Flutter
Dirty Region Management

SUMMARY

This document describes the issue of Dirty Region Management with respect to UI API design and implementation and how it relates to Flutter apps.

Author: jimgraham@ (flar)
Go Link: flutter.dev/go/dirty-region-management
Created: 09/2020 / **Last updated:** 10/2020

OBJECTIVE

This document should serve both as a general introduction to the concept of Dirty Region Management and the history of such mechanisms, as well as how Flutter manages these issues and what changes can be made to make it more efficient at doing so.

BACKGROUND

Dirty Region Management has been a staple of UI frameworks for decades, but the state of the art has changed quite a bit from the early days of CPU rendering on chips clocked in MHz to the modern age of GHz GPU rendering on pretty much any device.

The basic concept is to reduce the amount of time and resources you spend updating the screen when only a small number of things are changing on it. Originally the need was critical as the time spent was visible to the human operator, impacted their productivity, and measured in how quickly they could get their work done. More recently the shift has been to measuring the work saved with regard to how much else the machine can be doing in the background while the user is working in a UI program, or whether animations can maintain a consistent frame rate (>60+ fps), and also how much energy is used for devices that run on battery power.

Modern hardware is capable of maintaining a complicated 3D scene and updating it at 30-60fps even on a phone - mainly for games - so the need for this kind of mechanism isn't as clear. The distinction, though, is that games tend to be written to the capabilities of the hardware with fairly custom engines that are driven by those capabilities and by developers who are experts in the constraints of those engines. Games also tend to be the only thing the user is doing on the device at the time so reserve compute time for background tasks can take a back seat. The content of games also tends to be more actively involving than a simple text cursor blinking on a composition screen while the user thinks.

UI frameworks, on the other hand, are targeted at making it easy to present information to users with as much convenience as possible so that the developers can spend more time on their content ideas than on learning what the hardware does and being driven by those considerations. While the content of User Interfaces tend to be much less demanding than the content for games, modern graphical UIs are introducing concepts that bring 3D, antialiasing, and pixel effects to the simplest of apps at the conceptual cost of setting a flag or wrapping a widget with a boilerplate container. Mostly the latter feature of pixel effects tends to complicate the way you render a widget hierarchy and also cost quite a bit more rendering time than simple text, buttons, colors, gradients, and translucency. And the content

goal of a UI app is less to visually entertain than to use minimal processing and battery power to convey small concepts such as “this content is still loading” or “here is where you were typing once you decide what to say next”.

Even if modern hardware can gloss over the expense of some of these rendering operations with respect to keeping a smoothly flowing 30 or 60 fps frame rate, the time spent feeding the GPU takes away from other tasks the device may be doing in the background and can cause mobile devices to use more battery just to accomplish simple things like a blinking text cursor or flashing a button.

Glossary

- **Dirty Region** - A portion of an app where widgets and components are different in one frame as compared to previous frames that have been displayed.
- **Render Root** - An area of an app, typically associated with a single widget, that occludes all content behind it such that any changes in front of this area and bounded entirely within it can be repaired by starting the repaint at the Render Root.
- **Mahogany Staircase** - The term that refers to the Flutter framework design where a tree of representative objects is used to construct a parallel tree that describes the original objects in a new way. There are several such tree conversions that happen as Flutter refines from a tree of Widget objects into a tree of renderable Layer objects as described in this Google TechTalk.

OVERVIEW

Flutter has a number of mechanisms designed to reduce the effort of computing a new frame based on changes to the Widget hierarchy as specified by the application. But these mechanisms are not yet used by the rendering system to minimize the area of the screen that is repainted on each frame.

Non-goals

This document is mostly a list of issues to consider and potential/general solution ideas but does not propose specific implementations for use in Flutter.

DETAILED DESIGN/DISCUSSION

History

Typical techniques used to manage dirty regions are broken down into 2 steps - accumulation and handling/repainting.

Dirty region accumulation

Sources of dirty regions (historically) include:

- Changes to the state of items: color, shape
- Changes to the hierarchy: widgets added or removed, scrolling
- External factors: application life cycle, overlapping windows, GPU resources
- Animations (which are mostly just a frequent source of one of the above)

Some of these sources don't tend to happen on embedded devices (typically there is only one app running so there doesn't tend to be "damage repair"), and modern desktop systems often devote a backbuffer per window so even overlapping windows can be entirely handled by the system desktop compositing mechanism. The only remaining external factors tend to be life cycle issues like being mapped to the screen, minimized, maximized, and quit.

