Flutter

Preemption for 60 FPS

SUMMARY

Use preemption to achieve 60 FPS, no matter how heavy the tree is to build/layout.

Author: Jingyi Chen (fzyzcjy)
Go Link: flutter.dev/go/sixty-fps (still in PR)
Created: 09/22 / **Last updated:** 09/22

WHAT PROBLEM IS THIS SOLVING?

Many people face a jank for Flutter applications, because the build and layout of the tree is so heavy to be done within 1/60 second. Thus, I hope to make it 60 FPS even on low end devices, no matter how heavy the tree is to build/layout.

Preface: Thank you, Flutter! (And why I do this)

Flutter has saved me months, if not a year, by allowing me to write a single codebase and run on dual platforms (saving half of the time), to use the very productive declarative framework and hot reload, to utilize the quickly-progressing Dart language with expressive and safe type system, to easily customize the appearance as accurate as pixel level, to be able to dig down and modify engine code when I need a new feature (impossible with Web), and many more.

That is a big reason why I decided to contribute this PR (Preemption for 60 FPS) to Flutter. Flutter has saved me so much time, so now it is my turn to provide my time to Flutter to make it better. I also love open source very much, such as my flutter_rust_bridge open-source library (which could have been closed-source and only used by myself), and my previous PRs to Flutter fixing bugs, which is another driving force.

Scenarios

Here are some scenarios where build and layout janks:

- **A test case in the framework**, pointed out by @dnfield: He said, this ends up being janky because layout gets expensive for all that text (on a lower end phone it can easily take 20-30+ms just to layout all the text there, and the ListTile is a little deceptive because Material introduces expense - this is the kind of thing we want to figure out how to break up "automatically").
- **Bytedance's apps**: Infra team reports that they see such janks (around here). In addition, the infra team seems to have interest in my issue and chatted a lot discussing it - this surely will not happen if it were not a problem for bytedance.
- **All apps using keframe**: Keframe is a library to optimize such janks (with comparison below), with 94% popularity on pub.dev and 741 stars on GitHub. If there is no such jank, I guess the package will have no users.
- **Really slow devices**: There exist many slow, slower, and slower-than-slower devices in the world. Without this proposal, a Flutter app must be janky when running on devices beneath a certain computation power threshold; with the proposal, they will still be smooth, or at least such threshold is lowered by a magnitude.
- **Locality**: @gaaclarke said that the thing that might be holding back layout / build is locality which will be hard to fix, with more guesses here. (The guess will be fixed by this proposal.)
- **Heavy sync computation**: A little portion of real-world users may have to do heavy sync computation inside initState within the main isolate, because the data is not transferable or too big to send to a second isolate. (This exists, but is not the main target case that the proposal is going to solve.)
- **Implicit scenarios**: If I understand correctly, googlers (@Hixie, @dnfield, @JonahWilliams) and bytedancers (@JsouLiang, @Nayuta) have had some discussions around solving this problem, so the problem just exists. (See here for the chat history links)

Comparison

(Suggested @JonahWilliams for a comparison.)

There are some other methods (abbreviated as “OM” in below) related to smoothness optimizations, which can be roughly separated into two categories:

1. Modify the build phase, including the following (abbreviation: OM-B)
    a. The `keframe` package
    b. My (failed) experiments (abbreviation: OM-B-M)
2. Modify the layout phase, including the following (abbreviation: OM-L)
    a. Googlers (@Hixie, @dnfield and other googlers) had some discussions about it
    b. Several bytedance infra team people also had some discussions
    c. My (failed) experiments (abbreviation: OM-L-M)

Indeed, I have made many failed experiments and failed proposals before reaching this design you are reading :)

In the following subsections, problems of those approaches will be discussed. Those problems are overcome in the proposed method.

Unnecessary re-layout

OM-B and OM-L methods have the extra cost of re-layouting subtrees in each frame.

For a simple example, suppose we have a Column/ListView with five children. In frame #1, it renders the first, and children 2-5 return empty boxes. In frame #2, the Column has to perform a full layout (i.e. call `performLayout()` fully), so is frame #3, #4 and #5. Different OM-B and OM-L methods may vary about how many widgets are rendered in each frame, but the relayout overhead is still there.

If it were just a Column we may accept the overhead, but it can be a big ancestor tree. We have to re-layout over and over again in every frame, for the whole ancestor tree, up to the nearest relayout boundary. The consequences of such overhead will be discussed further below. The proposed method does not have the problem.

Unnecessary re-paint

OM-B and OM-L methods have the extra cost of unnecessary re-paint in each frame.

Suppose a heavy rendering needs 0.1s, then OM-B and OM-L methods will run the full paint phase for 6 times, while the proposed method only needs one paint call. This is especially troublesome when painting is slow, and still not very great even if each paint only takes 2ms - it adds up and occupies precious time of useful work. The proposed method does not have the problem.

Unnecessary whole-pipeline re-execution

OM-B and OM-L methods need to re-execute the whole pipeline while the proposed method does not.

For example, when Keframe replaces placeholders with real widgets, or when other OM-B and OM-L methods run build/layout on a few widgets, it is driven by the vsync signal to execute the drawFrame and submit to the engine, so it will execute the complete build/layout/paint etc process. However, the build/layout/paint other than the actual widget is not necessary.

On the contrary, in the proposed method, the UI thread just voluntarily submits a frame to the Engine after roughly 16ms of detection, and then returns to the normal rendering flow without much additional overhead.

(Suggested by @Nayuta)

Unnecessary CPU idle even when pending work

OM-B and OM-L methods will make CPU idle, even though there is a ton of work to be done, thus making more unnecessary perceptual lagging. The idle period for UI thread is after current pipeline ending and before next vsync.

Moreover, it is hard to remove such idle periods. If we halt too early (say, current frame ends at 12ms), then we waste 16.67-12=4.67ms; if we halt too lately (say, current frame ends at 19ms), then we even waste more - 16.67x2-19=14.3ms, because we are idle until the next vsync. As is discussed in other subsections, it is hard to know when to halt the existing build/layout can make the current frame end at 16ms.

