Tile Prioritization Design

vmpstr@, enne@, aelias@, epenner@, et al

Abstract

This document outlines a proposal to redesign tile prioritization work in order to improve performance of scheduling raster work. The areas of focus are the following:

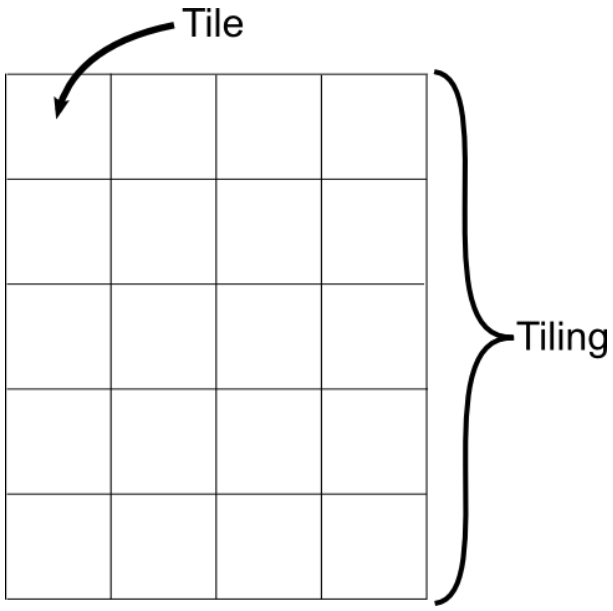
- Updating tile priorities. Current implementation is O(n) where n is the number of (theoretical) tiles created in the layer. The proposed solution is a constant improvement. That is, the proposal is still a linear algorithm but it acts on fewer tiles. It also does less work per tile.
- Identifying tiles that need to be rasterized next. Current implementation is O(n log n) where n is the number of tiles. This is due to the fact that tiles are sorted based on their calculated priorities before being processed. Furthermore, processing the tiles visits each of the tile. The proposed solution achieves the same result in O(n) where n is the amount of tiles. The constant of the proposed solution is also small, since the number of tiles to be visited is limited. As a parallel improvement, memory consumption of static pages should also be improved.
- Identifying tiles that need to be evicted. In the current implementation, this step happens while identifying tiles that need to be rasterized. That is, tiles that are processed after we reach our limit on memory are evicted. However, this step should be treated separately since the decision to evict can be done independently from identifying raster work.

tl;dr Solution Overview:

- It's too much work to give TileManager the universe of all prioritized tiles to make good global decisions about rasterization and evictions.
- To address this, generally move more responsibility to layers and tilings to make local decisions about what should be rastered and what should be evicted so that TileManager is then only responsible for choosing from these candidate lists when rastering or evicting.
- To do that, each layer only needs to choose and prioritize some fixed number of candidate tiles for eviction and rasterization each frame. The vast majority of tiles that aren't important enough to be rastered but too important to evict no longer need to be prioritized each frame.
- Additionally, if the layer and tiling are responsible for providing candidates for rasterization, then they can also greatly reduce the number of tiles that they create in non-ideal scenarios by not creating candidates that aren't provided.

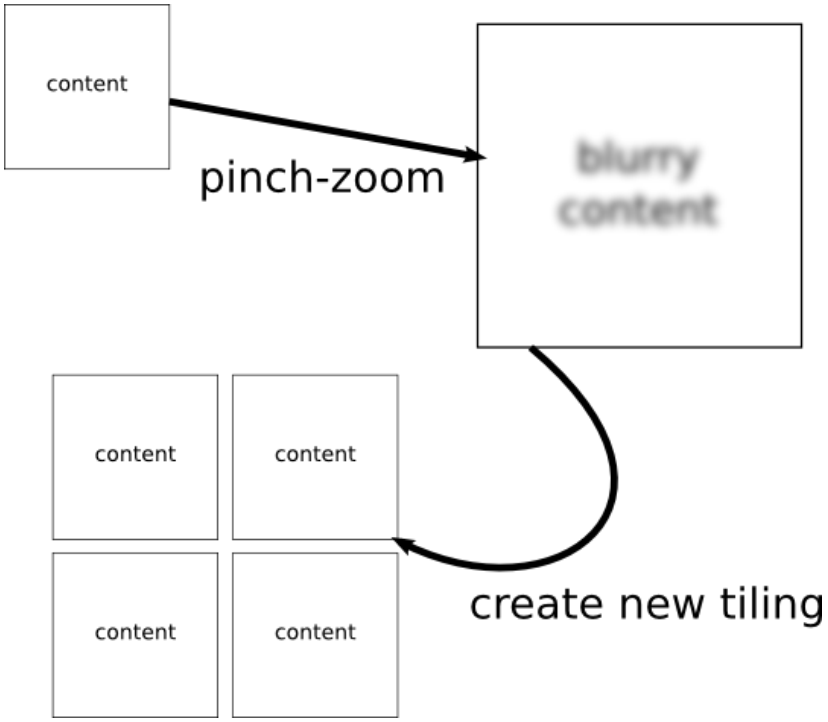
Terminology and general background

The impl-side painting system is designed to operate on *tiles* as the basic unit of rasterization. In the compositor code this is represented by the class `Tile`. The tiles are organized into a regular grid on a *tiling*:



Tilings in the compositor code are represented by the class `PictureLayerTiling`. Upon rasterization each tile gains a resource that contains the content of that tile within it. Note that it is important to keep in mind that these tiles do not necessarily map 1-to-1 to screen pixels. That is, each tiling has another property called *scale.* The scale of the tiling determines its mapping from tiling *content space* to *layer space*. That is, each unit of measurement on the tiling maps to (1.0 / content scale) units of measurement on a layer.

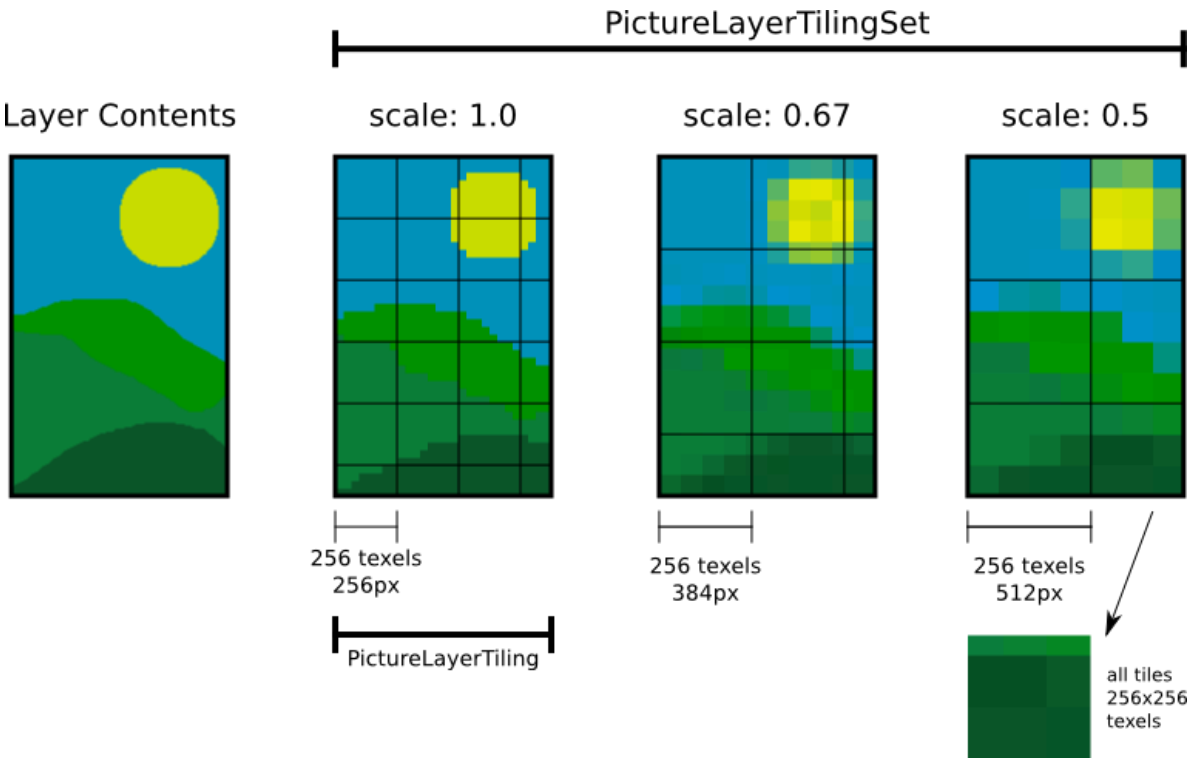
The tilings can be created or destroyed depending on user interaction. For instance, if the user pinch-zooms in, a new tiling will be created at a larger scale so that each resource will appear sharp on the user's screen:



A tiling is usually deleted when it is no longer required. That is, if we didn't use a single tile from a tiling to draw a frame on the screen, then that tiling is typically destroyed. Thus each layer can have a collection of tilings, at different scales, which are contained in the class `PictureLayerTilingSet`. Each of the layers that at some level generates tiles has a single `PictureLayerTilingSet`. Note that these layers are implemented in `PictureLayer` and `PictureLayerImpl` classes.