Most of the programmatic sources can be managed by including code in the various widget (or RenderObject) setters such that when a property that impacts its visual representation changes, it informs the framework that it needs to be repainted. Some systems allow a rendered object to optionally specify a rectangle that needs to be repainted if the attribute that is changing only affects a small portion of its screen representation.

Various bookkeeping strategies for tracking dirty regions include:

- One big bounding box of all changes
- List of N bounding boxes of consolidated changes
- Pixel-level Region object

Most systems start with the "one big bounding box" approach as that handles the needs of 99% of applications which may only be dealing with one widget changing on a screen at a time (blinking cursors or buttons/checkboxes changing state due to user input). The list of bounding boxes and region objects help with situations like spreadsheets, or dashboards where multiple isolated rectangular regions can

update on each frame, but not all of the content.

If a system has a well-optimized Region implementation and the ability to render clipped to that Region, the last solution was historically chosen. But GPUs don't tend to offer the ability to clip to a complicated list of rectangles so that option doesn't get chosen much any more. Another consideration is that minimizing pixel operations is just one of the goals here, but minimizing the overhead of producing the rendering operations is also important.

If a short list of N rectangles is accumulated, then the size of the list needs to be balanced between how expensive it is to process compared to how expensive it is to just rerender the entire scene. This decision takes a lot of factors into consideration including how easy it is to only perform the rendering operations for the list of renderable objects that intersect a rectangle. If the data is kept to allow efficient drilling down to just the affected renderables and if the list of rectangles mostly involve disjoint renderables, then the list can be larger. If the response to rendering a given rectangle boils down to setting a global clip rect and then executing the rendering operations for the entire scene, then this solution, repeated for each rectangle in the list, is usually not much better than just going with the "one big bounding box" approach. It might win if the cost of navigating the full scene and dispatching the operations is low and the savings of not rendering pixels that fall outside each of the component bounding boxes is huge, but with modern GPUs the savings on fill rate is fairly small so any overhead in processing each bounding box magnifies.

In the end, many systems use the "list of N dirty rectangles" approach, but use heuristics to punt on the complexity to a single bounding box. Some systems may even track the total proportion of clipped content compared to the size of the scene and potentially also stop the accumulation process early and punt to just repainting the entire scene. The heuristics might also take into account whether there are well separated Rendering Roots (a concept I'll explain soon), how fast the GPU is compared to the CPU, and how deep vs. wide the rendering tree is.

Handling a dirty region

The simplest way to repaint a dirty region is, as mentioned above, to set a global clip to the extent of the dirty region and then repaint the entire scene and let the

GPU or the rendering code elide whatever operations don't matter. This can be quite effective if the culling of rendering operations is efficient, but there are a few techniques that can help make this better.

One of the better "bang for the buck" techniques is to look for large areas filled with opaque pixels. Typically these are containers with an opaque background color, but any rectangle that appears in a system might provide some decent optimization. (In particular, by the time Flutter gets to rendering, the distinction between containers with background colors and random rectangles filled with colors is no longer tracked.) These areas can represent origins for repairing a dirty region in that there is no need to process any objects behind them when the dirty region falls entirely within their opaque bounds. One can refer to these areas as Rendering Roots because they can be noted and tagged while accumulating dirty rectangles. If an object needs to be repainted, then its bounds are recorded and compared against opaque rectangles registered by their parents for a Rendering Root that covers the dirty region and that parent would then be recorded with the rectangle. When it comes time to repair the region, only children of that Render Root and any of its siblings in front of it need to be visited.

Another data point that can be useful to track when dealing with Render Roots is the case of containers with mutually non-overlapping children. If such a container is a Render Root, then only a subset of its children need to be visited when repainting a region that lists it as a root. In some cases, the parent might know enough about the layout of these children that it can even immediately know which children are in the dirty region without having to process them all (consider a spreadsheet with fixed cell sizes - the affected children can be quickly computed by a pair of simple divisions on the LTRB properties of the damage rectangle).

More recent considerations

"Back in the day", UI rendering tended to be non-antialiased, opaque, flat (2D) and with no pixel effects. Dirty region management techniques were born in that day and the presence of large rectangles filled with a (presumably tautologically) opaque background color were plentiful and easy to find (basically, a widget's parent except for occasional "no bg" containers).

