) that it spans.

Pop

Users find interfaces that move suddenly to be quite disruptive and confusing. Examples of this include images popping in and changing the layout when they are fetched from the network, ads shifting content when they are loaded, UI elements changing when the user hovers over them, UI elements moving when the state of the application changes (e.g. when a network connection drops or when another user sends a message), etc.

For this reason, the default behavior for images should be to require a fixed size to be specified, even in the absence of image bytes. Thus, when the image is loaded, the image will not disrupt the layout. This also allows the framework to select images based on the pixel density of the output display. For example, if the image is 32×32 and the device pixel ratio is 2.0, the framework can explicitly request (from disk or the network) the 64×64 version of the image, but if the device pixel ratio is 1.0, it can save memory and CPU cycles by just directly requesting the 32×32 version.

Incremental (or interruptible) layout

Sometimes, when faced with slow layout performance, developers suggest that the layout system be interruptible or incremental. The framework would stop mid-way through computing layout, render what it has so far, and then resume in the subsequent frame. Unfortunately, because of the disruption caused by pop, this is rarely desirable for the user. In practice, it is usually preferable to merely delay the rendering until the entire layout can be presented at once, rather than showing data incrementally.

Constraints and geometry

A *constraint* in this context is a way for a render object to influence the rendering of its child(ren).

For example, a constraint could be "be no wider than this, and be taller than this".

Rather than hard-coding a coordinate system, this system at its core is agnostic regarding the geometry used. This allows the same layout system to be used for multiple styles of layout. This is a design Flutter pioneered. In practice, it has not found much use in the Flutter community, with one exception: scrolling views rely heavily on this mechanism.

Core layout loop

The basic approach used in this design is for each layout node (render object) to send constraints to its child(ren), and then return a geometry to its parent. Specifically, for each render object, the layout algorithm operates as follows:

1. The parent provides the render object with a set of *constraints*.
2. The render object iterates over its children (in some order defined by the render object); for each one:
    a. The render object creates a new set of constraints based on the constraints it received from the parent and on its configuration.
    b. The render object provides the child with those constraints.
    c. (This entire algorithm runs recursively for that child.)
    d. The child returns a *geometry* for itself.
3. The render object iterates over its children again (in the same or a different order), and for each one, the render object uses its configuration, the constraints from the parent, and the geometry information from the children to determine the child's position.
4. The render object uses the geometry and positions of its children, and the constraints it received from its parent, and its configuration, to determine its own geometry.
5. The render object returns that geometry to its parent.

During the first frame, the root node is laid out (with constraints derived from the environment, e.g. the window size), and this recursively lays out the entire tree.

Nodes mark themselves dirty when their configuration is changed, including when their children change in some way (e.g. adding or removing children). When a node is marked dirty, it is added to a tree-wide dirty list.

Each subsequent frame, the framework lays out the nodes in the list of dirty nodes. This list is walked in depth order. Nodes lay out their children when laying themselves out; those children may also independently be in the dirty list. This means that nodes in the list may have already been laid out when they are reached from walking the list. These nodes are skipped, to avoid redundant work.

(If the list was not sorted, then a deeper node may be laid out using constraints from the previous frame before their parent is laid out; when their parent is then laid out later, the parent might want to lay out the deeper node with new constraints that invalidate the earlier layout work. In the worst case where every node changes which constraints it applies to its children when it is laid out, with a list sorted in reverse depth order, that changes an O(N log N) algorithm[233] to O(N^2)[234].)

---

[233] Dominated by the sort; the actual walking of the list and laying it out is O(N).
[234] This is assuming every node of the tree is dirty. Computing the complexity is non-trivial when one assumes the dirty list contains a small fraction of the total nodes of the tree, and that laying out any one node of the tree will cause a cascade of (clean) nodes to be laid out with new constraints. A more pragmatic analysis would probably find that sorting the list is a trivial cost, and that not sorting the list increases the expensive layout work by some small but measurable amount.

Performance-improving shortcuts

Dirtying

When a node is dirtied (meaning it needs to rerun its layout algorithm in the coming frame to update its rendering), it must also notify its parent node if its geometry might change as a result, in case its parent derives *its* geometry from the child's.

This enables a few performance improvement opportunities:

- If the constraints that were provided to the child are such that only one geometry is possible (a "tight" constraint), then there is no need to notify the parent, because by definition the geometry cannot change despite the render object being dirty[235].
- If the parent has declared that it does not take the child's geometry into account, then again, there is no need to notify the parent, because even if the geometry does change, the parent will not use it.
- If the constraints given to a child have not changed, if the child is not dirty, then it does not need to be laid out again when the parent is dirty, even if the parent changes its position as a result.

The impact of these performance optimizations varies. In general, the more work can be skipped, the greater the benefit.

Cleaning

If a node is re-laid-out using the same constraints as its parent used the previous time, and the child is not dirty, then by definition nothing needs to happen; the result will be the same. Therefore the system can skip such nodes and immediately use the previously computed geometry.

Box constraints and geometry

More or less universally, frameworks use Cartesian coordinates as their primary mechanism for describing layout. The constraints for Cartesian coordinates are known in the Flutter ecosystem as "box constraints", and the corresponding geometry is called the "size"[236].

A box constraint is a minimum width and a maximum width, and a minimum height and maximum height. The maximums can be infinite to indicate an unconstrained environment. A size is a specific finite width and height.

For example, a "padding" render object would receive a constraint from its parent, "deflate" the constraint by the amount required to account for the padding, and pass the new constraint to its child. The child would then return a size that fits that constraint, and the padding render object would inflate that size by the amount required to account for the padding, then return that as its geometry. A render object that tries to enforce a specific width would create a new constraint that has minimum and maximum width values set to that fixed width (typically constrained by the minimum and maximum width provided by the parent), with the minimum and maximum heights set to the same values as those it received from its parent. This same pattern continues for all the various layout primitives to be supported.

---

[235] Other aspects of the render object might change, e.g. the positions of its children, so the render object itself still needs to be laid out again.
[236] Flutter does not formalize the concept of geometry the way this design does.

This is not strictly enough for all the primitives developers expect. For example, a primitive that takes a list of children and tries to size them so that they are all as high as the child with the tallest "natural" height cannot be implemented with only the features described so far. Such a feature requires some support for multipass layout (which, as discussed earlier, is inherently expensive).

One option is to have a way to query box render objects regarding their "preferred" metrics. The main disadvantage of this approach is that every render object must now support this query feature[237]. This is the approach used by Flutter. Another option is to design the layout protocol so that boxes can expect to be called multiple times with different constraints as some ancestor attempts to probe the tree to find the information it requires. This approach has a minor added complexity because some render objects must perform very expensive computations once their geometry has been determined. For example, resizing a render object that represents a web view multiple times per frame would be catastrophically expensive.

Instead, this model separates the layout phase from a post-layout "finalization" phase which render objects can opt into during layout. This allows the layout phase to restrict itself to computations that do not affect external resources, which enables ancestor to trigger multiple layouts per frame in the few cases where this is necessary.

In addition to this, experience suggests that giving box constraints the ability to explicitly request that a render object prefer minimizing its width or height, maximizing its width or height, or "best" fitting its width or height, might address a number of important use cases that otherwise require this more expensive mechanism[238].

In this design, therefore, the box constraints are described as the following parameters:

- minimum width.
- minimum height.
- maximum width, or infinity if the parent has no width constraint (e.g. it can scroll horizontally).
- maximum height, or infinity if the parent has no height constraint (e.g. it can scroll vertically).
- a horizontal mode and a vertical mode, each of which are one of:
    - default (child should size itself using its normal semantics).
    - minimum size (child should size itself as small as possible in this axis, but no smaller than is needed to render the contents correctly).
    - preferred size (a typical "preferred size" would be the smallest that the dimension could be without increasing the other dimension, but in practice there is little value in defining this rigorously).

When the mode for a direction where the maximum dimension is infinite is "default", it should be treated as "preferred".

---

[237] And they won't, which will lead to a never-ending stream of bugs being reported that, when investigated, reveal new render objects that fail to fully implement the protocol.

[238] Flutter does not support this in its library of box render objects. Instead, Flutter has a complicated mechanism that allows parents to ask their children for their "intrinsic dimensions", which it can then use to constrain the children. This is potentially expensive (though Flutter aggressively caches these values to reduce the cost), but the complexity is the real problem. Very few people (including those who work on Flutter's framework full time) could be said to fully understand how this API works. Certainly this document's author, who was instrumental in that API's creation, is not one of those who can say they understand it.

The geometry (size) should always be finite, and should always be within the bounds given by the minimums and maximums[239]. It may also be valuable for the geometry to include information beyond merely the width and height. For example, the geometry could include information such as the positions of various baselines, to enable baseline alignment[240].

In this design, therefore, the box geometry is described as the following metrics:

- the width.
- the height.
- the distance from the top of the box to the alphabetic baseline.
- the distance from the top of the box to the ideographic baseline.

When there is no baseline to report, the distance is the same as the height, so that baseline-aligned boxes without baselines are aligned on their bottom edge.

The *position* of a box is considered to be a matter for the box's parent, and is not part of the box's geometry.

Limitations of this approach

Because the constraints and geometry do not include position information, some layouts cannot be achieved. For example, this system does not allow a child box to signal to its parent that it must be shifted to the edge of the box in the way that the CSS 'float' properties allows. To support such layouts with this set of constraints and geometry types, the configuration regarding child positioning should be specified to the parent, rather than the child. This box constraint and geometry system also doesn't support flowing text around shapes in the way that 'float' enables in CSS.

Designing a dedicated set of constraints and geometry designed specifically for this kind of document layout might be an interesting exercise.

Scrolling component constraints and geometry

To efficiently render content in scroll views, the visible components must be generated on demand, based on the user's scroll position. This means there are two kinds of components in scroll views: those that represent actual content on the screen (typically using box geometry), which are created on demand and only exist while visible, and those that are managing this process. To render scroll bars, the scroll view must learn from its contents how much content *could* be rendered.

---

[239] One of the intentional design decisions that Flutter made that, in retrospect, was probably a mistake, was having cases where render objects could refuse to pick a geometry. For example, placing a scrolling container inside a scrolling container throws an exception, because the outer scrolling container gives the inner scrolling container an infinite maximum height (that is, after all, what it means for it to be a scrolling container), and the inner scrolling container's layout semantics are to grow to fit the maximum height constraint, which means the inner scrolling container wants to have infinite height, which is invalid. The thinking at the time was that catching errors as soon as they were detected would help developers fix the errors faster. This may even be true. However, it makes the developer experience frustrating, because the failure to lay out the scene means the developer cannot look at the scene to see where things have failed. A potentially better solution would be to make the inner scrolling container pick the *minimum* height in that situation, and maybe render a red banner in debug mode to flag the situation.

[240] It could also indicate that the constraints were insufficient to be able to render the content (i.e. that the content either will overflow, or that it will be incorrectly painted in some way due to insufficient space). This would be useful primarily for debugging purposes, e.g. it could be exposed in tooling, or nodes with this flag set could be highlighted in some special debug mode.

The latter components depend on the scroll position of the scroll view, the dimensions of the scroll view, and how much content has already been rendered into the scroll view[241]. Thus the constraints are very different than for boxes:

- scrolling direction (up, down, left, right).
- scroll offset from the scroll origin.
- the size of the scroll view in the scrolling axis direction.
- the size of the scroll view in the cross-axis direction.
- the distance from the start of the scroll view to the start of the child being painted.

Similarly, the geometry is very different than for boxes: it must describe how much of the scroll offset the render object can be responsible for, and how much space the rendering takes given the current constraints:

- scroll dimension.
- layout dimension.

In this design, the constraints for scrolling components are never tight. Indeed, strictly speaking, there is nothing in the constraints that actually constrains the geometry at all.

This model can be extended to support features like header pinning, floating toolbars that appear when the user scrolls in the reverse direction, pull-to-refresh interfaces, etc[242].

Representing alignment

Layout components are typically configurable, for example a layout component that controls the size of its child would take a size argument.

Some layout components have as one of their configurable properties an alignment. For example, a component could let its child size itself according to loose constraints (no larger than the component itself, but possibly smaller), then, if the child does select a small geometry for itself than the maximum possible given by the constraints, the component can select the maximum size for itself and then position the child within itself.

To represent alignment in a Cartesian space, the author recommends using a data structure that takes two real numbers (the *x* alignment and the *y* alignment), with 0,0 representing a centered child, -1,-1 representing a child positioned at the top left, and 1,1 representing a child positioned at the bottom right.

This representation has several advantages, the biggest of which is that it is simple to have a variant of this data structure that is direction-agnostic, where the *x* alignment is instead -1.0..1.0 for start to end rather than left to right; in right-to-left text, then, -1,-1 is the top right. Flipping between the two thus becomes as simple as flipping the sign on the *x* alignment.

---

[241] Some other information is also required, such as the actual direction of scrolling, and the direction in which the user is currently attempting to scroll. If the geometry provides more features than is described in this document, e.g. features to implement pinned headers, the constraints may also need to reflect that information.

[242] It's not clear that a perfect design for this kind of feature has yet been invented. Certainly, this YouTube video of the author and Filip Hráček exploring Flutter's version of this API demonstrates pretty clearly that Flutter's API is not perfect! Starting from a design similar to Flutter's and iterating on it with extensive usability testing would probably be a good way to make progress here. Of importance here is keeping the design focused on real use cases that developers are trying to solve.

The other advantage is that it is easy to scale this representation: multiplying the values straightforwardly pushes the alignment in the chosen direction[243].

Bidirectional support

The layout system itself is text-direction-agnostic; it is described in terms of width and height and offsets from a top-left origin, but has no directional bias beyond this[244]. Render objects themselves, however, may take parameters in terms of "start" and "end" (rather than "left" and "right"), combined with an explicit text direction (left-to-right or right-to-left), and then translate those values to offsets accordingly. Being consistent throughout the API about defining everything in those terms ensures that supporting right-to-left locales will work out of the box, though of course care should be taken throughout the design process to think about how right-to-left locales will be supported, to avoid other implicit assumptions creeping in. Avoiding having APIs ever *default* to left-to-right text direction is a big help in avoiding such biases.

One example of how to avoid a bias is to always default alignments of bidi-unaware properties to centering, rather than to left or right alignments. In bidi-aware APIs, the default can also be "start", meaning it depends on the text direction.

API

This design implies the following characteristics:

- An object representing the render tree, which has:
    - a dirty list for layout (actually a set).
    - a method for running layout that
        - sorts the dirty layout list.
        - for each render object in the layout list:
            - if the node is still dirty for layout, calls the private layout method defined below.
        - clears the dirty layout list.
    - a dirty list for finalization (also actually a set).
    - a method for finalizing layout that
        - sorts the dirty finalization list.
        - for each render object in the dirty finalization list:
            - calls the private finalization method defined below.
        - clears the dirty finalization list.
- An abstract class for constraints. The only property of this class is an expression of whether the constraints are tight.
- An abstract class for geometry. This class is empty.

---

[243] The more obvious way to represent alignment is two values in the range 0..1 with 0,0 being in the top left and 1,1 the bottom right. In principle this is mechanically equivalent to the recommended representation, but in practice the arithmetic is less clean. For example, instead of simply flipping the sign, flipping the direction requires subtracting the x direction from 1. Similarly, scaling requires multiple operations, and results in weirdness like -1 being the opposite of 2. Finally, zeroing out memory puts things in the top left instead of centering; this introduces a left bias that (as discussed in the next section) is generally worth avoiding.

[244] In principle one could put the origin in the center of the display surface (e.g. window), but that would be extremely unusual and experience shows it leads to quite confusing bugs. For example, trying to center something by putting its upper left at (width/2,height/2) places it just off screen, which leads the developer to trying to debug why it has failed to render at all, rather than merely why the position is incorrect.

