SVG in WebRender

Actions:

Terminology

"Blob images" is webrender's fallback mechanism. A blob image is a command list that is opaque to webrender. It is rendered in worker threads during the scene building phase using skia software into small tiles. WebRender treats the result as a regular tiled image. A blob corresponds to one command list which produces multiple tiles. SVG documents are mostly drawn using blobs, however one document can turn into multiple blobs.

We often use the terms "blob" and "SVG" interchangeably. This is because very few things use blobs other than SVG and most SVG content is currently rendered using blob images.

SVG performance issues

TODO: Put bug numbers in here, to find them:
https://wiki.mozilla.org/Platform/GFX/perf_triage#Blob_image_rasterization

Slow path rasterization on the CPU

Filling a large amount of pixels on the CPU for large paths is expensive.

- https://bugzilla.mozilla.org/show_bug.cgi?id=1637876
-

Malloc in worker threads

Turns out that often when blob rasterization takes a long time we are spending a lot of time in malloc. It's worth pointing out because we can probably improve that without massive changes. Note that we don’t have jemalloc pools in worker threads. We also don’t make any effort to reuse memory allocations in this code.

- https://bugzilla.mozilla.org/show_bug.cgi?id=1477371
- https://bugzilla.mozilla.org/show_bug.cgi?id=1519622
- https://bugzilla.mozilla.org/show_bug.cgi?id=1637876
-

Replaying svg commands on multiple tiles is slow

The blob code replays each drawing command on all of the tiles that intersectwith the bounds of the command. Skia doesn't deal with that well and there is a lot of overhead related to the content outside of the tiles. There is also a lot of duplicated work for each tile. For example skia turns strokes into fills once per tile, etc.

- https://bugzilla.mozilla.org/show_bug.cgi?id=1624304
- https://bugzilla.mozilla.org/show_bug.cgi?id=1519622
-

Recording blobs on the content side is slow

- https://bugzilla.mozilla.org/show_bug.cgi?id=1557069
- https://bugzilla.mozilla.org/show_bug.cgi?id=1663387
-

SVG filters

Some SVG filters are implemented in WebRender, some are not. Running the ones that are not in the blob fallback is very slow.

- https://bugzilla.mozilla.org/show_bug.cgi?id=1552405
- https://bugzilla.mozilla.org/show_bug.cgi?id=1566942
- https://bugzilla.mozilla.org/show_bug.cgi?id=1568027
- https://bugzilla.mozilla.org/show_bug.cgi?id=1578964
- https://bugzilla.mozilla.org/show_bug.cgi?id=1607890
- https://bugzilla.mozilla.org/show_bug.cgi?id=1551733

Large amounts of blobs (blob splitting/layerization)

If there are many SVGs or SVGs are split into many blobs (because of animations or other reasons), we can end up with a large amount of blobs. This can cause a lot of overdraw which slows down compositing and also a lot of individual texture uploads.

- https://bugzilla.mozilla.org/show_bug.cgi?id=1637580
- https://bugzilla.mozilla.org/show_bug.cgi?id=1662297
- https://bugzilla.mozilla.org/show_bug.cgi?id=1477371
- https://bugzilla.mozilla.org/show_bug.cgi?id=1594768
- https://bugzilla.mozilla.org/show_bug.cgi?id=1514047

Rayon consumes CPU

Rayon is the worker thread pool we use to dispatch blob rasterization on multiple threads. The way rayon is implemented is inefficient at waking up the worker threads each time a job is posted or split, to a point that sometimes we spend more time in rayon's glue than doing actual rasterization work.

Edit: This problem used to be much worse a year ago than it is today. The glue code still shows up in profiles but I haven't seen it overwhelm the actual useful work as much lately.

Huge blob images

Mostly related to bad decisions during display list building, often because a very large element is rasterized including the clipped out part even though only a small part is visible.
Tends to magnify all of the other issues.

