Jank in Flutter

March 2021

Compiled by Ian Hickson.

This deck is a status update that was presented to the Flutter leadership team. It is an internal artifact of the team's internal processes, and not intended as public communication. Since Flutter is an open source project, it is nonetheless publicly viewable, and you are welcome to read it to see how we are internally thinking about the topic. However, bear in mind that it may use terminology or conventions that differ from those used in official public documentation.

Project structures and incentives

Background

Good performance has long been a stated priority of the Flutter project.

Currently there is very little structure around performance issues: we treat them more or less like exceptions. You can get exceptions anywhere, the responsible team is the one whose code throws the exception. You can get jank anywhere, the responsible team is the one whose code janks.

In 2019 and 2020 we had a (one-person) "performance team". This was really an engineer on the engine team focused on performance, largely working on metrics collection and shader warm-up mitigations.

Background

Regressions in areas we have thought to monitor are caught in our daily benchmark audit (currently performed by Ray Rischpater), which motivates the team to maintain a performance status quo. This is similar to our use of continuous integration to catch functional regressions.

For Googlers specifically, large scale performance improvements can help with career development. Beyond this, we do not have strong processes in place to motivate contributors to improve performance specifically, any more than we incentivize bug fixing.

Background

We currently have five labels in GitHub for performance issues. The first is a general catch-all label:

- **severe: performance** applies to all performance issues

The others apply to subsets of performance issues with specific characteristics:

- **perf: speed** applies to jank issues
- **perf: memory** applies to memory usage issues
- **perf: energy** applies to battery usage issues
- **perf: app size** applies to binary size issues

Proposal

Since these efforts cut across multiple projects (Skia, Dart, Flutter) and are largely tactical in scope (many individual efforts focused on specific problems with no overarching theme beyond "performance"), a single point of contact (the "performance constable") that regularly collects and propagates status updates from the many relevant engineers to the various team leads is probably the most effective structure.

This could take the form of a TPM project or could be coupled to our existing critical issue triage process (e.g. by marking jank bugs P2 or auditing all bugs with a particular label each week).

Proposal

Identifying key user journeys (e.g. the GPay onboarding flow) that show particularly notable performance issues today, and having the "performance constable" periodically demonstrating progress on these specific interactions to leadership, may help focus efforts.

Legend

The following terms are used when describing timelines:

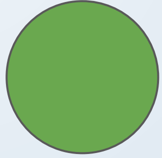
Weeks: At least a couple of weeks, but probably less than three months.

Months: At least three months, but probably less than a year.

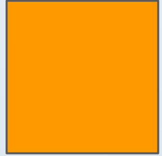
Quarters: At least six months, but probably less than a year.

Years: Over a year, probably several years.

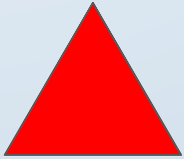
These estimates are highly speculative.



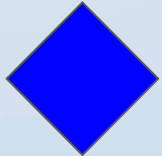
Making good progress. Sufficiently staffed.



Success is not imminent. May be incompletely staffed.



Nothing or very little is happening. Unstaffed or very low priority.



Information is lacking.

Early-onset jank

Background

We first noticed early-onset jank in 2015.

Over the years we have made numerous improvements to address this issue, the earliest and most prominent was moving from JIT compilation of Dart code to AOT compilation.