- An abstract class for render objects, which is generic over a constraints type and a geometry type (which most inherit from the corresponding base classes). This class has a variety of public, private, and protected members.
  - The render tree (private).
  - The depth (private).
  - The parent (a private field, with a public read-only property).
  - A public method to attach the render object to a new parent, which:
    - updates the parent field.
    - if necessary, updates the tree field to match that of the parent. If that was necessary, it must also update the tree field of all its descendants.
    - walks the subtree rooted at the render object, ensuring for each node that the depth field is deeper than its parent's. (Any subtree rooted at a node that does not need to have its depth updated can be skipped.)
    - calls the method defined below to mark the node as needing a layout update.
  - The last constraints (private).
  - The last geometry (private).
  - Flags tracking if the render object is dirty for layout or finalization, and whether the parent uses its geometry (all private).
  - A public method to mark the node as needing a layout update, which:
    - returns early if the node is already dirty for layout.
    - adds the node to the tree's dirty layout list.
    - marks itself as dirty for layout.
    - if the node has never been laid out, or, if the node's last constraints were not tight and the parent does use its geometry, then, if it has a parent, calls the method recursively on the parent.
  - A non-virtual public method for performing the layout and returning the geometry that takes a set of constraints, which:
    - if the parent previously did not use its geometry, marks the node as dirty for layout and sets the flag that tracks that the parent uses the node's geometry.
    - calls the private layout method defined below, giving it the provided constraints.
    - returns the last geometry.
  - A non-virtual public method for performing the layout that takes a set of constraints and returns nothing, which:
    - if the parent previously did use its geometry, marks the node as dirty for layout and clears the flag that tracks that the parent uses the node's geometry.
    - calls the private layout method defined below, giving it the provided constraints.
  - A private layout method that optionally takes a set of constraints, which:
    - if the node is not dirty for layout, and either no constraints were provided or the provided constraints match the last constraints, returns early.
    - sets the new constraints as being the last constraints, if new constraints were provided.
    - sets the new geometry as being the result of calling the protected method for computing the geometry defined below, giving it the last constraints (which are now the new constraints if any were provided).
    - clears the dirty for layout flag.

- An abstract protected method for computing the geometry that takes a set of constraints.
- A public method to mark the node as needing layout finalization, which:
  - returns early if the node is already dirty for finalization.
  - adds the node to the tree's dirty finalization list.
  - marks itself as dirty for finalization.
- A private layout finalization method, which:
  - calls the protected method for finalizing geometry defined below.
  - clears the dirty for finalization flag.
- A protected method for finalizing layout, which does nothing[245].

The box and scrolling component constraints and geometry build on this design as described above. Render objects for the box coordinate system subclass the render object class with the type arguments specifying the box constraint and geometry types, and equivalently for the scrolling component render objects.

Automatically generating layout components

Because layout concepts are well understood in the literature[246], if the protocol is very well-defined and sufficiently simple, generative AI could be used to create many of the core layout components of the framework. Having some mechanism to automatically test that a layout component correctly obeys the protocol (e.g. injecting different constraints and observing if the resulting geometry is valid) would also help with this, as generative AIs can iterate their design until the tests pass.

Painting

Many UI scenes are largely static with some small component of the scene changing each frame (e.g. a blinking cursor, a progress indicator, a video). Historically, the expensive part of updating a screen was moving pixels, and UI frameworks would determine "dirty rectangles" and their design was built around only updating the dirty pixels.

This is less of a concern with modern hardware. Today a more effective approach is to separate the scene into logically separated components, so that the parts of the scene that need updating can be recomposited with the pre-computed static parts. This conveniently corresponds relatively closely with slicing the render object tree into nested subtrees of display lists. Each frame, the parts of the render tree that have changed are rebuilt, then the complete scene is rebuilt by grafting in the changed subtree.

The painting infrastructure is thus a combination of an immediate-mode painting API to describe the subtrees as pictures, and a retained mode API to combine those pictures.

Overview

This design assumes an underlying graphics API that takes instructions for combining and transforming textures, which we will call layers. There are different kinds of layers, e.g. pictures (lists of vector commands), textures for video playback, or textures for embedding another

---

[245] Or alternatively, which throws an exception reporting that this object should not have been marked dirty for finalization.
[246] ...and thus well represented in LLM training data.

application like a webview. This design also assumes there is an API for creating pictures by emitting vector commands.

For simplicity, we shall reduce render objects to two forms: those that create a layer (composition roots), and those that draw parts of pictures and insert children into some ancestor layer, but do not manage a layer of their own. As briefly noted earlier, a render object may interleave pictures with its children. This means a render node that draws parts of pictures may actually correspond to multiple layers. Having the simplified assumption removes a number of complexities that might otherwise arise[247].

In general, the number of layers should be kept small. Parts of the UI that change frequently should have their own layers; parts of the screen that are mostly static should share a layer.

API

This design is an extension of the layout design described above, with the following characteristics:

- An abstract class representing layers, which has:
  - an abstract public method for updating a child layer, which takes an old layer and a new layer.
- A concrete class representing a group of layers, which has:
  - a data structure that holds the child layers and maintains their order, which should have O(1) insertion and O(1) lookup (for efficient replacement).
  - a public getter that returns the API for creating pictures[248], which:
    - checks if the last entry in the child list is a layer that represents a picture, and if not, adds one.
    - returns the last entry in the child list.
  - a public method that takes a layer and adds it to the child list.
  - an implementation of the public method for updating a child layer declared in the superclass, which replaces the given old layer in the child list with the given new layer.
- The render tree object has, in addition to the members described above:
  - a dirty list for painting (again, this is actually a set).
  - a method for painting that:
    - sorts the dirty paint list.
    - for each render object in the dirty paint list (each of which must, by definition, be a composition root):
      - takes a note of the current layer for the render object (the old layer).

---

[247] There are many optimizations that can be attempted here. For example, Chromium detects when separate layers happen to have geometry that can be combined without conflicting with the geometry of intermediate layers, and internally "squashes" them. Vector paths can be simplified, combined, or otherwise transformed to reduce the computation cost. Some systems will cache the rasterized pixels of layers rather than maintaining them as GPU instructions. Whether such optimizations are worth it is a matter for benchmarks, and probably a matter for consideration only once specific bottlenecks are identified. Simplicity is often as valuable, if not more valuable, than performance at the start of a project.

[248] The point of this design is to make it easy for developers creating render objects that draw both below and above their children to do so in a way that transparently supports their children having layers. It is important that the API be intuitive to developers, because a mistake here (e.g. caching the canvas and reusing it after attempting to draw children on the canvas) will result in incorrect renderings but only in edge cases where children need a separate layer (which is not the common case). The proposed API can be made to catch such errors by having a way to flag a canvas reference as "out of date" when painting a child, and having canvases assert that they are not "out of date" when their drawing functions are called.

                - calls the render object's method for creating the new layer.
                - tells the node's nearest ancestor compositing root render object's layer to update its child layer, passing it the old layer and the new layer.
            - clears the dirty finalization list.
- The render object has, in addition to the members described above:
    - An abstract public getter that returns a boolean that reports if the node is a compositing root or not[249].
    - A flag tracking if the render object is dirty for paint.
    - A field optionally holding a layer, which should always be null if the render object is not a compositing root.
    - A public getter that returns the layer of the nearest ancestor render object that is a compositing root.
    - A public method to mark the node as needing painting, which:
        - returns early if the node is already dirty for paint.
        - if the node is a compositing root, adds the node to the tree's dirty paint list.
        - marks itself as dirty for paint.
        - if the node is *not* a compositing root, and if it has a parent, calls the method recursively on the parent.
    - A public method for painting the render object into a group of layers, which takes an instance of the class representing a group of layers as an argument, and which:
        - if this is a compositing root:
            - if the node is dirty for painting, calls the method for creating the layer, and stores the return value as the render object's layer.
            - adds the render object's layer to the group of layers' child list.
        - otherwise:
            - calls the method for drawing the render object, passing it the group of layers.
        - clears the dirty for paint flag.
    - A protected method for creating a layer, which returns a layer, and whose default implementation:
        - creates a layer of the type that represents a group of layers.
        - calls the method for drawing the render object, passing it this new layer.
        - returns the new layer.
    - An abstract protected method for drawing the render object.

Concrete render objects would override one of the last two protected methods (and the getter for reporting if the node is a compositing root) to define how the render object paints. Render objects with children would call each child's public method for painting the render object.

This design does not handle the possibility of a render object changing whether or not it is a compositing root. The ability to do this can be useful, and would require a separate phase and dirty flag to handle that case by more aggressively updating the layer tree.

---

[249] This is not a very satisfying solution (it can get out of sync with the later methods for creating layers and painting, not to mention that an implementation could return different values each time it is called which would violate the expectations of this design). Depending on the language used, it may be possible to find a more type-safe design.

UI description

Various models for describing UIs were discussed earlier. In this section, we will consider the details for implementing the model suggested in that earlier section.

As a reminder, the basic approach is to have an immediate-mode API that maps to a retained-mode render tree built of the components described in the previous two sections.

The expression of UI

In this immediate-mode UI API, each part of the user interface is described using nested expressions, which are themselves composed together using nested expressions.

Because this API is the main face of the framework, it is important, if popularity is a goal, that it appear as clean as possible . To achieve this, some frameworks use a DSL[250], often HTML-inspired. However, with a sufficiently expressive language, function calls (or constructor calls) are sufficient and avoid the need for bespoke syntax.

For convenience, these expressions should be reusable[251], both in the sense of being used multiple times in one frame, and in the sense of being used in multiple different frames. For example, it should be possible to describe a part of the user interface that represents a vertical divider, and then reference that expression multiple times when building a list; similarly, the items in the list should only need to be generated once, and then used when needed as the list is scrolled. This implies that these expressions create values that are inherently stateless, can be long-lived, and do not have lifecycles that are tied to their usage in the interface.

Finally, these values should be cheap to build and discard, as they may need to be recreated every frame (e.g. during an animation).

If we assume that the language supports garbage collection with cheap allocation, and that it has some mechanism akin to objects, and that the constructors for these objects can take named arguments, the logical design is for these expressions to be nested constructor calls. Following Flutter's design, we will call these objects *widgets*. Because of the constraints just described, these objects should be deeply immutable (possibly by convention; for example, it is not unreasonable for a widget to hold a reference to a mutable object, so long as either there is a way to subscribe to that object to be notified when it changes, or, the object is never changed once it is given to a widget).

Kinds of widgets

When building a user interface, there are different components that can build the final picture.

Layout widgets

The ultimate goal of the API is to create or update a tree of layout components, therefore some of the widgets must correspond to layout components.

---

[250] There's basically two ways people have found to describe nested data structures: having opening and closing punctuation (as used, e.g., in Lisp, JSON, HTML, and C-style function calls), and indentation (as used, e.g., in YAML). The former style tends to have cascades of closing punctuation; the latter tends to be brittle when edited. While the closing cascades tend to be mocked, they are probably the lesser of the two evils in practice.

[251] This also helps performance.

These may have a variable number of children (zero for leaf nodes like images or text; one for most components that decorate the layout; a list of children for most nodes that perform layout like rows and columns; possibly a more complicated structure for those that have more complicated needs like a table layout).

These widgets should have the ability to create a corresponding render object, and the ability to configure a render object of that type to match the arguments that were passed to the widget.

Templates

In order to be able to compose widgets, there must be some mechanism wherein an expression of nested widgets can be given a name and used as a widget. In the simple case of an expression that cannot be reconfigured, this could literally just be a global variable that holds an expression; however, it is convenient for widgets to be configurable.

In principle, a function could be used for this purpose, but for consistency with the rest of the API, having a kind of widget that merely builds more widgets based on some constructor arguments is quite convenient.

Logic

Some UI components have state and logic. For example, a button renders differently when it is being actively tapped.

Because widgets themselves are, by definition, stateless, there must be some mechanism by which a widget, once it is being used, can instantiate an object that represents that instance of the widget[252] (the "state" object). That object is responsible for building more widgets, in the same way as a template widget, while also tracking internal values and having the ability to spontaneously request that the interface be rebuilt from this widget.

Configuration

The fourth kind of widget is one that provides configuration information for others in the tree[253].

For example, a widget could be used to set the stylistic choices (theming) of a subtree of the interface, or set the default language for localization, or provide access to the application's data model.

A simple implementation of this mechanism would simply walk up the tree any time the configuration was needed, but this O(N) algorithm effectively becomes $O(N^2)$ (because many widgets will need to walk many widgets up the tree). For this reason, it is beneficial to track the configuration widgets in play, and pass that information down the tree so that it is accessible in O(1) time (making configuration lookup O(N), just like building the widgets in the first place).

Additionally, there should be a mechanism so that when a widget obtains its configuration in this way, if necessary, it can subscribe to be notified when the configuration changes.

---

[252] Flutter calls this a "stateful widget", which is in some ways an oxymoron, but does correspond closely with how the widget is used by developers.
[253] The mechanism described here is essentially Flutter's "InheritedWidget" system.

This mechanism as described is a bit coarse. In practice, developers desire more fine-grained abilities to request configuration and subscribe to configuration changes. This is an area that will probably require additional research with usability studies.

Infrastructure

Because widgets are stateless, and don't map directly to render objects, there needs to be a data structure that represents the fully composed widget tree[254], so that updates can be tracked, and so that it can be mapped to a render object tree.

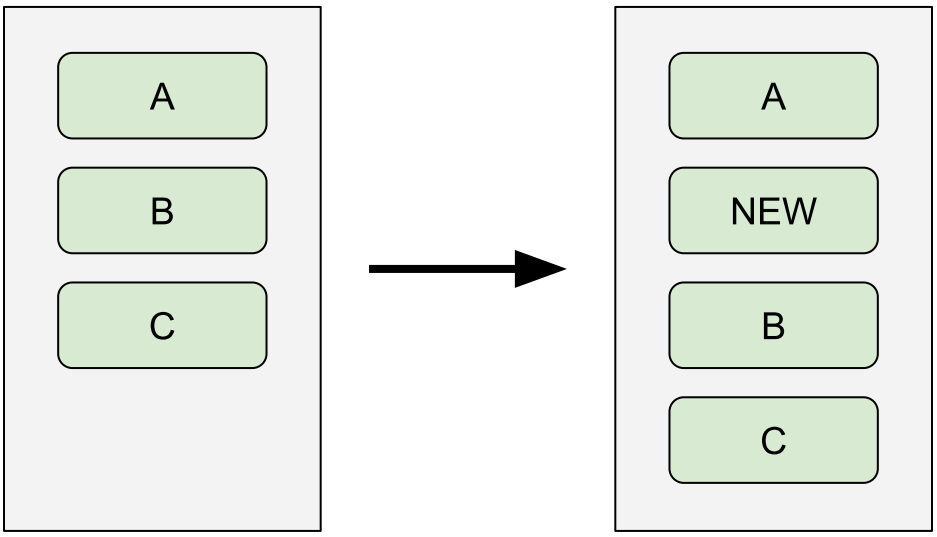
This tree can be created by starting from a root widget, creating a node that represents that widget, and then asking the widget for its child widget (asking it to "build" itself). The process then iterates, child by child, with some nodes (the multi-child layout widgets) having multiple children[255], and some (the leaf layout widgets) having no children.

As each child is processed in this way, any special behaviors relevant to the widget should be triggered as well: a parallel render object tree should be created from the layout widgets, the logic widgets should have their state objects created, the configuration widgets should have their configuration propagated, and so forth.

When the root widget is replaced, or when a logic widget's state object requests a rebuild, the tree is rebuilt by asking the affected widget to provide a new child node, and then iterating down the tree until one of the widgets returns a child widget that is identical to the previous widget at that place in the tree (this is sometimes known as the reconciliation phase). As nodes are replaced, any associated special behaviors must be updated as well: the render object tree must be updated to match new layout widgets, state objects must be notified of updates appropriately, and so forth.

Keying

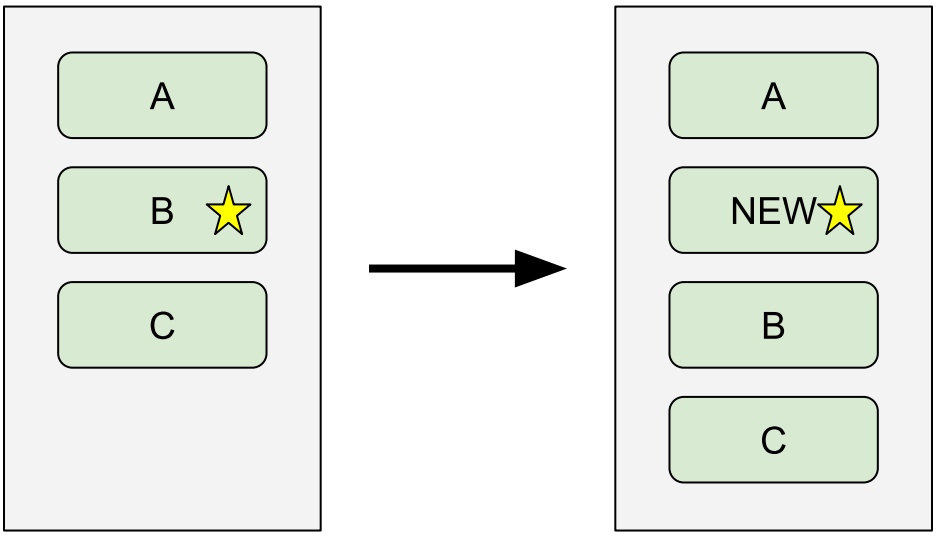
In some scenarios, e.g. when content is inserted or removed from a list, there is a difference between the identity of a widget as the user sees it, and as the framework sees it. For example, suppose a user taps a button in a list, which causes another button to be inserted before it:



[254] Flutter calls this the "element tree".
[255] In principle other kinds of widgets could have multiple children also. Flutter uses this in a widget that represents the list of application windows, for example.