- https://bugzilla.mozilla.org/show_bug.cgi?id=1621532
- https://bugzilla.mozilla.org/show_bug.cgi?id=1623634
- https://bugzilla.mozilla.org/show_bug.cgi?id=1493466
- https://bugzilla.mozilla.org/show_bug.cgi?id=1644514
-

Slow texture uploads

Texture uploads are slow on many systems. Rasterization on the CPU means we have to get the result across to the GPU. Which can be slow. For some systems the cost of texture uploads is more of a driver overhead per upload than relative to the amount of uploaded data.

- All "Huge blob images" and "Large amounts of blobs" bugs are affected by slow texture uploads to varying extents depending on drivers.

This will always be a potential bottleneck, however bug 1681310, bug 1690247 and which landed in February improve upload performance on windows (from 3x to 20x depending on how silly your drivers are).

JeffM is currently investigating further upload performance improvements.

Proposition

What I propose to improve SVG performance is split into two independent pieces:

Blob webrenderization

Today there are many things that we could draw in WR but end up always rendering in blobs. For example if an SVG element only contains rectangles and text it could be rendered in WR but is currently entirely rendered in WR (we make decisions based on whether we are in an SVG rather than based on what we are actually drawing). Even if an SVG element contains more complex primitives, being able to do in WR the leading and trailing primitives that can be rendered that way would help reduce the size and overdraw related to blobs.

A corollary to this problem is that the way we choose what goes into a blob and what is drawn by WR is simplistic and not flexible. So we can run into cases where we alternate between blob and non-blob content, leading to creating a lot of blobs stacked on top of one another and in extreme cases that hurts performance spectacularly by producing a lot of overdraw. This is mostly orthogonal to how we render SVG items. It's more of a layerization issue and skia-gl wouldn't change it, so it should be addressed regardless of what we chose to do to render complex vector graphics.

At the display list building level I want to have more flexibility to decide whether an item should go into a blob or not. This decision would take into account:

- can we render the item without a blob at all,
- are we already pushing items to a blob,
- are there many blobs already,
- how big are the bounds of the item.

Once we have support for choosing whether an item goes in a blob or not, we can allow a lot of primitives to be drawn by webrender when it doesn't split blobs (or not up to a reasonable threshold). images, text, rectangles, circles, etc.

Ideally at this point we have moved a lot of content that is currently drawn via blobs even if all of it could have been done in WR, but we still often fallback to blobs because filled and stroked paths are common (especially fills).

Blob layerization

Potential improvements to blob layerization, the goal is to avoid creating too many overlapping blob layers which cause excess rasterization and texture uploads.

- Not layerize all of the children of a transform until we get to one that is needed to be active.
- For transforms we don't need to make the transform active if one of its children is active
    - This helps with testcases which have “inactive, transform(inactive, active)”, because we’d be able to merge the two inactive layers, and have the active layer on top
- Take overlapping into account when splitting blobs (probably harder to get right)
- A pref to not layerize unless we really need to (for investigation purposes)
    - Chrome doesn't always layerize for willchange
- General rule: want to layerize until there is overlap
- Layerizing less would help creative cluster.lu

Path filling in WebRender

Nical has a prototype at https://github.com/nical/misc/tree/master/tiling

WebRender already has good support for applying solid/image/gradient patterns on any kind of arbitrary masks but it can't generate the masks for filled and stroke paths.