In my OM-L-M experiment (can see a timeline figure there), about 39% of the UI thread time is idle, though there is still build/layout work to do. This may be tunable to be less harmful with careful choice of parameters, but by nature it cannot be fully removed.

On the contrary, the proposed method has exactly 0% idle time while work is not finished, without any need of tuning parameters.

Unnecessary FPS drop: 30FPS when could be 59FPS

OM-B and OM-L methods will immediately drop to 30FPS even if the frame is only 0.01ms longer than 16.67ms, while the proposed method will be 59FPS.

The “a little bit longer than 16.67ms” situation is inevitable because of two reasons: On one hand, as described above, when we decide to suspend/halt, we still have an unpredictable non-negligible amount of work remaining to do within the current frame. On the other hand, there may not be enough positions to halt, such as when a single widget layout can take several milliseconds. Thus, we will either halt too early (cause problems pointed out above) or too late (the drop-to-30FPS problem).

The analysis of 30FPS is as follows. To simplify math, suppose each frame needs 16.67+0.01ms and continue for one second. Then, those approaches will miss half of the vsync, i.e. will only get vsync per 33.33ms. Therefore, they will simply run the pipeline per 33.33ms, which is 30FPS. On the contrary, the proposed method will call `window.render` to submit a frame to the rasterizer per 16.67+0.01ms, regardless whether it misses a vsync or not, so we will see roughly 59 frames on the screen in one second.

Remark: The “average FPS” in DevTools seems to be wrong for such cases.

Remark: Given this discussion, when reading something like “17ms” in the “*_frame_build_time_millis” in benchmark results, it indeed means a completely different end-user feeling (30FPS vs 59FPS). In the “When to call preemptRender” section later, there are also some discussions.

Unnecessary perceptual slowness

OM-B and OM-L methods take more frames to render all elements in the whole UI than it could have been.

This is a direct consequence of the problems above, since the system has less time to deal with the real heavy subtree needing build/layout.

In my OM-L-M experiment, if we want each frame to be under 16ms, it seems the overhead is so big that we will only have <10ms for handling the real widgets (rough numbers), and I even fail to make it be under 16ms in total for some devices. On the contrary, the proposed method has much less overhead (0.53ms per frame, to be discussed in the experiment section).

Costly suspending

This point is related to some points above, using another perspective to view it. In some of the OM-B and OM-L methods, the layout phase will be suspended via various approaches, such as early-returning the layout function (OM-L-M), build a placeholder widget (OM-B), etc. However, it seems that all has overhead in terms of memory and CPU. On the contrary, the proposed approach does not have overhead when suspending.

Coarse suspending points

OM-B and OM-L methods can only act per Widget/RenderObject, so if one Widget/RO is too slow to build/layout, it still janks. A real case may be a text widget containing long content.

On the contrary, the proposed method can pause in the middle of any arbitrary function, as long as the function is called during the build/layout phase. This is because the `maybePreemptRender` call can be inserted anywhere you like.

Problems specific to a subset of methods

The following drawbacks mainly apply to a subset of the OM methods.

- Some of the OM-B methods add exactly one widget in one frame, no matter how fast or slow it is. As we know, the build/layout time for one widget varies greatly on different devices. For example, on low-end devices it may still be too much to add one widget in one frame, then we still get jank. On high-end devices, it may be OK to add five widgets in one frame, then we are rendering the UI using 5 frames even if we could have done it within 1 frame. The proposed method does not have the problem.
- Some of the OM-B methods always have one frame lag. For example, if you provide a child in frame #10, it will never be visible until frame #11 ends, no longer how high-end and how spare the device is. In addition, as @DanField points out, “this will cause problems with scrolling/touch events”. The proposed method does not have the problem.
- For those “build/layout as many as possible, until it is near timeout” OM-B methods, like in OM-B-M, there are also problems: When we see it is nearly timeout and do not provide further build/layout, we will still have to do a lot. We have to finish the build/layout of the remaining non-managed widgets, and we have to paint the whole tree, finalize (dispose widgets) the whole tree, etc. All of them take UI thread time, and takes a lot in scroll-ListView case in my experiment. Then, even if we halt at, say, 12ms, we may still miss

the 16.6ms deadline, and it is not 60fps now. On the contrary, the proposed method will not have so much overhead, but only do a little job (send existing layer tree to raster thread), which is much more predictable in terms of time, so we have less risk of missing 16.6ms.

- In some of the OM-L methods, users of Flutter framework may need to modify their code because implicit assumptions such as "build/didUpdateWidget happens on each frame" has been broken. More details: When suspended, those approaches will have some subtree of RenderObject whose performLayout has not been called in this frame, i.e. still dirty, even if a frame ends. Given the existence of LayoutBuilder, we will also have some Widget.build not called within that frame. This requires each and every widgets and RenderObjects to update their code to allow such behavior, which will not only be a lot of work inside Flutter framework widgets, but also a lot for all package and app developers. The proposed method does not have the problem.
- In OM-L-M, those approaches paint nothing (i.e. do not call child.paint) if a Suspendable is suspended. This will destroy the layer tree and C++ engine layer trees, making performance much worse. This may be overcomed but I have not experimented. The proposed method does not have the problem.
- In OM-L-M, if a child under Suspendable marks itself as needed to relayout/rebuild, and there is a relayout boundary between that child and Suspendable, then the suspending mechanism will not work at all. The proposed method does not have the problem.

BACKGROUND

Audience

Flutter contributors who want to improve Flutter's performance.

Glossary

N/A

OVERVIEW

Non-goals

Jank caused by raster threads (probably addressed by Impeller etc).

EXPERIMENTS (THE PROTOTYPE)

In this section, I will show the experiments (prototype) I have done with quantitative and qualitative results.