If the framework has no way to identify the buttons over time, it can only use the placement of the buttons in the list to maintain their identity. This means that when the new button is inserted, any state that was associated with the button labeled B is now associated with the button labeled NEW, and C's state is now associated with B, and the new C button is considered a new widget.

This is problematic if the buttons, for example, are animated.



To avoid this, the widgets can be given an identifying *key*, which is used in two ways. First, when the key of a new widget is different from the key of the widget it is replacing, the old widget's state can be discarded. Second, when there is a list of children, the old and new children can be reconciled using the keys first[256].

Tree grafting

Because changing the shape of the tree in anything but the most trivial of ways will cause state to be discarded, there is a need for a mechanism that will allow parts of the tree to be moved (with state objects surviving). One way to achieve this is to have a way to identify subtrees (the aforementioned key mechanism, for instance), and to place subtrees in a table when rebuilding the widget tree, rather than immediately discarding them. Then, when that same subtree is specified for insertion elsewhere in the tree, it can be obtained directly from that table and inserted into the widget tree without rebuilding the state and render objects.

This, unfortunately, is a somewhat brittle approach, but it currently appears to be the state of the art.

Projection

While the simplest kinds of template widgets merely expand into more widgets, it can be useful to allow a widget to insert into its output a widget that was provided to it as an input. This is similar to how a layout widget takes another widget as a child. This is sometimes referred to as *projection*.

Projection is especially useful when the template widget is animating, but the projected widget is static. For example, a widget that animates its background color, while rendering a text label

[256] An algorithm that worked well for Flutter, when reconciling lists of widgets, is to start at the top of both child lists and pair up widgets until one that does n't match is found, then start from the bottom and do the same, and then with the remaining widgets in the middle, use a hash table to map the widgets using their keys. This is near-optimal because in most cases only one or two widgets are added or removed, and the algorithm manages to correctly pair up every widget in O(N) time.

unchanged. If the text label is provided as a child in the widget's configuration, and included in the widget's output verbatim, the framework can tell that it has not changed when the template widget updates itself. This can result in significant performance benefits (and can propagate down to lower layers; for example, the text does not have to be remeasured each frame).

This contrasts to designs that require the template widgets to build their children (e.g. by having this example take a string instead of a widget, and then having the template create the text widget child itself each frame).

Accessibility

As discussed above, the UI tree (widget tree) is mapped to a layout tree, which is mapped to a layer tree and rendered to the screen. Another tree that must be created is the *accessibility tree*. This is the representation of the interface for use with screen readers and other accessibility tools.

While every platform has different APIs for accessibility, they all broadly follow a pattern of having trees of nodes that are annotated with information representing the state of the tree, e.g. a checkbox can be checked or unchecked, can have a label, can be disabled, etc.

The accessibility tree depends on the geometry and positions computed in the layout and layer trees.

When using an aggressively composition-first approach for controls as suggested in this document, it is often the case that different parts of the widget tree will need to contribute independent parts of the accessibility settings for a single node in the accessibility tree. For example, the label of a check box will come from a different widget than the checkbox state, which will come from a different widget than the bounding box (geometry). To this end, the accessibility tree can be built by having widgets that contribute different components, with some widgets representing the collection point.

For practical purposes, it is generally reasonable to make accessibility widgets directly map to equivalent layout widgets, and then to have a pass similar to painting that builds the accessibility tree in a manner similar to how the layer tree is built.

Input

Gestures

Introduction

Multitouch devices are conventionally expected to support gestures, meaning sequences of inputs that have different effects on applications, such as:

- Taps (e.g. pressing a button), consisting of a brief contact on the touch surface.
- Double-taps (e.g. zooming at a point, selecting a word in a text field), consisting of a brief contact on the touch surface followed by another brief contact at the same point on the touch surface. In principle, UIs could also support longer sequences of taps, e.g. triple-taps.
- Long-press (e.g. activating a context menu), consisting of a tap where the contact lasts longer than some threshold time.

- Drags (e.g. moving an image), consisting of a single finger making contact with the surface, moving some distance, then being released.
- Two-finger pan, zoom, and rotate gestures (e.g. navigating a map), consisting of two contact points being tracked, and the vector between them, and relative to their original contact points, being used to compute a scale, translation, and rotation.

There is not a fixed set of these gestures; developers can invent new ones. (This is generally not a good idea because of discoverability, but there are always exceptions.) For example, Google Maps supports a double-tap-and-drag gesture to zoom with a single finger, and a two-finger pan gesture to change the perspective. Apple supports many gestures on its laptop trackpads, for example two, three, and four-finger pans.

Frameworks are responsible for providing mechanisms to convert raw touch input messages into high-level gesture descriptions.

Frameworks that are built atop other frameworks, e.g. web frameworks building on the HTML, JS, CSS, and DOM features shipped in web browsers, can defer the problem of recognizing a gesture to their host environment. This may prevent developers from creating custom gesture recognizers, but that trade-off may be considered reasonable as it removes some complexity from the framework. For the purposes of this document we will assume that the framework implements low-level gesture recognition itself, however.

The anatomy of a touch gesture

Multitouch devices report touches as a sequence of events, each associated with a *pointer* (essentially, the contact point that a finger makes with the screen). Each pointer goes through the same lifecycle:

1. At some point, a pointer is detected touching the screen (a "down" event).
2. The pointer is tracked moving around the screen ("move" events).
3. The pointer eventually loses contact with the screen (an "up" event).

These events typically also include information such as the force of contact or the parameters of an ellipse describing the contact area.

In principle, hardware could also detect "hover" events as a pointer approaches the screen, in which case a single pointer might receive multiple "down" and "up" events. Such hardware is relatively rare and is largely ignored in practice, but frameworks should support it.

A pointing device such as a mouse or stylus can also be modeled using this architecture, with the addition of logic to track the device's buttons. In the case of a mouse, "contact" would be defined as events happening while at least one button is depressed. Modeling a touch pointer or stylus in contact with its surface as having its primary button depressed helps unify the mouse and touch models.

The mouse-emulating part of a touchpad (moving the cursor around, clicking) can be approximated to this architecture also, but in practice touchpads require a separate approach for gesture handling due to the way touchpad gestures are not associated with a specific *screen* location.

Touch gestures consist of one or more pointer event sequences (sometimes in parallel, sometimes in sequence, depending on the gesture).

Multiple gestures can happen at the same time on a single multitouch device. Each pointer can contribute to at most one gesture (the alternative is not intuitive to users). There is therefore a disambiguation step required whenever a particular pointer could in principle contribute to multiple gestures. This is gesture recognition.

Gesture recognition

While there are various ways to implement this, the author would recommend a modular approach where each kind of gesture is implemented independently, and parts of the user interface that support multiple gestures disambiguate between only the gestures that are supported at that time.

For each pointer, the framework should determine which controls might be interested in the pointer. This can be achieved by performing a hit-test at the location of the "down" event, i.e. by walking down the layout tree[257], determining which nodes consider the point to be "inside" them. Once a list of candidate controls is built, these controls can each be sent the relevant pointer event (e.g. the "down" event) and given the chance to contribute one or more gesture recognizers for which they believe the pointer could be relevant. (This requires providing some object that can be obtained from all the controls.)

The gesture recognizers provided are now all in contention for the pointer. As events for this pointer arrive, the gesture recognizers each decide whether they should decline the pointer, claim the pointer, or continue to wait for more events. (In this way, there is no need to hit-test pointers more than once; the down event uses the layout tree, but subsequent events are delivered directly to the gesture recognizers.)

For example, a horizontal drag recognizer might reject a pointer if the move events report more vertical movement than horizontal movement. A long-press recognizer might reject the pointer if the up event is received before a sufficiently long period of time has passed, but might immediately claim the pointer after a threshold time has passed with no significant movement.

When only one recognizer remains in contention, it should automatically be considered to have claimed the pointer.

A gesture recognizer claiming a pointer does not necessarily mean that the gesture recognizer will then report to its control that the gesture itself has been triggered. It just prevents other gestures from claiming the pointer. For example, a double-tap gesture recognizer will be placed in contention for any pointers within its region, and will reject pointers that significantly move between their down event and their up event. However, the double-tap recognizer will only report a double-tap to its client if it ends up claiming two pointers in a row within a threshold time.

---

[257] Another approach to hit testing is to create an off-screen texture that represents the entire interface, with each pixel of the output being colored to identify the source widget. In principle, this enables O(1) hit testing, but in practice it has three major limitations. The first is that only one (or a small number of) node(s) can be specified for each pixel. This is not enough for interfaces where multiple controls may all have contributions to make to the decision on how to handle touch input (e.g. a button inside a long-tappable rectangle inside a horizontal scroll view inside a vertical scroll view). The second is that it does not provide sufficient information to handle transformations for the input. For example, a vertical scroll view rotated a quarter turn must have its input transformed accordingly so that it acts intuitively for the user. Doing this with the texture approach requires an O(N) walk up the layout tree to apply transformations; once that walk is required, the benefit of having the texture is defeated and the hit test itself could just be done directly on the tree. The third problem is that updating the texture requires a complete paint every frame; it cannot be updated incrementally (e.g. consider if a widget is removed — at a minimum, anything below it must update the texture, but to determine what was below it requires walking the entire tree).

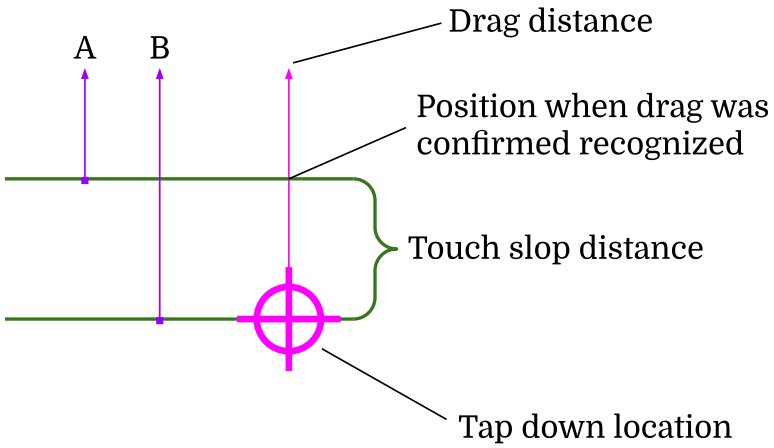
Grouping gestures

There are times where gestures in contention can defer to each other. For example, a switch control can be both dragged and tapped, but in most cases (when no other gestures are in play), the user experience is better if the tap gesture is ignored and the drag always claims the pointer. However, when a switch is in a vertically scrolling list, there is a third gesture in play, the vertical drag gesture, and therefore the disambiguation mechanism needs to determine if the user merely tapped the control (in which case it should trigger a tap), or dragged horizontally (giving the gesture to the switch), or vertically (giving the gesture to the scrolling list).

To handle these cases, a mechanism that allows gestures to be grouped and prioritized within the group is important.

Special considerations for drag gestures

In this design, when a drag gesture competes with other gestures, e.g. a tap, it cannot be recognized until either the other gestures reject the pointer, or the drag is sufficiently unambiguous that the drag gesture can claim the pointer. This means the pointer is at some distance from the original tap location, known sometimes as the touch slop distance, when the drag is recognized. If the drag gesture then reports the entire distance from the original tap location to the final position of the pointer as the drag distance (B in the diagram below), the user will perceive a jump. If, on the other hand, the drag gesture acts as if the drag started from the point where the drag was confirmed as recognized (A in the diagram below), the user will instead perceive some lag, as the drag they see on the display is shorter than the total distance dragged.



In practice, the lag is generally preferable to the jump. This is easily made configurable for those cases where a particular reason exists for preferring one over the other.

Mouse wheels and touch pads

As noted above, mouse input, stylus input, and the pointer-movement and clicking parts of touch pad input can be modeled using the same basic framing as touch input.

Mouse wheels (scrolling) can be modeled as a kind of out-of-band signal that is dispatched to controls in a similar way to other pointer events, but without needing the gesture disambiguation. However, in some cases, additional disambiguation is nonetheless needed; for example, in the case of nested scrollable components, where it is important that only one of the components, not both, scroll in response to a mouse wheel input. A disambiguator can be exposed to the controls in a

similar way as the gesture disambiguator, but its logic can be simpler as it is merely a matter of tracking whether any control has decided to handle the event or not (so that only the first does so).

This can be extended to also work for other kinds of events that are generated by track pads, such as zoom or rotate gestures.

Keyboards

Keys can be pressed and released, and can auto-repeat. Key input can therefore be represented as a series of events, down, repeat, up.

There are three ways to identify a key on a keyboard.

The first is the physical key's identity, for which the USB HID code is the most common representation. This value is useful for setting up keyboard shortcuts, though some mechanism for mapping physical keys to their actual labels is necessary when displaying such key bindings to the user (it is not helpful to tell a user that they have bound "0x1D Z" when they are in a Swiss French locale and their key is labelled "Y").

The second is the key's "logical" identity after the key has been mapped according to the user's keyboard mappings (locale). There is no standard set of codes for these keys; each platform has its own set. The framework should have a method of identifying logical keys so that keys supported by multiple platforms can be recognized as the same key, while also supporting platform-specific keys if necessary[258]. Logical keys are the means by which users reason about their keyboard.

The third means to identify a key is not strictly speaking identifying the *key* so much as the intended input of the key. For example, it may be affected by previously-pressed keys ("dead keys", e.g. hitting an accent then a letter, in some locales, generates an accented letter, instead of two separate inputs; also toggle keys such as Caps Lock) or other keys pressed at the same time (modifier keys, e.g. shift).

In addition to providing key events with physical, logical, and (where applicable) relevant input, a framework should also expose the current keyboard state as a whole (i.e. for each key, whether it is currently up or down). This makes it easier for developers to handle key combinations (e.g. Control + A).

Operating system API limitations make all of the above somewhat more complicated than necessary. For example, on desktop operating systems, a user can press a key, change the focus to another application, and release that key, confusing both the first application (which never receives an event to the effect that the key was released) and the second (which never receives the event to the effect that the key was depressed in the first place). To avoid confusing application code, the framework can synthesize events that are necessary to maintain a self-consistent representation (e.g. in the earlier example, the framework in the first application would eventually, upon receiving information that implies the key is no longer depressed, synthesize an up event, and the framework in the second application would synthesize a down event immediately before sending the real up event). Similarly, some system APIs do not provide the underlying USB HID codes, and some do not

---

[258] Flutter has a comprehensive solution here (the LogicalKeyboardKey API) which would serve as a solid starting point. It is automatically generated from data collected from every supported platform, carefully curated to map keys supported by each platform to a common list, while still defining an unambiguous mechanism for unknown keys (e.g. defined in versions of operating systems that post-date the time a Flutter application was built) to be identified unambiguously.

provide sufficient information to implement the physical-to-logical mapping. For each system, the implementation of the framework's API must provide as much information as possible within the constraints of the OS API.

On some systems, keyboard events may be emulated in response to users interacting with virtual on-screen keyboards. Depending on the actual implementation of these system features, there may once again be difficulties in providing all the information expected by key events. For example, a virtual on-screen keyboard may not have the concept of a physical USB HID code at all.

Focus

The target of mouse and touch input is implicit in a mouse or touch event, because the event corresponds to a screen location and the framework can (through hit testing, as discussed above) identify specific controls that could be affected.

For keyboard input, however, there is more ambiguity: the keyboard is not specifically associated with any particular text field!

To resolve this, a framework keeps track of the *keyboard focus*, and key input is sent to the currently focused control.