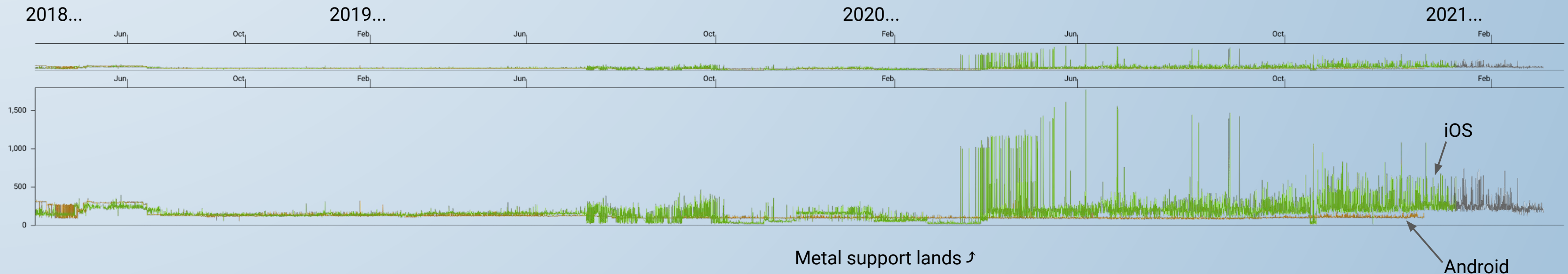
Background

Unfortunately our focus has primarily been on sustained performance, not early-onset jank, and therefore we have historically been willing (often unconsciously) to sacrifice start-of-app performance for sustained performance.

The most notable example of this was our migration to Metal, which was driven by Apple's deprecation of OpenGL. While we believe Metal has better sustained performance characteristics, it has worsened our early-onset jank situation on iOS quite significantly.

Background

We migrated to Metal on iOS on March 31st 2020. This caused a noticeable regression in our "worst rasterizer frame time" performance, as shown below. However, since we were focused on sustained performance and have a culture of ignoring worst-frame times, we didn't understand the significance at the time.



Gallery transition worst rasterizer frame times (ms); >16ms indicates jank

Background

We have also recently observed an increase in reports about early-onset jank from our developers.

To obtain specific real-world scenarios with which to evaluate solutions, we encourage people to file issues.

Naturally, only a subset of users actually file bugs. There is therefore a risk that these may not be representative, and that reducing them to test cases may lose relevant context that affects real-world performance.

Background

In recent history we've received about five actionable reports of actual reproducible jank. Shader compilation jank is the primary problem seen in these issues. It is especially notable on iOS with Metal.

Our focus on sustained performance has led us to create more and more specialized shaders, which need to be compiled on first use, and which contributes to this issue.

For example, some animations require shaders to be compiled for each frame, and it can take multiple iterations to find and compile all the shaders used.

Background

Q. Why can't you compile all the shaders ahead of time?
A. There are an intractable number of shaders due to the level of specialization we currently apply.

Q. Why can't you enumerate the shaders an application will need ahead of time?
A. It may depend on many factors, such as screen size, user preferences (font size, color schemes), the nature of dynamic or user-generated content shown by the app, etc.

Q. Metal on iOS made this worse. Does Metal on macOS have the same issues?
A. Yes, and we believe the same fixes will resolve them.

Overview of efforts addressing early-onset jank

- Shader warm-up on Metal
- Static shader set
- Optimising specific shaders
- More general path shader
- Tests for Skia
- Thread prioritisation
- Task prioritisation
- Diagnostics
- Tooling integration for precaching shaders
- Documentation

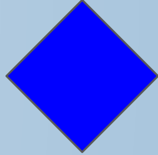
Shader warm-up on Metal

When we migrated to Metal we lost shader warm-up on iOS.

Risks: Implementing shader warm-up for iOS to the level we see in OpenGL may be technically challenging.

Timeline: Weeks to months.

Status: Work by Jim Van Verth (Skia) complete. Work from Flutter team to integrate with new Skia APIs is not yet started. Will probably need follow-up work from Skia team to move warm-up to asynchronous operation.



Static shader set

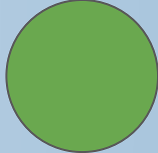
Since the root cause is shader compilation of specialized shaders, one solution would be to use a finite set of much more general shaders.

Risks: May be impractical (can this integrate with Skia? how many shaders are needed to cover everything we support?). General shaders may be insufficiently specialized to achieve full performance. Static shader set may be too big to compile on startup, defeating the purpose of the effort.

Timeline: Investigation results in a few months, deployment in a few quarters.

Status: Chinmay Garde working on design doc.