The system always strives to have the ideal tiling on the screen: A tiling at such a scale that one pixel in the tile resource would map to one pixel on the device screen. This tiling is referred to as the *high resolution tiling* or *ideal tiling*. Note that we also keep a *low resolution tiling* around. This tiling has a scale that is 0.25 times the high resolution tiling scale. That is, one tile in the low resolution tiling covers the same area as 16 (4x4) high resolution tiles. We use this tiling for getting content on screen quickly. We can generate one of these tiles fairly quickly, and it would cover a large area. Although the content will appear very blurry, it is deemed better than checkerboarding. Low resolution tiling is also special in the sense that it is not deleted if a tile was not used to draw on screen. The reason for this is that it might contain tiles outside of the visible area, viewport, of the screen. When the user scrolls or flings around, the low resolution tiles are likely to be displayed before the high resolution content is rasterized.

If we transform each tiling into layer space, the picture looks as follows:



If a region of the page changes, the corresponding tiles in those regions are *invalidated* in each tiling. An invalidation causes a copy of the layer tree to be created called the *pending tree*. By contrast, the layer tree that is displayed on screen is called the *active tree*. Tiles that were not invalidated are shared between the tilings on the pending and the active trees. For convenience, when referring to a tiling that is on the other tree but has the same scale, we refer to it as a *twin tiling*.[1]

Background on the old implementation

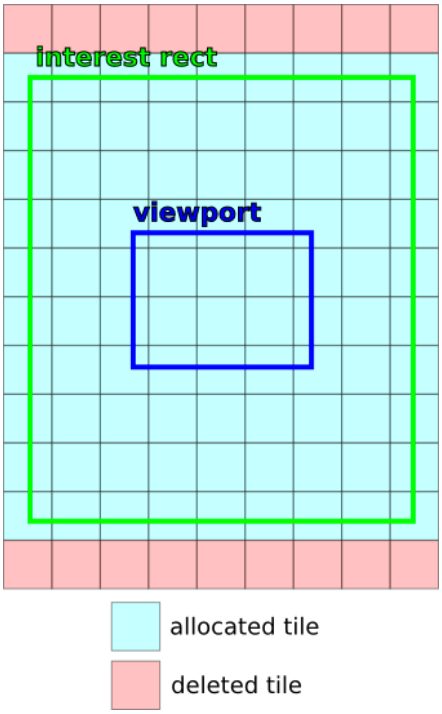
Now that we are familiar with the general structure of the code and the terminology used, we can talk about the implementation details of the current system.

Updating tile priorities

A tiling that is created by a PictureLayerImpl initially contains no tiles. It consists only of a placeholder grid where tiles would be populated. This remains so until we update the tile priorities. As a first step, we calculate an *interest rect*, which is essentially an inflated viewport.

[1] Note that some of the discussion in this document omits the pending tree for simplicity.

All tiles within the interest rect are created.[2] At the same time any tile outside of the interest rect is deleted[3]:



Tiles that are allocated are created via *tile manager*, implemented in the class `TileManager`. Although tiles are ref counted on each of the layers, it is sometimes convenient to think of the tile manager as the ultimate owner of the tiles. That is, every tile that is created within the system is registered with the tile manager, and the tile manager keeps a reference to each of the tiles that is created.

The second step in update tile priorities is to iterate over all of the tiles in the interest rect and calculate two values that comprise the priority of a tile:

- distance_to_visible_in_pixels: This value is a manhattan distance from the edge of a tile to the viewport.
- time_to_visible_in_seconds: This value is an estimated time when the tile will enter the viewport, calculated using the current viewport, the last frame's viewport, and the time difference between the two frames.

This concludes the process of updating tile priorities. When it is done, tile manager is requested to perform its work via `ManageTiles()` call.

---

[2] Note that some of tiles will not necessarily be allocated, they can be retrieved from the twin tiling.
[3] Tile deletion simply informs the tile manager that the tile is no longer required, which is freed and destroyed at the tile manager's discretion

Managing tiles and scheduling raster work

TileManager performs the rest of the work needed to get the tile rasterized and ready to be displayed on screen. In particular, this involves identifying which tiles are required for rasterization and the order in which they should be rasterized. This also involves creating *raster tasks*, any supplementary tasks that are needed, such as image decode tasks, and handing off a set of prioritized tasks to the raster worker pool.

TileManager has a reference to every tile in existence, and at the time ManageTiles is called, each of those tiles had its priority updated. So the process is straightforward (right?)

We iterate over each of the tile to determine its *bin*, and *unified priority*. The bin calculation is fairly complicated and is evolving frequently, but it boils down to the following bins[4]:

- NOW_BIN: the tile is currently in the viewport; this is the most urgent bin.
- SOON_BIN: the tile is close to the viewport, or the time to visible is short as determined by a constant; these tiles should be rasterized after NOW_BIN tiles are rasterized.
- EVENTUALLY_BIN: these tiles will eventually be visible; they are somewhere in the interest rect but have low priority.
- AT_LAST_BIN: this is the lowest priority bin, which can be used for tiles that aren't required but shouldn't be freed.
- NEVER_BIN: this tile will not be needed and should be relieved of all resources.[5]

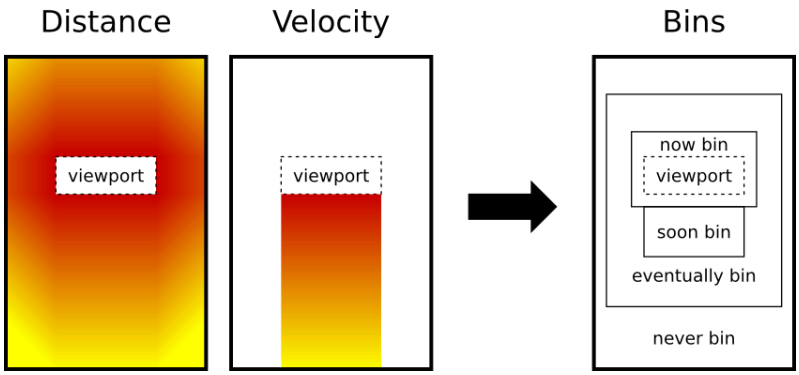
As the tiles are assigned bins, they are inserted into a *prioritized tile set*, which is an iterable container responsible for returning tiles in unified priority order.

Unified priority is a result of tiles being shared in the pending and active layers. Since these layers can have different transforms (e.g. pending layer is scrolled further and active layer), each tile results with two priorities: pending priority and active priority. Based on the current requirements of the system as specified by tree priority, we combine the two priorities into one priority, the unified priority. Tree priority determines whether we should prioritize active tree, pending tree, or if they are equivalent.