Focus navigation

Most platforms enable focus navigation through a pair of keyboard shortcuts, typically tab and shift-tab, which move the focus through an ordered list of controls. The framework is responsible for determining the order of these controls, and can provide affordances for the developer to control that order.

Some platforms also have directional focus navigation (e.g. TV interfaces typically have a directional pad on the remote that moves the focus around the screen). Frameworks are responsible for making sure that directional navigation can reach every control, without leaving "dead zones" in the interface where there are controls that cannot be reached by the user.

On some platforms, some controls should not receive focus by default; it is the responsibility of the framework to match the conventions of the platform.

Mouse and touch selection

On devices with both mouse or touch input and keyboard input, mouse activations can move the keyboard focus (for example, clicking a text field focuses it).

Web

On the web platform, tapping or clicking outside a text field (or a component associated with a text field) should unfocus the text field. This is unique to the web; other platforms don't act in this way. Typically this is implemented by having the whole application be a focusable region (in web browsers, that is the outer scrolling region of the web page).

Text input

The single most complicated part of any high-quality UI framework is its implementation of text input, as seen in text fields. In this section, we will cover the various features that text fields must have. (This list is probably incomplete.)

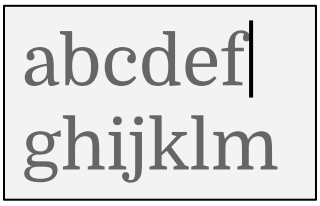
The cursor

The main feature of a text field is that it has a text insertion cursor (also called the caret). This is a position within the contents of the text field that represents where content will be inserted when the user types text.

Representation

Directional bias

While it seems intuitive to use a single integer to represent the position of a cursor in a text field, this is insufficient. In addition to the precise position in the content, an extra bit must be stored that represents the cursor's directional bias (sometimes called affinity). This allows the text field to disambiguate between being at the end of a line and being at the start of the next line when the text has wrapped at the end of a line in the middle of a word:



It also allows the cursor to be positioned both on either of the two possible positions between right to left and left to right segments. For example, consider the text "Hello world" with the word "world" in Arabic (عالم, which is read from the right to the left). When the cursor is after the space, it could be rendered either to the right of the space after the world "Hello", or to the right of (before) the word عالم:

This is important especially when dealing with cursor positioning in response to mouse clicks, as described later.

Only position between extended grapheme clusters

The positions themselves should be between Unicode extended grapheme cluster boundaries, not Unicode scalar values (and definitely not code points).

For example, consider this UTF-8 string:

> 0xf0 0x9f 0x87 0xa8 0xf0 0x9f 0x87 0xad

It is equivalent to this UTF-16 string:

> 0xD83C 0xDDE8 0xD83C 0xDDED

This corresponds to two Unicode scalar values:

> U+1F1E8 REGIONAL INDICATOR SYMBOL LETTER C
>
> U+1F1ED REGIONAL INDICATOR SYMBOL LETTER H

This corresponds to a single extended grapheme cluster: 🇨🇭

When considering what valid positions there are in this string, there should only be two: at the start of the string, and at the end of the string. Allowing the user to position the cursor between the regional indicator symbols would allow the user to "break" what they perceive as a single character, which is unintuitive behavior.

Rendering

Generally, the cursor is represented as a vertical line at the insertion location. The height of the cursor is typically defined by the size of the text that would be inserted. The width and exact rendering varies based on platform conventions (e.g. on some platforms it is a perfect opaque rectangle, while on others it has subtly rounded corners).

On some platforms, if the content of the text field has both right-to-left and left-to-right content, the cursor will have a little indicator showing the direction in which text will be inserted. On some platforms, when there are two possible renderings of the cursor because of bidirectional text (not because of line breaks), two cursors are rendered, one at each possible position.

Blinking

On most platforms, the cursor blinks. On some platforms this is an on/off toggle; on some platforms the cursor fades in and out over multiple frames. The precise rate of blinking is generally a platform-provided user-configurable value that the framework should use directly.

All of this is actually relatively awkward from the perspective of a UI framework, because users generally consider an application that is awaiting input to be idle, and expect it to take basically 0% CPU and GPU, but depending on the scene, blinking or animating a cursor could require recompositing a lot of the screen. Historically, cursor blinking was performed by merely xor'ing the relevant pixels of the display buffer on a timer, which is quite efficient, but this is no longer practical.

To the extent possible, cursor blinking should leverage features that minimize the amount of code that needs to run to perform the blinking. For example, the cursor could be represented by an

animated shader, moving the bulk of the logic into the GPU, requiring little if any coordination from the CPU, and requiring none of the core framework to be aware of the blinking.

On most platforms, the cursor should immediately revert to its most visible state when the user inputs text, causing the cursor to remain visible at all times while the user is typing.

Trailing spaces

When positioning a cursor for rendering, spaces at the end of a line are generally ignored. They are counted as cursor positions in the text content, but the spaces themselves are treated as having zero width at the end of the line.

Cursor keys

Text fields should support moving the text cursor around in response to arrow keys on the keyboard.

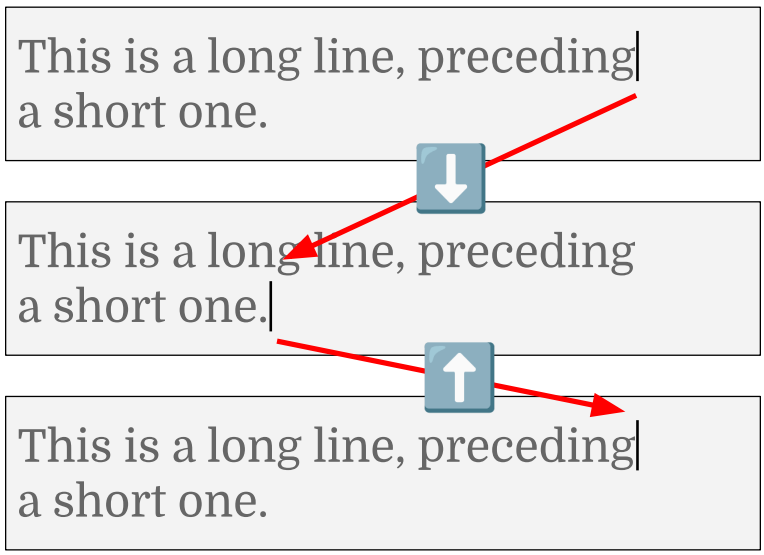
Vertical movement

Text fields are either single-line or multiline.

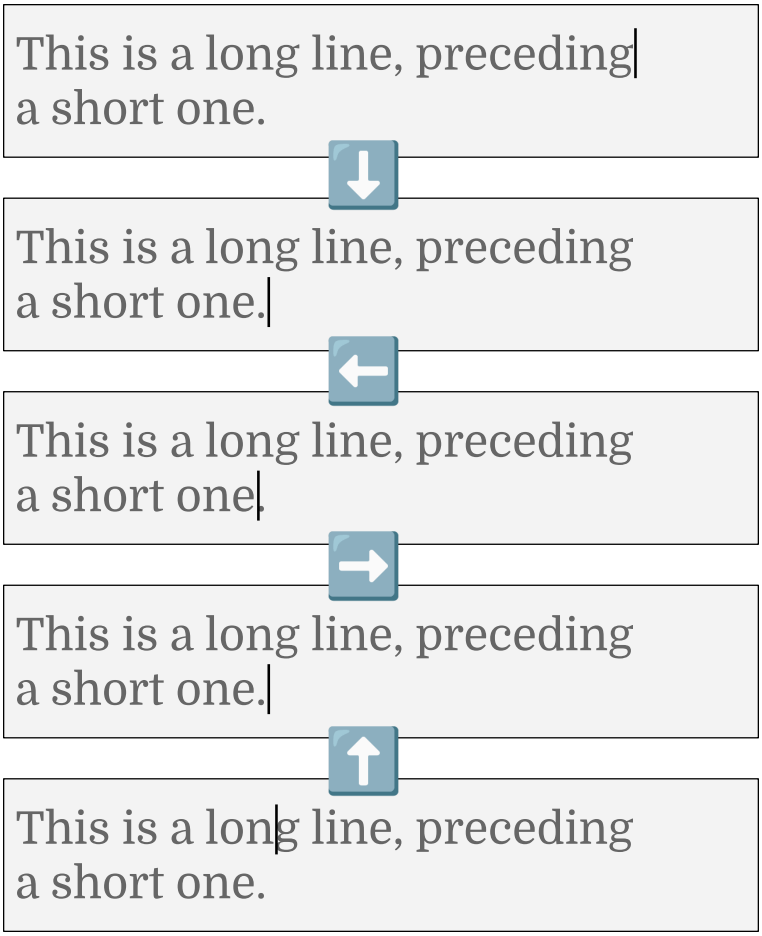
In single-line text fields, the up and down arrow keys should be configurable — they might be made to navigate in a history (as in a command line prompt), or for example the up arrow key could be made to edit the previous input. By default, they should move the cursor to the start (up) or end (down) of the text field.

For multiline text fields, the vertical arrow keys should move the cursor to the previous or next line while maintaining the same rough horizontal distance in pixels (not characters). The position being maintained should be the position of the cursor when vertical movement is started; this is sticky and only resets when the user does something other than vertical movement.

For example, consider this sequence (down, then up):



The cursor returns to where it started. On the other hand, if the sequence of vertical cursor movements is interrupted while the cursor was on the second line, returning to the previous line should cause the cursor to end up in the middle rather than the end:



Bidirectional text

Platforms differ regarding how to handle cursor navigation near mixed bidirectional text. Most platforms use the convention that the arrow keys move logically through content in a manner consistent with the overall direction of the text field, so for example, a left-to-right text field that contains right-to-left text will move the cursor left through that text when the user presses the right arrow key. Some platforms use a physical mapping (the right arrow key always goes right) but this leads to confusing behavior where holding an arrow key starts looping around mixed directionality text rather than advancing the cursor to the other end of the text field.

Keyboard shortcuts

Text fields should support the full range of keyboard shortcuts supported by the platform's text fields, e.g. the Page Up and Page Down keys to navigate by large jumps in multiline text fields, the Home and End keys to navigate to the start and end of the line, Ctrl+Home and Ctrl+End to go to the start and end of a multiline text field, etc. The precise key bindings vary quite significantly from platform to platform.

Mouse and touch input

Clicking or tapping the text field should position the cursor at the given position. In the event of bidirectional text or force-wrapped text, it's possible that a single position corresponds to two positions, as discussed above. It is generally unintuitive for users if tapping one location puts the cursor in the other equivalent location, so using the directionality bias is important here.

iOS floating cursor

On iOS, there is a feature known as the "floating cursor". Frameworks should implement this feature on iOS.

Selections

The text cursor can expand into a selection. (Most platforms do not separate the selection from the cursor.)

The selection consists of two positions in the text field contents[259]. The absence of a selection can be represented by having the two positions be the same[260], at which point the selection is the cursor position.

Keyboard shortcuts

Selections can typically be made and grown or shrunk using keyboard shortcuts. On some platforms, these shortcuts affect only one side of the selection; for this reason, the representation of the selection must include whether one side or the other has been chosen as the primary side, and which side is currently the primary side.

Mouse input

A drag should establish a selection from where the mouse down event was received to where the mouse up event was received. For the purposes of keyboard shortcuts (on platforms where this matters), the side of the selection where the mouse is released is the primary side.

Touch input (selection drag handles)

Platforms with touch input have a variety of gestures that can be used to create a selection, e.g. long presses. These should, naturally, be implemented.

Once a selection is made in this way, the selection typically has selection handles that can be used to adjust the selection. These differ in style and behavior based on the platform, but can largely be implemented using the same underlying logic.

Selection drag handles in scroll views

One notable trickiness of drag handles is that typically they are expected to render above all content, but must scroll with the text field when the text field itself is in a scroll view. The naïve implementation of such a feature leads to lag in the rendering, where the scroll view is scrolled in one frame, then the position of the drag handles is updated in the next. This can be avoided by having the drag handles' position be computed during compositing rather than during layout.

Drag-and-drop

Selections should be draggable to move content from one place in the text field to another place in the text field (or even to another text field, or another application). This requires a drag-and-drop model, where the framework can track that there is an active drag of content, so that it can be rendered, and so that the source and target can be updated when the drop is completed. This same

---

[259] Two offsets, or one offset and a length.
[260] Or, equivalently, the length be zero.

model can also support the operating system's drag-and-drop APIs; ideally, this should be done in a way that abstracts out the source and target details so that dragging out of, and dragging into, the application is indistinguishable to the developer compared to dragging and dropping entirely internally to the application.

Clipboard actions

As with drag-and-drop, clipboard actions should be abstracted out into some model that interacts with the operating system, and text fields should hook into that mechanism to offer cut and copy for active selections, and allow content from the clipboard to be pasted into the text fields.

Care should be taken to be resilient to potentially hostile content when pasting from the system clipboard, be it malformed UTF-8, deliberately misencoded PNGs, or some other kind of attack.

Selection replacement

When content is inserted into a text field, it should replace any selected content (if the selection is empty, it should be inserted at the cursor). This applies both for direct insertion (as in from key input) and for pasted content.

Other important features of text fields

There are a number of other features that are important for text fields, but for which there is not much to say:

Rich Text: Some text fields support formatting, inline images, and more. Having a control with such features is expected of UI frameworks because even basic chat applications typically support basic formatting of text and inline images.

Magnifiers: Most mobile platforms support a gesture in text fields to move the cursor that also triggers the display of a magnifier effect. These UIs vary slightly from platform to platform but, as with drag handles, are sufficiently similar that a common architecture can be shared.

Accessibility: As with any control, exposing a text field to accessibility APIs is a required component of the framework's implementation.

Focus: Text field should be focusable. When the control is not focused, the cursor should typically be hidden, and the selection typically is shown in a different color. The specifics vary from platform to platform.

Keyboard input: Naturally, a text field should support keyboard input for inserting text. Even on mobile devices, it is possible for users to plug in external devices.

Input Method Editors (IME): Most platforms have mechanisms for inserting text via software. These include mechanisms for inserting ideographs (e.g. for Chinese or Japanese) while using a Latin keyboard, mechanisms for inserting diacritics, and all the on-screen keyboards and voice input on mobile devices. Supporting these requires implementing system APIs. One particular aspect of this is that some IMEs require the text field to show parts of the text in specific styles (composition). Some of these IMEs have configurations, for example mobile on-screen keyboards can be configured to show special keyboards (e.g. for e-mail addresses or numbers), and can have their default capitalization changed by the application (e.g. a text field for names can be told to default to word capitalization).

Media insertion: In rich text fields, it should be possible to insert images via paste and via media insertion features of virtual keyboards.

Requesting and dismissing virtual keyboards: On mobile devices, users expect very specific behavior with respect to when virtual keyboards are shown or hidden. Matching the platform conventions on this front is therefore important.

Automatic substitutions (e.g. smart quotes): On some platforms, there are expectations around platform-defined text substitutions. Whether a text field enables or disables these features should be configurable.

Platform-specific features: Some platforms have very specific features that are expected of text fields. For example, iOS has the Scribble feature that requires dedicated support in text fields.

Autofill: Many platforms now have system-level autocomplete for user details like addresses, credit cards, etc. These require coordination with platform APIs.

Password autofill: Often implemented separately from regular autofill, some platforms also have a password manager and frameworks have to interact with that API as well to provide this important security feature.

Obscuring field contents: Password fields are expected to be shown in a manner where the number of characters is visible but the precise contents are not. (On some platforms, the most-recently inserted character is expected to be visible for some time as well.)

Spelling and grammar checking: Most platforms today have some form of spelling and grammar checker; exposing this requires coordination with how the text field is rendered, and may require exposing other user interface features (e.g. to change language), depending on the platform.

Context menus: Most platforms have some form of text input field context menu. This menu is expected to have basic controls like select all, cut, copy, and paste, as well as formatting options if the text field is a rich text field. In addition, some platforms provide an extension mechanism whereby applications can inject commands into this context menu (e.g. "Translate" or "Search"). Integrating into these features is yet another feature frameworks must implement for text fields.

Undo/redo: Users expect to be able to undo and redo edits to text fields, so an undo history must be implemented and keyboard and menu shortcuts for this feature must be provided.

Hints and labels: Developers expect to be able to place labels before or above text fields. This might be supported by having a separate control, but in some design languages (e.g. Google's Material Design), the label is an integral part of the text field. Text fields should also support having hints (text that shows when the text field is otherwise empty), as otherwise developers will find workarounds involving changing the text field contents dynamically and this tends to lead to UI bugs.