Optimising specific shaders

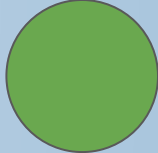


Studying specific scenarios has flagged some shaders that are especially troublesome due to being over-specialised. By generalizing these we may improve matters, potentially with little downside.

Risks: Optimizing for particular animations may make matters worse for other animations. Generalized shaders may require more battery.

Timeline: Weeks.

Status: Brian Saloman (Skia) is starting work on the reduced geometry mode variation mode now.



More general path shader

By having fewer shaders we reduce the number of times we stall for compiling shaders. Paths specifically have many specialized shaders today.

Risks: This is a trade-off between lots of simple shaders (lots of compilation jank) versus fewer more complex ones (potential for bugs). Also some complications around anti-aliasing.

Timeline: Quarters. First production results expected Q3 2021.

Status: Chris Dalton (Skia) has implemented some new algorithms. Testing and enabling the shaders remains to be done.

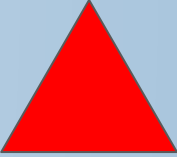
Tests for Skia

Having the Skia team be able to reproduce these issues in their own harnesses and development environment would help the Skia team debug and address the parts of those issues that are in their wheelhouse.

Risks: Having specific tests as targets risks over-optimising for unrepresentative cases. Dependency on another team to address a critical issue increases uncertainty on timelines.

Timeline: Tests available early Q2 2021. No commitment for improvements.

Status: Discussions ongoing to determine optimal form for tests.



Thread prioritisation

There are a number of situations where we could tweak thread priorities; for example, Apple has suggested that insufficient priority may be the reason for Metal shader compilation hangs.

Risks: May not have any meaningful effect.

Timeline: Not currently scheduled. Weeks to months once staffed.

Status: Some early investigations happened in each case, but no current activity.

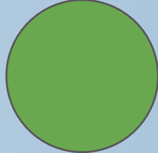
Task prioritisation

There are a number of situations where we could tweak *task* priorities; for example, processing rendering-related tasks ahead of other background tasks.

Risks: May not have any meaningful effect.

Timeline: Not currently scheduled. Weeks to months once staffed.

Status: Chinmay Garde has written a design doc; no active work ongoing.



Diagnostics

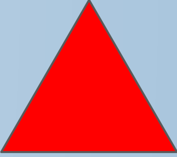
A number of ideas are being considered to improve the developer experience for dealing with jank, for example exposing shader compilation more obviously.

Risks: Minimal risk.

Timeline: Weeks.

Status: Brian Osman (Skia) has been doing work on identifying shaders as they are compiled; this still needs Flutter-side work to integrate with the new APIs. DevTools work is relatively small and would start after Flutter integration is complete.

Tooling integration for precaching shaders



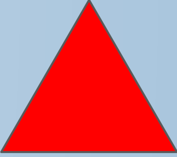
The Flutter tool and IDEs could expose features to enumerate the shaders that need to be warmed up.

Risks: IDE integration could be confusing to users who would otherwise not be aware of this feature at all. May not help with iOS if iOS shader warm-up does not fully warm-up the shaders.

Timeline: Not currently scheduled. Weeks once sufficiently staffed.

Status: IDE work currently in the discussion stage. Probably needs a PM. Multiplatform test-based shader warm-up work currently unstaffed.

Documentation



We would like to turn recent lessons learnt from GPay and gSkinner collaborations into public documentation (videos, web pages, etc).

Risks: Performance debugging is a specialized skill, teaching it is non-trivial.

Timeline: Not currently scheduled. Weeks once sufficiently staffed.

Status: Filip Hracek has some ideas but no time to work on them. Being considered for Q2 2021.

...

Risks:

Timeline: Not currently scheduled.

Status:

Other performance issues

Background

We continue to make performance improvements in general. For example, Kaushik Iska switched macOS to Metal, which is expected to improve sustained performance.

Performance is an area where the work never ends, however.