But, today's GUIs have a number of attributes that complicate both the process of accumulating a damage rectangle, and the process of handling it.

Opacity

The simplest advancement, in terms of complicating dirty region management, is the inclusion of non-opaque colors. While these colors do affect the rendering of a scene, they basically disqualify a rendering operation from being considered a Rendering Root. This is not much different than containers with no background color set, or “input only” containers in classic dirty region management.

Transforms

The first coordinate issue to be dealt with when the GUI has 2D or 3D affine transforms is that the bounds of a widget that changed need to be transformed to a common space for tracking the dirty regions - and typically that is screen or pixel space. The transformation of bounds is not rocket science and the screen bounds of rendered objects is often tracked anyways so this isn’t a huge complication.

Secondly we have to deal with rectangles that want to be Render Roots: “Hey, I’m a rectangle! And I have an opaque background! I’m a Render Root!” Well, not any more. If you are transformed by any affine operation other than a translate or a scale, then you don’t align well with the screen-space rectangles typically tracked for dirty regions. It is still possible to compute a sub-region of your bounds that is opaque and screen-aligned, but how often will that matter? And if you are transformed away from the Z=0 plane then it is probably much less likely you’ll work as a Render Root and it just became a lot more complicated to analyze that potential.

Antialiasing

It used to be simple to compute the opaque bounds of a widget - it was just its actual bounds. With appropriate choice of rounding operations, you would know the exact bounds even of a transformed widget. Antialiasing complicates these computations only a little in that you want to include any pixel affected by rendering something and usually that involves just doing a floor on the upper left of the bounds and a ceil operation on the lower right.

It also comes up in computing the opaque area of a potential Render Root, but here you do the rounding in the opposite order - ceil the upper left coordinates and floor

the lower right.

Pixel effects

For the purposes of this section, I am only going to consider the issues raised by a pixel effect that is applied to descendents of the effect. In particular, Flutter includes a mechanism to apply a pixel effect to the area behind a widget, but I will discuss that in a later Flutter-specific section.

Many pixel effects have no impact on bounds, but can change the opacity of pixels. The more interesting effects (from the perspective of dirty region management) are the ones that change the bounds. This isn't a huge complication - one simply asks the effect to recommend new bounds based on the raw bounds of its content, or on the bounds of a "dirty region" within its content - and that becomes the dirty region you track. This does mean that the parents of a widget that is changing need to be able to not only check if they provide a Render Root background of the dirty region, but also be able to recommend a different dirty region as well (in other words, the calling sequence that propagates a dirty region to an associated Render Root needs to include a way for the bounds to be updated as the search progresses).

These effects have similar impact to the considerations mentioned in the Antialiasing section above, but with more complicated analysis. An effect that produces a result that is larger (as in a blur effect) or moved compared to the bounds of its content (such as a transform effect) would need to modify the dirty region of its content as it propagates to the common space that all dirty regions are managed. An effect that causes parts of the content to become less opaque (such as a blur effect) would also need to modify the area considered as a good opaque backdrop for a Render Root. Typically, though, for simplicity such an effect just records the fact that it can reduce the opacity of its content and then its bounds are ignored for the purposes of finding a Render Root. Computing the precise opaque region of a coordinate transformed blurred region is typically not computationally worthwhile to avoid using a more ancestral widget as the Rendering Root.

Flutter-specific issues

BackdropFilter

The primary complication here is that the BackdropFilter is a backwards-looking effect meaning that it changes the appearance of the things behind it. Most dirty region management is focused on every node in the tree constraining or tracking its children so that the results and interactions of the rendering of various nodes can be tracked forward via typical tree traversal techniques.

But the BackdropFilter, by changing the rendering of objects behind it, violates this forward-acting approach in a couple of ways.

First, by default, the BackdropFilter applies to the entire screen/app. A clip widget can be inserted to restrict its rendering, though, but in general its output is not associated with its children - its “content” so to speak - it is instead associated with an open-ended impact on its inherited ancestors.

Primarily that means that it draws over everything, so its bounds in the default case are “the screen”. But, even if the programmer has installed a clip widget to restrict its output, its bounds are “bigger than I can tell from examining my children, but I don’t know how big because some parent might restrict me”. This can be managed by simply having it process dirty regions from its children by changing the dirty rectangle to be “the screen” and then passing it along to its parent. If one of its parents is a clip, then that parent can restrict the dirty region to the bounds of the clip. The clip could also be accumulated as you descend the tree and presented to the children who are determining how their changes might contribute to the overall dirty region - they would first pass their own dirty region through the clip before recording it.