Prefixes and suffixes: Text fields should have the ability to prefix text before and after the actual editable contents, e.g. currency symbols. Having prefixes and suffixes support arbitrary widgets allows the most flexibility for developers.

Maximum length: Some use cases require specifying a maximum length for the user-inserted contents. Such maximums can be specified in code points (e.g. UTF-8 bytes), Unicode scalar values, or extended grapheme clusters; any of these might be required depending on the specific use case

(e.g. a backend database may have a specific data width in bytes). In any case, the maximum length should be enforced for all kinds of input, e.g. pasting, drag-and-drop, insertion via accessibility APIs, composition in an IME, etc. It should not, however, prevent text from being inserted that is longer during composition than after composition. For example, if the user is using an IME that takes pinyin (romanized Chinese), it is possible that the pinyin input would exceed the maximum length even though the final Chinese does not.

Input formatters (e.g. telephone numbers): A common feature of text fields is to convert user input in some way as it is entered. For example, a user might enter "14154369333" and the text field might display "+1 415 436 9333", or, equivalently, a user might enter "1234.5" and the text field might display "$ 1,234.50".

Implementation

Unfortunately, most platforms do not expose clean APIs for all of these features (e.g. password autofill or text field accessibility). As a result, on some platforms a framework may be required to create hidden text fields using the platform's own framework, and then implement these features by probing those text fields. At the time of writing this was definitely required on at least iOS and the web.

Tooling

An SDK requires a number of tools to be considered viable in the market:

- **A debugger** with breakpoints, which can step through code. The state of the art includes backwards debugging (stepping in reverse through code) and time-travel debugging (where the entire application's execution is recorded and can then be examined *post facto*).
- **A profiler**, which can record a program's execution and report where time was taken. For a UI framework, this feature is absolutely critical, as debugging performance problems can be extremely difficult without such a tool. A profiler reports the aggregate cost of functions or statements during the inspected period. Typically, entire animations are profiled.
- A **timeline inspector**, which is similar to a profiler but allows one to see program execution visually. This is usually used to examine the precise operations happening during each frame. A timeline inspector reports the operations performed during the inspected period as a visual graph. Typically, a single frame is examined at a time. A timeline inspector should be able to highlight particularly expensive frames to examine, and should allow developers to immediately determine what the frame's budget is being spent doing.

The need for a profiler and timeline inspector is especially critical when aiming to enable application developers to create high-performance applications, as discussed earlier.

Animation

While some frameworks take the approach of providing a dedicated animation mechanism that optimizes certain animations by limiting the scope of expression (as discussed in the performance

goal section), this approach sacrifices overall expressiveness and is really only appropriate in a space where the framework essentially determines the entire platform's look and feel[261].

In the context of a composition-based immediate-mode UI framework, animation can be implemented directly in the UI layer as self-updating logic widgets. By triggering an update every frame, and interpolating a value between an initial state and a target state over a given duration (with an animation curve applied to the fraction of time elapsed), any item of configuration in a widget can be animated.

To enable this to be applied broadly, the API can adopt a general interface for interpolation, where given two values of the same type, and a fraction *t*, a new value of that type can be generated. When this approach is applied deeply to the entire framework, even very complicated types (e.g. font styles, elaborate decorations, styled shapes, etc) can be animated with minimal effort from the developer.

Particularly common animations, e.g. fades (animating opacity), may benefit from dedicated fast paths. However, if the same API shape is used for all animations, replacing the normal implementation with a dedicated implementation can be done transparently as the need arises based on benchmarking in real applications.

Documentation

A fantastic UI framework can remain obscure forever, if nobody can ever learn enough about it to get started.

For this reason, having high-quality documentation is paramount.

In this section we will discuss API documentation, but high-level introductions, tutorials, overviews, and other guides are equally important. Those are briefly discussed in the operational strategy section below.

Content

The most important thing to remember when writing API documentation is that someone is reading the documentation, and that person is doing so for a reason. The question to ask oneself is thus, why are they reading the documentation?

The more reasons one can find, and then provide answers for, the more useful the documentation can be. In this section we will cover some questions that *every* entry in API documentation should answer, but this is not an exhaustive list; it is the bare minimum. The more questions one can imagine, the more useful the documentation is.

Every page of the API documentation could be the developer's first exposure to the API documentation. The documentation should never assume that the reader has any context whatsoever. This may mean pointing the reader to pages giving background information, or it may mean providing the necessary background inline. It certainly means that any terms of art or abbreviations should have brief explanations.

---

[261] This is why Apple can do it with CoreAnimation. For Apple, limiting the scope of possible animations is a benefit, rather than a disadvantage, as it helps keep the platform coherent.

Encouraging team members to actively contribute to the API documentation may help make the API documentation more useful. In particular, team members should be encouraged to add to the documentation the answers to any questions they have while working on the framework that they did not find answered there.

Documentation should not be removed when an API is deprecated. While the API is available, developers may need to use it. Even if the documentation guides developers to newer APIs, it should still explain how the older APIs work.

Why would I ever use this API?

The API documentation should give a high-level description of the purpose of the API, providing context for the feature.

This should answer the question in a way that goes far beyond what the reader could guess from the name of the API.

For example, an API called "FileDocumentReadConfiguration" should not be described as "The configuration for reading file contents". That is woefully inadequate. The documentation should provide the context for what other APIs expect to use this one, and what can be achieved by using this API.

The reader should leave any page in the API documentation knowing for certain why they would need the feature.

For example, a type definition should mention some places where the type is used, and a constant should mention APIs that use that constant in their logic.

How should I use this API?

This includes the contract that any user of the API must follow, as well as how other APIs interact with this one. It includes the lifecycle of the feature, other APIs that must be used first, and any configuration that's required to use the feature.

How does the API work?

This includes the contracts that the API attempts to follow, the internals of the API's design, any relevant performance characteristics, expected memory usage, the nature of any edge cases that the developer might not expect (e.g. how does the API handle invalid inputs, or valid inputs that they may not have considered, such as NaN, infinities, empty strings, nulls, or callbacks that throw exceptions), interactions with other APIs (e.g. does this enable or disable the use of some other API?), and potential race conditions or other timing considerations.

What is an example of this API in idiomatic use?

The documentation should show actual code showing the API in use in a non-trivial way. Showing the API in actual use is important; if the reader could look at the sample code and say "I could have guessed that" then the sample is insufficient. The whole point of the documentation is to answer questions the developer had.

If this is not the API I wanted, what do I actually want?

Sometimes, a developer may be looking at an API not because they wanted to use it, but because it sounded like what they wanted. For example, an API wanting to draw an arc might look at an API called "Curves". The API documentation should include pointers between such APIs to help lost developers.

What else might be interesting?

Readers may be reading the API documentation to explore the framework. Thus, every page in the API documentation should suggest more reading, and such pointers should not result in small closed loops; the reader should be able to keep exploring the documentation for hours, going from one feature to another, seeing how they connect.

Testing

Documentation is not exempt from testing. Any sample code should be statically analyzed, executed, and, where possible, unit-tested, all as part of the continuous integration pipeline for the framework as a whole. This prevents a very common problem in API documentation, where sample code doesn't actually work.

Useless documentation

Any documentation that could have been written by someone who knew nothing about the API other than its name is pointless. Such documentation fails to answer questions, as described above.

Diagrams

Sometimes the clearest way to explain a feature is a diagram. (For example, a UI widget can use a diagram to show how its various components relate, which simplifies explaining how the various properties control how the widget looks.)

Diagrams should be easy to edit, and diagrams that include the rendering of UI widgets should use the actual widgets' rendering. One way to do this is to have the diagrams generated from code, and generate the code automatically in the continuous integration pipeline.

Using the documentation as a sanity check

When documenting an API is difficult because the API is hard to explain, the solution may be to redesign the API.

Supply chain security

While still a rare feature among libraries in general, one area where an application SDK could distinguish itself is in having a very clear and compelling supply chain security story[262].

Ideally, when the SDK compiles an application it should be able to create a provable attestation that lists the precise configuration and source of every byte in the resulting binary, traced back to specific build environments on specific machines at specific times with specific dependencies, such

[262] See, for example, https://slsa.dev/.

that someone could reproduce the exact compiled bits independently to prove that the binary was not tampered with.

This requires support throughout the build system, in the compiler, in the CI/CD pipelines, in the ecosystem's package manager, etc.

Updating the SDK

Having the UI framework support self-updating dramatically increases adoption of newer versions of the framework. It also provides hooks for migrating developers when a breaking change is required.

This is a feature that is difficult to retrofit, and as such would be best implemented as one of the the first, if not the first, feature of the entire product.

Analytics

A question that the project leadership will eventually ask itself is "how many developers use our framework?". This question, and many variations on it (e.g. "how fast do people upgrade when we release a new version", or "what crashes are common in the compiler", or "which features of our profiler are underused") requires some form of analytics to be recorded by the tooling.

Adding analytics to an existing product causes much uproar among users. This can be avoided by adding the feature when there are no users. As such, it is best to add this feature very early in the product cycle.

Ethically, analytics must support a simple and permanent opt-out feature, and must provide explicit notice to developers when they are first using the product (with the opportunity to opt-out before any analytics reports are sent). There may also be legal notice or consent requirements, which are out of the scope of this document.

The analytics tooling should be designed to distinguish developers invoking the UI framework's tools from automated systems doing so. Otherwise, every new installation on a CI/CD system will count as a new user, which will completely defeat the purpose of the analytics.

Operational strategy

Building a UI framework and application SDK is a long-term investment that requires a large team, significant infrastructure, and an engaged community.

In this section, we will examine some of the non-technical decisions that would support such a project.

This section assumes that the project operates primarily as an open source venture, as suggested in the Design choices section above. All source is developed in public (e.g. on GitHub), with a core team that accepts contributions from the public. Core contributors may be from multiple employers or be volunteers.

Team culture

The single most important factor in determining a team's success is the team's culture. Culture comes from the top. It cannot be delegated. The leadership must truly believe in it.

When done well, maintaining team culture requires very little effort. Unfortunately, destroying a functional team culture is even easier.

Mission

The ultimate guiding principle behind any effort is its mission. Having a clear mission allows team members to make consistent high-level decisions.

There are many possible missions. "Profit" is a common one: ultimately, making money drives much of the engineering work in our society.

This document is based on the premise that the mission has something to do with making the most popular UI framework. There can be nuance in how to frame that as a mission however. For example, there is a difference between "make the most popular cross-platform framework", and "make a single framework that is the most popular framework for each platform". In the former case, one is competing against other cross-platform frameworks (Qt, Flutter, React Native, the web). In the latter, one is competing with the leading single-platform frameworks as well (e.g. SwiftUI, WinUI).

In the author's opinion, aiming to create the best or most popular framework for each platform is a more ambitious goal than merely creating the best cross-platform framework; it is easy to dismiss the single-platform frameworks if one is focussed on being a good cross-platform framework, but if one is focused on being the best framework for targeting just one platform as well, one must take a more holistic view. This is especially important as it is not uncommon for developers to use "implement the application multiple times" as their cross-platform framework.

In the author's opinion, therefore, a good mission statement would be something like:

Create the most popular UI framework for	Android. iOS. Linux. macOS.

OpenHarmony.
robots.
the web.
TVs.
wearables.
Windows.

Values

Fundamentally, culture is drawn from values.

Values must be sincere

To reiterate: these *must* be sincerely held beliefs of the leadership. One can't claim "weekends are sacred and we will not work on weekends" as a team value if the team's manager regularly sends e-mails on Sundays. Values should always be top of mind for the team's leaders — not artificially, but because they truly represent how the leaders think.

Values are a trade-off

When describing values, it is important to present the trade-off being made. By definition, values are not self-evident. They are something the team believes in *in contrast to something else* that they could instead have believed in, and that other teams might believe in, but which they believe is less conducive to their ultimate goals.

For example, a team might value speed over quality because the team feels it needs to have a product in the market sooner than its competitors, even if the product is lower quality, to be able to secure funding to continue, and that if the focus was on making a quality product, the team would run out of funding before being able to have a product in the market at all. Similarly, a team might value quality over speed because they feel they have sufficient funding to make a quality product, and that even if they are not first to market, their long-term success will come from having the best product.

Values for a productive team

Work-life balance

Stressed team members are less productive. Values that speak to having a healthy work/life balance can mitigate this. For example, asking team members to ensure they have a life outside of the project. For paid employees whose job is the project, this could include encouraging team members not to work more than their designated work hours, or making it clear that there is no expectation that e-mails or text messages receive immediate replies while someone is not working.

Supporting this could be values around not having deadlines. For example, never pre-announcing features.

The trade-off

Values that encourage healthy work-life balance are betting on long-term team productivity instead of short-term success. A team *could* instead believe that they will be more productive if they have a high turnover of individuals who are each intensely productive while on the team. They may also

believe that having more intense work commitments from each team member means they can have fewer team members overall, reducing costs.

Decision-making

Being data-driven, and encouraging decisions to be driven from debating data rather than personalities, can help focus the project on the needs of users rather than on team politics. In the case of a project like the one described in this document, where usability studies are proposed as an important path to finding the optimal design, ignoring the data would be a big waste of resources.

The trade-off

Teams could instead value charismatic leadership, giving individuals the credit (and responsibility) for making decisions. This would enable faster decisions, and, assuming the leaders are experienced, have a good intuition, and are unusually lucky, could result in a higher quality product than one "mired in data collection and nit-picking" (which might be the way data-driven decision making is characterized). A leader with very strong vision can also bring a project to a more *coherent* design than one built purely from data and committees (at the risk, maybe, of it being a less perfect fit to the market).

Values for an open source project

If the project is open, then having its openness be a core value is important. The "team" should be everyone contributing, not just those paid by an employer. It is very easy for a team all working for a single company to become insular and treat other contributors as outsiders. This can quickly defeat the benefits of having an open source project.

The trade-off

A project could value the closeness that comes from a colocated team, and see having their source being open as a cheap way to get some credibility in the community.

Explicit values

Having determined the values for the team, it is very useful to write them down. This lets new team members see what the team aspires to. Having the values written down can also help when facing a difficult question. Sometimes, rereading one's values can clarify the correct choice[263].

It can be very helpful in terms of aligning the team to the values to repeatedly call them out. Explaining team decisions in terms of the team's core values teaches the team the importance of those values and how they apply, while demonstrating that they are real[264].

---

[263] For example, consider a situation where a contributor proposes a really clever change to the API that makes it much more usable, at the cost of hurting performance by a small but measurable amount. If the team's values explicitly put performance above usability, or vice versa, then the choice is easy to make. If the team has no (relevant) values, then it becomes a judgement call that is hard to justify, and even harder to make consistently across the lifetime of the project. (These kinds of situations can sometimes be used to crystalize one's values, documenting the decision that was made and what that implies about the team's values.)

[264] Obviously this works best when the values are being consistently applied. If the leadership team makes decisions that ignore the documented values, and pulls out the values to justify their decisions when they happen to align, all it does is teach the team that the values are meaningless and that the leadership team is manipulative and dishonest.

Values are axiomatic

A team's core values should rarely change. Many of the benefits of having documented core values come from the stability of their existence over time. As such, they are not generally up for debate. It is healthy to allow discussions, including talk about the team's core values and how they apply, but entertaining regular conversations about potentially changing the core values reduces their credibility as pointers on the team's moral compass.

Values are threatening to some individuals

New team members (especially paid employees for whom the project is a job, rather than a passion) who find themselves misaligned with the core values can feel threatened by them, and can be harmful to the project. For example, hiring a strong-willed new senior engineer with strongly held opinions that differ from past decisions of the project will cause problems if one of the core values is that decisions should be backed by data. They might find their usual approach of making decisions and pushing their implementation gets pushback from team members who expect decisions to be based on research. This will lead to them pushing against the team's values.

This can lead to dissatisfaction, stress, and damage team unity. It is therefore important for the team leadership to notice such misalignment early, and either help new team members align to the core values, or remove them from the team.

Value deterioration

As the team's leadership changes over time (e.g. due to leaders moving on to other projects), the moral core of the project can weaken, leading to the team's "true" values (as demonstrated by actions) drifting away from those written down. It is important to empower everyone on the team with the responsibility for enforcing these values and passing them on to new team members.