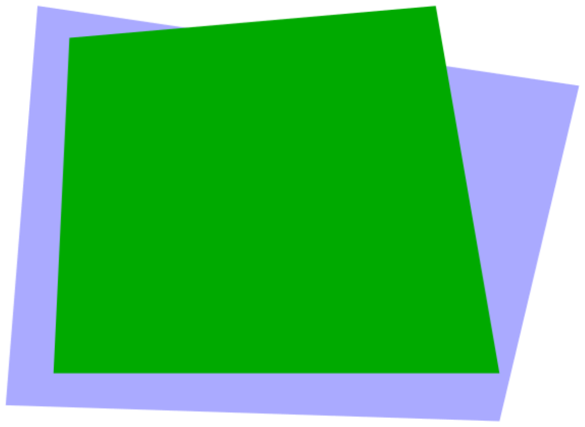
Path filling in WebRener would consist in:
 - Take all of the paths that are children of the same animated transform (important later).
 - Segment the path into a grid of small tiles (32x32 or 16x16 pixels). I have written code that does that, it's quite simple and fast (see “How does tiling works” below).
 - Discard all tiles that aren't covered at all.
 - Remember all tiles that are fully covered (especially if they have an opaque pattern).
 - Discard tiles that are under fully covered opaque tiles from a path above.
 - Rasterize alpha masks for the remaining partially covered tiles on the CPU.
 - Upload these small tiles in a texture atlas in the texture cache.
 - Render the tiles with a few draw calls using WebRender's existing shaders (one draw call for all opaque tiles, and one draw call per sequence of mask tiles, but at this point we are already in webrender's normal batching code, no changes here).

If the paths are affected by a complex clip, we can use a shader with two mask samplers. Supporting different types of patterns does't change from how WebRender works in most cases: patterns are rasterized into the texture cache and masked using an image shader.

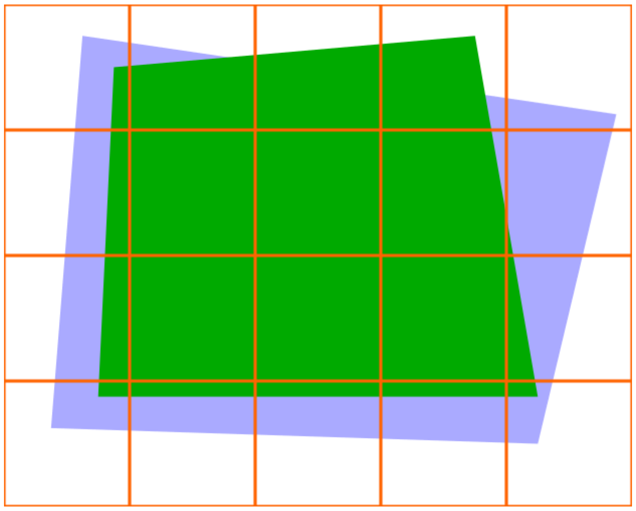
The motivations behind this approach are:

- Doing this lets us keep a very simple implementation for the mask rasterization routine on the CPU which is the only complicated part (if we are feeling fancy later, it will be very easy to move the mask rasterization to the GPU without changing the rest).
- It's faster than rasterizing stacks of large paths on the CPU because we only rasterize/upload tiles along the edges.
- It's also a much more efficient way to segment into tiles than how the blob code replays each path on all tiles that intersect their bounds.
- In addition, this integrates with how webrender does occlusion culling letting large opaque portions of the SVG occlude non-SVG web content behind it. Every time we have moved content into WebRender's opaque pass it has produced good performance improvement.
- For very large shapes and masks, optimizing out fully opaque and fully transparent tiles would be a massive win in GPU memory usage, uploaded data and overdraw. Bug 1623634 is an example of a bad case where this would help a lot.

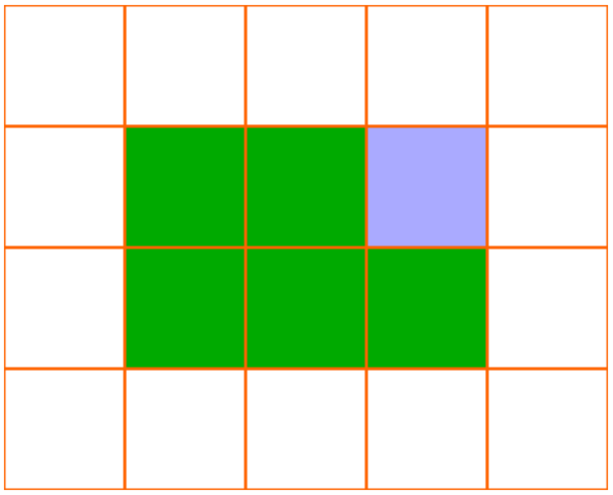
Let’s take for example two simple paths in the image below:



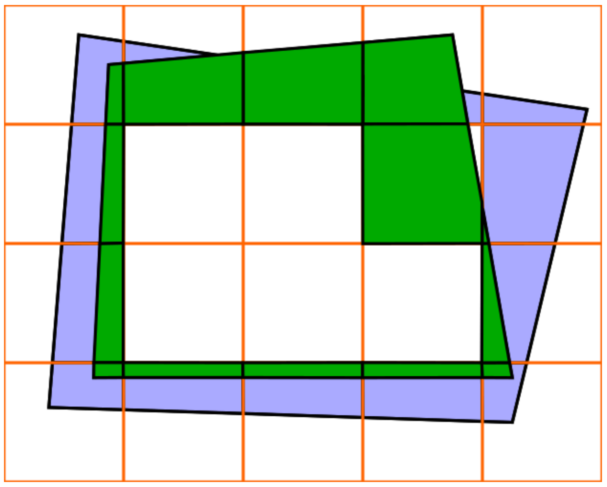
Split into tiles on the CPU as follows:



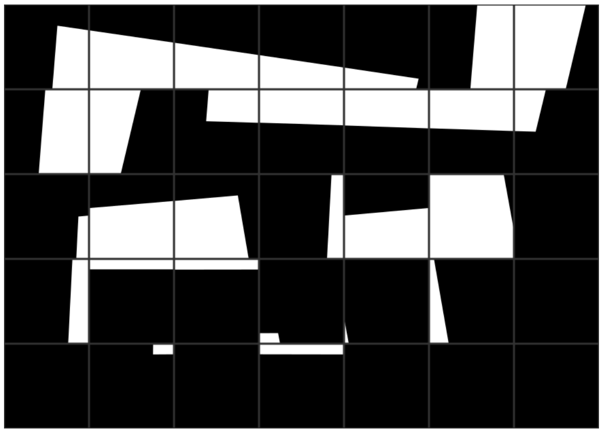
Some of the tiles are fully covered (and opaque), they can be rendered without a mask. If they are opaque they can go in the opaque pass and we can discard content behind them like 5 of the purple tiles behind the green ones:



We still have to deal with the partially covered parts:

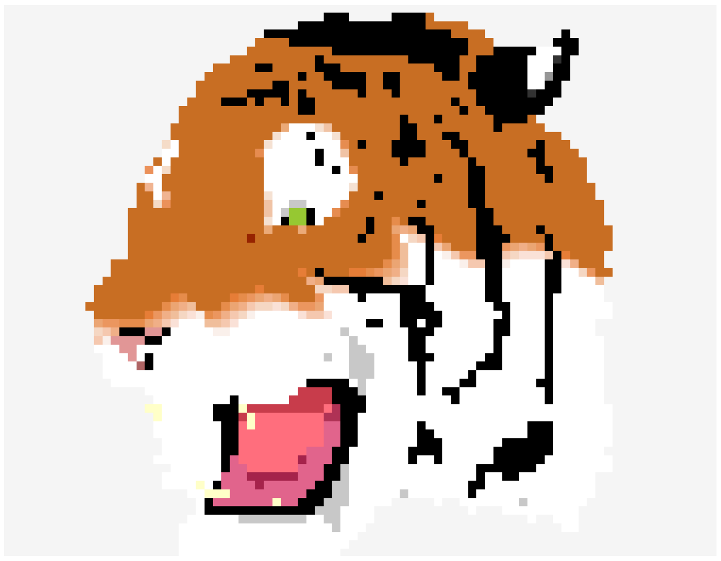


Masks that are not occluded by opaque tiles are rasterized into an atlas on the CPU:



The masked tiles are rendered in the alpha pass as we do for other primitives in WebRender.

Now with a more real-world example, here is the opaque tile pass when rendering the ghostscript tiger at 1080p with 16x16 tiles, before rendering any of the masked parts:



Then the alpha pass adds the masked parts (on the left). Masked parts only on the right. Notice how little of the actual drawing requires masking. Also note that all of the black area inside and outside of the drawing is very cheap to render with no overdraw.