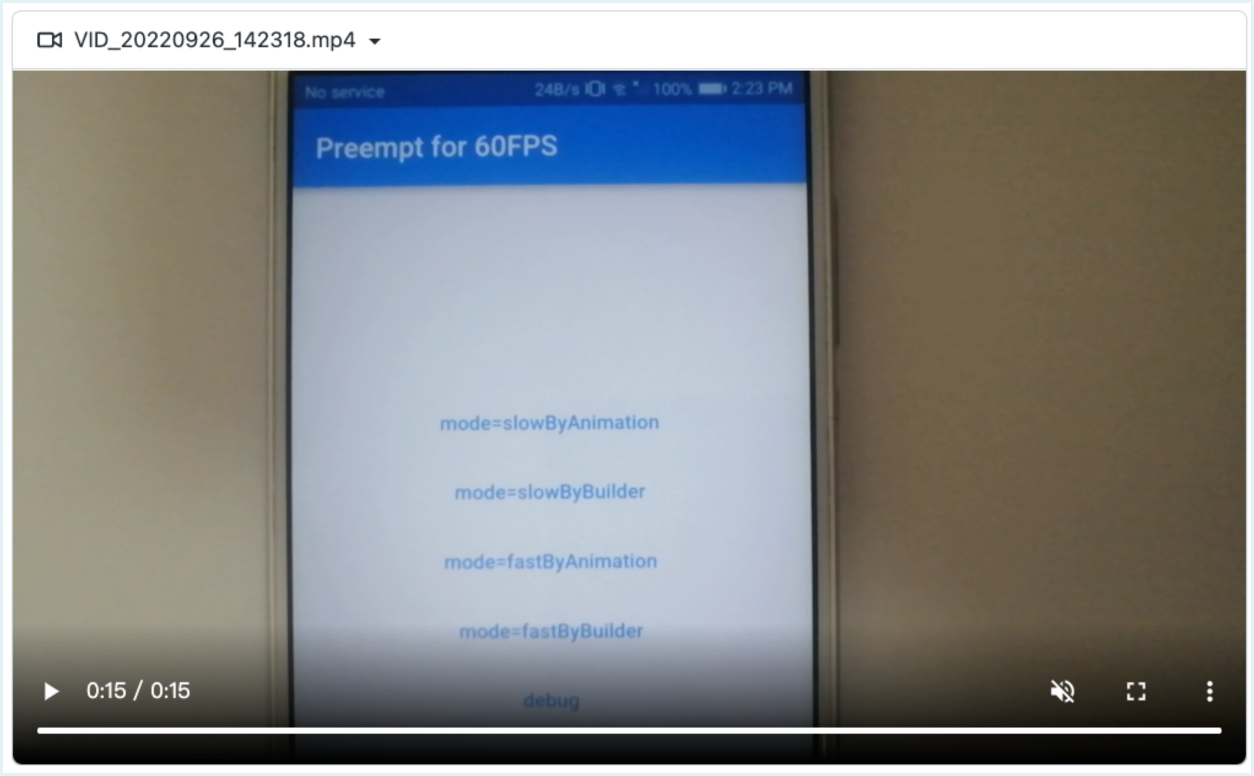
Scenario: Entering a new page with heavy subtrees

As is suggested by @dnfield, I made this prototype by reproducing a very common case: Enter a new page, with slide animations, and that page contains subtrees which are heavy to build/layout.

To demonstrate more clearly, the subtree is set to be so heavy that it needs 500ms to show on my test (low-end) Android phone. Surely, without the proposed method, the page transition animation does not animate at all.

Video

Video Link: Tap here, or tap here.



The video contains:

1. Case without optimization, repeated twice, showing jank.
2. Case with the proposed method, repeated twice, showing smoothness.
3. Content for sanity check (a jumping number animation; if you see it janky then the video/display may have problem)

The video is recorded by a second phone camera at 60FPS, in order to better mimic the human eye perception.

User-facing API

Well, just the standard things, except for the PreemptBuilder, which looks like AnimatedBuilder.

```
// the new thing: PreemptBuilder
PreemptBuilder(
  builder: (context, child) => _MyAnimation(child: child),
  child: TheVeryComplexPage(),
);

// just the very standard things…
class _MyAnimation { … }
class _MyAnimationState extends State<_MyAnimation> with
SingleTickerProviderStateMixin {
  var controller = AnimationController(...);
  var offsetAnimation = Tween(begin: const Offset(1, 0), end: const Offset(0,
0)).animate(controller);
  void initState() => controller.forward();
  Widget build(BuildContext context) => SlideTransition(position: offsetAnimation,
child: widget.child);
}
```

Full code

The latest code is in https://github.com/fzyzcjy/engine/tree/experiment-smooth and https://github.com/fzyzcjy/flutter/tree/experiment-forest.

FPS analysis: From ~2FPS to ~60FPS

As can be seen, without optimization, it takes more than half a second to show the new page, so it is <2FPS. With optimization, it is near 60FPS by the following analysis.