When values do change, at a minimum, the written values should be updated to match reality. Otherwise, they become a work of fiction.

One must recognize that a team has values regardless of whether they are consciously chosen or not. When the leaders stop believing in the team's purported values, that does not mean the team has *no* values. It means the team has *different* values. These values may be toxic, they may be dangerous for the health of the project, they may be demoralizing for the team members who joined believing the original team's values, but they are still real. The direction of the product and of the project will be influenced by these real values, whatever they are.

The impact of decisions

Developing software fundamentally involves making trade-offs. This document describes many: popularity, or performance? simplicity, or vertical text support? risk creating a new language, or risk using an existing one?

Trade-offs, *by definition*, will disappoint some users in the interest of making other users happy (if a decision makes everyone happy, it's not a trade-off). This means that every time one makes a decision, there is the risk that someone will complain, either immediately — in which case it's usually easy to respond by explaining the decision, as it is fresh in everyone's mind — or later, in which case it may not be immediately apparent that the complaint is the result of an intentional

decision at all. In the latter case, attempting to resolve the complaints may lead to making a decision that selects the worst of both options that were originally selected.

For example, consider the vertical text decision mentioned earlier. Suppose the team makes an early decision to not implement vertical text, in order to have a simple API. The trade-off is between having a simple API (by not having the feature), or having high-quality vertical text (by baking the feature in early). If, many years later, the team receives complaints that vertical text is not implemented, they may be tempted to retroactively add the feature. Doing so, however, would mean that the project has neither a simple API (since it now does have vertical text), nor high-quality vertical text (since it isn't baked in from the start). This is the worst of both worlds.

The team must therefore always be vigilant about how it responds to complaints. In some cases, they are expected, and represent success (or at least, represent the earlier decision), rather than being an unexpected problem[265].

Surveys

As the team grows, it will become more and more difficult for leaders to remain connected to the day to day experiences of the team's more junior members, as well as those members with very narrow interests who contribute only in one specific area of the product. Periodically polling team members (by use of anonymous surveys, to encourage responses and remove any possible concern about retaliation or shaming) can help illuminate developing problems.

Such surveys must be conducted with the sincere goal of identifying trends and learning about problems in the team. If the survey responses are ignored, or if results are collated but then ignored, the team will quickly and correctly determine that the exercise was a waste of time and that the team health is not a priority. This will destroy morale and will erode team spirit.

This is especially important when the results illustrate a growing disconnect between team members and the project's leadership. Leads should attempt to learn why such a divergence has happened. It can be difficult to recognize that change must happen, especially if it runs counter to the preferences of the leaders. At the very minimum, if the team's opinions have diverged from the values espoused by the leads, the leads should attempt to explain the reasoning behind their (unpopular) opinion. A team is much more willing to follow a policy they disagree with if they understand it.

Dogfooding

As discussed earlier in the context of avoiding complacency during usability testing and then again in the context of designing a programming language, teams should regularly use their product. There are several reasons for this.

First, there is a category of bugs and frictions that users (developers, in the case of a UI framework) will not typically report. For example, a command line tool's line wrapping may be ugly. This is a trivial bug that few users would ever report, but which makes the tool look shoddy. It is easily fixed by a team member when noticed[266].

---

[265] For another discussion of this phenomenon, consider the author's "When complaints are a good sign" blog post from 2024.

[266] Ideally by a team member primarily working on that part of the code base, but *anyone* on the team should be empowered to make this kind of change. It is the team leadership's responsibility to ensure that the team culture normalizes this kind of cross-team contribution.

Second, there are bugs that users have no chance of recognising, but which team members will immediately notice. For example, if a minor feature fails to trigger, users may be completely unaware that it even exists. Only a team member who is aware of the feature will notice the absence. Not catching this kind of bug is a waste of time (the time spent creating the feature).

Third, having experience with the product helps prioritize and contextualize bugs. Users often do not have a full understanding of the product and may file a bug and describe it as serious when its severity is entirely the result of a more obscure issue exacerbating some underlying problem. Team members who are aware of the bug, and who then experience the obscure issue or underlying problem may be able to use their knowledge of the product to connect the dots in a way that a regular user would not be able to.

For all these reasons, the team culture should encourage team members to regularly put some time aside to *use the framework* (specifically, the development version of the framework, so that any problems found can be fixed and the fixes immediately experienced). The amount of time put aside for this needs to be enough for folks to be able to make real applications, so that they run into the same real issues as other users (and experience them sufficiently to motivate them to fix those issues).

These reasons all have one thing in common: in each situation, they only apply to the few individuals on the team who happen to have all the relevant context and experience. For *this* reason, it is critical that *everyone* on the team spend some time using the product, and do so specifically with an eye towards improving the product.

The ideal application is one that the member cares about, so that the bugs have the same emotional impact as it would on other users. This typically means it should not be something directly associated with the project itself, but should instead be a personal project. Team members who work for corporations who will attempt to claim ownership of such projects or otherwise stifle creativity outside sanctioned projects should find some official exception to such rules to enable team members to genuinely have a personal project created (partially) on company time and company hardware using the framework.

It is possible to leverage this dogfooding requirement to further help bond the team. Team members could work in groups on these projects; if these groups are different from the usual organisation of the team, this will help with cross-team bonding. Team members could be occasionally encouraged to work on these projects all at the same time in a hackathon arrangement. Projects could be regularly demonstrated to the wider team, both to garner (positive) feedback and encouragement, and as a way to encourage everyone to participate.

Communications

Creating popular software necessitates communicating with users and potential users of the software. In the case of a UI framework, this means outreach to application developers. This takes various forms. The most technical form, API documentation, is discussed in an earlier section. Other forms include accepting bug reports and feature requests, community building, short-form educational content (videos, blog posts), long-form education content (videos, web documentation, technical talks), product announcement live streams, product announcement in-person events, maintaining a social media presence, and engaging in social media discussions.

Reputation

A critical component of having a popular software product is maintaining a positive reputation. A team viewed as inclusive, attentive, responsive, fun, and competent will have a much easier time maintaining and growing a loyal user base than one lacking any of these attributes. It is tempting to think that the product will speak for itself, but all the positive goodwill that a technically sound offering will create can be destroyed by having a reputation for being a team that ignores feedback and views its users as an annoyance.

The easiest way to maintain a positive reputation is to legitimately earn it. This starts by having a positive team culture (as discussed in the team culture section), and by being explicit about the team's values, and then sincerely upholding those values. While this will generally push the team in the right direction, however, a reputation can be further maintained by additional explicit choices around communications, which we will discuss presently.

Voice

A project's *voice* is the tone and style used by all its communications. For example, it could be playful, or business-like. It could be intentional about using simple English, or it could be intentional about using technical English[267]. It could be antagonistic or saccharine, it could use youth slang or be strictly formal.

Having an explicit description of the project's writing style allows the text written by various writers on the team to maintain a consistent voice over time.

As with the recommendations in the previous sections, usability testing can be used to find the most effective voice to achieve the goal of a popular UI framework. That said, by default, one would recommend a playful yet humble voice that uses technically accurate language while minimizing unnecessary complexity, avoiding sounding condescending, antagonistic, or infantile, and avoiding slang.

Under-promise and over-deliver

Schedule

An important way to gain trust from the community is to avoid making claims that have to be walked back or promising features that are then delayed. An easy way to achieve this is to not promise anything that isn't already implemented.

This requires discipline from the team, but is fundamentally quite simple to execute. Instead of announcing features six months before they are ready to launch, the entire announcement schedule can be slipped nine months at the start of the project, meaning that features are announced three months after they are ready (giving some time for finding critical issues before the announcement).

With this approach, the announcement rate is unchanged, the announcements themselves are unchanged except for being more precise (as they can be tailored to the actual state of the feature as shipped), all demos can be real instead of simulated, and there is never the risk that an announced feature slips.

---

[267] It could also be another language, of course. Or the voice could be defined in multiple languages.

The impact of this approach is an increase in trust from the community. This leads to the community being willing to defend missteps in the rare cases where they might happen. It also gives the team a lot of credibility when the team needs to ask the community for patience. For example, in the event that the team is caught unawares by a platform vendor changing an API, the team can promise that a fix is coming soon, and the community is more likely to give the team the benefit of the doubt than if the team repeatedly promises features and then either slips the launch date, or launches something less impressive than originally promised[268].

This is culturally very different than the norm in our industry, where as a rule we like to pre-announce features. As such, it would require very explicit choices to be made by the leadership team and communicated to the customer-facing parts of the team, as well as buy-in from the team to work in this unusual way. This is an example of a "brave" choice that may get push-back from those with industry experience.

Scope

In a similar vein, when describing features, aiming for understated claims rather than bombastic claims will build a reputation for trustworthiness. It will also lead to the community feeling compelled to "correct" the record and make the more impressive claims on behalf of the team, which makes them more credible to the general public.

Engage positively

To the extent that the team responds to the public (e.g. in social media, in issue databases, etc), an unfailingly positive attitude reinforces the team's positive reputation. "Positive" must be sensitive to the context; it can mean being respectful, cheerful, supportive, as appropriate.

Examples include never saying anything negative about a competitor, avoiding subscribing to the premise of a zero-sum game (i.e. favoring "we want to grow the pie" instead of "we want a bigger slice of the pie"), highlighting successes in the community, emphasising the positive aspects of proposals when evaluating them, and phrasing criticism as opportunities for improvement rather than as failings in the proposal.

Obviously, one cannot be positive in response to everything. However, where one cannot find a way to write a positive response, one should instead avoid responding at all. There is no value to responding to *ad hominem* attacks, for example.

Some brands will sometimes respond to negative sentiment with clever retorts that seem positive on the surface but are actually mocking the critic. While this may work in other contexts, in the context of a software project it is generally not a good strategy to mock a potential user. In general, ignoring such feedback is the best way to avoid encouraging more or drawing attention to it.

Admit errors

The easiest way to accumulate credibility is to admit mistakes and agree with valid criticism. Adding context, talking about the constraints that led to the mistake, and describing changes the team has made to avoid such mistakes in the future can help as well, but fundamentally being seen to take responsibility for errors can only improve one's reputation.

---

[268] Both of which are inevitable if the promises are made ahead of the implementation.

Denying one's mistakes will not retroactively make those mistakes go away. Attempting to justify mistakes makes one sound out of touch. Admitting a mistake costs nothing[269].

This is another example of a "brave" choice, in that it sadly flies in the face of industry norms.

Code of conduct

The reputation of a community is generally driven by the worst members of that community. By extension, the worst behavior the team is willing to allow in their community spaces will determine the reputation of that community. For this reason, the code of conduct[270] defines the reputation that the team's community spaces will eventually earn.

It is common for a code of conduct to allow any behavior short of the most egregious. For example, a code of conduct might specify that sexual assault is a violation of the code of conduct. This may be sufficient if the goal is to avoid permanently traumatizing community members, but if the goal is to grow a community that has the reputation of being welcoming and helpful, then it is inadequate to the task.

Instead, the author recommends raising the minimum bar so high that even the slightest grumpiness is considered out of line. For example, requiring that all communications always be actively welcoming and helpful, that criticism should be constructive, that people listen to each other before responding, that all interactions be respectful, kind, and courteous towards other members of the community as well as to people and cultures who are not present.

Enforcing such a code of conduct requires a nuanced approach. New community members are especially likely to fall afoul of rules with such a high bar. Having existing community members gently let them know that "we don't do that here" will help them course correct. Continued unwelcoming behavior after being redirected should result in a short ban (e.g. 24 hours), with subsequent violations receiving progressively longer bans (a week, a month, a year). Moderators should never appear gleeful about banning a community member, but nor should the bans be done secretly; the purpose of the bans is as much to indicate to the community the seriousness with which the code of conduct is taken, as it is to convince the participant of the error of their ways and to limit the damage they can make to the community.

The most difficult aspect of enforcing a code of conduct like this is the question of how to enforce it against privileged members of the community, such as those in leadership positions or those who are paid employees of a core contributing company. For example, if a company is responsible for the bulk of the project's work, and an employee of that company is rude towards a volunteer member of the community, the code of conduct would call for a ban, but applying a ban would be politically complicated.

Having the top leadership of any major corporate contributors be fully invested in this code of conduct is therefore key, as it removes the conflict. When an employee violates the code of conduct, the company's leadership would take it as seriously as the moderators, and would support a temporary ban (or, possibly, consider reassigning the employee to a role that does not involve public interactions).

---

[269] Except one's pride, but really, that cost should have been paid when the mistake was found. Acknowledging it does not make the mistake any worse, and avoiding doing so does not make it any better.

[270] Specifically, the code of conduct that is actually enforced, whether or not that is what the written code of conduct claims.

Issue database

A software project must have a way to track defects and feature requests.

Any project with a scope as large as that described here has an infinite number of bugs (let alone missing features). At any one time, only a small number of these bugs are known, and even fewer and understood well enough to be actionable. A well-constructed bug report is a valuable gift. It's therefore important to encourage the filing of high-quality bug reports.

A bug report is of high quality if:

- It clearly states the *problem* (as distinct from the desired *solution*).
- It has clear steps to reproduce the problem, ideally in the form of a unit test.
- The team agrees that the problem is a real problem.
- It is actionable.

It can be helpful to provide guidance for users explaining how to best report issues.

Triage

Once a project is popular, it will attract issue reports. This will require dedicated staffing. Ideally, every issue would be examined, and an attempt would be made to reproduce it. Issues that can't be reproduced would be categorised as likely rare intermittent issues that warrant more study by the team (potentially in the form of collecting more issue reports from more users), or as issues requiring more information from the original reporter. In the latter case, the issue should be automatically closed if the reporter cannot provide the required information in a relatively short time frame (at least a week, but less than a month[271]).

The ideal state for a bug database is for it to have an infinite number of immediately actionable, well-annotated issues with clear steps to reproduce, up to date metadata (specifying the general area of the product affected, the priority of the issue according to some mostly objective criteria, and tagged with as many relevant labels as possible to make finding the issue straightforward), and no more comments than is necessary to make further progress. The bug database should have no duplicates.

Not having a triage team will result in the issue database becoming useless as it is overwhelmed with ignored issues. The worst outcome is for the development team to use another system as their source of truth for planning, rather than the public database. When this happens, there is a failure of the team culture that may be very difficult to redress.

In general having a dedicated team (as opposed to a rotation where each member of the development team is responsible for triage on a periodic basis) is preferable, because triage requires a unique skill set and knowledge of the entire bug database is especially useful (e.g. for finding duplicates).

[271] The goal is to allow flexibility for developers who file a bug and then go on a vacation, while still keeping the number of open but not actionable bugs to a minimum.

Closing issues

Automatically closing issues due to lack of activity sends a strong message to users that the team is uninterested in their feedback. This will certainly result in the bug database having few issues filed, but it will also blind the team to the real state of the project.

Issues should be closed when they are not actionable, e.g. because they have been fixed, or because they do not have steps to reproduce the issue in question, or because the proposals run counter to the values of the project. Essentially any low-quality issue (by contrast with the definition of a high-quality issue given earlier) should be closed. This keeps the bug database useful for the team.

Issues should be left open when they represent real problems, or valid feature requests (in the sense that it is plausible that the team would eventually implement that feature).

Priorities

It is important for team health and for transparency with the public that priorities assigned to issues be accurate. If an issue is labeled with a priority whose definition is that the issue will be immediately taken up and will have daily updates, then that is what should happen. If the issue is not being regularly updated, then it should not be given that priority.

This is another example of honesty and transparency being important for a healthy project.

Community building

For a developer product like a UI framework, there are two communities to consider: people using the framework, and people developing the framework. These communities overlap, but are distinct and have different needs.

The larger community (assuming the project is popular!) will be the community of developers using the framework in their applications. They will need spaces to discuss how to use the framework, ways to report their needs and desires for future versions, access to the bug database to report issues, documentation in all forms (API documentation, short and long videos, tutorials and codelabs, etc), and in-person events (e.g. conferences). A subset of this community is the library developers, creators of third-party libraries for developers to use in conjunction with the UI framework.

Generally, this is best handled by a dedicated team.

The second community is the development team itself. For an open source project, this is likely a mixture of employees from sponsor organisations and volunteers. For best results, the project should have a uniform set of policies that apply to all contributors, regardless of affiliation. These policies should cover how to get commit access, how to get access to CI/CD systems, how to access the bug database, how to participate in security incident processes, how to participate in triage, how to write and review design docs, how to participate in regular team meetings, etc. The sign of success here is if team members can change employers without it affecting their participation in the project.

