SKIA COMPOSITING

Feature support for the viz display compositor

Summary
Authors: michaelludwig
Last Updated: 2019-5-29
Public Copy

Objectives

- Add new functionality to Skia's SkCanvas and GPU backend to provide explicit control over per-edge anti-aliasing of rectangles and quadrilaterals.
- Add bulk image-drawing APIs to SkCanvas, and specialized GPU ops to efficiently process the image content provided by the viz system.
- Add support for arbitrary convex quadrilateral rendering, without relying on a path renderer, to improve performance in 3D sorted compositing contexts.
- Update `skia_renderer`'s draw functions to make better use of Skia and the new APIs.
- This document focuses on the rendering, geometry, and shading aspects needed to correctly composite content, so does not go into detail about the resource management and threading improvements that have been made in Skia.

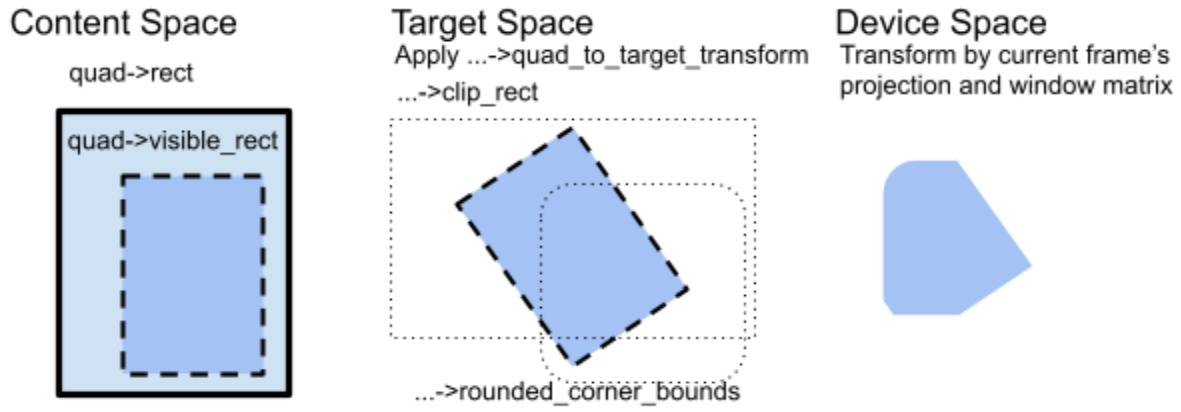
Background

The viz display compositor component of Chromium is working to replace their custom OpenGL compositor (`gl_renderer`) with an implementation that uses Skia directly (`skia_renderer`). Previously, there was a CPU-only compositor (`software_renderer`) that also used Skia directly, albeit only its CPU backend. `skia_renderer` started as a clone of the software compositor, but has diverged more substantially to support backend textures, deferred display lists, and YUV video streams. By moving viz away from the low-level graphics platform, it will be easier to add Vulkan and Metal support to Chrome.

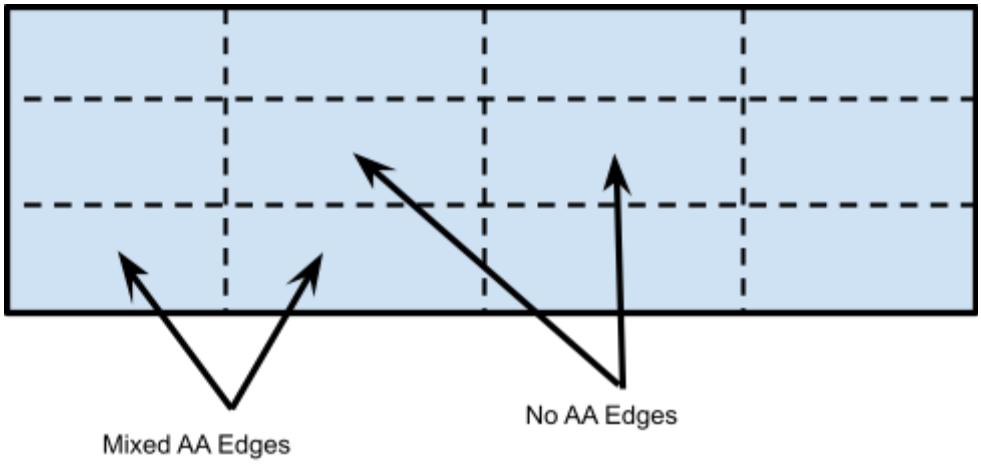
The renderer subclasses represent the lowest level of display logic for the compositor system, responsible for drawing the actual content to the screen. The content is produced by layers from cc and processed at a higher level to limit overdraw and offscreen drawing. Decisions are also made to rasterize page content into tiled images for fast reuse when a page is scrolled. This means that the renderers only need to draw rectangular, textured content with some set of effects applied to it.

DrawQuads

DrawQuads represent the base pieces of content the compositor needs to display. There are three coordinate spaces relevant to the geometry of a DrawQuad. Its local coordinate system is referred to as the *content* space, in which `rect` and `visible_rect` are specified. The visible rectangle may be the same as the base rectangle, or inset along edges to eliminate occluded draws. The target coordinate system is the global coordinate system shared by all DrawQuads. The shared quad state optionally specifies a clipping rectangle and a clipping rounded rectangle. After projection into the window's device space, the intersection of the transformed visible rectangle, clipping rectangle, and clipping rounded rectangle is blended using the quad's blend mode and opacity.



Some types of quads form grids, grouped by their shared quad state. All grouped quads share the same content coordinate system. Per-edge anti-aliasing must be used so that interior edges between quads do not form transparent seams, while exterior edges are smoothed. Edges where the visible rectangle has been inset from the original rectangle also have anti-aliasing disabled[1].



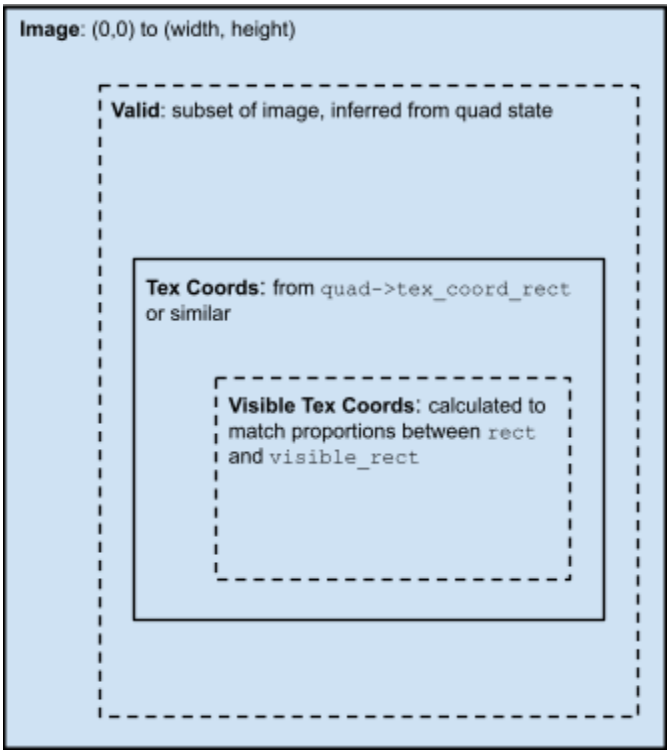
[1] `gl_renderer` behaves this way for certain types, and it disables AA for video stream quads. It may be worth trying to make `visible_rect` clipped edges smoothed.

Texturing

Most of the quads that are composited involve texturing. Their distinguishing factors are where the data comes from (video, rasterization, etc.), effects that are applied, and simply how the data is specified. The image coordinate system is defined by a texture coordinate mapping each corner of the quad's `rect` to pixels in the image. There are four image-space rectangles to think about: the image itself, the valid texel content in the image, the texture coordinates corresponding to `rect`, and the texture coordinates corresponding to `visible_rect`. The image size and base texture coordinates are defined with the quad; the valid texel content must be inferred from quad type and state, and the visible texture coordinates must be calculated to match the relationship between `rect` and `visible_rect`.

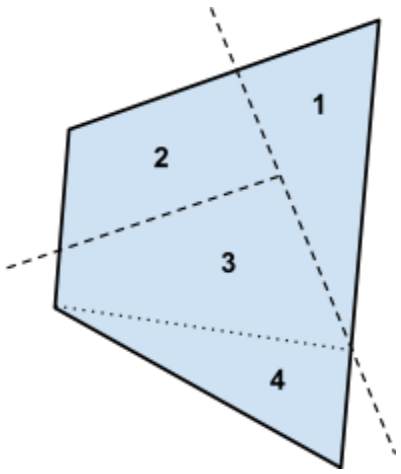
When quads are rendered with bilinear filtering and/or anti-aliasing, default sampling behavior may cause the GPU to sample texels outside of the provided visible texture coordinates.

Bilinear filtering will outset by half a texel; depending on scale factor, anti-aliasing may require outsetting more or less than half a texel. As long as these outset coordinates are still within the valid texel bounds, the filtered texture color is valid. If it were to sample outside the valid bounds, it might read dirty or undefined image data caused by texture reuse. The exception to this is if the valid bounds completely fill the image dimensions, at which point hardware clamping can be relied on. The solution is to clamp the visible texture coordinates to a safe inset of the valid bounds in the fragment shader, but this incurs a performance hit.



3D Splitting

The majority of content that needs to be composited is two dimensional, but the compositor does need to handle quads that exist in a 3D coordinate system. The quad's local content space is still 2D, but their transformation to the target space or window space will have perspective. To produce correct rendering in a painter's order system, quads are grouped into sorting contexts. Each group is then processed by a BSP tree and split into polygons that can be drawn from back to front. This means that a given DrawQuad instance can be submitted to the renderer more than once, with an additional `draw_region` specifying the subset of its geometry to draw. The BSP tree may create arbitrary polygons when it splits planes, but these are converted into multiple quadrilaterals as needed. As part of this, degenerate quads (triangles) may be submitted to the renderer, and must be safely handled. Thus the renderer is still only required to handle quadrilateral-class geometry, although they may be arbitrary convex quads instead of rectangles.



In the above example, the blue DrawQuad is split by two recursive planes (dashed lines). These splits are determined by how the quad intersects with other geometry in its 3D sorting context. These two splits turn the DrawQuad into three polygons: #1, #2, and the union of #3 and #4. Polygon #1 is degenerate but can be submitted directly to the renderer. Polygon #2 can also go directly to the renderer, and is a plain quadrilateral. The last polygon is 5-sided and must be converted into a quad-fan before sending to the renderer as #3 and #4 (with the last also being degenerate in this case). The renderer would be invoked 4 times with the same DrawQuad and a `draw_region` matching the particular subset of its `visible_rect`. However, these invocations can be interleaved with other split DrawQuads to convert the BSP tree into a visually-correct set of draw commands from back-to-front.

Per-edge anti-aliasing must be used so that the separate regions seam together correctly. Edges of the draw regions representing the exterior of the quad need to be smoothed, but interior splits should not be anti-aliased.

Skia Updates

It is entirely possible for the viz display compositor to use Skia's standard API, such as `SkCanvas::drawImageRect` combined with appropriate `SkCanvas::clipRect` and `clipPath`, to composite the DrawQuads correctly. `drawImageRect` accepts a source and destination coordinate rectangle, along with a constraint mode, which together can emulate the required valid texel clamping. Mixing rectangular and polygonal non-AA clips with an AA rectangle can simulate per-edge anti-aliasing. This strategy could work for both tiled quads and 3D split quads.

However, this approach adds a lot of overhead, so we added a new experimental API meant to support the needs of `skia_renderer` directly.

Requirements

- Provide a bulk API for image rendering. The majority of composited content will be rectangular textures without any complicating effects. Pre-batching can reduce the amount of overhead in Ganesh's internal batching and opens up the possibility for more internal optimizations.
- Provide a similar API for solid-color rendering since some DrawQuads are pre-determined to be a solid color and can be drawn without any image.
- Directly handle per-edge AA. Instead of mixing draws and clips to simulate mixed anti-aliasing, support quadrilateral primitives that provide the same behavior in a single draw without any clipping.
- Accept rectangles and arbitrary convex quadrilaterals. This allows 3D split DrawQuads to be drawn without the use of the expensive `SkCanvas::clipPath`. Must support degenerate/triangular quadrilaterals.
- Support per-quad transforms in the bulk API so that batching is not prevented by differences in their `quad_to_target_transform`'s. Normally, a different transform would have to modify the SkCanvas' CTM, requiring any accumulated DrawQuads to be drawn before changing the current matrix.
- Support existing effects and functionality of SkPaint. Depending on the DrawQuad state, it may be more complex than just texturing. For some page content, SkMaskFilters, SkColorFilters, and SkImageFilters are required to draw the quad content. The new APIs should support these effects so that there's no need to switch to an older API that did not support per-edge anti-aliasing.
- Support all source and destination color spaces currently supported by `gl_renderer`.

New APIs

Two new entry points have been added to SkCanvas to support `skia_renderer`. Two supporting types have also been added to SkCanvas: QuadAAFlags and ImageSetEntry. A special factory for SkColorFilters written as SkSL snippets has also been exposed. The details provided below are based on the state of Skia at the time of writing; the source code is the definitive resource.

QuadAAFlags

```
enum SkCanvas::QuadAAFlags : unsigned {
      kLeft_QuadAAFlag    = 0b0001,
      kTop_QuadAAFlag     = 0b0010,
      kRight_QuadAAFlag   = 0b0100,
      kBottom_QuadAAFlag  = 0b1000,

      kNone_QuadAAFlags   = 0b0000,
      kAll_QuadAAFlags    = 0b1111,
};
```

A set of masks to define which edges of a rectangle or quadrilateral are anti-aliased. This is accepted in DrawEdgeAAQuad and stored in per ImageSetEntry, so each quadrilateral has a corresponding bit field that specifies the AA state for its left, right, top, and bottom edges. The mask flags are named to correspond to the edges of the rectangle, in the draw's local coordinate space. This means that a rectangle with a smoothed, locally-left edge can be rotated by the canvas' matrix so that, visually, the top edge appears smoothed.

When drawing an arbitrary quadrilateral, specified as an array of SkPoints, the concepts of left/right or top/bottom are less well-defined. In this case, the edges of the quadrilateral are labeled by their vertex indices, following the vertex ordering specified by `SkRect::toQuad()` (top-left, top-right, bottom-right, bottom-left). The edge between vertex [0] and [1] is controlled by `kTop_QuadAAFlag`, [1] to [2] is controlled by `kRight_QuadAAFlag`, [2] to [3] is controlled by `kBottom_QuadAAFlag`, and [3] to [0] is controlled by `kLeft_QuadAAFlag`. This index-based approach affords some flexibility in `skia_renderer` when determining the final AA state for a 3D split DrawQuad.

These flags replace the anti-alias setting of an SkPaint and is more expressive. `kNone_QuadAAFlags` is equivalent to `SkPaint::isAntiAlias()` returning false, and `kAll_QuadAAFlags` is equivalent to it returning true. All other combinations of edge flags represent new anti-aliasing possibilities, and are sufficient to represent all per-edge anti-aliasing needs of `skia_renderer`.

ImageSetEntry

```
struct SkCanvas::ImageSetEntry {
      sk_sp<const SkImage> fImage;
      SkRect               fSrcRect;
      SkRect               fDstRect;
      int                  fMatrixIndex;
      float                fAlpha;
      unsigned             fAAFlags;
      bool                 fHasClip;
};
```

A data type that describes all parameters needed for a single image-based draw, consumed by DrawEdgeAAImageSet. It is modeled after the parameters used in the conventional `SkCanvas::drawImageRect` API, which draws a sub-region of an image to a destination rectangle. This struct adds additional parameters needed to support per-edge anti-aliasing and allow for greater batch sizes (per-entry alpha and view matrices). The specifics of the struct fields are detailed below in the DrawEdgeAAImageSet documentation.

DrawEdgeAAQuad

```
SkCanvas::experimental_DrawEdgeAAQuad(
      const SkRect& rect,
      const SkPoint clip[4],
      QuadAAFlags aaFlags,
      SkColor color,
      SkBlendMode mode);
```

This draws a single filled quadrilateral. The shape will be filled in with `color` and blended with the destination via `mode`. Its anti-aliasing is controlled solely by `aaFlags`. The geometry drawn is `rect`, clipped by the optional `clip`. It is assumed that `clip` is convex and is fully contained in `rect`, so setting aside the support for per-edge anti-aliasing, this is equivalent to:

```
      SkPaint paint;
      paint.setBlendMode(mode);
      paint.setColor(color);
      If (clip) {
            SkPath path;
            path.addPoly(clip, 4, true);
            canvas->drawPath(path, paint);
      } else {
            canvas->drawRect(rect, paint);
      }
```

This function call is provided exclusively for drawing solid-color DrawQuads, which require per-edge anti-aliasing support but are not image-based nor apply any additional effects. An SkPaint is not an input parameter to reinforce the limited use cases this is intended to support. A bulk solid-color API is not provided for two reasons: keep the public API as-simple-as-possible, and pre-batching was not found to provide substantial benefits over Ganesh's internal batching capabilities.

DrawEdgeAAImageSet

```
SkCanvas::experimental_DrawEdgeAAImageSet(
      const ImageSetEntry imageSet[],
      int count,
      const SkPoint dstClips[],
      const SkMatrix preViewMatrices[],
      const SkPaint* paint = nullptr,
      SrcRectConstraint constraint = kStrict_SrcRectConstraint);
```

This function draws `count` textured quadrilaterals described by `imageSet`. Each entry will be textured with its provided `fImage`; `fSrcRect` describes the subrectangle of pixels to draw from the image. In addition to optional per-entry quadrilateral vertices and transforms, the entire batch will be affected by `paint`. The paint's alpha will be multiplied with each entry's `fAlpha` to determine its final opacity. This supports SkColorFilters, SkMaskFilters, and SkImageFilters. When the image is an alpha-only texture, it can also use an SkShader on the paint, matching the behavior of the simpler `SkCanvas::drawImageRect`. If `constraint` is strict, it guarantees that the quadrilateral will not sample the image outside of `fSrcRect` due to anti-aliasing or bilinear filtering.

If provided, `dstClips` must have at least 4 x (the number of entries with `fHasClip` true). If `dstClips` is null, then every entry must have `fHasClip` set to false. The destination clip coordinates will be read consecutively with the image entries, advancing by four SkPoints every time an entry with `fHasClip` is encountered. When an entry has a clip, its geometry is the convex quadrilateral specified by the clip's four vertices. When it does not have a clip, its geometry is specified by `fDstRect`. This is consistent with the geometry specification of DrawEdgeAAQuad().

Entries that have a non-negative `fMatrixIndex` will pre-concat that matrix from `preViewMatrices` with the SkCanvas' current view matrix to determine that entry's final CTM. This allows DrawQuads with different `quad_to_target_transform`'s to be batched together without explicitly transforming them into a shared coordinate system.

All told, ignoring the handling of per-edge AA, this can be thought of as a more efficient form of:

```
int clipIndex = 0;
for (int i = 0; i < count; ++i) {
      canvas->save();
      if (imageSet[i].fHasClip) {
            SkPath path;
            path.addPoly(dstClips + clipIndex, 4, true);
            clipIndex += 4;
            canvas->clipPath(path, false);
      }
      if (imageSet[i].fMatrixIndex >= 0) {
            canvas->concat(preViewMatrices[imageSet[i].fMatrixIndex]);
      }

      SkPaint perEntryPaint;
      if (paint) {
            perEntryPaint = *paint;
      } else {
            perEntryPaint.setBlendMode(SkBlendMode::kSrcOver);
            perEntryPaint.setFilterQuality(kNone_SkFilterQuality);
      }
      perEntryPaint.setAlphaf(imageSet[i].fAlpha *
                              perEntryPaint.getAlphaf());

      canvas->drawImageRect(imageSet[i].fImage, imageSet[i].fSrcRect,
                            imageSet[i].fDstRect, &perEntryPaint,
                            constraint);
      canvas->restore();
}
```

The only required reasons to make separate DrawEdgeAAImageSet calls is when the SrcRectConstraint mode changes; when the SkPaint settings are different; or when the clip state of the SkCanvas is different.

Initially, this bulk API was designed to match `drawImageRect` and allow for lightweight rendering of images without the need to allocate an SkPaint with an SkImageShader. Per-entry alpha was added to make it easier to combine DrawQuads with different opacities into a single Skia draw call. Per-entry view matrices were added for a similar reason. Without it, `skia_renderer` could only combine DrawQuads with the exact same view matrix, or had to explicitly apply scale/translation matrices. The matrices are not stored on the ImageSetEntry so that specific matrices can be shared by entries. The `dstClips` points array is also separate from ImageSetEntry, since the majority of DrawQuads do not require 3D splitting and can be more compactly represented with just SkRects.

Originally, the goal was to avoid accepting an SkPaint, but RenderPassDrawQuads require per-edge anti-aliasing *and* need to use advanced SkPaint features like image filters. Adding support for an SkPaint also allows `skia_renderer` to easily render YUV and textured quads that need SkColorFilters with the same API.

RuntimeColorFilterFactory

```
class SkRuntimeColorFilterFactory {
      SkRuntimeColorFilterFactory(SkString sksl);
      sk_sp<SkColorFilter> make(sk_sp<SkData> inputs);
};
```

Provides a factory for creating an SkColorFilter from a string of SkSL shader code. SkSL is the internal, high-level shader language used by Ganesh that can be compiled to GLSL, MSL, and SPIR-V. This is not ready for general consumption, so it is part of `SkColorFilterPriv.h` that `skia_renderer` can include. The provided SkSL snippet must define a main function: `void main(inout half4 color)`, where `color` provides the input color in the source color space, and after execution contains the output color in the destination color space.

The compositor has to support many esoteric color spaces that are outside of the color space representation used by SkColorSpace. These were originally implemented as custom GLSL shaders in `gl_renderer`. With the need to support Vulkan, a more backend-agnostic solution was needed. Exposing runtime SkSL compilation to `skia_renderer` allows the compositor to implement the esoteric color space logic without complicating SkColorSpace and still easily deploy to Vulkan and Metal backends.

Deferred Display Lists

While a key part of the overall compositor system, DDLs are somewhat orthogonal to the nitty-gritty of the geometry that must be rendered to achieve the correct composited results. It is also worthy of its own design document:
https://docs.google.com/document/d/1i5qvAaG7heDxDJfdXiuWdL3PoibJZnZRN8KA3Altero

Other Ganesh Improvements

A number of improvements have been added to Ganesh to increase performance for the draw commands used in compositing:

- Added the specialized GrTextureOp for drawing textured rectangles and quadrilaterals.
    - Able to preserve pre-batched ImageSetEntries in a single op instance.
    - Avoids allocating GrFragmentProcessors and implements texture sampling directly in its geometry processor shader.
    - Uses sampler objects to reduce overhead of binding multiple textures with the same sampling parameters.

    - GrTextureOp was capable of batching draws of different images together in a single glDrawElements call using a shader that selected between multiple samplers. This could be a big performance win or loss depending on the particular GPU hardware and the number of pixels drawn in the destination. It used more power on mobile GPUs. This proved too hard to tune to a clear win and was removed.
- Unify rectangle and quadrilateral geometry processing code, shared between GrTextureOp and the regular GrFillRectOp.
    - Uses vectorized CPU operations to determine final anti-aliased geometry to send to the GPU, specialized for simple cases (no AA edges, or all AA edges), but has complete solutions for mixed-AA edges and non-rectilinear quadrilaterals (from perspective or skew, or from the `clip` parameter of the new APIs).
    - Uses vertex attribute interpolation to produce the anti-aliased coverage at edges, instead of evaluating edge equations per-fragment, which is significantly less GPU work on lower-end and mobile devices.
    - Adds direct support for anti-aliased perspective rectangles, which previously were rendered using a path renderer.
- Support multiplanar YUV images as a single SkImage, and automatically use fragment shaders to sample and convert to RGB.
    - Allows consuming YUV encoded video directly without having to flatten to a separate RGB texture on each frame.
- Optimize GrTexture and GrResourceAllocator for compositing use patterns.
    - Increase arena interval size.
    - Ignore borrowed resources that are managed by Chrome.
    - Mark wrapped textures as read-only and skip adding dependency edges in a DDL DAG to improve oplist sorting.
    - Assist with caching improvements in `skia_renderer` to reduce cost of re-wrapping textures as SkImages.
- Improved backend resource management APIs required for OOP-D and OOP-R. This is not required for geometric and color operations of compositing, but is tied heavily to DDL and is critical for getting the image content in the right place to drawn at the right time.

Notes and Improvements

Batching Strategy

When `skia_renderer` was updated to use the new per-edge drawing and batch APIs, DrawQuads were grouped into three categories based on their material:

1. *Always batchable* - The DrawQuad's material always results in a textured quad, and only needs to specify filter quality, alpha, blend mode, and texture constraints. These quads will be grouped into `ImageSetEntry[]` arrays and drawn deferred (i.e. not until a quad is incompatible with the accumulating batch).

    a. TileDrawQuads and VideoStreamDrawQuads are currently the only types that are always batchable.
2. *Sometimes batchable* - The DrawQuad's state may result in a draw compatible with batching, in which case it will be enqueued as above. However, some material-specific state may require additional SkPaint configuration or extra SkCanvas control. This may be setting an SkColorFilter or SkShader on the draw's paint, or saving a layer with SkImageFilters before drawing. When these complexities arise, the current batch is flushed, and a single draw is immediately sent to Skia for the complex quad.
    a. TextureDrawQuads may be batchable, but can require an SkColorFilter for their background color blending, or an SkMaskFilter for their vertex opacities.
    b. RenderPassDrawQuads may be batchable (e.g. when they simply wrap a tiled quad), but can require layer management for image filters and backdrop filters, or using SkMaskFilters.
3. *Never batchable* - Either the DrawQuad always requires a complex SkPaint, or requires the use of APIs other than DrawEdgeAAImageSet.
    a. YUVVideoDrawQuads currently always require a custom SkColorFilter. They could be upgraded to *sometimes batchable* if we could detect when an SkColorSpace stored on their attached SkImage was sufficient for the color transformations. In this case, `DrawEdgeAAImageSet` is sent a singleton set.
    b. DebugBorderDrawQuads and PictureDrawQuads describe primitives that are not simply filled rectangles or quadrilaterals. They have very specialized rendering implementations that do not use `DrawEdgeAAImageSet` or `DrawEdgeAAQuad` at all.
    c. SolidColorDrawQuad uses `DrawEdgeAAQuad` since it does not have any associated image data. This API does not support batching because Ganesh internally is already able to batch these ops together in an efficient manner.

These three categories were chosen to provide a balance between using a bulk API and efficiently detecting when quads were compatible together. The *sometimes batchable* category could almost go away entirely if the batch just maintained its SkPaint, and a new batch was started if the current quad required a different SkPaint. Comparing paints is non-trivial so this was avoided.

Coordinate Systems

While the rendering and layering systems in Chromium have a number of coordinate systems[2], `skia_renderer` manipulates Skia's SkCanvas so that it operates either in device space or content space. When SkCanvas' total matrix is the identity, it represents the device coordinate space. All primary clipping is specified in device space, which was partially chosen because `direct_renderer` already ensures its scissor rect was in device space. The 'target' coordinate space is avoided since the actual draw operations are easier to specify in their content space. Because DrawEdgeAAImageSet accepts per-entry transforms, this means that

[2] See DrawQuads.

as long as the device-space clipping rules agree between DrawQuads, they can be batched regardless of their content transforms.

This separation of coordinate systems should remain valid going forward, with the main complicating factor being backdrop filters. These filters create an interaction between quads that were already drawn and the layer saved for the current render pass. Currently, `skia_renderer`'s backdrop code assumes that the backdrop filter rounded rectangle bounds exists in the render pass' content space. If that ever becomes untrue, it will need to be updated to support the rounded rectangle in its own coordinate space.

Per-edge AA Performance

When tessellating a batch of quads in an op, there are two factors that directly affect performance: the geometric class of the quadrilateral and the anti-aliasing required. The ops are capable of batching simpler geometric or AA'ed quads as more general types in order to send fewer--but larger--draw commands to the GPU.

There are four classes of quadrilateral geometry that is tracked internally: axis-aligned rectangle, rectilinear quad (e.g. rotated rectangle), 2D quadrilateral with non-right angles, and 3D quadrilateral with perspective.

- Axis-aligned rectangles and rectilinear quads require the least CPU and GPU work.
- Arbitrary 2D quadrilaterals require additional CPU overhead for AA-enabled tessellation to safely handle corners that are not right angles, with additional fallbacks when the geometry is degenerate. The GPU fragment shaders also require a geometric clipping domain to prevent anti-aliasing at acute corners bleeding farther than half a pixel from the corner.
- 3D quadrilaterals require the most CPU and GPU work: the projected shape must first be processed, and then un-projected back to 3D. This ensures the screen-space anti-aliasing is correct while preserving the perspective-correct interpolation of the quad's shading. Since the projected shape will not have right-angled corners, its shaders also require a geometric clipping domain, along with the extra vertex component and perspective division.

There are four anti-aliasing states that a quad can be in: no anti-aliasing, no anti-aliasing but batched with an AA-enabled op, full anti-aliasing, and mixed-edge anti-aliasing.

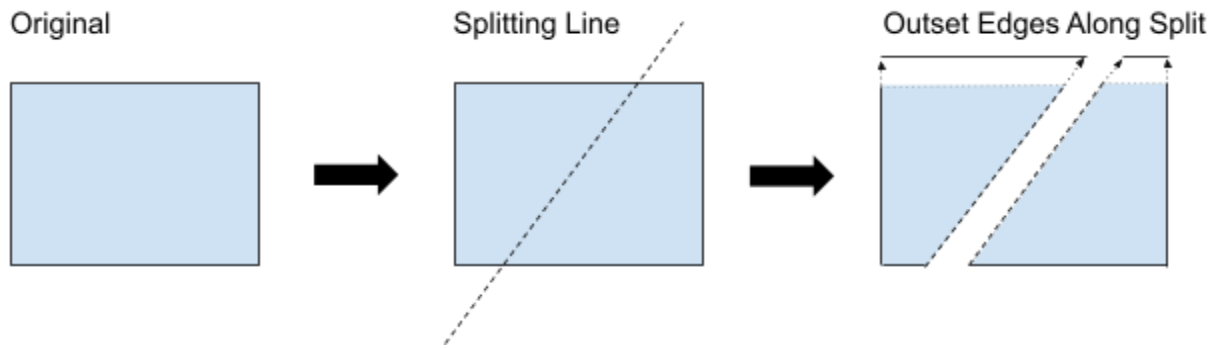
- No anti-aliasing is cheapest for CPU and GPU. A non-AA quad is drawn as two triangles specified with four vertices.
- Full anti-aliasing is next cheapest for the CPU, and increases the complexity of GPU shaders and number of triangles. All quads in an AA-op are drawn as 10 triangles specified with eight vertices.
- Mixed-edge anti-aliasing is the most expensive CPU cost, but the same GPU cost as any other anti-aliased quad.
- No anti-aliasing rendered in an AA-enabled op has roughly the same CPU cost as simple non-AA, but the same GPU cost as any other anti-aliased quad.

Currently all quads with compatible paints are batched together, which means that simpler geometric types and non-AA quads get combined into more general ops. All quads in a batch are tessellated based on the most general geometric class in the batch. Unlike geometry class, the CPU optimizations for anti-aliasing are applied per-quad, so the only downside to batching non-AA quads with AA-quads is their higher GPU overhead. Under that assumption that fewer draws is better, this is reasonable. However, if performance tests suggest otherwise, it is easy enough to prevent combining quads with different AA or geometric classes.

Per-edge AA and BSP Trees

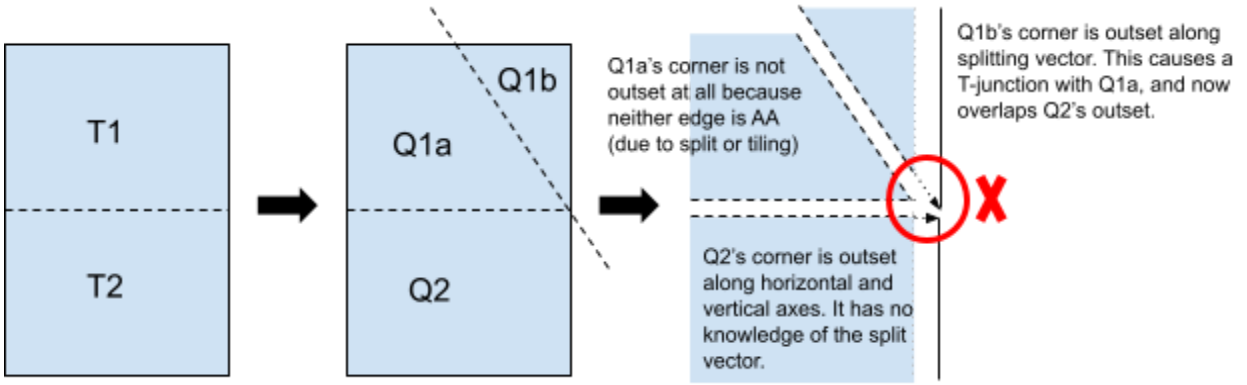
There are three sources for quads that have mixed edge AA flags: tiled quads with explicitly defined exterior edges, clipped quads with a `visible_rect` inset from the original geometry, and 3D quads split by a BSP tree. The first two types do not create any significant difficulties when preparing geometry, but the current division of work where `skia_renderer`'s parent class makes the BSP tree and provides clipped quads one at a time makes it difficult or impossible to guarantee seamless reconstruction[3].

When preparing the AA geometry, Skia must outset by half a pixel along the edge vectors. This ensures that a rectangle cut by an arbitrary line will seam correctly:



This almost always works and produces correct results, but care must be taken to arrive at exactly the same vertices. Even if they differ by hundredths of a pixel, the two quads non-AA edges may evaluate differently at each pixel and form holes or overlap. When there's perspective, or the interior of the quadrilateral has a degenerate interior, this is not always feasible and increases the risks that gaps appear from these unintentional T-junctions. Correctly outsetting these edges can get even more complicated however. Consider two adjacent tiled quads--so their shared edge is already non-AA. If one of them is split, the resulting quadrilaterals may share a corner that's adjoined to the other tiled quad. In this circumstance, an individual quadrilateral no longer has enough information to correctly reconstruct AA coverage geometry:

[3] Given Skia's vertex-based AA approach; evaluating edge distances per-pixel on the GPU simplifies the vertex structure, which would mitigate this issue at the cost of performance. Given how rare this likely appears in the wild, we're avoiding that cost for now.



In this example, three quadrilaterals meet at a vertex, but they all have different edge vectors and anti-aliasing requirements. This causes them to make different decisions when forming the inset and outset geometry. This can be mitigated if Q1a is actually a triangle, in which case its degenerate edge can be marked as anti-aliased and positioned to be at this complicated corner. `skia_renderer` has been written to attempt this in its `GetClippedEdgeFlags` function. T-junctions are still a problem and can result when one half space is further split, creating vertices that are on the original splitting plane but are unknown to the first half space.

The only safe way to handle per-edge anti-aliasing in these circumstances is to know about the entire BSP tree, in which case Skia can determine the ideal geometry and then split according to the BSP. In the future, it may make sense to provide an API in Skia that can accept all of the DrawQuads at once, and is responsible for sorting, 3D splitting, and occlusion culling. It may not be a high priority change, if the current splitting decisions are unlikely to cause these conditions.

Texture Coordinate Constraints

Both Skia and `skia_renderer` have the concept of texture constraints, where it is guaranteed that a texture will not be sampled outside of the constraint. However, in Skia's APIs that accept texture coordinates (i.e. the src rect), they are used as both the corner coordinates and the constraint bounds. The `DrawEdgeAAImageSet` function continues this convention. Unfortunately, this is not entirely compatible with how `skia_renderer` and Chromium use texture constraints.

When drawing tiled content, exterior tiles may need to be constrained on a single edge but use bilinear filtering beyond the provided coordinates on the other axis. Others may use SkImages larger than their valid contents, but because of occlusion culling, will provide texture coordinates that reflect the drawn subrectangle and so are safely inset from the valid constraint along one or both axes.

With careful manipulation of the `fSrcRect` and `fDstRect` geometries in an `ImageSetEntry`, and providing a rectangular clipping region, `skia_renderer` can draw with the proper texture constraints and effectively a separate set of texture coordinates. This is because the destination

clipping region becomes the actual drawn geometry, and the transformation between `fSrcRect` and `fDstRect` is used to determine that region's new texture coordinates. This is implemented in `skia_renderer`'s `ResolveTextureConstraints` function.

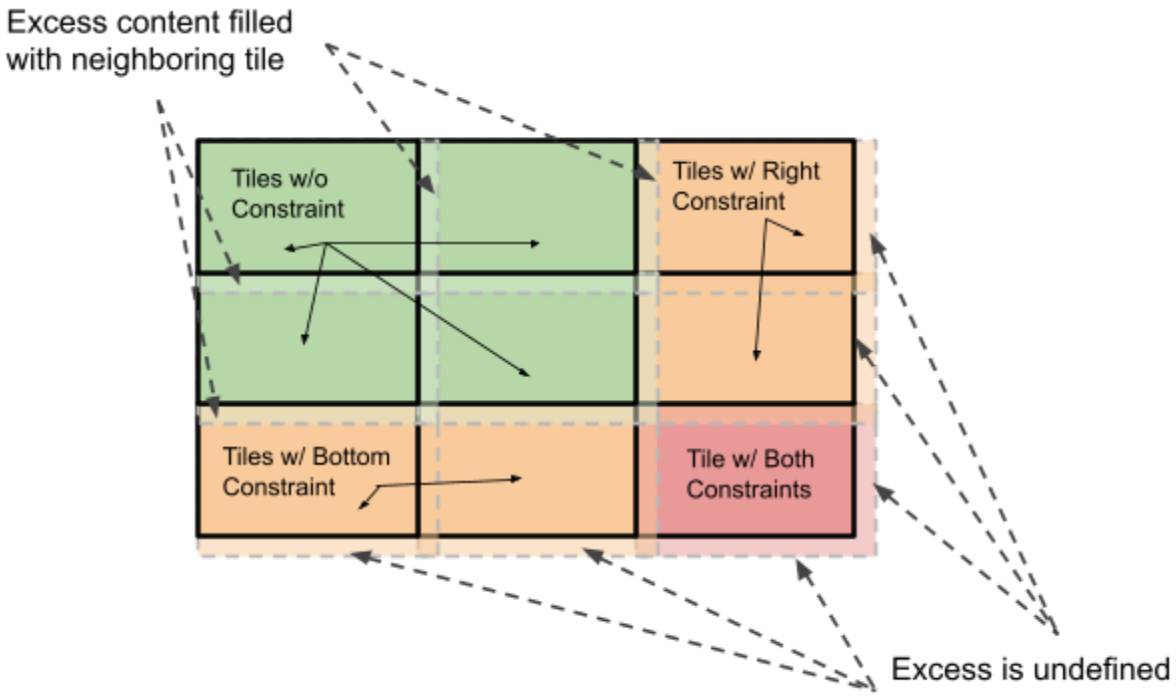
The overhead of calculating this artificial clipping region is minimal. However, in general, switching between fast and strict constraints does force `skia_renderer` to split otherwise compatible batches. Furthermore, when the strict constraint does not align with the underlying SkImage's dimensions, the constraint has to be implemented in shader code, which can be much slower than hardware clamping (particularly on lower end hardware). The shader-based constraint overhead is a cost paid by both `skia_renderer` and the existing `gl_renderer`. For some DrawQuads, it may be possible to avoid texture constraints entirely with upstream changes to how the textures are filled.

When the valid texels[4] of an image match the dimensions of the SkImage (and critically, its backend texture), hardware clamping can be used to automatically respect the texture constraint so it can be effectively dropped. Unfortunately, for a number of reasons, the actual valid content in the SkImages seen by `skia_renderer` regularly does not fill the image. If it is because a hardware video decoder requires a power-of-two buffer, there is not much leeway. If it is too support texture re-use, it may be possible to avoid the use of texture constraints by preparing the data more carefully. Here are two possible strategies, that may not be valid once we consider the needs of OOP-R or CC.

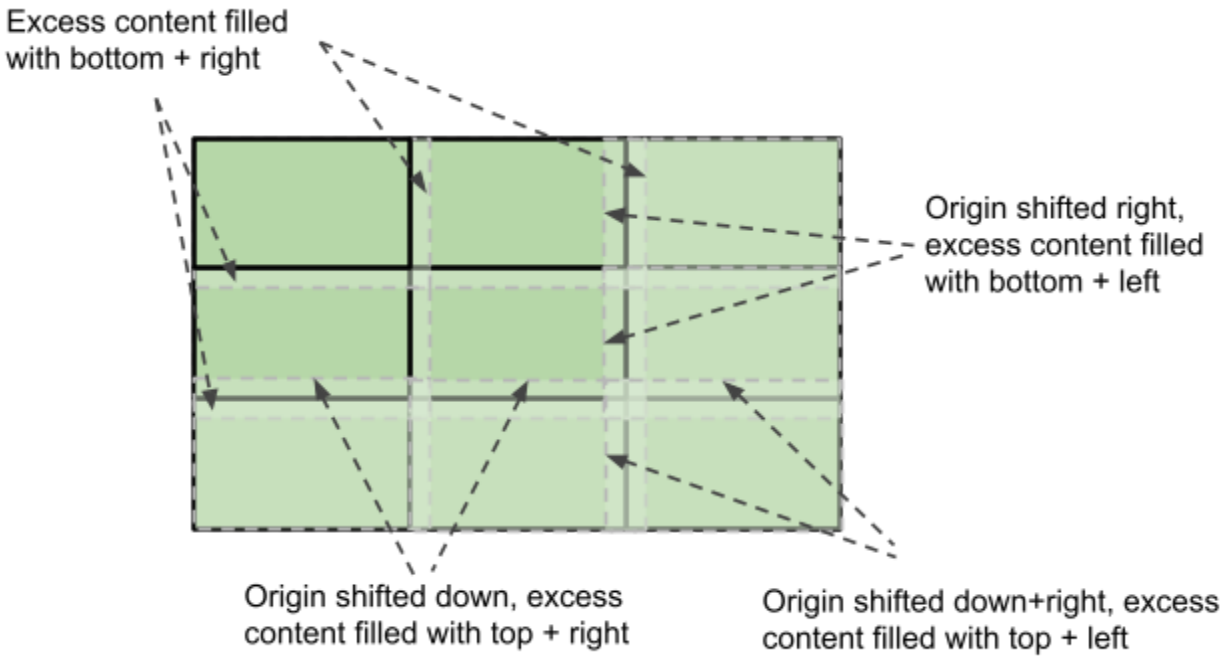
First, if the valid image content is CPU-resident, the right and bottom edges of valid texels can be copied into the adjacent rows and columns. When drawn at 1-1 scale with bilinear filtering and anti-aliasing, only a single extra row and column is necessary. The bumper must be increased when the image is drawn at a smaller scale, in which case filling the remainder of the image with the valid boundary colors will produce a visually equivalent outcome to shader-based texture constraints. With an increased upfront cost, rendering with the texture becomes cheaper. Second, as a more general approach, when tiles are rasterized, their overlapping arrangement can be modified to require no constraints.

Currently, a layer is segmented into rectangular tiles that form a contiguous, larger rectangle spanning the content. However, the image data for each tile is larger than the geometric area used to draw each tile, so the extra texels match the content of adjacent tiles to the bottom and right. When there are no tiles to the bottom or right, the extra texels have undefined data and a constraint must be used. If using CPU rasterization of the tile content, the row and column copying strategy could be used, but there is no convenient way to expand a subimage into the entire texture on the GPU. This is shown in the figure below:

[4] See Texturing for details on nomenclature.



Instead of always using the top-left corner of the image as the origin for the rasterized content, which then leaves undefined texels to the right and bottom, tiling could choose to adjust the origin so that tiles are completely filled. Essentially, tiles on the right edge of the grid would be shifted so the right edge of the rasterized content aligns with the right edge of the texture; similarly for the bottom edge. This would update the tile overlapping to resemble:



Under this approach, the backend textures are completely filled with valid content so there is no need to add a shader-based constraint.

YUV Color Spaces and Batching

Chromium requires a much wider range of color spaces than what is modeleable by SkColorSpace[5]. To support these color space transformations, the YUVVideoDrawQuads can use the RuntimeColorFilterFactory to create an SkSL shader snippet to perform the more advanced conversions. However, this now means that the SkPaint used to draw that particular DrawQuad's geometry has a complex effect that is not compatible with the currently implemented batching rules in `skia_renderer`. This is not a limitation of `DrawEdgeAAImageSet` itself; it is entirely permissible to submit multiple entries that share the same SkPaint (and thus the same color space conversion). Unless a page had numerous YUV quads with the same color space, supporting custom color filters in `skia_renderer`'s batching rules wouldn't behave any differently than the current mode of drawing each YUV quad by itself.

If better batching is needed for performance reasons, it'd make more sense to try and map the YUVDrawQuad's color space to an acceptable SkColorSpace. When the color space is represented as an SkColorSpace, it can be entirely encapsulated in the SkImage so there is no need to end the current batch of DrawQuads. Internally, Skia may split the image sets into compatible color space groups, but that is transparent to `skia_renderer`, and will still be more efficient than pre-splitting the groups and invoking Skia multiple times.

[5] See RuntimeColorFilterFactory.