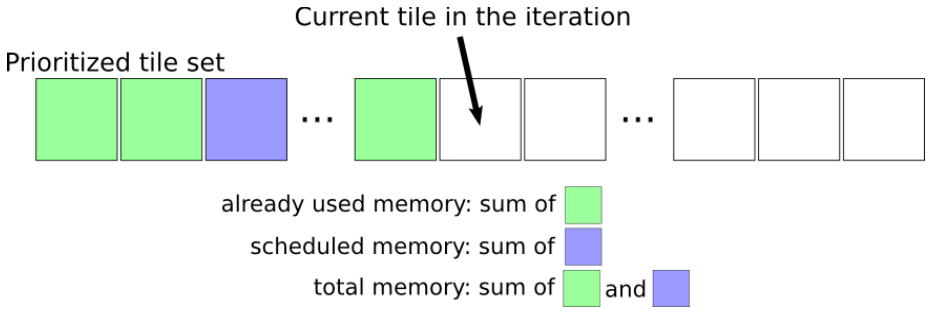
---

[4] For completeness, the total set of bins is as follows: NOW_AND_READY_TO_DRAW_BIN, NOW_BIN, SOON_BIN, EVENTUALLY_AND_ACTIVE_BIN, EVENTUALLY_BIN, AT_LAST_AND_ACTIVE_BIN, AT_LAST_BIN, NEVER_BIN.

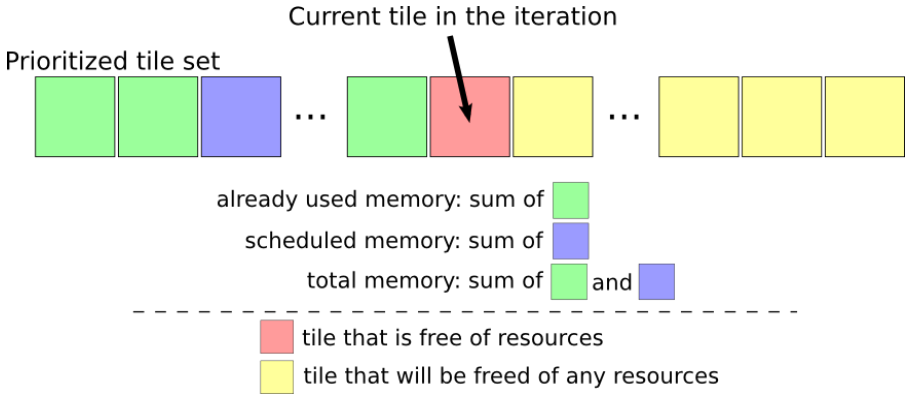
[5] This bin is used in the cases where a tile has an active priority, but not a pending priority. In this case the pending priority results in a NEVER_BIN value. However, the overall tile bin is promoted to AT_LAST_BIN if the twin bin is not NEVER_BIN.



The second pass through the tiles iterates over the prioritized tile set. This process is called *assigning GPU memory*, since that is the task it performs. For each tile in the iteration, regardless of whether it currently contains a resource, we tally up both the currently used memory as well as memory we will use once the tile is scheduled. As well, we keep track of tiles that need to be rasterized in a separate vector[6].

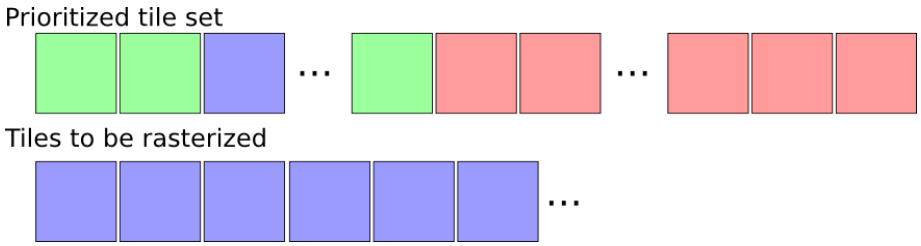


If during this iteration, we reach our memory limit as specified by the *memory policy*, we enter the *out of memory* state, or *OOM*. The iteration continues but only with intent of releasing memory from any tile that currently has memory.

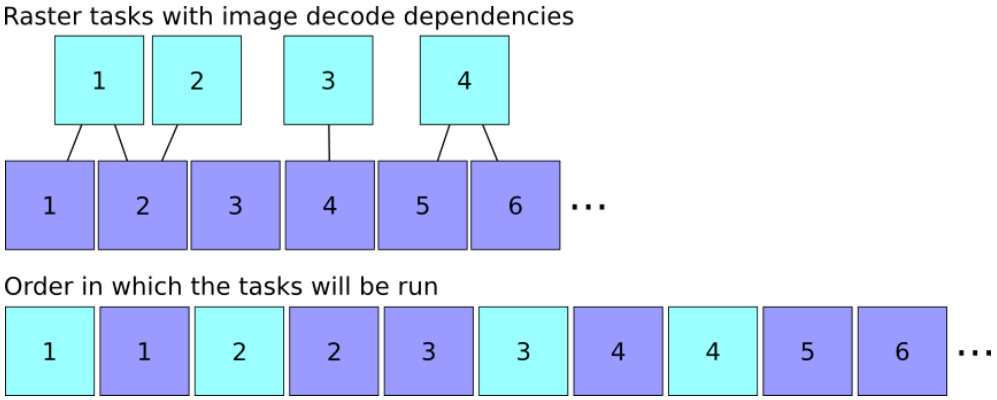


[6] For the sake of completeness, it's worth mentioning that tiles that are currently in the process of being rasterized are considered as needing to be rasterized.

After the process is done, we have a list of tiles that need rasterization in a vector. Important to note is that at this point these tiles are in a priority ordering.



Next, we construct a raster task for each of the tiles to be rasterized. Each raster task can have several dependencies, namely image decode tasks[7]. After all of the tasks and the dependencies are created, the vector is given to the raster worker pool for the work to begin. Note that in the current system, any task that was previously scheduled but is not scheduled in the next call will be cancelled by the raster worker pool.



Upon task completion, tile manager is notified that a tile was rasterized into a given resource. At this point, the only work that is required is to pass the resource to the tile. Hence, the tile becomes rasterized and ready to be drawn on screen.

Inefficiencies within the tile system

Given the current system, we can now analyze the time that we spend in each of the separate sections. In general, the idea that tilings' only responsibility is to create and update priorities, whereas TileManager's responsibility is to resolve cross-layer priorities limits our ability to perform work on only the tiles that will be rasterized. As a result, we end up in a situation where

[7] Image decodes run as separate tasks, and can be cached for other raster tasks to use. Additionally, this allows us to cancel a raster task in-between an image decode and the task actually starting to run.

TileManager has to perform a global sort via multiple passes through the tiles. It is worth it to note that the number of tiles in the system can grow to over 1,000 during rapid pinch zoom and possibly more. As an example, while idle on nytimes.com on a typical desktop machine[8], the system has approximately 150 tiles.

Updating tile priorities

The process of updating the tile priorities favors correctness over efficiency. The calculations of distance and time to viewport intercept are performed in device screen space. This means that a transform that maps a layer to the device is passed down to the tiling and is used to map a tile into device screen space before doing the calculations. This process is done independently for each tile within the interest rect.

Note that some optimizations are done here:

- Transforms that only have translations are handled separately: tile rects are scaled in order to map into device screen space
- Other transforms that do not have a perspective are also handled separately: basis vectors are computed for both current and last frame and are used to generate a screen space quad, bounds of which are used to calculate distances and velocities
- All other transforms fall into the general case: each tile rect is mapped to screen space using the transforms from the current and last frames; the resulting rect is used to calculate distances and velocities.