Stroking paths

The simple solution (what skia-gl does in most cases), is to turn strokes into fills. It's not very hard. We can salvage pathfinder's implementation.

How does tiling work

There are different algorithms to break a vector path into a grid of smaller paths tiles.
We want vector paths as input and as output we want a grid of tiles, each tile containing the winding number at the top-left corner and the list of tiles that intersect the tile (optionally with a threshold around the tile).

There's one I have implemented here:
https://github.com/nical/misc/blob/master/tiling/src/lib.rs#L129

It works by first accumulating segments into rows of the tile grid, then for each row, sorting the segments from left to right, and assigning segments and winding numbers to tiles within the row via a left-to-right traversal.

This is pretty straightforward and fast. It is easy to speed up with SIMD and the per-row work is embarrassingly parallel, although I think it's already fast enough that we don't need to bother.
There is also an algorithm described in the *Random Access Rendering of General Vector Graphics* paper http://hhoppe.com/ravg.pdf (Patrick eventually implemented this one in pathfinder).

Putting it together

The path filling work and display list building changes can happen independently. path filling for CSS clip paths for example doesn't need changes to display list building (and would address a number of performance issues on file). The display list building changes would address blob layerization issues that we currently have and would continue having regardless of how SVG content is rasterized.
With the two together I think that most of the SVG content can be rendered entirely in WebRender. There isn't that much in SVG that isn't already implemented in WebRender outside of path and stroke filling.

We would get a nice win out rendering text in WebRender instead of blobs from not having fonts in both WebRender and skia’s font cache.

Followup improvement #1: rasterizing masks on the GPU

A natural evolution of this scheme is to rasterize mask tiles on the GPU (the tile decomposition still happening on the CPU).

I suspect that rasterizing masks on the CPU will be enough and that we’ll have more important things to do than move rasterization to the GPU. I’m mentioning it here to illustrate that the tiling approach lends itself to simple incremental improvements.

After tile decomposition each tile is only affected by a few edges (especially with very small tiles like 16x16). A brute-force approach of evaluating the winding number at each pixel by visiting all edges of the tile in a loop performs well as shown by piet, pathfinder and the RAVG paper. Note that the tile decomposition already computes the winding number at the top-left corner of the tile, so only edges that intersect the tile are needed (not the edges that are further on the left of the tile). In pseudo code the rasterization shader looks like

```
winding_number = tile.top_left_winding
for each edge in tile.edges:
     winding_number += coverage(edge, position) * edge_sign(edge)

mask_color = fill_rule(winding_number)
```

The `coverage` function returns 0 if the edge doesn’t intersect the pixel’s row or if it is on the right of the pixel, returns 1 if it is on the left of the pixel, and a value between 0 and 1 depending on coverage when the pixel is close to the edge.

The `fill_rule` function computes the in/out based on the integer part of the winding number, and shades based on the fractional part (which represents coverage.

Pathfinder’s implementation:
https://github.com/pcwalton/pathfinder/blob/compute/shaders/fill.cs.glsl#L51

Jeffm will have noticed by now that this simple shader does not deal with conflation artifacts from coincident edges. This can be dealt with reasonably easily during the tile decomposition when we traverse edges left-to right on the CPU. At this stage it’s easy to detect coincident edges and coalesce them.

Followup improvement #2: single-pass compositing

Another optional improvement that is independent from the first one and also illustrates the flexibility of the tiling approach.
Instead of emitting a quad per masked tile for each path, we can have a single quad for each tile that renders all paths at once. This is the approach taken by piet. The shader simply loops over all masks affecting the tile and composites them in one pass, emitting a single write to the framebuffer.
This requires that the masks for that tile are on the same texture and the patterns are also on the same texture. This is similar to batching requirements and where the requirement cannot be met, the behavior is the same as batching: we can still group consecutive mask tiles and split into a separate draw call wherever needed.