Moderation

As in any situation that interacts with the public, not every interaction is positive.

Bug databases in particular, as the most visible parts of an open source project's development processes, attract a variety of contributions that a team could do without. Problems, however, can appear in any of the team's venues, from group chats to convention floors.

Moderators

The team should have members who are empowered to apply the code of conduct (primarily applying bans, but also hiding harmful content and generally cleaning up after violations, e.g. from spammers).

For small to medium teams, it may be reasonable to have all members who have passed some minimum bar (e.g. the same bar as getting the right to review contributions) have moderator privileges.

Categories of nuisances

Well-meaning individuals whose values do not align with the project's values

This kind of interaction usually manifests as a series of long arguments about how an issue should be resolved (or even *whether* it should be resolved). Among well-aligned individuals, these discussions usually make evident forward progress, even if they are lengthy (issues can be complicated!). With individuals whose values are not aligned with the project's, the arguments tend to go around in circles and feel futile and frustrating[272].

Identifying the cause of such an argument as a value misalignment can be difficult, but once that has been identified, it is important to firmly state the project's values, and conclude the discussion. Allowing a discussion to continue can be misleading for the wider community (such discussions often become relatively high-profile and the community will read between the lines).

Well-meaning individuals whose skill level is not equal to their ambitions

With an SDK, where users are writing their own code and creating their own bugs, it is not uncommon for less skilled developers to misattribute their issues to a problem with the SDK, and file issues that turn out to be their bug rather than a bug in the SDK. Determining this can be a time-consuming process.

Similarly, a developer may run into a bug and describe it with confidence, only for it to later become clear that the nature of the actual issue is entirely different.

Alternatively, a contributor may attempt to help other people who are filing issues, by unintentionally giving them bad advice, frustrating and confusing them.

Another way this dynamic may manifest is if the individual is a high-achieving child. This may be their first interaction with a "serious" project, and they may simply not yet know how to engage productively[273].

In all of these cases, it can be difficult to notice the problem before it has wasted a lot of time. In the case of someone "helping" other users, it can be even worse than wasted time: the other users may

---

[272] It's important to recognize that these discussions may nonetheless be extremely respectful and welcoming on all sides. Merely disagreeing does not itself make a code of conduct violation, even with the most stringent of codes.
[273] This is not uncommon. There's really no way to know the age of people online.

mistake the individual as a member of the core team, leading them to form negative opinions of the project.

How to deal with such an individual depends on the specific case. The first step is usually to reach out to them privately to attempt to determine their situation. They may have skills that are better put to use in a different part of the project, for example. If they are young[274], they may benefit from some mentoring; very little effort may quickly pay for itself, if they are competent but inexperienced, and have a lot of spare time.

On the other hand, if they are a lost cause, it may be appropriate to ask them to refrain from contributing further. This can be a difficult conversation to have, but it is better to have it early than after someone more productive has become frustrated and left the project, or snapped and broken the code of conduct.

Well-meaning individuals whose socials skills are in need of further development

The nature of an Internet-hosted open project is that a large fraction of the entire population of Earth has the opportunity to attempt to contribute. Sadly, some of those people are socially inept. As with technical competence, this can be the result of inexperience; it can also be the result of cultural differences. In both cases, privately contacting such individuals to explain the social norms of the project can turn a problematic contributor into a productive one, and it is therefore often worth the small effort.

Sometimes, attempting to shut down a discussion based on value misalignment, or asking someone to refrain from contributing due to their inexperience, leads to them becoming intransigent or rude. In that case, and in the case of other people who prove to be beyond hope, applying the code of conduct and banning them for progressively long periods of time is the only practical solution.

It's worth noting that people do change. Sometimes, a ban is what it takes to make people realize that the moderators are serious about their requests. Sometimes, a problematic contributor will return after several years away and will have experienced personal growth and become productive (they may or may not ever acknowledge their past transgressions; it is frankly best to not bring them up). It is therefore reasonable to give people a second chance after their latest ban period has passed — but equally, it is important to be vigilant and prompt in banning people again (with a longer ban) when they show that they have *not* changed.

Some individuals exhibit behavior so egregious that an immediate lifetime ban is appropriate. The low probability of their becoming valuable contributors is not worth the message that would be sent to the rest of the community if they were ever to be allowed to return. Where to draw that line depends on the values of the team, but blatantly racist, sexist, or homophobic behavior, any sort of unwanted physical interaction at an in-person event, personal attacks, etc, are all obvious examples.

Entitled individuals who believe that because they use the project, they have a right to control what contributors work on

It is sadly the case that every open source project attracts users who think they are owed something for having decided to use the project. Such individuals are rarely redeemable and luckily make

---

[274] Or young at heart. Adults between jobs, or who are retired, may also have a lot of free time; if they are inexperienced in contributing to open source but are open to feedback, they may well become significant contributors given the right guidance.

themselves very obvious; they are easily dealt with by an immediate ban (acting entitled is not welcoming, so this is a simple code of conduct violation).

This is another case where being lenient tends to send the wrong message to the wider community. If people see that making demands in the issue database works to change the team's priorities, very soon the team will be overwhelmed by demands.

Individuals who feel slighted by the project and wish to use the bug database to vent their frustrations

SDKs like UI frameworks are often used by people for commercial purposes. They may therefore depend on the UI framework for their livelihood. Defects in the framework may lead to them losing customers and money.

If they did not pay for support (see the Funding section below), then they are in no way owed anything by the project, and it is not the team's responsibility to ensure each user's financial success. Unfortunately, when faced with difficulties, some people lose perspective and may find the product's bug database (or indeed any of the product's community spaces) an easy place to vent their frustrations.

This kind of behavior is extremely harmful to the wellbeing of the team, and should be immediately shut down as a code of conduct violation without any attempt to respond to the points raised.

Trolls

The Internet is sadly full of people who find amusement in causing other people grief.

This can take many forms, but luckily they are usually pretty obvious and there is no controversy or difficulty in banning trolls.

Spammers who see the project's popularity as a way to make money

Spammers also abuse open source projects. This takes many forms; in a bug database, the most common approach is straight forward link spamming.

Nobody whose opinion is important to the project will shed a tear if spammers are immediately and permanently banned.

Developers of automated systems that use the project's bug database as a playground for their creations

A curious artifact of having a large public open source project is that developers writing bots have a higher probability of (accidentally? intentionally? who can tell) using one's project as a testing playground. This can manifest as weird comments, issues being opened with minimal useful content, or other *non sequitur* behavior.

As with spammers and trolls, it is best to just ban the offending accounts promptly with minimal fanfare.

Content

Text

Detailed API documentation should be supplemented by higher-level documentation describing the overall architecture of the framework and introducing cross-cutting concepts, tutorials that take a developer along the journey of creating complete applications, discussions of topics such as state management, performance debugging, integrating with other build systems, and generally anything that does not fit neatly into the bucket of API documentation.

Videos

Some developers prefer text documentation, but others prefer documentation in video form. Short videos can be very effective for introducing a specific feature. For example, Flutter has "widget of the week" videos inlined into the API documentation for many widgets, which provides a compelling introduction to the widget.

Long-form video content is also useful. One form of video content that a lot of developers appreciate is watching other developers create applications, *including their struggles and how they overcome them*. This kind of content is cheap to create as it requires very little more than just putting experienced developers and developer relations engineers in a room with a camera (and screen capture software), and showing their work with very little editing.

Product announcements

It is undeniable that sudden marketing splashes, combined with coordinated press coverage, will drive persistent increases in usage. This requires occasionally having big events; in-person events tend to create more compelling content and also provide the opportunity to bring community leaders together and generally increase the goodwill from the community.

Such events should focus on already-released features rather than being forward-looking, because that maximises the extent to which the content can resonate with developers and be immediately actionable. ("Download our product now to try this amazing feature" is more compelling and results in more downloads than "Wait 6 months then download our product to try this hopefully-amazing feature".)

Social media

Maintaining a presence on social media can increase engagement and lead to conversions (this is marketing speak for "you'll get more users if you talk about your thing"). It is important to maintain a presence consistent with the project's values and voice, and in particular to avoid getting caught in controversy[275], while taking responsibility for the project's failings (as discussed earlier).

Which channels are most appropriate and effective changes every year, so no recommendations are made in this document.

---

[275] Undesired controversy, at least. A project's values may justify courting controversy. For example, a team could list social equity as a core value. Doing so and then not standing up for the rights of oppressed folks, especially those within the wider community of the project, would be a betrayal of those values. Doing so only when it is deemed convenient or safe will appear transparently and shamefully opportunistic and disingenuous, which has a high likelihood of backfiring (people have become very sensitive to artificial virtue signalling).

Team member blogs

It is best to avoid relying on blog posts from team members as a form of documentation. Any advice team members post on their blogs should be included in the official documentation as well. Blogs tend to be a very brittle form of content; they can disappear as team members move on to other projects, they are hard to find, and they don't fit well with the rest of a project's documentation.

Team member blogs are best used for revealing the human side of the project, showing behind the scenes of the community, talking about how the team approached a problem or discussing current events on the project. Information which will not be broadly useful after a few months, but which is interesting contemporaneously.

Funding

UI frameworks have been funded in various ways over the years.

Licensing

Developers can be charged for the privilege of using the UI framework at all. Usually, a free variant of the license for non-commercial use is provided; this has the benefit[276] of allowing potential customers to get locked in to the framework before committing to payment.

Support contracts

The framework can be made available for free, with the option to sign a contract offering the developer prompt technical support as needed.

Bundled

The framework can be created as a necessary part of some larger commercial operation, e.g. as part of an operating system, which provides the funding for the project.

First-party customer

The framework can be funded by a corporation's need for the framework, and offered to the wider community as an afterthought.

Second-order effects

The framework can be funded by second-order effects, e.g. increasing the use of some other commercial offering from the same vendor, such as an advertising framework or cloud services.

The risk with this funding model is that the vendor will lose interest in the project, or will demand tighter integrations, which may compromise the value of the project.

Crowdfunded

A UI framework could be an entirely open-source project funded purely by voluntary contributions.

It is unlikely that funds raised in this manner from individuals alone would be sufficient to fund an engineering team. Expanding the source of funding to corporations ("sponsors") would help;

---

[276] From the perspective of the UI framework vendor.

corporations using the framework could also pay employees to be full-time members of the engineering team.

Unfunded

Some frameworks are created as hobby projects, with all infrastructure being paid out of pocket by volunteers.

Politics

External politics

In general, ignoring the competition and focusing on solving the real problems of real users is the best plan, but it is important to eventually prepare for how other teams will react to success.

Teams like Apple and the Android team at Google will likely have opinions about a new framework if a significant fraction of applications on their platform use that framework, and may attempt to find ways to disadvantage such a framework[277] (e.g. introducing new OS features that are harder to support in a third-party framework than in the platform's native frameworks, using hidden APIs to advantage native frameworks, introducing platform policies that happen to be easier to comply with when using the vendor's tooling rather than the third-party framework's tooling).

Other frameworks (e.g. Flutter, React Native, Qt) may similarly feel threatened if a new framework begins to attract a significant number of developers, and while their leverage is less direct, they may also react (e.g. by more aggressively marketing specifically against the new framework[278]).

Success can also affect funding. Other vendors may try to use their leverage with the framework's patrons to cut off funding. If the framework's development is largely operated by a single private corporation, threatened vendors may attempt to influence that corporation's leaders, or may attempt to take control of that corporation in some way, so as to cut off funding directly. (The more distributed the development team, the less the framework is vulnerable to these kinds of shenanigans. This is one reason to aggressively pursue a truly multivendor development strategy.) In extreme cases, vendors may take direct action in the form of litigation, e.g. using patents, either against the framework vendor or against high-profile users of the framework (the appearance of risk can very effectively poison the well; witness, for example, how this very document recommends against using Java or Kotlin because of the perceived risk of legal action in that ecosystem as a result of a single event).

In many ways, of course, these are problems of success, and one can only hope to be so lucky as to need to be concerned by these issues.

---

[277] There's no evidence that this has happened in the past, but one sometimes wonders at some of the decisions made by platform vendors. For example, implementing a text field on iOS that implements all the features that the built-in text field has without literally using a built-in text field is impossible due to Apple's use of hidden APIs. LLVM has historically been changed in ways that have broken Flutter and required workarounds. iOS has changed in ways that prevent debugging of applications outside of Xcode. Apple does not document the protocols used by Xcode and iOS to enable development. Is all this incompetence or malice? Is indifference to allowing competitors equal footing itself anti-competitive, or merely negligent? These questions are thankfully outside the scope of this document, but they are nonetheless questions that may arise when one is successfully competing with a platform vendor.

[278] Platform vendors may find it more difficult to market directly against a third-party framework because of the optics (it may look like punching down, with the third-party framework in the position of underdog). Other frameworks, however, do not have this problem.

Internal politics

Once a team is sufficiently large to have subteams, there is a risk that teams will compete for limited resources (e.g. headcount), or will attempt to direct blame at each other. These kinds of insidious tendencies can grow very slowly and imperceptibly; it is ultimately the responsibility of the leads to rout out such cultural aberrations and find ways to foster cross-team collaborations and trust.

It is also possible for the existence of subteams to inflict a kind of tunnel vision on team members. For example, when an engineer working on a low-level component of the framework writes an error message, they may not have the full context for how that error message would be exposed to the developer. They may genuinely attempt to write a high-quality error message, but because of their lack of context, the result may be one that developers find baffling.

Team members should be encouraged to reach out to others to get context for their work (and just as importantly, team members should be encouraged to help those who come to them with such questions). Understanding the full context of one's work is critical to creating a coherent complete product. Leads should work to avoid the kind of internal politics that discourages this kind of cooperation and collaboration.

Pressures

It is common for team members working on large projects such as a UI framework to feel pressured. Project managers (PMs) may feel pressured to make announcements ahead of the code being ready. Engineers may feel pressured to write code fast to meet some arbitrary deadline[279].

Take the time to do it right

Never cut corners. Build it right the first time, even if it takes longer. This will save so much effort in the long run that it will dramatically dwarf any short-term benefits that cutting corners would have achieved.

This is one of the most "brave" choices one can make. The benefits of cutting corners are felt immediately. The benefits of doing it right may never be felt (because they are the absence of problems).

Don't do it at all if you can't do it right

Sometimes, a problem would require so much time to fix properly that the investment cannot be justified given the project's priorities. In that situation, making a temporary fix is a mistake. Temporary fixes reduce the pressure to fix the problem well; since the problem already had insufficient importance to get fixed properly, the temporary fix will make it even less likely that the

[279] They're all arbitrary; the universe doesn't care about UI framework release dates. They may feel real, and may feel imposed upon us, e.g. because funding is running out or because some important customer wants to make an announcement by a specific date or because some operating system vendor has an impending release that changes the UI conventions and wouldn't it be terrible if our UI framework wasn't ready in time, but the reality is none of these things actually matter. If the funding is so close to running out that attempting to speed up engineering would make a difference, then the source of funding is playing stupid games. If a client has overextended themselves and is depending on a feature that isn't available yet, then that is their problem, not the framework's. If an OS vendor has made some UI change that the team feels must be taken into account, a few extra weeks or months will not make much difference a year from now. All these deadlines are just arbitrary pressures we place on ourselves and having good boundaries around this is imperative for a healthy team and a good project.

team will *ever* fix the problem well. Each such temporary fix ends up eroding the project's overall quality. Temporary fixes are a prime example of taking on technical debt.

Furthermore, if a problem is not worth fixing well, then spending any time on it at all is time that is *by definition* time that should be spent on something else.

This again is an example of a "brave" choice. It is easy to see the benefit of addressing a customer need: the customer will be happy. It is easy to feel the *cost* of not solving it: they will be unhappy, and likely quite vocal about it! Unfortunately, the benefit of this strategy is subtle (keeping the quality up), and again is typically only felt as the absence of a problem. Not having a problem is rarely seen as the benefit of a strategic choice, it is more likely to be just taken for granted.

Ship early, ship often

Waiting until the framework is perfect will waste a lot of time during which the framework will have been good enough for some developers. Shipping early diffuses the issue of high expectations. Shipping *often* will reduce the need for every release to be a blockbuster. Having users refocuses the team on what matters.

Test

Always test everything. If something is untestable, redesign it to be testable. The time saved by having tests catch regressions before they land in the codebase is staggering and having the discipline to actually test everything is absolutely worth the cost.