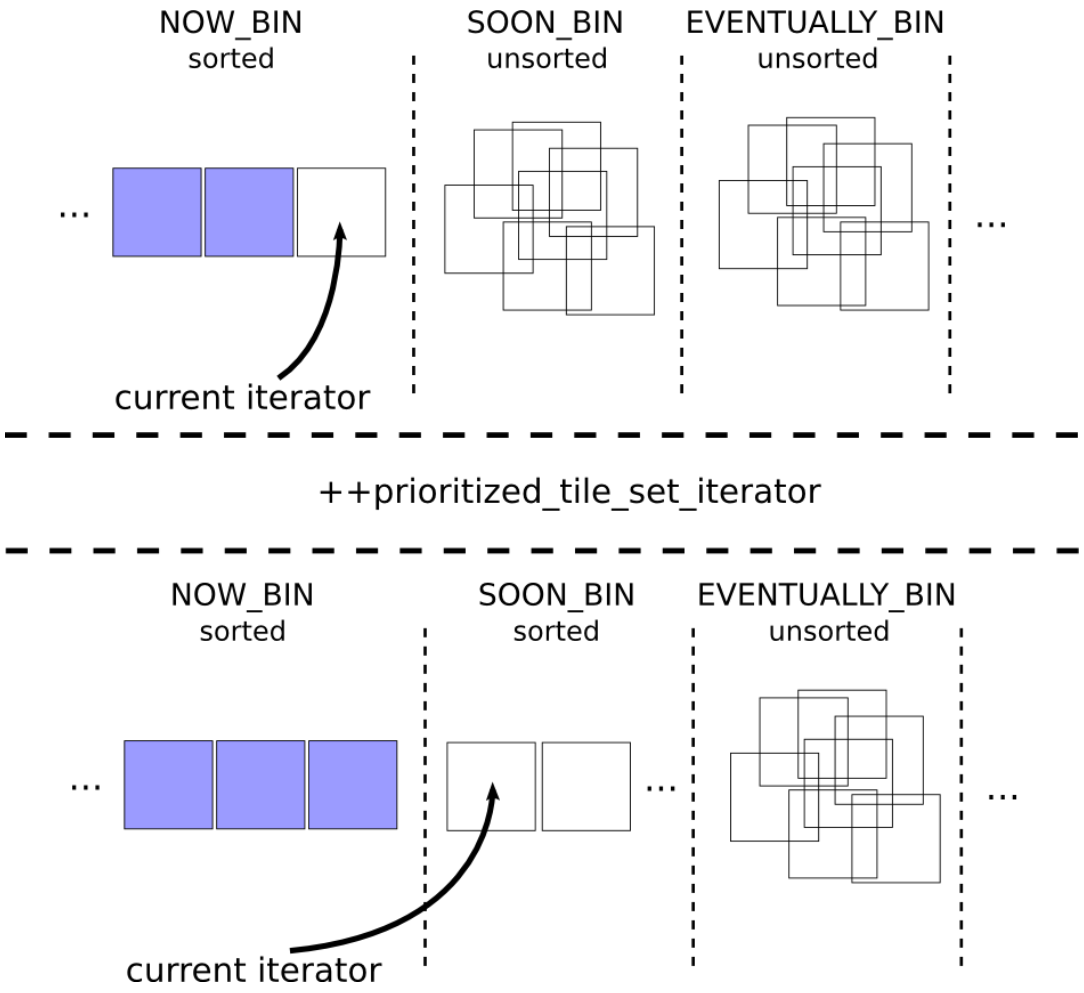
The overall complexity of updating all tiles across all tilings and layers is linear in the number of tiles that exist. Furthermore, the constant cost per tile can be high if layers have perspective transforms.

Managing tiles

Due to the current design, tile manager operates on all tiles that exist within the system. The first pass through the tiles assigns tiles a bin as well as calculates the unified priority of a tile. This is a operation is linear in the number of tiles with a high constant calculation per tile. Additionally, the process of determining the bin value is complicated and somewhat rigid. Short of introducing new bins, it is very difficult to identify a set of tiles and promote their priority to be higher than a different set. The tile manager is not in a good position to judge relative priority of tiles and has to rely on distance and time to visible calculations, as well as the containing tiling resolution, in order to determine relative ordering.

[8] The type of machine does not actually matter, but the size of the viewport and whether the display supports high DPI affects the number of tiles generated.

The second pass through the prioritized tile set is linearithmic. The reason for this is that while iterating the set, when we cross a boundary to the next bin, the next tiles in the next bin are sorted (using a comparison function that contains multiple conditionals).



The reason that bins are sorted lazily is that when we reach and out of memory situation, the iterator is flagged as not needing the sort any longer. We free the resources, or evict, the rest of the tiles so the order does not matter. This is an optimization to prevent unnecessary sort. However, in the worst case all of the tiles will eventually be sorted and processed.

It is also important to note that the current system will attempt to schedule as many tiles for rasterization as the GPU memory limit would allow. This means that we could be scheduling hundreds of tiles for rasterization although only be able to process a handful before the next frame. During the beginning of the next frame the process will repeat: tile priorities will updated, and the tile manager will perform the two passes over the tiles to determine the set of tiles to schedule.

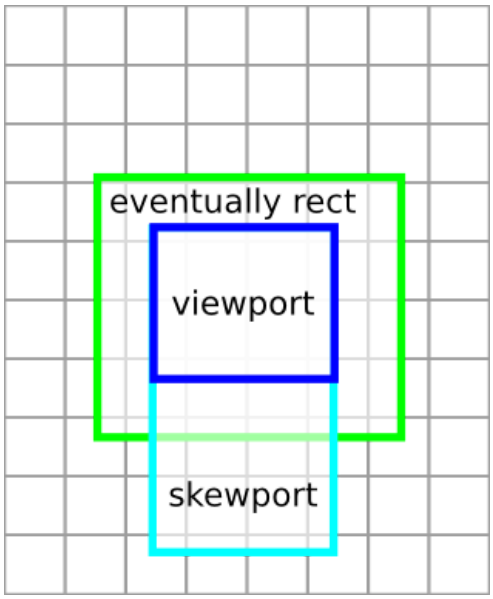
Proposed solution

The proposed solution is a redesign of the tiling system in order to retain more knowledge about required tiles, and allow for incremental updates to schedule only a handful of work for the raster worker pool. In general the idea is to shift decision on which tile is important to the layer and only have the tile manager determine which of the layers should contribute the next tile. That is, the layer will provide a set of tiles that it wants rasterized to the tile manager and the tile manager will merge different candidates from all layers. Note that the design will have to cover eviction separately, as it will no longer be a byproduct of scheduling tasks.

Updating tile priorities

Updating tile priorities will now not require an interest rect. Instead, three other regions are generated:

- Visible rect - a rect in tiling content space which contains all tiles that are visible. This is the tiling content space representation of the viewport. This is also referred to in the tiling as the viewport in content space.
- Skewport - a rect in tiling content space which contains tiles that are either visible or will become visible in the next few frames. This rect is calculated using finite differences of the current visible rect and the last frame's visible rect.
- Eventually rect - a rect in tiling content space which contains visible tiles as well as tiles in a thin border around the visible rect.



By comparison to the interest rect, these rects collectively contain fewer tiles. The reason we can afford to do this is that these rects govern the creation of tiles only. That is, tiles outside of these rects will in general retain their resources. On average, 25 tiles is enough to fill the viewport[9]. Assuming a 5x5 viewport, this means that with a static viewport and a single tile eventually border, we only generate 36 tiles. Rapid fling can, of course, generate more. Note however that tiles aren't generated unless they are next to be rasterized, which means that skewport tiles will be filled up over multiple frames. By comparison, a 3 layer nytimes.com generates approximately 150 tiles, coming mostly from one base layer[10].

Additionally, the interest rect currently generates tiles on all tilings. Hence, the number of tiles increases dramatically if the number of tilings is increased, as is the case during pinch-zoom. With the proposed solution, we can be smarter and not generate new tiles for non-ideal resolution tilings. The thought being that non-ideal resolution tilings are likely to disappear in the coming few frames.

The priority calculation now only computes the distance to the viewport, instead of distance and velocity calculations. Furthermore, to distinguish the location of tiles one of three bins is assigned to each tile:

- NOW - tiles within the viewport get this priority as the most important tiles
- SOON - tiles within the skewport but outside the viewport will get these tiles as tiles that need to be processed after the NOW bin
- EVENTUALLY - tiles within the eventually rect but outside of the skewport get this priority as something we should prepaint if we have enough time.

As update priorities takes place, tiles that do not have resources will be placed in a separate vector by bin. That is, NOW tiles are put into a now_tiles vector, and so on. These vectors will serve as sources of tiles that need to be rasterized. The SOON and EVENTUALLY vectors can be sorted lazily based on the distance to viewport. NOW vector does not have to be sorted, since the order in which tiles are processed within the viewport does not matter as much as outside of the viewport.

Note that the priority only needs to be updated for tiles that fall into one of the three rects, thus we save time updating priorities for tiles that will not be required.

Some optimization thoughts:

- We could cache when all visible/soon tiles that are ready and use that information to not look at as many tiles on the next frame
- Even if the visible rect changes, we could then only look at the difference between the old and new visible rect

[9] This varies from device to device.
[10] Two scrollbar layers are relatively small and at most contribute a handful of tiles.