Concerns

Display item overhead concerns

WebRender has a high overhead per display item. This is a problem that affects all web content, addressing it is arguably as important as improving SVG rasterization. That said, SVG primitives don't need 1-1 mapping to WR display items. All we need is a representation that is opaque to WebRender. We could:
 - Keep a blob-like recording format that contains multiple items and simply isn't opaque to WebRender.
 - Group SVG items into separate pipelines (like we do for videos). In order to re-build the SVG's display list independently from the rest. We could also just do that for big SVG, and we could also do that for non-SVG content. It's not an SVG problem and its solution could benefit the rest as well.
 - Improve WebRender's display list format. Display list building happens very frequently. Any improvement we do there will have a big impact. This on its own could be a few well placed optimizations or a large project, It would make sense to not block SVG improvements on it, so we have the other two solutions to work with in the meantime, but it's time well spent in any case.

Blob splitting concerns

Splitting blobs so that some of it is handled in WebRender can cause an explosion of the number of blobs, which in bad cases produces massive overdraw and kills performance.
If more SVG primitives are handled by WebRender, we could cause more blob splitting.

This is a layerization problem that would be solved by the display list building changes I propose at the beginning of this document. It is already a big issue and needs to be fixed independently of how we render SVGs (the work would need to be put even if we used an external renderer like skia-gl).
WebRender being able to rasterize some SVG content doesn't imply all paths must be filled by WebRender. It only means we can choose whatever option is fastest. Thankfully it's easy to decide:
 - Anything small can stay in a blob if it avoids blob splitting. SVG content that is small and can be rendered in WebRender without splitting a blob should be done in WebRender.
 - Very large paths should be handled in WebRender even if they cause blob splitting.
 - for anything in between, favor webrender at the cost of splitting up to a certain total number of blobs and after that threshold force content to be in a software blob.

Note that this is a problem caused by having two separate renderers (WebRender and blobs). When most SVG content can be done in webrender, the problem goes away. However, if we use a faster but still opaque way to render blobs like skia-gl, then the problem continues to exist and the splitting continues to be a problem in the long term.

Concerns about the performance of the proposed path filing solution

If there are concerns here I would love to hear them. I think that integrating with WebRender's occlusion culling and batching scheme is a very strong advantage. The tiling scheme would greatly reduce the amount of rasterization happening on the CPU for large SVGs where rasterization matters, and would reduce the amount of texture uploads as well in the common case. This is to say I am confident it would be a big improvement over what we currently have.

It may be possible to get even better path rasterization performance with skia-gl. If this is a concern, I believe that we can beat skia-gl at path rasterization with further improvements, by doing the tile rasterization in a compute shader. There is open-source prior art for this, it's well understood and not very complicated. I don't think that it's worth looking into for now.

Concerns about memory usage

Generating all mask tiles on the CPU before rendering means the amount of memory required for mask tiles depends on the amount of paths that will be rasterized. There are concerns about not having a limit to this, as opposed to, say, generating masks until we reach a certain limit, compositing using the masks, then going back to generating masks reusing the previous mask texture memory.

This is a valid concern, experience with pathfinder shows that even with complex SVGs the area occupied by masks tends to be much smaller than the overall drawing. This is because for each path we need only to generate masks for non-occluded tiles along the path which is typically a very small fraction of the bounds of the path.

That said there can definitely be SVGs that are adversarial to this method and would generate an unreasonably large amount of mask tiles. This is similar to a page containing a very large amount of different glyphs all visible at the same time, or bad blob splitting cases as we have today.

I think these bad cases are going to be rare enough that we'll have more important issues to fix. I have a prototype implementation of the tiling algorithm which we could use to validate this assumption on real-world SVGs which we worry might cause issues.

One way to fix this problem entirely is to move the tile rasterization to the GPU. This way we can generate as many tiles as fits in a fixed budget, do some compositing, rinse and repeat.