Tests are sometimes viewed as an expensive luxury that takes time away from developing new features and fixing bugs. This may come from having tests that do not find bugs — the quality of tests definitely matters. If a developer is writing tests but they never fail, then the tests are insufficient[280]: the reality is that no developer writes perfect code, and it should almost always be possible to find a bug in new code by writing tests for that code. It may also come from thinking that other people writing tests will mean that one's code is covered even if one does not write tests; again, aside from the obviously selfish aspects of that attitude, this misses that the point of tests is to find bugs and therefore the tests must be written specifically for the code that they are testing.

Deflake

Never, ever, accept flaky tests.

This is something that *cannot* be deferred. As soon as a test flakes, that test *must* be fixed immediately. This is more important than any feature or other bug fix that is pending, except for a security incident. Flaky tests absolutely destroy the team's confidence in the testing infrastructure. A team that completely believes its tests will move much faster and with more confidence than a team that questions tests. Similarly with flaky infrastructure. This should be considered the highest possible priority.

Deferring dealing with flakiness does not work. All it does is hurt team morale and cost resources that could be spent doing something useful.

---

[280] A test that never fails for the entire lifetime of the project is effectively a waste of resources. By definition the test never found a bug; the time spent writing it could have been spent on something else, the CPU spent running it could have been spent running something else, the disk space spent storing it could have been spent storing something else.

With sufficient resources, the best way to deal with flaky tests is to run every test multiple times[281], and to consider any failure, even a failure on a test that nominally "passes" sometimes, to be a full test failure that blocks the code from landing[282], or, if detected after landing, causes the code to be reverted. (This is the opposite of the common pattern of running failing tests multiple times and considering them "acceptable" if they pass on at least one occasion[283].)

This may be the most "brave" choice of all.

Initial strategy

Target platform

It is tempting to create a cross-platform UI framework by porting a basic engine to 7+ platforms on day one. However, this way lies madness. Instead, a framework should be designed with the ability to be cross-platform in mind, but with only one platform as the initial target[284]. That platform should be the most difficult platform (probably iOS). The work should be done with the needs of a concrete first customer in mind, with the goal of shipping that customer's app to real users as the first milestone of the project.

This allows a steel thread to be drawn that proves that the framework can be used for its intended purpose. It allows immediate customer feedback to drive the project, keeping it grounded. It allows the team to clearly see what subteams will be required to create a real product. Depending on the funding strategy, it enables funding to grow from real customers as soon as possible, which can enable the project to reach self-sufficient funding as early as possible, removing a significant source of risk and pressures.

Team structure and funding

Creating a UI framework, programming language, and corresponding ecosystems is a massive endeavor. Securing funding for the long term is critical to ensuring success.

---

[281] For performance reasons it may be reasonable to scale how often tests are rerun by how often they have ever failed (even in precommit testing). A test that never, ever fails is not a useful test and it may make sense to stop running it at all, or at least run it some smaller fraction of the time. A test that fails regularly is an excellent test that is finding real problems and running it multiple times to ensure it never develops flakiness is extremely valuable.

[282] To be clear, the statement here is that if a test fails in precommit testing, whether because the test reliably fails, fails some small fraction of the time, or fails because of some sort of infrastructure problem with the testing harness, the code that first detected this failure should not land until the failure is resolved. How the issue is debugged depends on the kind of failure; this is out of scope for this document. The cause of such failures could be any number of things. It could be that the failure is actually already in the product, and never revealed itself before, and the proposed change in question was just unlucky in being the first to hit it. In that case, the tip-of-tree testing will eventually fail too, the tree will be closed to any new commits, and an investigation will need to be done, and work should stop until the issue is resolved — whether the test is a flaky failure or a continuous failure or a test harness failure. It could be that the test is triggering the infrastructure in some bad way that tickles a bug in the infrastructure that the team hasn't seen before. It could be that there's just a regular bug in the proposed code, or there's a race condition that isn't directly related to the code but the change tickles it in a way that causes tests to occasionally fail. In all of these cases, the proposed approach here is that the code should not land until the issue is resolved, because that is the only way to avoid ever having flaky tests, which are a morale destroying, productivity destroying, confidence destroying scourge.

[283] This is an extreme position that few, if any, teams follow in practice, but the author is confident that actually following this advice would pay for itself and lead to a better experience for everyone involved, as well as making a higher-quality product with a more productive team.

[284] Plus a second platform at a minimal proof of concept level just to ensure no one-platform assumptions seep in.

However, the team should start small. A single engineer and a usability researcher may be enough for several months[285]; the team should grow slowly from there, only adding people when there is concrete work to be done. It is easy to see the scope of such a project and begin by hiring a twenty person team, but in practice such a project is a research project for a long time, not a full engineering effort.

It is also important that the team lead(s) be experienced in the field, yet be humble enough to be ready to change their preconceptions based on the results of the usability research. While this document has attempted to document much knowledge about creating UI frameworks, it is no substitute for personal experience.

Security

A key component of the creation of any developer product (and indeed most software products of any kind) is the establishment of a vulnerability reporting pipeline. This could be combined with a vulnerability bounty program, though this is not critical.

There are industry best practices that should be followed for establishing vulnerability reporting pipelines; in the interests of avoiding spreading outdated information, this document will not provide specifics. In general, such a program should have a report ingest channel, e.g. an e-mail address. Reports should receive a response within a defined timeline, e.g. 24 hours, which typically will require a trained team with a defined process for handling an incoming report. There should be a policy for how to escalate the report to the relevant team members, and how to report a vulnerability to the public.

For a UI framework, there is no clean way to deploy an urgent security fix to *end users*, instead the application developers must update their framework then update their application then redeploy. For this reason, security by obscurity (keeping vulnerabilities secret) is largely moot; any bad actors could easily get themselves into any program intended to keep vulnerabilities secret while disseminating the information to application developers.

Noise

A security vulnerability strategy typically gives the required steps to follow for a real security vulnerability, but the reality is that the report ingest channel will receive mostly trash.

Most e-mails received will be untargeted spam. These may be filtered by automated spam filtering software, but there is a caveat: because most e-mails received to the security list will be spam, spam filters will learn that merely having the security bug report e-mail address in the recipient list is itself a high-signal sign that the e-mail is spam. This will cause legitimate e-mails to be marked as spam. This somewhat defeats automated spam filtering.

Of the e-mails received that are not untargeted spam, most will be targeted spam, for example asking if they can post on the team's blog (in an attempt to seed link spam for SEO purposes). A fraction of these will be actually security related; most of *those* will be automated e-mails, or

[285] This would be a novel way to construct a team, and as such is an example of a "brave" choice. A slightly less "brave" choice would be to start with two engineers who are well versed in the basics of usability testing, possibly with a part-time usability researcher who can advise them on early testing efforts, but having a usability researcher at the team's inception would ground the team's culture from the start and really set the tone for the entire project. Certainly there is plenty to test even from day 1, for an effort such as this.

e-mails from folks using automated vulnerability detectors, reporting non-issues and then demanding, with various levels of assertiveness, to receive a payment from the bug bounty program (whether or not the team has one).

This unfortunately means that one must remain vigilant at all times if one is not to miss a security vulnerability report.

Testing infrastructure

Cross-platform UI frameworks have testing needs that are a little more elaborate than most software. Code must run on each target platform, including mobile devices. Useful performance numbers can only be collected from actual devices in controlled conditions (including, e.g., climate control, as devices vary their performance based on their temperature).

While there are some third party services that nominally provide this kind of service (e.g. Firebase Test Lab), in practice these services are insufficiently reliable, to be used for a continuous integration pipeline.

This therefore requires the maintenance of a bespoke laboratory with dedicated hardware that can be invoked from the continuous integration system. Such a lab requires continuous supervision (devices are notoriously high-maintenance). Ideally such a system would have enough capacity to run tests for all commits as well as offering pre-commit testing[286].

It also requires supporting software. Specifically:

- There must be a queuing mechanism to queue the execution of tests on the physical hardware.
- There must be integration with the rest of the testing system.
- There must be a system that manages the hardware, rebooting devices regularly, running tests on the devices, etc.
- There should be a system that tracks benchmark systems over time and that can show graphs.

Unfortunately, to the author's knowledge, no such systems are available off-the-shelf.

[286] If pre-commit testing is offered, consideration must be made for how to mitigate the risk of hostile code being executed on the lab hardware. For example, it could require a code review to have already been provided. Otherwise, a hostile team member (or non-member contributor) could upload code that damages the lab or abuses the lab hardware for purposes like bitcoin mining.

Conclusion

The premise of this document was to answer the question "how should one build a UI framework", given no constraints except four goals.

As was discussed, creating a UI framework that minimizes power consumption is straightforward, and creating one that maximizes performance to the exclusion of everything else is similar. The goal of maximising UI effects is achieved by providing low-level access to the rendering layer, including enabling custom shaders. This is also relatively straightforward, but does not provide much guidance in terms of how to design the rest of the framework.

This leaves the most difficult goal, creating a "most popular" UI framework, which the bulk of this document discusses.

Based on the discussion herein, the author would recommend one of the following approaches, if the goal is purely being popular: either a Python-based framework, or a new framework heavily based on usability research.

Python-based framework

The first option is trying to create a Python-first UI framework that speaks directly to a low-level graphics backend; essentially, Flutter but using Python instead of Dart. While the final product is likely to be less performant than Flutter, and the development experience likely less pleasant, the benefit of providing a UI framework native to the Python ecosystem may outweigh these costs.

Cost

A Python-based UI framework would be a relatively inexpensive proposition. The language and ecosystem already exist, the community is large and enthusiastic, and there is demand (as demonstrated by the many attempts to provide Python bindings for non-Python frameworks). The project should probably start with a couple of engineers building the core, but eventually the team would need to grow:

- A few engineers for the core framework, and a few engineers for the backend. Much of this would be in Python, but the lowest levels would likely be in C++ (and in shaders).
- A few engineers for creating platform-neutral controls based on the framework.
- At least one dedicated engineer for each supported platform, plus at least one engineer for each platform-specific set of controls.
- A few engineers for the developer tools.
- A developer relations team.
- A community manager.
- A usability research team.
- Product management.
- Technical product management.
- A release engineer.
- A marketing specialist.
- A developer operations team.

- A team and budget to manage on-device continuous integration testing.
- Management.

Some of these roles could be combined, especially in the early years.

Risk

There is a risk, in that it may not be possible to create a sufficiently compelling framework in Python. Creating a proof of concept early would be important to reduce the exposure, as would working with real independent customers early in the process.

Potential

A cross-platform Python-first UI framework could revolutionize the Python community and empower an entire generation of developers to create applications that might otherwise never exist. While not necessarily a path that would lead to an objectively fantastic UI framework, it would nonetheless be valuable, and, in the author's opinion, is an achievable goal.

A new framework based on usability research

Alternatively, one could create a new language and corresponding UI framework with a team that prioritizes data-driven language and API design, relying on extensive and continuous usability studies combined with a lead designer who has a strong vision for the entire product but who is able to quickly pivot the design based on new information.

It is important to emphasize that the novel feature here is the heavy reliance on usability research to guide the design. This is what differentiates this potential language and framework from those that exist today[287]. There are lots of languages and frameworks, creating a pair that is more popular than the existing options requires an edge, which is what the usability research can offer.

Cost

This approach is *much* more expensive than the Python option.

Even creating a proof of concept would be expensive, due to the need to start from a blank slate and essentially act as a research project for months or years. It should start with an engineering lead and usability researcher, and slowly grow over time, as the lead deems it necessary.

Once a proof of concept is established, the initial productionization effort would need to be focused primarily on the new language:

- A lead engineer with expertise in language design, who is eager to work with usability testing to design a new language and framework.
- A lead engineer with expertise in UI framework design, who is eager to work with usability testing to design a new language and framework.
- A usability research team (lead researcher and several team members), with a budget for paying research participants, funding surveys, and funding travel to events with a broad range of developers.

---

[287] With very few exceptions, Flutter being one (as mentioned earlier), though even Flutter uses a language, Dart, that is largely *not* based on usability research.

- A few engineers for creating the core language tooling.
- A few compiler engineers per platform (starting with one platform, ideally iOS, with a second platform using a different architecture as a proof of concept; then, later, supporting both platforms fully).
- A small QA team to test the tooling and compilers.
- A part-time graphics engineer to assist in early framework design work.
- Product management.
- Technical product management.
- A release engineer.
- A developer operations engineer.
- A community manager.
- Management.

This effort will likely last *at least* another year before the first customer (of the language, not yet the framework, probably) can be approached, probably much longer.

At that point, the effort will likely need to expand to include work on the framework, and all the work that supports a shipping developer product, eventually easily reaching two hundred people:

- An overall lead engineer.
- A large usability research team with corresponding budget.
- A lead engineer for the language.
- A language team.
- A compiler team, with subteams for each supported platform.
- A standard library team.
- A team for bootstrapping the language's ecosystem.
- A lead engineer for the UI framework.
- A graphics backend team.
- A core framework team.
- A design language team.
- A framework team per supported platform.
- A team for bootstrapping the framework's ecosystem.
- A developer tools team that works across the language and the framework.
- A large QA team.
- A triage team.
- Product management.
- Technical product management.
- A release engineering team.
- A developer operations team.
- A team and budget to manage on-device continuous integration testing.
- A developer relations and community management team.
- Event staff for developer events.
- Marketing.
- Management.

An important question to ask is how the work would be funded. This effort requires a big team and non-trivial capital investment.

Risk

A new language might get dismissed by the developer community. A new UI framework might be ignored or ridiculed. The author could be wildly mistaken regarding the premise that usability is what would make a framework the most popular. The leads could be insufficiently careful about how they interpret usability research results, e.g. allowing their biases to blind them to what the research is telling them, or, equally, being too rash in their application of the results, leading to an incoherent overall experience.

In general, creating a new software library is high risk, creating a language is high risk, using a novel approach is high risk, and coupling all three is therefore an extremely risky endeavor.

Potential

While creating such a product is probably a fantasy, a properly-funded effort with a good team would, in the opinion of the author, have a very good shot at creating a sufficiently better developer experience that a large fraction of the developer community would come to recognize it as the *de facto* way to write applications across its supported platforms.

Final thoughts

Different frameworks are built with different goals. The web, now the world's most-used application platform, and the underpinning of much of the world's economy, was originally designed for sharing research results at CERN. Flutter was born out of an effort to improve Chrome's performance on mobile devices. React was originally a Facebook-internal framework for their applications. It is likely that none of the most popular frameworks were really designed with popularity as their primary goal. Most were designed out of a strong vision that had little to do with the needs of the global developer population.

These frameworks are nonetheless popular because they solved real problems that, as it turns out, other developers also have. There is a risk, in designing a framework specifically to be popular, that the lack of concrete focus and vision will mean the framework does not actually solve real problems. As has been discussed several times in this document, it is therefore imperative to ground the design in real projects, with real customers. It is important to be open to new ideas, to be ready to discard one's emotional attachment to prior inventions and to go where the data (and customers) are leading. The most difficult part of designing a product such as this is balancing having a strong coherent vision with this flexibility in the face of new information.

In writing this document, the author has attempted to put down in writing the wide variety of hard-learned lessons from years of work in the field. It is the author's most profound hope that this document helps someone avoid some of the pain of learning those lessons the hard way, so that they can learn new lessons, and further advance the state of the art. To whomever this may be: good luck.

Table of contents

Acknowledgements

This document was written with funding from Futurewei Technologies.

The author wishes to thank the expert reviewers who provided valuable technical feedback on this document, correcting many of the author's mistakes. Blame for any remaining mistakes lies entirely on the author. The author especially wishes to thank Adam Barth, who provided detailed notes on the architectural proposals and technical discussions in this document, Tao Dong, who provided expert feedback on the usability research aspects of this document, Michael Goderbauer and Kate Lovett, who pointed out several areas that needed more detail, Josh Levenberg and Vijay Menon, who provided careful review of the programming language section, Mike Smith, whose detailed feedback provided the author with a new perspective on many subjects in the document, Nic Barker and Matthew Kramer, who pointed out a number of omissions, and Mew Hewitt, whose review highlighted many parts of the document that were unclear.

Thanks also to Tor Arne Vestbø for catching some spelling and grammar issues.

Finally, thanks to the members of the skeptical Discord community, especially skeptick, gwillen, ohAitch, and TheSkeward, who provided equal parts sanity checking and motivating snark.