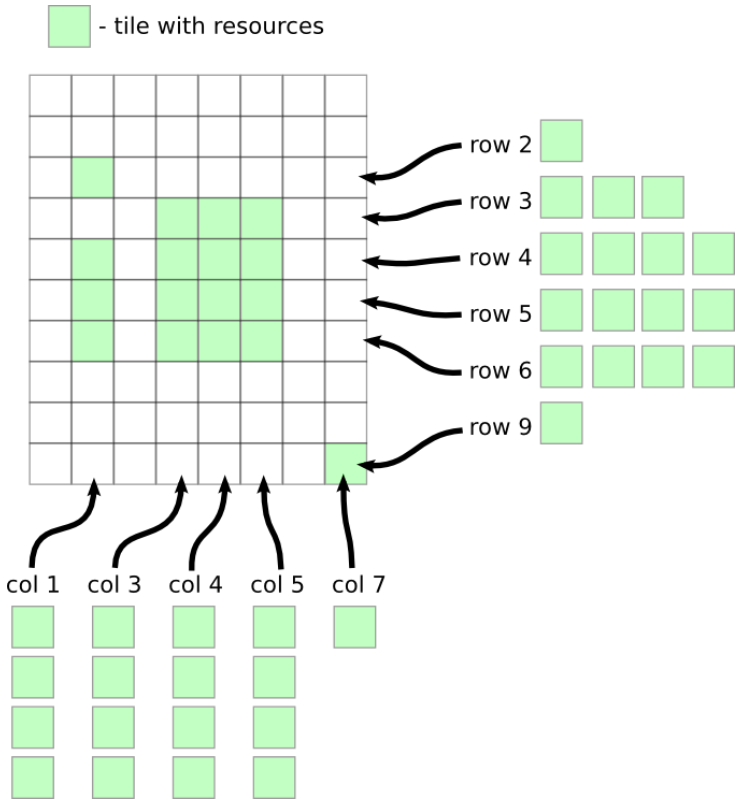
- Note that the old tiled_layer had a similar algorithm as our Eventually bin: https://code.google.com/p/chromium/codesearch#chromium/src/cc/layers/tiled_layer.cc&l=774

Eviction

Since updating tile priorities does not naturally determine which tiles should be evicted, a separate structure is needed to maintain this information.

The naive approach would be to simply iterate over all tiles that have resources and put them into a heap ordered by distance to the viewport. This heap will become the source of eviction tiles for the tiling. This approach is relatively slow, because it will require updating priorities for all tiles that have resources, but it is simple to implement. Furthermore, there should be fewer tiles with resources in this tiling than the total tiles in the interest rect that would have been iterated previously.

A more involved approach would maintain a more elaborate data structure. This structure would maintain a separate list of tiles with resources by both row and column index. These lists will be updated both when a tile gains a resource as well as when we evict tiles:

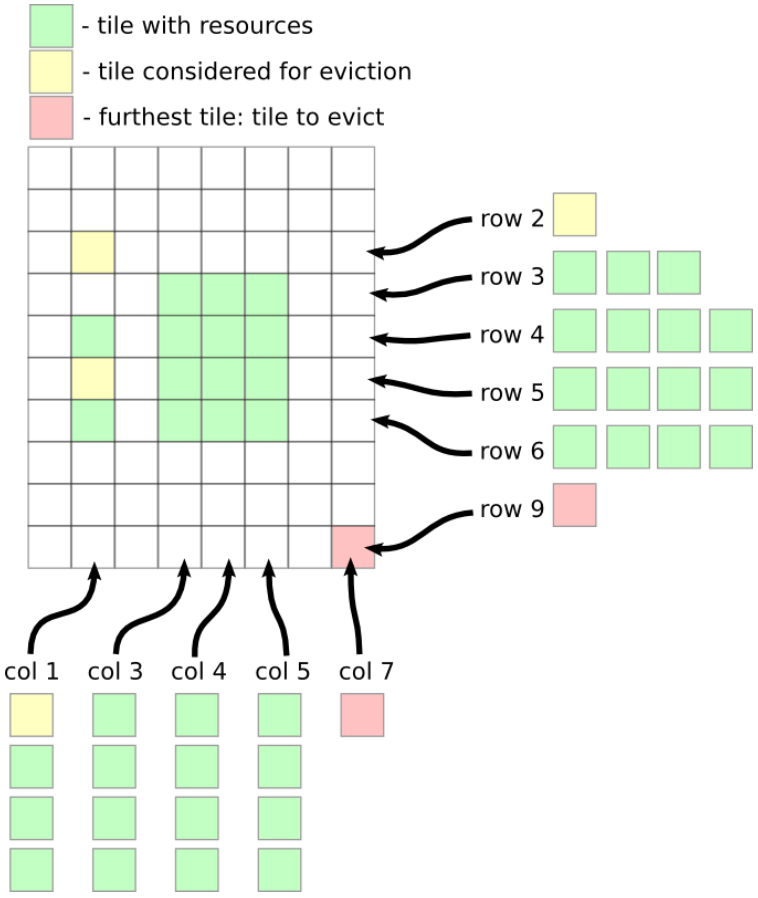


Additionally, we keep track of bound values: minimum column, maximum column, minimum row, and maximum row. In the example in the diagram, the values would be min column: 1; max column: 7; min row: 2; max row: 9.

These are linked lists that are unsorted, and can be updated in constant time when a new tile gains a resource or a tile is evicted.

Now to determine the next time to be evicted, we first ensure that the extreme values (min/max) point to non empty lists. If the lists are empty, we walk towards the other extreme (min increments to max, max decrements to min) until we find a non-empty list. Then, we need to check 4 values: first tile in min column, first tile in min row, first tile in max column, and first tile in max row. Checking involves computing the distance to the viewport and picking the one that is the furthest away. Note that this might not be strictly correct, as we'll only be picking a random tile from the furthest rows/columns, but that is sufficient to get a tile that is far enough away.

After the tile is evicted, it will be deleted and the lists can be updated to remove the tile. In the example in the diagram, this would cause col 7 and row 9 lists to be empty, but next time we need a tile for eviction, we will simply walk the index until we find a non-empty list.



Overall, this modifies the tiling to provide the following API:

- `Tile* PeekTileForRaster()`
- `void PopTileForRaster()`
- `Tile* PeekTileForEviction()`
- `void PopTileForEviction()`

This API uses the vectors and lists described in this section to return most of these values in constant time, with the exception of lazy sorting SOON and EVENTUALLY tile vectors. As well, the tiles that are returned are guaranteed to have their priorities updated.

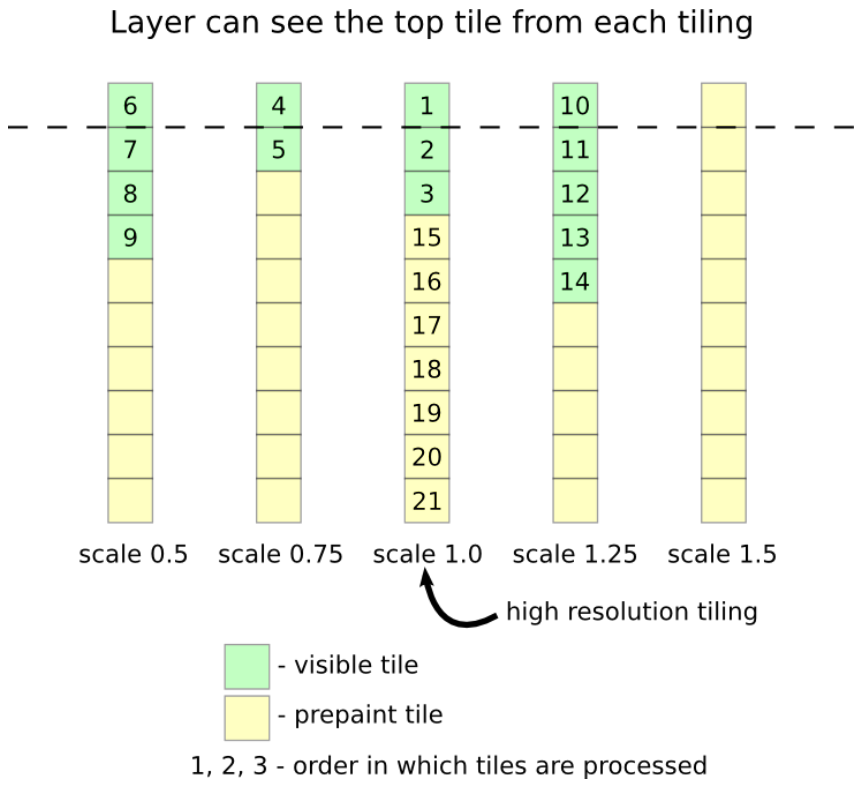
Layer changes

Each layer has a single tiling set, which is a collection of tilings at different scales. The task that the layer has to solve is to use its knowledge of which tilings are important and which are likely to disappear in order to provide the same API as each of the tilings provides:

- `Tile* PeekTileForRaster()`
- `void PopTileForRaster()`
- `Tile* PeekTileForEviction()`
- `void PopTileForEviction()`

In order to provide this type of API, the layer needs to merge information from each of the tilings. The heuristic used here is fairly open to discussion. One such heuristic is outlined below:

For rasterization, the top priority tiles should be visible tiles from each of the tilings. Starting with high resolution tiling and walking first towards lower resolution tilings and then towards higher resolution tilings, we can peek from the current tiling. In case the tile is visible, return that tile. If more tiles are requested, we can return the high resolution tiling prepaint tiles. After this, we can inform the caller that no more tiles are required by returning NULL. Note that we explicitly do not return prepaint tiles from non-ideal tilings, since they are likely to disappear once we have enough high resolution content.



Eviction can happen in a similar manner, except traversing the tilings in the reverse over: overly high resolution first, lowest resolution next, and finally ideal resolution. Additionally, when requested to return a tile for eviction we can return NULL if the next tile that we would otherwise return is a visible tile. That is, we should never evict visible tiles.

Note that from the performance point of view, each operation can be done with constant overhead. The only cost incurred for each tile is the underlying cost of a tiling returning and popping off a tile. Other than that, the iterations over the tilings can be thought of as constant time iterations.

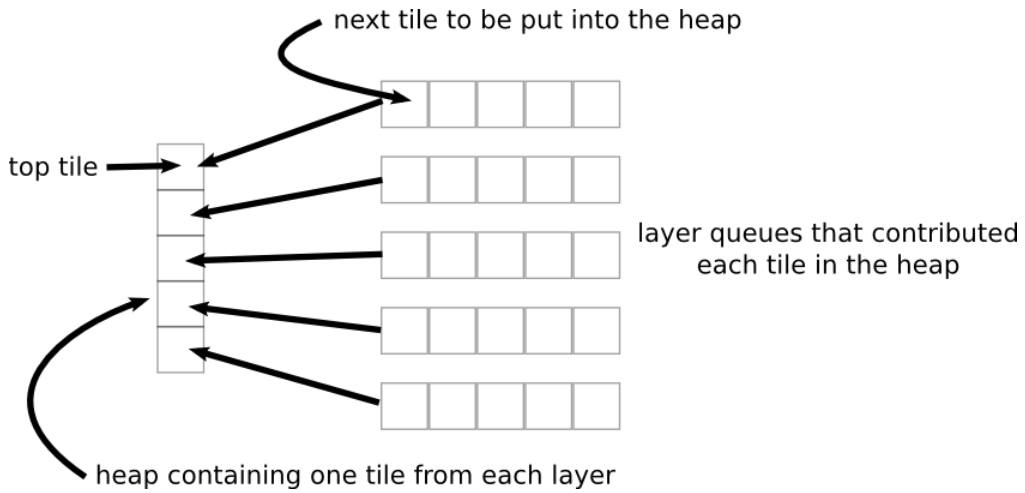
One additional consideration is the low resolution tiles. There has been a bit of discussion whether we need to keep this around. For the purpose of this document, we assume that we require low resolution tiles when high resolution content is not available. Observe however, that the layer is in a very good position to guess whether low resolution content will be required. It has to simply check whether the top tile on each of the tilings stack is a visible tile. If it is, prepend a low resolution tile before proceeding.

Ideally, it would also be able to query how many visible tiles each of the tilings will provide before the next time priorities will be updated. This information should be readily available to the tiling. Hence, the layer will know how many of its tiles are visible and are not rasterized. If that amount breaches some arbitrary constant, a low resolution tile can be prepended to the queue that will be returned to the caller.

Managing tiles

One structural change that has to happen in order for this design to work is that the tile manager needs to be aware of all tile contributing layers. We will call these *tile sources*. Each PictureLayerImpl, the only layer that currently can contribute tiles to the system, will derive from a base TileSource class and register itself with the tile manager upon its construction. As well, it will unregister itself upon destruction. TileSource class itself will define the API as described update **Layer changes** section.

The actual work that tile manager has to perform is very similar to the work that layer has to perform on its tilings: it has to merge tiles that come from all layers within the system into a final set of tiles to be rasterized. The comparison to decide which tile should come next are the values calculated during update tile priorities: distance to visible and the coarse bin (NOW, SOON, EVENTUALLY). This is sufficient information to decide which layer should contribute the next tile. One efficient implementation is to keep the first tile from each layer in a heap. Once the heap value is popped off, the tile that needs to replace it comes from the same layer.



Note also that at this point, the limits that are imposed on the amount of work that the tile manager needs to schedule are the GPU memory limit, and any other arbitrary constant limits that can be varied depending on the need. For instance, if we know that the system can't rasterize more than 10 tiles in a frame, we can limit the amount of tiles we process to 10.

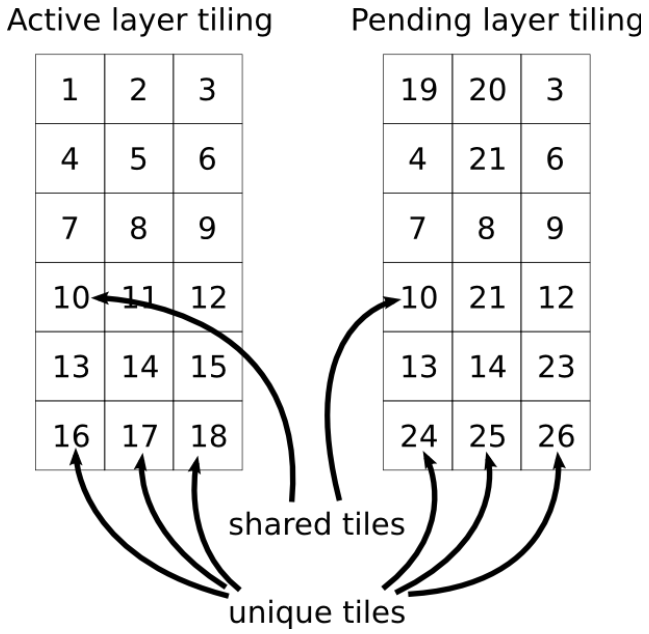
An important thing to note here is that `ManageTiles()` can be called multiple times in a frame. This means that we might call it twice after only one update tile priorities. We have to be careful in selecting which tiles we schedule, since anything that was scheduled previously but not scheduled currently is cancelled by the raster worker pool.
One solution to this is to keep a list of tiles that were last scheduled as a member vector on the tile manager. If ManageTiles is then called before the next update priorities call, we have to

begin with that member vector as the set of tiles to process, and then append more. That is, we need a signal from the system when a new frame has begun and priorities have been updated. In that case, we can simply ask the layer for new tiles. However, if we are scheduling tiles, then we first have to check all previously scheduled tiles and seed the vector of tiles to be rasterized with those tiles that were previously scheduled but have not yet finished. This is to ensure that work that is in progress and the tiling expects to be done will not be cancelled.

Another, possibly better solution, is to shift this responsibility to the tiling. That is, the tiling will keep track of which tiles it has returned and popped off for rasterization. If update tile priorities call comes it, it will clear this vector. However, if tile manager enters a new cycle of scheduling work (as indicated by a new call to PictureLayerTiling::PrepareTiles() or something similar), then the tiling will first return tiles that it has previously returned but that have not been rasterized yet. This would ensure that that successive calls to ManageTiles will consistently produce the same set of tiles to be rasterized.

Active and pending trees revisited

We briefly mentioned that in the current system tilings share tiles. That is, any time a tiling needs to create a tile and that tile was not invalidated in the current frame, we instead get the tile from the twin tiling. The overall picture looks as follows:



Since in the current system the tile manager only knows about each tile, not necessarily which layer the tile came from, we have to store two priority values with each tile: one for the pending tree and one for the active tree. If a tile is unique to one of the trees, then that tile's priority for the twin tree is essentially null. In practice, we assign a priority that guarantees that it will be

assigned a NEVER_BIN. Specifically, we set both distance to visible and time to visible to be infinite.

In the proposed solution, we now request that each layer gives us the candidate tiles for rasterization. We run into a problem when we have two layers that share tiles. That is, we need to be able to resolve duplicate tiles in the tree since each of the layers can independently return the same tile.

There are several ways to approach this problem. The simplest approach is to designate one of the layers as the twin that returns shared tiles. Since tiles are shared, we know that they can be accessed from either one of the trees, so we can guarantee that shared tiles will be definitely returned.

The remaining concern is tile priority. Since each of the trees can have a separate priority, we need to be able to pick which priority takes precedence. There are three situations:

- Smoothness takes priority: In this situation we prioritize the active tree.
- New content takes priority: In this situation we prioritize the pending tree.
- Same priority for both trees: In this situation we take the highest priority out of the two trees.

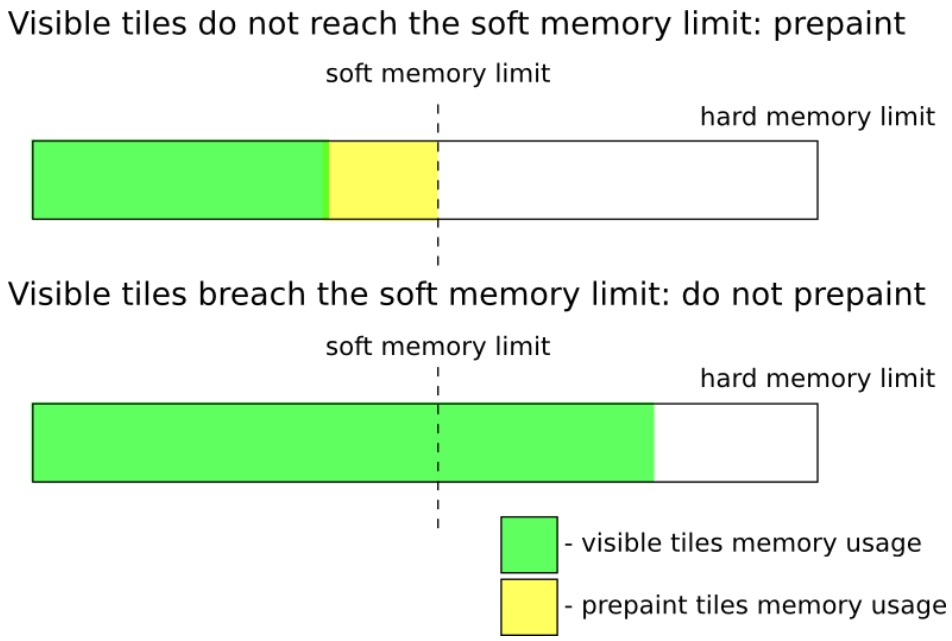
The first two approaches can be implemented fairly easily. We need to instruct the tile sources from the tile manager as to which tree is designated to return the shared tiles. In smoothness takes priority mode, it will be the active tree. In new content takes priority, it will be the pending tree.

The last case will involve some interaction between the twin layers. Since we take the highest priority out of the two trees, we can say that the tree that has tile priority higher than in its twin will return the shared tile, with ties going arbitrarily to the active tree. Tilings will update the priority of the tiles they are interested in, so when the twin requests the priority for a tile, it will already have this information. If the tiling did not update the priority in this frame, we can assume that it has the lowest priority and the twin does not need that tile rasterized.

GPU memory revisited

The problem of a single GPU memory limit is somewhat tangential to the discussion of tile priority redesign, but it's appropriate to mention it here. The concern with having a single large GPU memory limit is that the system will tend to prepaint enough tiles to fill all available memory. This is typically unnecessary work since invalidations come in causing the rasterized tiles to disappear. Our only real requirement is to make sure that *visible* tiles are always ready when we need them. This means that if we have prepainted enough tiles to fill the memory and the user switches the direction of the scroll, we have to start evicting prepainted tiles to rasterize content in another direction.

The proposed solution to address this is to introduce two memory limits: a hard memory limit and a soft memory limit. The hard memory limit will be a high memory limit that is to be used for the visible tiles. Once the visible tiles are rasterized, the prepainting tiles will only be rasterized if doing so would not break the lower soft memory limit.



This ensures that in most situations where the visible tiles occupy a small amount of memory, we will prepaint tiles up to the adjustable small soft memory limit. However, we are still OK if there are many visible tiles and we breach the soft memory limit: we simply use enough memory for the visible tiles. This allows us to make the hard memory limit larger, since it does not mean that we will prepaint up to that limit.

Implementation plan

There are several parts to the proposed solution that can be implemented on top of the current system. In this section, they are listed in the order in which they can be implemented without any major regressions. Most of the time savings, however, should come from the last patch in this sequence.

Tile priority simplification

crbug.com/339142

As a first step, we can simplify tile prioritization. This means that we can eliminate the velocity calculation in favor of skewport calculations. Note that we have to keep the eventually rect as big as the interest rect used to be, not to regress performance. That is, because the eviction is still governed by the interest rect, we have to make it large enough not to evict tiles that are fairly close to the viewport.

Additionally, we can also opt in for faster calculation on rotated and perspective layers by doing the math in the tiling space. This means that for perspective and rotated layers we might mark more tiles as visible than before, but we deem this is OK due to two things. First, rotated and perspective layers are fairly rare to find in practice and most of the time they are small enough for us to rasterize the extra tiles. Second, this simplification greatly simplifies the code, which makes it easier to reason about and maintain.

Two memory limits

crbug.com/339144

This task can be treated independently from the rest of the system and can be implemented separately. The currently system will gain the same benefit from this task as the redesigned system. That is, the idle memory usage will go down to the soft memory limit without risking out of memory situations on the visible tiles. If we have enough memory to prepaint, we will do so. At the same time, we will have a large memory limit for the visible tilles.

Incremental rasterization and eviction

crbug.com/329686

Finally, the incremental scheduling of rasterization can be implemented. This will involve each tiling generating a set of tiles that it wants rasterized. These tiles will be merged by the layer to generate a layer level set of tiles to rasterize. Finally, the tile manager will combine layer tiles to generate tiles that will be rasterized. The amount of tiles to be rasterized will be limited by a constant number and the two memory limits. This will allow us to process fewer tiles in order to get the same content generated. Thus, we will save time on processing tiles that will not be rasterized in this frame.

There are three major areas in which implementation will take place: PictureLayerTiling, PictureLayerImpl, and TileManager. However, these cannot be done independently, since the system will not work properly if all three components are not in place.

Discussion and other considerations

What coordinate space do we use?

1.) Screen Space

The current impl-side painting approach does things in screen space. This is better in some cases like big rotated layers and more accurate direction/velocity/time-to-intersect. But it is likely much more costly per tile because we have quads of points instead of simple rectangles.

2.) Layer Space

Distance calculation is made trivial (this is nice!), but the “viewport” can be problematic. The “viewport” in layer space is technically an arbitrary quad, clipped polygon, or empty polygon. The most common approach elsewhere is to just take the AABB in layer space and use that as the viewport. If we do this, we’ll make strange/incorrect decisions on rotated/perspective layers.

We’ll go with **layer space** because we care a lot about CPU cost, but not much about rotated layers (the use cases will at least not flicker if we conservatively expand the rectangle, and that’s good enough).

Random Things to Consider (you may skip reading this)

Active and pending trees: Tiles are currently shared among the active and pending tilings. This means that if we strictly stick to getting tiles from layers, we will have a set of tiles from an active layer and its corresponding pending layer that share tiles. We could try and unify tilings so that the tiling is shared across pending and active tree (crbug.com/326560). This might help alleviate the problem, but I think there is still some details here that need to be worked out.

Invalidation: Invalidation happens by simply deleting a tile and recreating a blank tile. I think this might be ok in either scheme we pick, but is also something that of which we need to be mindful.

Skewport: This is the idea to skew the viewport before computing priorities with hopes of eliminating the need to keep velocity information as part of tile priority. This might fit in as a patch that lands first? We can simplify the current priority calculations before we actually begin implementing this.

Pinch/zoom: This could be punted to v2, but I think it might be worth mentioning that tiling-creation heuristics could be improved about which tilings we create and when.

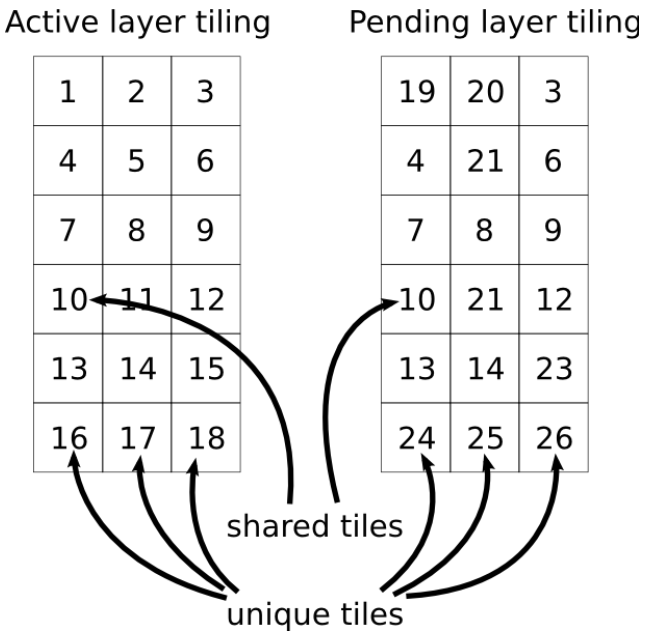
Other thoughts and concerns?

The process of scheduling raster work can be summarized as the following:

```
kNumTilesToSchedule = a constant number of tiles to schedule (32)
tile_queue = TileQueue with all tile sources added
tile_queue.PrepareTiles()
tiles_to_raster = new vector
soft_limit = soft memory limit
hard_limit = hard memory limit

while tiles_to_raster.size() < kNumTilesToSchedule:
      raster_tile = tile_queue.PeekTileForRasterization() || break
      memory_usage = memory if scheduled raster tile
      while memory_usage > soft_limit:
            eviction_tile = tile_queue.PeekTileForEviction() || break

            // This check includes checking visibility
            if (eviction_tile.priority < raster_tile.priority):
                  evict(eviction_tile)
                  adjust memory_usage
                  tile_queue.PopTileForEviction()
            else:
                  break
      if (memory_usage > soft_limit && !raster_tile.is_visible) ||
          (memory_usage > hard_limit):
            break

      tiles_to_raster.push(raster_tile)
      tile_queue.PopTileForRasterization()
schedule(tiles_to_raster)
```

Other approaches we're not going to take

We briefly mentioned that in the current system tilings share tiles. That is, any time a tiling needs to create a tile and that tile was not invalidated in the current frame, we instead get the tile from the twin tiling. The overall picture looks as follows:

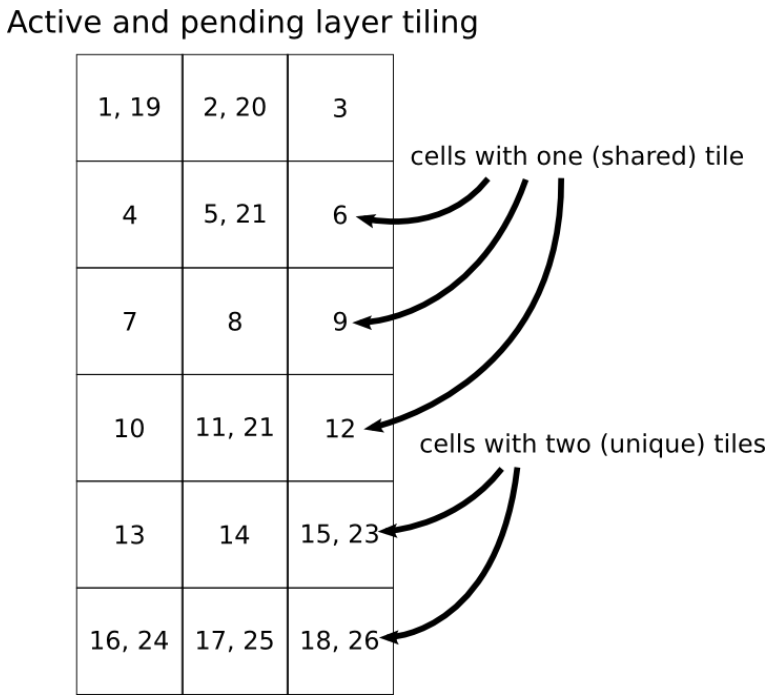


Since in the current system the tile manager only knows about each tile, not necessarily which layer the tile came from, we have to store two priority values with each tile: one for the pending tree and one for the active tree. If a tile is unique to one of the trees, then that tile's priority for the twin tree is essentially null. In practice, we assign a priority that guarantees that it will be assigned a NEVER_BIN. Specifically, we set both distance to visible and time to visible to be infinite.

In the proposed solution, we now request that each layer gives us the candidate tiles for rasterization. We run into a problem when we have two layers that share tiles. That is, we need to be able to resolve duplicate tiles in the tree since each of the layers can independently return the same tile.

One possibility is to keep track of which tile was already processed in the tile manager, essentially de-duping the tiles. However, there is an approach that yields a nicer design.

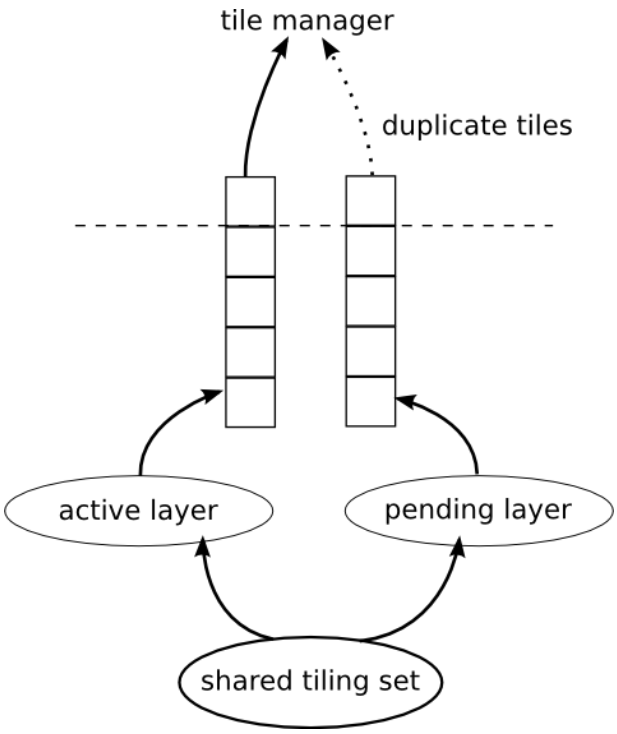
What should conceptually happen is for the layer to share tilings. That is, instead of getting a tile from the twin tiling when appropriate, the tiling and its twin should be one and the same. This means that instead of storing a single tile in each cell of the tiling we would sometimes store two: one representing an active tree tile and one representing a pending tree tile:



The restriction that allows us to do this merge is the same restriction that establishes the twin relationship: both active and pending tiling have to share the same scale. This means that the tile will be in the same place, in the spatial sense.

Once the tilings are merged, merging the PictureLayerTilingSet becomes trivial. It is simply a container of tilings and the tilings are already shared. It makes sense to share the tiling set as well. The only caveat is that creating a new tiling on a pending tree would create the corresponding tiling on the active tree. This needs to be handled with care, since it would be a deviation from the current system's behavior.

If the tiling sets are shared, however, we have a nice property that both the active layer and the pending layer returns the same tiles in the same order of priority. This means that from the tile manager's perspective, we can get away with only asking one of the twins for each tiles, effectively reducing the number of layers queried by two and eliminating duplicate tiles:



Again, we must be careful not to eliminate all pending layers, for instance, since it might be a new layer that doesn't have a corresponding active twin. This de-duping of twins however, operates on a much smaller set of layers and can be done ahead of tile generation.