Some numbers collected from the tiler prototype on a few test cases with different tile sizes. The percentage of mask pixels rasterized (100% means the amount of pixels of the image flattened into one layer), The number of bytes is just the number of pixels divided by four to reflect that we only need to rasterize and upload the alpha channel. For the tiger which has a lot of opaque paths, this shows the amount of culled out mask tile pixels and solid tile pixels (also compared to the total surface of the image):

https://bug1682398.bmoattachments.org/attachment.cgi?id=9193391

- 8x8 tiles: 68.32% px (17% bytes)
- 16x16 tiles: 135.86% px (34% bytes)
- 32x32 tiles: 278.64% px (70% bytes)

The ghostscript tiger

- 8x8 tiles: 74.26% px (18.56% bytes), culled: masks: 16.35%, culled solid: 108.24%
- 16x16 tiles: 145.95% px (36.48% bytes), culled masks: 25.63%, culled solid: 85.93%
- 32x32 tiles: 293.55% px (73.39% bytes), culled masks 35.14%, culled solid: 49.81%

Skia-gl would fix most of our SVG performance issues

I don't believe this to be true.

skia-gl would only help with performance when the time is spent rasterizing and hopefully texture uploads if skiaGL knows the ANGLE dances. We would still have performance issues related to:
- Splitting/layerization discussed earlier.
- Overdraw in general which can come from the splitting issue but also simply having large SVGs in general, and affects compositing every frame.
- Blob recording performance.
- Blob replay performance on multiple tiles. We can address that just like we can address it for the current implementation, but it doesn’t come for free.
- Filters. The implementation work for filters is orthogonal, it’s already in WebRender.

Concerns about the amount of implementation work

On the WebRender side there isn't a lot of work to do to support path filling the way I described. All of the pieces are simple, well understood and map well to WebRender's drawing model. It's not trivial but not unreasonable.

On the display list building side, having better infrastructure to choose what goes into blobs is in my opinion more work than what needs to happen on the WebRender side. It's also work that needs to happen regardless of how SVG items need to be rasterized, because of the blob layerization/splitting issues.

Vertex shader overhead

Rasterizing many small tiles means rendering many small rectangles. If we use the brush shader infrastructure it means we will render a lot of vertices via some rather heavyweight vertex shaders. This may be an issue.

The right thing to do here is to measure when we get there, and if need be spend some time optimizing the vertex shaders, or cache the rendered paths in larger targets (losing fine grained opacity information, or have some specialized shaders)

skia-gl would be less work

I don't believe this to be true.

We would need changes to the current blob rasterization infrastructure to support skia-gl. We currently rasterize blobs into small tiles which would be very inefficient for skia-gl. Rasterizing large tiles means it wouldn't work as well with WebRender's texture cache, so we would have to do something to avoid batching issues. Large tiles means either very coarse invalidation (it's per-tile), or implement a new way to track invalidation for blobs.

Letting skia manipulate WebRender's GL context would be a recipe for disaster. I think that slow SVG with skia software is preferable to dealing with that. We could let skia render into another context and use texture sharing. On some platforms switching between GL context causes massive driver stalls, perhaps we could just let these configurations fall back to the skia software blob path. Texture sharing as we all know is not trivial and not bug-free driver side setting that up neads work and testing.

skia-gl on windows means ANGLE. Just with the simple things we do in WebRender ANGLE is already a pretty sizable maintenance burden. We spend time figuring out what it does and how to work around it. It's already complicated but we control what we ask ANGLE so it's doable. Now throw something we don't have control over like skia-gl in between and the puzzles become much harder to solve.

When we used it for canvas, skia-gl was not bug-free. Bugs in skia-gl were complicated to investigate and fixes were complicated to upstream. Lee had to deal with skia-gl bugs a lot of the times he updated it.

A lot of the points here are maintenance concerns so it could be argued that if we ignore maintenance burden it would be less work to do the initial implementation with skia-gl. Compared to what though? Compared to the proposed path filling and display building changes? Maybe. However the displaylist building changes mean to address issues that would still exist with skia-gl and should still be implemented if we had skia-gl blobs. Compared to implementing path filling in WebRender, skia-gl would require more work.

Path filling and stroking is not enough

Add your thoughts here.

Risks related to driver issues

The solution I propose for filling and stroking paths does not introduce anything we don't already do in WebRender. Everything new happens on the CPU.

skia-gl (on top of ANGLE), introduces new interactions with the GPU. In addition these are further away from our control so our main tool to work around incoming bugs is a global on/off switch to fallback to software blobs.

Orthogonal SVG performance issues

Some SVG performance issues are independent of the main approaches discussed earlier.

Rayon issues

Nical started experimenting with a new parallel job scheduler, the code is at https://github.com/nical/misc/tree/master/parasol

The general ideas behind this experiment are:

- build the scheduling around having worker threads that help a submitting thread with a parallel workload via job-stealing. This is in contrast with rayong which does not let threads outside of the thread pool help with work (so they have to got to sleep and be woken up once the work is done, which adds a lot of latency.
- Avoid as much as possible to have to put the submitting thread to sleep. This is done by letting it do work and even give it more work than the other threads.
- Avoid spinning worker threads aggressively (rayon tends to do that to avoid letting worker threads go to sleep. With this approach we add some latency but reduce CPU usage allowing the cores to do useful work on the other processes.
- Support priorities so that
- Optimize for a small amount of worker threads (typically 4 workers, 8 at most).

Texture upload overhead

There has been some big improvement to texture upload performance on windows in 2021, it is still a bottleneck, Jeff is investigating potential further improvements.

Display item overhead

Not really SVG-specific but part of the discussion. Miko is working on it and will continue in 2022.

SVG filters

WebRender currently has incomplete support for SVG filters:

There is some code for:

- Blend
- ColorMatrix
- Flood
- ComponentTransfer
- Opacity
- Composite
- Offset
- GaussianBlur
- DropShadow

But among these, the following appear to not be enabled during DL building (and fall back to blobs instead):

- Flood
- Composite
- Offset
- Blend

In addition WebRender has no code to support:

- Morphology
- Tile
- ConvolveMatrix
- DisplacementMap
- Turbulence
- Merge
- Image
- DiffuseLighting
- SpecularLighting
- ToAlpha

Some of them are rather trivial.

In addition, WebRender does not fully support primitive-subregion (equivalent to clipping after the filter, which is different from the filter region acting as a clip before the filter). This only affects filters that sample sources outside of the destination coordinate (blurs, shadows, offset, etc.).

The DisplayList building code also assumes that webrender only supports filter nodes using the previous node as input. In other words it only emits straight linear filter chains but no graphs. I think that we support proper graphs now.

Another performance issue is the cost of converting between srgb and linear color spaces.

Actions:

To reduce the cost of srgb/linear conversion, we can add support for inline srgb-linear conversion as filter input and linear-srgb at the output to avoid a full render pass at the beginning and end of most svg filter chains. This can be implemented in the current filter chain implementation in parallel with other SVG related work (but would be re-implemented as part of SVG filter graph support). It should be a low-ish hanging fruit.

For the missing filters and functionalites:

There are the existing filters that aren't plugged in. We can try to enable them and fix issues until they pass the tests (I can only assume there are issues since they aren't enabled).

For missing filters, we can add new render task types. SVG filters are currently mostly implemented in a single shader but I think it's a mistake and we should have a shader per filter both for simplicity and performance. Some filters will be very straightforward to implement and we'll probably run into limitations of the render task or cs_shader systems when implementing others but I don't expect any major roadblocks. This is very parallelizable work if several people want to contribute. We also don't *need* implement all filters, if we can figure out which ones are used most.

Some filters require a bit more than adding a shader, for example the Image one needs some plumbing to interact with the texture cache.

Primitive sub-region needs a bit of extra work at the render graph level but I don't expect we need big fundamental changes there. Needs a bit more investigation to evaluate the amount of work.

The spec is quite large so there's going to be a bunch of time spent just understanding it.

https://www.w3.org/TR/filter-effects-1