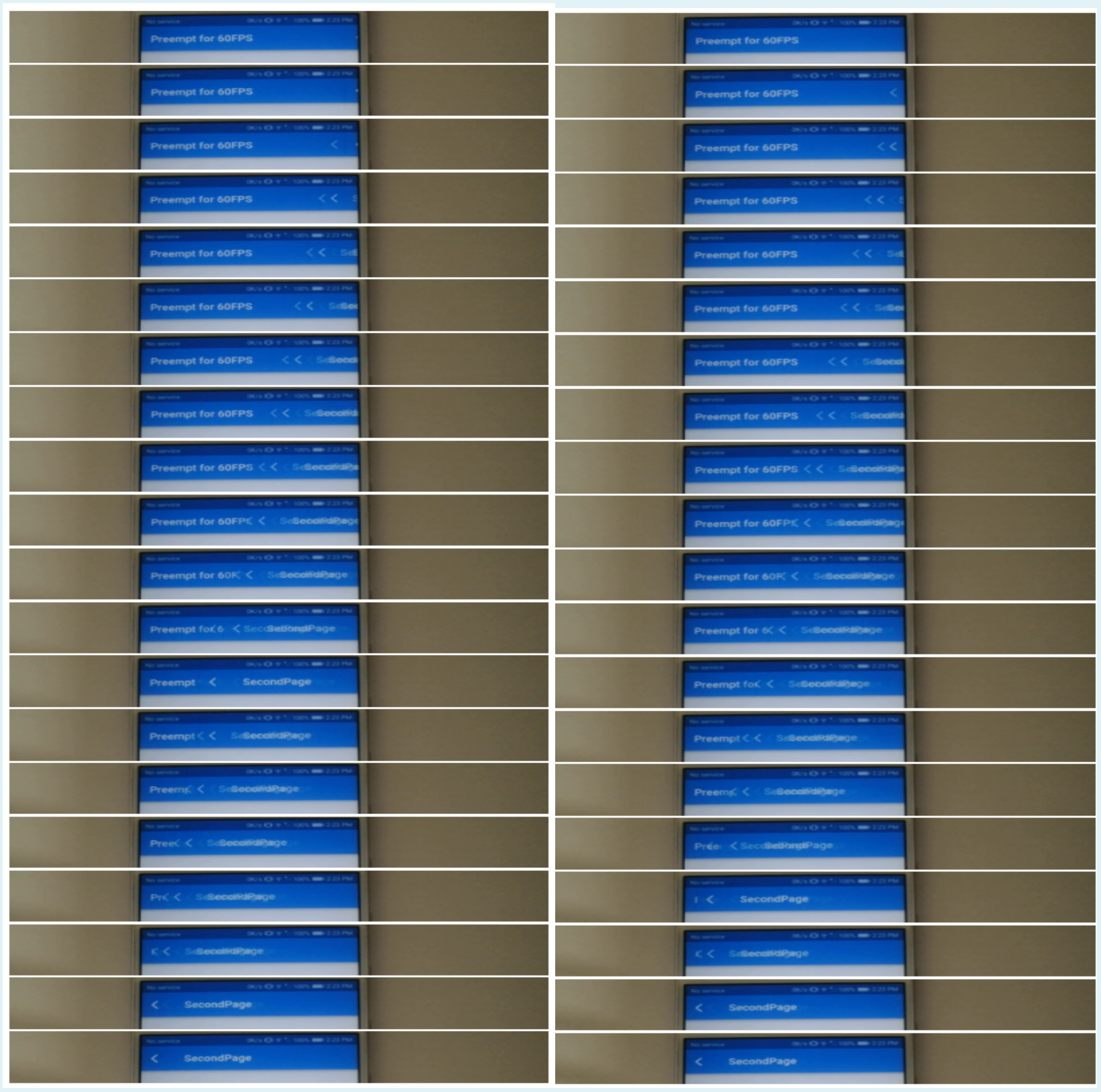
I use the following command to extract each frame as JPEG files out of the mp4 file:

```
ffmpeg -i $VIDEO -pix_fmt rgba -vsync 0 -frame_pts true -vf
drawtext=fontfile=/usr/share/fonts/truetype/freefont/FreeMonoBold.ttf:fontsize=80:t
ext='%{pts}':fontcolor=white@0.8:x=7:y=7 -filter:v "crop=1920:250:0:0"
~/temp/video_frames_cropped/output_%04d.png
```

Below are the frames during smooth animation (open them in a new page for best view). There are two series of photos, because the operation is repeated twice in the video.

Since the content (mainly the Icons.back in AppBar) is different in each frame, we know the Flutter UI refreshes at ~60FPS. We do see two photos that move the "left arrow" twice the shift compared with other photos, thus there are 5% frames crossing vsync, making maybe 57FPS is more accurate (I may explore later why 3FPS is lost when making the PR).

There often exist two or three "left arrows" inside one photo. This is inevitable since we are using a camera to capture a video, and the Flutter UIdoes not have the problem.



Overhead analysis: 3.9%

Indeed, I am suspecting whether it is an "overhead" or a "must". In order to have a 60FPS smooth UI, we must render something onto the screen, so must pay some extra UI thread time). What is done in `preemptRender` looks quite minimal.

I use `DateTime.now()` to simply record the time of `preemptRender`, and later print it out in a `PreemptRender times=[...]` log. Raw data:

```
times=[1.123, 9.671, 0.626, 0.694, 0.426, 0.625, 0.535, 0.614, 0.735, 0.608, 0.578, 0.6, 0.737, 0.852, 0.56, 0.832, 0.694, 0.58, 0.574, 0.52, 0.508, 0.58, 0.434, 0.562, 0.457, 0.435, 0.507, 0.503, 0.482,
0.411, 0.495, 0.468, 0.483, 0.458, 0.562, 0.442, 0.454, 0.634, 0.768, 0.797, 0.777, 0.757, 0.767, 0.607, 0.537, 0.69, 0.562, 0.737, 0.653, 0.462, 0.75, 0.613, 0.74, 0.626, 0.513, 0.447, 0.64, 0.429, 0.415,
0.503, 0.488, 0.513, 0.492, 0.502, 0.542, 0.451, 0.584, 0.455, 0.443, 0.512, 0.48, 0.48, 0.435, 0.574, 0.42, 0.719, 0.78, 0.772, 0.911, 0.853, 0.698, 0.662, 0.643, 0.453, 0.625, 0.641, 0.656, 0.576, 0.466,
0.615, 0.6, 0.484, 0.472, 0.456, 0.414, 0.524, 0.427, 0.435, 0.44, 0.566, 0.459, 0.507, 0.474, 0.518, 0.436, 0.455, 0.492, 0.46, 0.517, 0.434, 0.448]
```

The statistics is:

- P95: 0.81ms
- P99: 1.10ms
- Medium: 0.54ms
- Average: 0.65ms

So 0.65ms/16.67ms = 3.9% and 0.54ms/16.67ms = 3.2%.