Overview of topics

- Web scrolling
- Reducing painting
- Garbage collection
- Tooling improvements for Flutter for web
- Other tooling issues
- Faster SVGs
- Faster vector graphics format
- Framework performance issues
- Reducing rendering latency
- Other engine performance issues

Web scrolling

A very noticeable problem with the web target is the performance of scrolling large regions, a common operation on the web. This problem is will hurt adoption of Flutter for web since it presents performance cliffs.

Risks: Web issues may require updates to browsers (e.g. weak references on Safari), which implies a very long development cycle (updating standards, implementing those standards in browsers, deployment of the browsers).

Timeline: Unknown. Could be months to years.

Status: This is the web team's primary focus.

Reducing painting

We could reduce painting by only recomputing pixels in regions of the rasterized image that have changed from scene to scene.

Risks: It may be more expensive to determine the regions that need repainting than just repainting the scene as we do now.

Timeline: Months once the effort is started; this work is not currently scheduled.

Status: Some early exploration was performed Jim Graham last year. Matej Knopp may look at this after current desktop work.

Garbage collection

There is a slow trickle of reports of issues involving garbage collection, e.g. that GC events are poorly timed, or that concurrent GC is descheduling time-critical work. Some reports may be the result of poor diagnostics, some may be real.

Risks: Not applicable.

Timeline: Not currently scheduled. Weeks to years depending on issue.

Status: No recent efforts have looked at garbage collection timing.

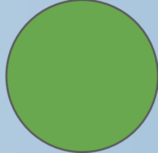
Tooling improvements for Flutter for web

Examination of performance issues with Flutter applications deployed to the web has discovered some specific problems which tooling could help with, e.g. not using lazy building for scroll views, repainting too much, laying out too much, etc.

Risks: Minimal risk.

Timeline: Planning expected in late March 2021, with fixes starting in Q2 2021.

Status: Project is in the planning stages. Has UX and PM involvement.



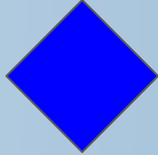
Other tooling issues

Other ideas are also being considered to improve the developer experience for dealing with performance issues beyond jank, for example making the Dart DevTools show performance live rather than requiring manual refresh.

Risks: Minimal risk.

Timeline: Weeks to months once sufficiently staffed.

Status: UX studies determined a number of areas that would benefit from improvements, work planned to start Q2 2021. May need PM support. Alibaba are currently contributing some tooling improvements for exposing SVG render times.



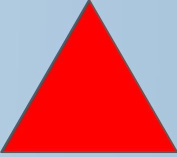
Faster vector graphics

SVGs often are used in preference to PNGs due to binary size concerns. Applications using SVGs in Flutter (e.g. GPay) are experiencing high rasterization thread times. Transparency and gradients in particular are proving very expensive. Skia is investigating improving performance for these SVGs.

Risks: Optimising for specific SVGs may negatively impact other scenarios.

Timeline: Months.

Status: No current work is focused on rendering SVGs in particular.



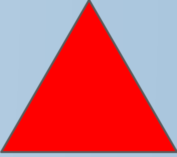
Faster vector graphics format

Vector graphics are commonly used but current solutions are expensive to parse, render, and animate. We could develop a new vector graphics format optimized for first-frame rendering speed and efficient animations.

Risks: Getting adoption of a new standard is not guaranteed. A new standard may not solve the underlying performance issues.

Timeline: Years.

Status: Collecting requirements.



Framework performance issues

There are many performance improvements we can make in the framework, starting with making our benchmarks more representative of real workloads.

Risks: The opportunity cost of some of these may be high (i.e. the impact of some of these improvements may be less than other non-performance-related work we could work on instead).

Timeline: Not currently scheduled. Solving all known issues would take years.

Status: Some bugs get resolved as part of larger framework planning, but no specific performance-focused effort is underway.

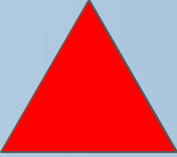
Reducing rendering latency

If we have multiple frames in flight, we currently render them in sequence, even if we could skip one and jump straight to the most recent frame. By skipping intermediate frames we could improve performance.