OK, so the output side of the impact of the BackdropFilter is really only just a special case of what we’ve seen with other rendering tree operations, but there is a case where the input is also surprisingly non-tree-like in a way that is harder to handle naturally in a single pass tree traversal.

Because the BackdropFilter takes the layers behind it as input, and because those layers are not restricted by any clip container surrounding the BackdropFilter, there

are cases where dirty regions that don't intersect the clip container can still modify the way that the BackdropFilter renders. Normally if you had two adjacent ClipRect containers that don't overlap, and a dirty region is recorded inside one of them, then the dirty region would be clipped to the parent Clip container and then compared to the bounds of the other Clip container and discovered to be non-intersecting. But, if one of the children of that second Clip container happens to be a BackdropFilter, then it might read the pixels from inside the other Clip container (the one that was dirty), perform a blur on them, and those pixels will now affect what appears in the clipped bounds of that BackdropFilter - even if the BackdropFilter and its part of the tree are stable.

The net result of all of these considerations is that when a BackdropFilter is in a scene, we need to take special care that trees that contain them are repainted even when they don't appear to intersect the dirty region.

But, wait! There's more!

Consider a simple case of a BackdropFilter that has no clip. When any object in its background changes, we need to reapply the blur. Typically, we only repainted the object that changed, but because we need to reapply the blur, we have to also repaint everything under it that was inside the blur radius. In this case, the optimizations we'd perform when accumulating the dirty region of the object that changed are invalidated by the presence of a BackdropFilter above it.

So, generally, we then need to track any regions of the screen on which a BackdropFilter applies its filter, and repaint the entire "blurred area" when any part of that area is repainted.

As an exception to this. If a child of a BackdropFilter is modified and its (blurred) dirty region is encapsulated by a Render Root before it gets to the BackdropFilter parent, that optimization can remain valid. Also, any sibling of the BackdropFilter that paints above it can capture a repaint region as a Render Root without having to invoke the "repaint everything in the blur" area. This optimization would require us to track not only the regions of the screen that are blurred by a BackdropFilter, but also the parts of the tree that live underneath the BackdropFilter.

Layers

The Flutter framework combines all of the output of describing a widget tree in rendering terms into a series of layer objects which it then passes to the rendering engine to process and express as calls to various platform rendering APIs such as OpenGL or Metal or Vulkan. On most platforms it uses the Skia rendering library as an intermediary to handle the specifics of these various graphics APIs for it.

Typically these layers represent a fundamental break in the rendering of the various pieces of the widget tree - places where an opacity, transform, clip, etc. is applied to a number of widgets. Layers for those particular operations are pushed and then a number of children are added to them. These layers are hierarchical and form a tree, but it is not a 1:1 translation of the widgets. Widgets may combine to render their output in a single layer or a single widget may produce more than one layer (less common). Some common rendering layers would be:

- OpacityLayer
- TransformLayer
- ClipLayer
- PictureLayer
- ColorFilterLayer
- ImageFilterLayer
- BackdropFilterLayer

These layers might modify the graphics context with some additional information (such as a modification to the current transform) and then pass to their children for more rendering to occur. But many of them require their children to render separately to an offscreen texture and then composite that result into the existing scene with some possible modifications. For example, the OpacityLayer renders its children into a texture and then paints that texture into the scene with an additional opacity modulation attribute on the drawTexture call. A ClipLayer might just install a clip on the rendering context if the clip type requests hard edges, but it may use an intervening temporary texture for its children if the clip type requires soft/antialiased edges or is not a simple rectangle.

A BackdropFilterLayer is a special case and not only uses a separate texture for rendering its children, but also requests that the graphics machinery fills this texture with a filtered copy of the existing scene before it renders the children.

With respect to managing dirty regions, though, these layers represent just about any rendering technique that might require special casing of the output regions of other parts of the scene. More importantly, the layers sometimes attempt to reuse these layers from scene to scene if their attributes did not change, so their reuse in more than one scene can be used to measure the stability or dirtiness of the output.

Pictures

Much of the regular rendering of a scene (filled background shapes, text, checkboxes, icons, lines) is combined into a recorded list of basic rendering operations called a Picture. The advantage of a Picture is that it can tightly represent a large number of rendering calls in a small number of objects. One of the disadvantages of the Picture is that it can represent the rendering calls of a large number of widgets including a mix of widgets that are stable from frame to frame and some which are animating on every frame.

A further complication of these Picture objects is that they are currently opaque objects both at the framework level and at the engine level. The Dart object at the framework level holds a reference to a Skia SkPicture object at the native engine level. Neither the Skia API, nor (consequently) the Dart API for these objects provides any methods for determining what kinds of rendering operations are inside them. The Skia API does provide some minimal information such as the number of operations so that the engine can analyze the complexity of a picture in loose terms as it makes decisions about caching the output.

The only mechanism provided with regards to dirty region management and stability is that if the Picture is not reconstructed by the framework then its persistence will be noticed by the engine level as each object is associated with a unique ID (a fingerprint so to speak) for caching purposes, but only if the rendering commands are not recomputed by the framework. Unfortunately, that is the only opportunity to detect their stability. If but a single widget that contributed to a Picture object changed its rendering commands then the Picture will be reconstructed for the new frame and its identity will not match. Furthermore there are occasions when these Picture objects are not directly associated with any widget that changed but are nearby some other widgets that changed and so their

painting must be repeated, only to produce a new Picture object that contains the same rendering as the previous object. Techniques such as comparing the serialized forms of 2 Picture objects have been used to discover that such cases exist, but a deeper analysis of what causes them has not been performed yet.

To help ameliorate this issue, the framework does provide a RepaintBoundary widget which will isolate its children from sharing a Picture object with any of their ancestors or siblings and if such an object is used judiciously by an app then animating widgets will not disrupt the recorded output of static widgets in the tree around them. This mechanism requires manual intervention by the app developer, though, and their use of these widgets requires some understanding of the various layers of the Flutter framework, which can be an obstacle for new developers and sometimes even seasoned developers.

Another Picture-based issue to consider is that Render Roots cannot be computed if the actual rendering that signals them is hidden inside these opaque objects.

Mahogany Staircase

There are multiple tree-based levels in Flutter and each has similar, but not identical, rules in tracking state changes. These levels include:

- The Widget tree, maintained by the app developer
- The Element tree, reconstructed from the Widget tree when it changes
- The RenderObject tree, managed by the Element tree
- The Layer tree, constructed by the RenderObject tree
- The EngineLayer tree, mostly 1:1 native counterparts to the Layer objects, but they do have representative “Handle” objects seen in the Layer tree. The Dart handle class is called “EngineLayer” and the native C++ class is called flutter::Layer.

It is important to note that each of these layers can be bypassed by an application to achieve custom results. The final determination of stability must lie in the lowest level in order to take into account the possibility that external sources have directly manipulated the eventual EngineLayers. With proper documentation developers who create their own RenderObjects can preserve good information on stability as can the code in the Flutter framework, and as long as the stability is an optional optimization for the engine layers, their non-compliance will not produce incorrect results - it just might not be as efficient as it could be.

But much of the work for detecting tree stability can be done at higher levels where we have more information and, in fact, it already does so in order to reduce allocations and computations in each of those staircase steps. Every time one level decides to keep the same object in the next tree to represent the state in the previous tree, vital information about the stability of the tree is preserved for the next level to use. If that reuse makes it all the way down to the EngineLayer tree then the bottom level has the best chance to minimize repaints. How well each level achieves these goals will become much more noticeable once we instrument the engine level to start reducing the repaint areas.

OPEN QUESTIONS

- The issues described here are represented by an open GitHub issue listed in the Flutter repository: Track damage rect and only draw inside it

MIGRATION PLAN

- Work has started on a possible solution that could be entered into the Flutter engine repo in a number of opt-in stages. The various PRs for these efforts should be linked to the Issue mentioned above under OPEN QUESTIONS:
    - Stage 1 will be a mechanism to compute the differences between engine layer trees. It will not be plumbed into the existing rendering pipeline at this point, but will include tests to verify its function and to increase confidence as the work progresses through the stages.
    - Stage 2 will be to use the difference mechanism created in Stage 1 to restrict the rendering of frames depending on the double buffering strategies and mechanisms of the various supported platforms. The new rendering modifications will be gated behind an opt-in flag.
    - Stage 3 will be to change the enabling flag from opt-in to opt-out after an appropriate period of debugging and feedback from developers.