I also record the time from `initState` to the post frame callback, in order to determine how long the page is needed to be visible to the users. The raw data:

```
slow_all=[745.347, 653.874, 720.377, 720.496, 731.236, 720.158, 731.52, 646.012, 734.658, 701.606, 733.994]; fast_all=[746.954, 701.56, 685.499, 688.476, 681.832, 692.165, 685.588, 676.315, 688.286, 682.473,
689.403]
```

The statistics:

- Without optimization: p95=740 p99=744 med=720 avg=712
- With the proposed method: p95=724 p99=742 med=688 avg=692

To my surprise, the proposed method has negative overhead. I honestly report it here for completeness, and may explore further if making a PR. Anyway, we also do not observe a significantly longer loading time, so there is at least no perceptible problem.

Remark: The speed will be much slower (even a 400% increase) if there is logging and printing. So when reproducing the experiment, you need to remove all before testing if the branch tip contains some temporary log at that time.

Needed code change

With the prototype, I roughly know what code change needs to be done to the engine and the framework. In the following paragraphs, I will list them, and also discuss their effects. Indeed, there are not many changes excluding the completely isolated PreemptBuilder and its underlying implementation.

The framework

- Add the whole `PreemptBuilder` and its underlying implementation. The new code will be separated from all existing code, so will not affect any existing code or user.
- Add `TickerRegistry` (InheritedWidget) and let Tickers report their existence to the registry. The change will not affect existing things.
- (Optional) Add `maybePreemptRender` to the `RenderObject.layout` function. Alternatively, adding a new `PreemptPoint` widget is also enough. The widget

solution will not affect existing things, and the add-to-layout solution will only minor affect the implementation.

The engine

- Add a few lines to `Animator::Render`, allowing the UI thread to submit to the rasterizer thread multiple times (but the Dart code will control itself such that it does not submit more than once per vsync).
- Change `VsyncAwaiter` to save the latest vsync time to a thread-shared variable, and later the UI thread will read it when needed. (P.S. The current prototype is merely a hack when implementing this; will do it seriously in the real PR.)

USAGE EXAMPLES

The new widget

We will introduce a new widget. Usage is like:

```
PreemptBuilder(
  builder: (context, child) => build_animation_that_needs_60_fps,
  child: arbitrary_sub_tree_such_as_a_new_page,
)
```

Well, it is very similar to what we are all familiar with, such as AnimationBuilder.

Example 1: Animation at 60fps, while having a heavy tree at the same time

Remark: The next example will be most interesting. This example only serves to understand the design step by step.

Suppose we have a page with a very heavy widget tree to build/layout. In the meantime, we want to show a 60-fps loading animation when it is not ready (building or fetching data or something else).

It can be implemented as follows. Indeed, just wrap with PreemptBuilder, nothing more.

```
Stack( // can be any parent, just for example
  children: [
    your_fancy_other_widget_trees,
    PreemptBuilder(builder: (_, __) => CircularProgressIndicator()),
    more_about_your_fancy_other_widget_trees,
  ]
)
```

Example 2: Smooth ListView

Suppose we have a page with ListView, and the contents in it are very heavy to build/layout. This is very common in the real world. In the old days, when scrolling, it may have been janky; but now it will be smooth as 60fps.

Users should not need any change except for adding a bool flag - `ListView(preempt: true)`. We can also make that flag on by default, then users need to do nothing when upgrading.

As for how it is implemented inside the ListView, it is also simple:

```
PreemptBuilder(
  builder: (_, child) => SlideTransition(
    position: scrollController.compute_extra_shift_at_this_extra_frame_by_inertia,
    child: child,
  ),
  child: ListView(...),
)
```

Example 3: Page transition (enter new page needs to a big new tree, while needing transition animation at 60fps)

UPDATE: A demo (prototype) is done for it. Please see the “Experiments (Prototype)” whole section.

When entering a page, the transition animation may be janky if the new whole page is so heavy to build/layout. But with this proposal, it should be smooth.

Same as above, users need to do nothing except for using PreemptBuilder just like AnimatedBuilder.

```
YourParentWidgets(
  child: PreemptBuilder(
    builder: (_, child) => YourFancyAnimationWhichNeeds60FPS(child: child)),
    child: YourNewPageAndSoOn(),
  )
)
```

Example 4: Custom scenario

The users should be able to implement their own arbitrary want-to-be-60-fps UI. Just mimic the examples above - add a PreemptBuilder.

DETAILED DESIGN/DISCUSSION

In short: Preempt build/layout (by calling a synchronous function), update the layer tree a little bit for animation, render the extra frame, resume build/layout (by naive function return).

The analogy / mental model

Some researchers or Nvidia DLSS have some algorithm such that, they can input some low fps frames, and output high fps frames. Consider this proposal as such a kind of "tween creator". In other words, originally we have janky rendering (say, 15fps). And now, we add three extra animating frames after each of the 15fps frames, to get a 60fps smooth feeling.

The flow chart

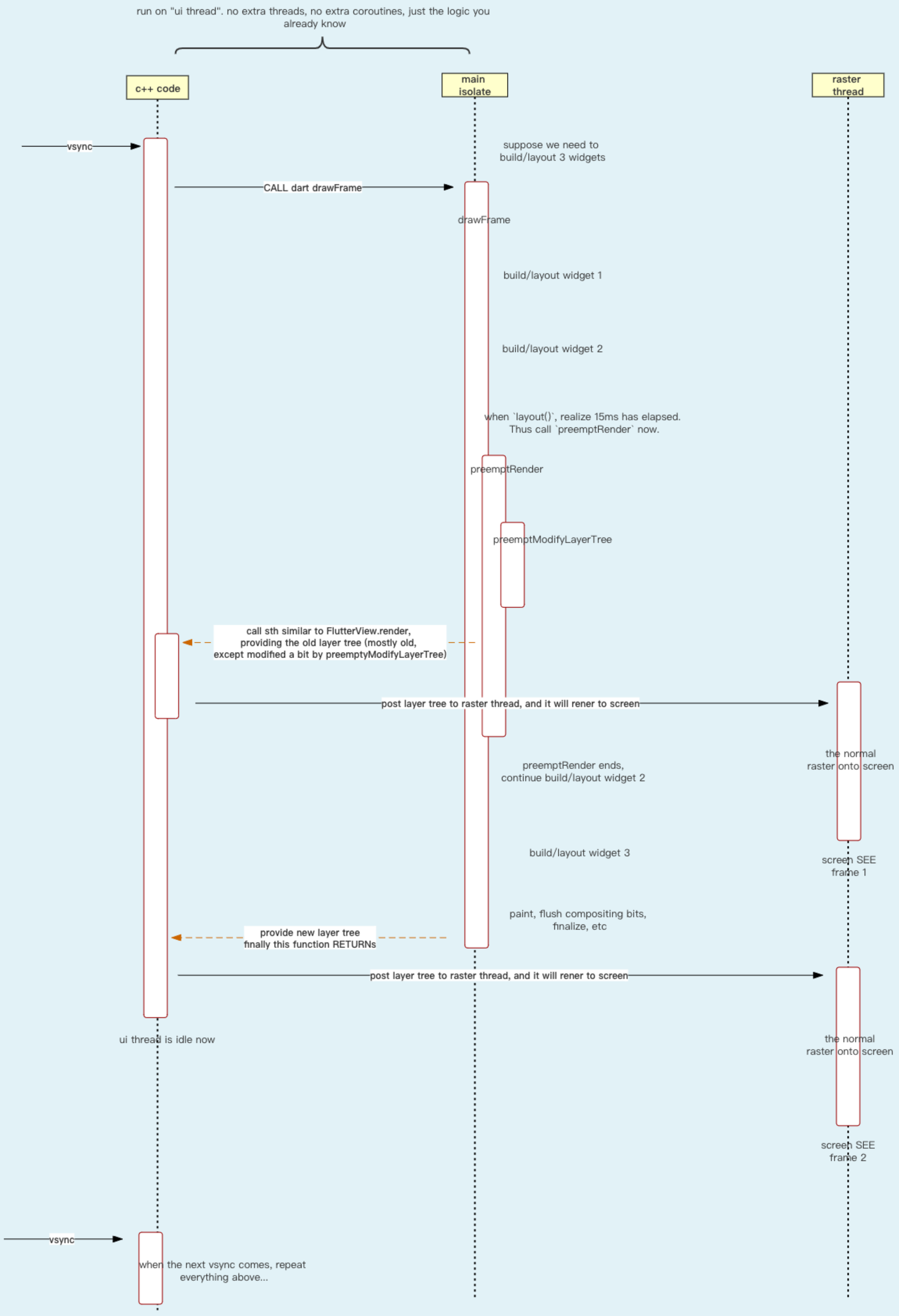
EDIT: As an optimization, we may move `preemptModifyLayerTree` from where it exists now to the beginning of each 60fps frame. Then, we will do even less work between observing a near-timeout and submitting things to raster thread, thus even reducing the risk of accidentally be over the deadline further. But just think about it as it is now since it may be easier to understand firstly.

Below is the flowchart of one cycle. (I will refine and reformat the chart if this doc is later approved.)

The C++ code and main isolate runs on the UI thread, just like what we already do currently. In other words, there is no extra thread or coroutines in this proposal.

Description of the figure:

- vsync comes.
- As normal, C++ calls Dart's drawFrame.
- Suppose Dart has 3 widgets to build/layout. It build/layout the 1st, then 2nd.
- Then it realizes time has up (say, 15ms has come), when layout() the 2nd widget. Then it calls preemptRaster() (a dart function).
- In preemptRaster, we firstly call preemptModifyLayerTree to modify the layer tree a bit. See below for examples about this function.
- In preemptRaster, we then call a probably modified version of FlutterView.render. In other words, we provide layer tree to C++ code, and c++ code provide it to raster thread. Notice what layer tree we provide here: Because preemptRaster is called within a layout(), the paint phase has not started, so the layer tree is completely old (instead of mixed). ThusIn addition, preemptModifyLayerTree will modify the layer tree a bit. That's all. We will send this to raster.
- Raster thread renders that layer tree as usual, so we see beautiful things on screen.
- UI thread C++/Dart goes on, because preemptRaster function returns. The Dart code will continue from where preemptRaster is called (you know, just very plain function calls; but this solves the "how to suspend a layout call" implicitly indeed). In Dart's view, it thinks it is still the 1st frame. Let's say it continues layouting the 2nd widget. Then 3rd widget. Then paint, flush compositing bits, semantics, etc.
- Then finally, as a normal pipeline stage, dart provides the new layer tree and let c++ to throw it to the raster thread.
- Raster thread renders it to screen in the background.
- Then, just like what will be done normally in frame 1, call post frame callbacks, c++ calls dart for some callbacks, etc.
- Now ui thread is idle. When next vsync comes, the same loop will go.


run on "ui thread". no extra threads, no extra coroutines, just the logic you already know
c++ code
main isolate
raster thread
vsync
suppose we need to build/layout 3 widgets
CALL dart drawFrame
drawFrame
build/layout widget 1
build/layout widget 2
when `layout()`, realize 15ms has elapsed. Thus call `preemptRender` now.
preemptRender
preemptModifyLayerTree
call sth similar to FlutterView.render, providing the old layer tree (mostly old, except modified a bit by preemptyModifyLayerTree)
post layer tree to raster thread, and it will rener to screen
the normal raster onto screen
preemptRender ends, continue build/layout widget 2
build/layout widget 3
screen SEE frame 1
paint, flush compositing bits, finalize, etc
provide new layer tree finally this function RETURNs
post layer tree to raster thread, and it will rener to screen
ui thread is idle now
the normal raster onto screen
screen SEE frame 2
vsync
when the next vsync comes, repeat everything above...

The code

The high-level code will look like this:

```
class RenderObject {
  void layout(Constraints constraints, { bool parentUsesSize = false }) {
    if (nearTimeout) { preemptRender(); }
    … the original layout code …
  }
}

void preemptRender() {
  // remark: At this location, the layer tree has not yet been touched by the
current frame pipeline. So we have plain old layer tree.
  preemptModifyLayerTree();
  FlutterView.render(layer_tree);
}
```

To understand the core concept, suppose preemptModifyLayerTree is nothing but a very naive function that modifies the existing layer tree by hand (such as, `renderView.layer!.offset += 123`). We will refine it later.

From preemptModifyLayerTree to PreemptBuilder

If all sections above are done, we now have a mechanism for 60FPS smooth animation, no matter how heavy the tree is to build/layout. However, that 60FPS animation is very hard to build - we have to manipulate the layer properties by hand inside preemptModifyLayerTree.

In this section, we will see how we create a simplest PreemptBuilder widget, and no developer (or Flutter framework dev) will ever know the existence of preemptModifyLayerTree.

The core idea is to use a second tree in addition to the main tree. In other words, we create a separate BuildOwner/PipelineOwner/root-widget/etc (correspond to SecondTreePack in prototype code). Then, we are free to call its buildScope/flushLayout/flushPaint at *any time* at any frequency we like. Its input is a widget tree (indeed PreemptBuilder.builder output), and its output is a layer tree (indeed to be inserted to the main tree).

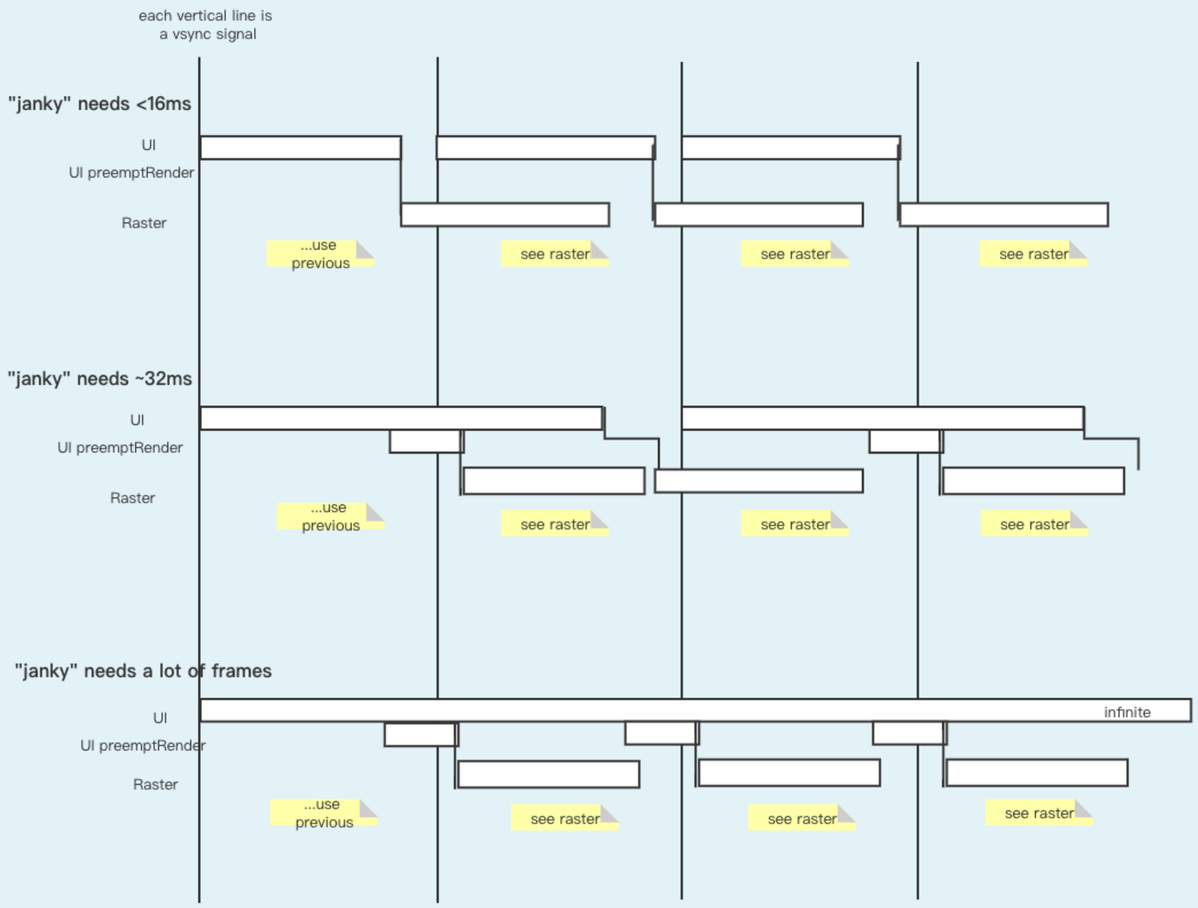
Then, we need to graft the second-tree's layer tree and the main-tree's layer tree. Shortly speaking, we do so in `paint` function by `context.addLayer` and so on. Details can be found in the code.

When to call preemptRender

Just call it when `now - lastVsyncTime > threshold` where threshold is smaller than and near 16.67ms.

Indeed, this approach is robust to the choice of threshold, and the execution time of preemptRender itself. If the preemptRender misses the vsync deadline, nothing bad will happen. This is shown in the figure below.

For completeness, three cases are discussed - a janky frame needs <16ms, ~32ms, and infinitely long. In the diagram, we deliberately assume the `preemptRender` *all* misses the vsync deadline. Surely, if they meet the deadline, the scenario can just be better instead of worse. As can be seen in the diagram, in each 1/60s, we see the rasterizer thread produce one outcome, so it runs exactly at 60fps.



ACCESSIBILITY

None

INTERNATIONALIZATION

None

INTEGRATION WITH EXISTING FEATURES

Well it is just a bool flag “preempt=true/false” (in the users’ view), so I guess the API looks natural.

OPEN QUESTIONS

- Will it work?
- How to check whether it is near the timeout, such as one check per `RenderObject.layout`. One approach may be reducing it. Such as check every N times, or check only when layout depth is lower than K. Another approach may be utilizing C++. Maybe we have a C++ atomic variable `is_near_timeout`, which is set in another thread and read here. I have not dealt with the details yet, since I want to focus on the main problem now.

TESTING PLAN

- Build a prototype to verify it works
- Enhance flutter testing package so it understands that, one “tester.pump” can result in multiple rasterized frames
- Test modified widgets, e.g. CircularProgressIndicator and ListView

DOCUMENTATION PLAN

- API should document the “preempt” flag
- Maybe add an article explaining what is going on (I am willing to write one)

MIGRATION PLAN

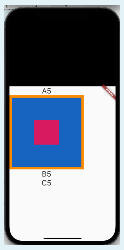
For end users, no need to migrate - it is just a bool flag (“preempt”) that is optional.

OUTDATED CONTENT

All contents below are outdated (outdated API, and outdated prototype). They are not deleted because otherwise may delete some Google Doc comments bound to them.

~~There are more details, which can be seen in full code prototype about this "from preemptModifyLayerTree to PreemptBuilder): https://github.com/fzyzcjy/flutter/blob/1332f1e8b46f16efe26ba679f3f815695bdfe349/packages/hello_package/lib/main.dart . Indeed, in that prototype, the last step - assembling PreemptBuilder - is not done, but quite close. Just combine AdapterInMainTreeWidget + AdapterInSecondTreeWidget + secondTreeWidget and wrap them into one single PreemptBuilder and we are done. This prototype does not aim at anything like speed up, but only shows ingredients of PreemptBuilder, i.e. this section.~~

~~Here is a screenshot of the demo (I know it is not beautiful in terms of art...). If you run it, you will see all colorful things change color in every second (i.e. every new frame tick). The orange border is from main-tree ancestors, the blue is from second-tree, and the red is from main-tree children. Thus, I show that they coordinate well together.~~



~~Implement "Example 1: CircularProgressIndicator, or any single DisplayListLayer animation"~~

~~Let's start by building it from the low level API. Then, we can later abstract out some high level APIs.~~

~~When user puts a `CircularProgressIndicator(preempt: true)`, we may convert it to:~~

```
PreemptDisplayList(
  child: CircularProgressIndicator(preempt: false, ... other argos ...),
)
```

~~The `PreemptDisplayList` may be a SingleChildRenderObjectWidget. Its render object is:~~

```
~~class RenderPreemptDisplayList extends RenderBox {~~
~~  @override~~
~~  void paint(PaintingContext context, Offset offset) {~~
~~    layer = paint_child_subtree_inside_a_DisplayListLayer();~~
~~  }~~

~~  @override~~
~~  void preemptModifyLayerTree() {~~
~~    layer = paint_child_subtree_inside_a_DisplayListLayer();~~
~~  }~~
~~}~~
```

~~In other words, we just let the subtree paint on a DisplayListLayer, and that is what we do inside preemptModifyLayerTree. Then, when later the layer tree is submitted to rasterize, we will see the CircularProgressIndicator is having 60fps.~~

~~Implement “Example 2: Smooth ListView”~~

~~There are multiple ways to do it, and here is one of them (maybe simplest to explain and see the idea, though not the most flexible).~~

~~Firstly, a `ListView(preempt: true)` is converted to:~~

```
~~PreemptShiftByGesture(~~
~~  child: ListView(preempt: false, … other argos …),~~
~~)~~
```

~~The PreemptShiftByGesture is again a SingleChildRenderObjectWidget. The render object is:~~

```
~~class RenderPreemptShiftByGesture extends RenderProxyBox {~~
~~  var preemptShift = 0.0;~~

~~// REMOVED~~
~~//  @override~~
~~//  void preemptHandleTouchEvents(PointerMoveEvent e) {~~
~~//    preemptShift += e.current_shift_amount;~~
~~//  }~~

~~  @override~~
~~  void paint() { // Well maybe not exactly do this in paint, this is just rough~~
~~idea~~
~~    // reset it, because the ListView will handle the touch events and shift its~~
~~real content now~~
~~    preemptShift = 0;~~

~~    layer = pushOffsetLayer();~~
~~  }~~

~~  @override~~
~~  void preemptModifyLayerTree() {~~
~~    layer.offset = preemptShift;~~

~~    // in each preempt painting, consider the inertia about how much ListView wants to shift and shift it.~~
~~    // May not be proper to put the code exactly here, and we may need another callback etc, but this is rough idea~~
~~    preemptShift += get_list_view_current_speed_caused_by_inertia();~~
~~  }~~
~~}~~
```

~~In other words, suppose the ListView runs at 20fps. Then, between each of the janky 20fps frame (which means ListView receives user inputs, shifts it, layout/build it, and paint it), we add two more extra frames by the following logic: Inside the 60fps preemptRender, we use preemptModifyLayerTree to shift the content a little bit (the extent is derived from ListView's current inertia).~~

~~At a first glance, it may seem that it is still janky, because we do not accept gestures at 60fps, but only at 20fps (in the example). However, normal gestures about shifting a ListView seem to be smooth, i.e. the speed of dragging does not change abruptly. Therefore, by using inertia to calculate the ListView shift, it should still be smooth. In addition, consider the "mental model" we mentioned above. Gamers using Nvidia DLSS (which creates additional frames to games) seem to be happy about that, and "get extra tween frames to scrolling a ListView" may mimic "get extra tween frames in a game".~~

~~Implement "Example 3: Enter page"~~

~~Well, just mimic the two examples above.~~

~~The prototype with video~~

~~Please refer to https://github.com/flutter/flutter/issues/101227#issuecomment-1257098789 to see a working prototype (a page-transition example). It has a video demonstrating that it runs at ~60fps, while widget build/layout needs ~500ms.~~