Risks: Generating more frames could harm CPU performance and battery usage.

Timeline: Started in March 2020; ongoing.

Status: Kaushik Iska has been working on this. The most recent attempt to land this feature had performance and tooling regressions. Work continues.



Other engine performance issues

There are some issues filed regarding potential engine performance improvements (other than those already listed in this deck).

Risks: The opportunity cost of some of these may be high (i.e. the impact of some of these improvements may be less than other non-performance-related work we could work on instead).

Timeline: Not currently scheduled. Solving all known issues would take years.

Status: Some bugs get resolved as part of larger engine planning, but no specific performance-focused effort is underway.

...

Risks:

Timeline: Not currently scheduled.

Status:

Memory

Background

Memory usage can directly contribute to jank because it reduces the space available for caching and increases the time taken by GC.

Overview of topics

- Compressed pointers
- Tooling
- Other Dart memory issues
- Flutter memory issues

Since most programs are happy to operate within a 4GB address space and don't need the full 18EB address space that 64 bits provide, Dart could store pointers using 32 bits and expand them to 64 bits on the fly. This is estimated to reduce heap memory usage by 20-30%.

Risks: Additional complexity in the VM, additional complexity exposed to developers (since they need to decide which mode to use), runtime performance.

Timeline: Should be complete in 2021.

Status: Work is ongoing.

Tooling for memory diagnostics

A number of ideas are being considered to improve the developer experience for dealing with memory issues, for example clearly describing where memory is being used.

Risks: Minimal risk. Investigations using a prototype suggests a lot of apps are leaking, so there will be follow-up work needed to resolve issues.

Timeline: Leak detection expected to be productionized in Q2 2021.

Status: Some UX research uncovered particularly important pain points. Terry Lucas working on this. May need PM support.

Other Dart memory issues

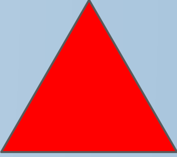
There are a few ideas for memory improvements currently being considered.

Risks: Improving memory usage often involves trade-offs (e.g. memory vs time).

Timeline: Resolving known issues will take months to years.

Status: Previously-mentioned projects are ongoing, and will give way to further issues as they get resolved.

Flutter memory issues



There are many areas where we believe that memory usage could be improved.

Risks: The opportunity cost of some of these may be high (i.e. the impact of some of these improvements may be less than other work we could work on instead).

Timeline: Not currently scheduled. Known issues would take months to years.

Status: Dan Field has done substantial work in this area but that work is currently on hold.

...

Risks:

Timeline: Not currently scheduled.

Status:

Binary size

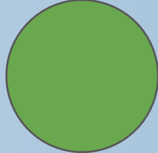
Background

Binary size contributes to load times and to memory usage (which itself contributes to jank as described in the previous section).

We have made significant gains in binary size for Flutter applications over the years, but it remains an area of concern. The smallest Android application we can build is still about 4.5MB, the smallest iOS application is about 9.1MB. These numbers also correspond roughly to the overhead of adding Flutter to an existing application, which makes Flutter a more difficult decision than we would like.

Overview of topics

- Deferred loading
- Other known binary size issues



Deferred loading

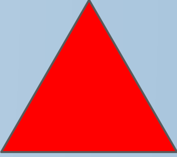
Android supports a model where applications sections are downloaded on demand ("Split APK"), reducing binary size and runtime memory usage when a section of the application is not activated by the user.

Risks: Current offering is Android-only and requires some manual configuration, which may put it out of reach for some developers.

Timeline: Should be available in the next beta.

Status: Product is ready; Gary Qian is currently writing documentation.

Other known binary size issues



While the low-hanging fruit is largely picked, there are still some areas where binary size may be improved.

Risks: There are often trade-offs involved, e.g. trading precomputed data (which costs bytes) for runtime computation (which costs startup time).

Timeline: Not currently scheduled. Known issues would take months to years.

Status: No recent efforts have looked at binary size other than deferred loading.

...

Risks:

Timeline: Not currently scheduled.

Status: