Pathfinder 3

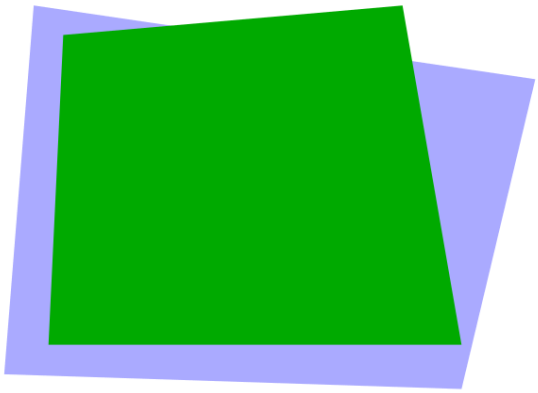
Pathfinder 3 renders vector graphics using the GPU.

The code has been merged into pathfinder master: https://github.com/pcwalton/pathfinder

Demo: https://github.com/pcwalton/pathfinder/tree/pf3/demo/native

General idea

I'll go through some of the steps using the following drawing as example, composed of two simple shapes. To keep things simple it only contains line segments, but the same principle can be applied with curves.

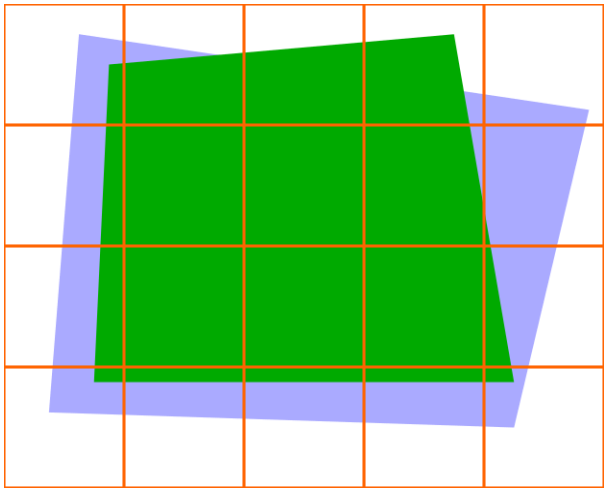


Tiling

On the CPU, the paths are sliced into 16x16 pixel tiles. The GPU part is analogous to a stencil-and-cover renderer, with some notable differences that I'll mention later in this document.

Pathfinder's tiling scheme is meant to split the problem of rendering complex paths into smaller independent tasks. To render each tile independently on the GPU, pathfinder only needs to know the edges that intersect that tile as well as whether the top-left corner of the tile is in or out of the shape.

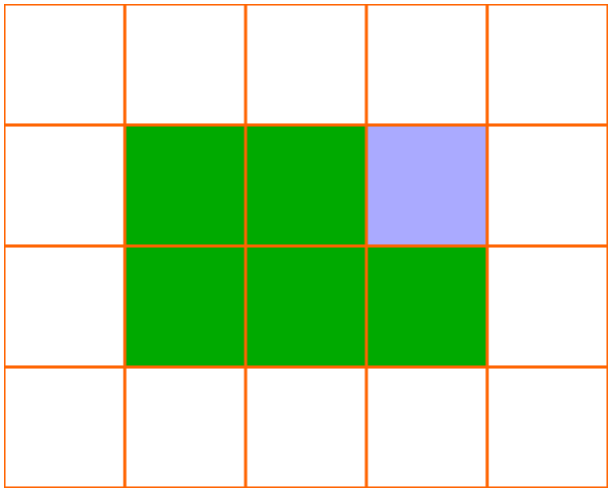
This tiling scheme applies in device pixel space, so all paths are tiled under the same coordinate space.



Occlusion culling

Pathfinder mitigates overdraw in SVG composed of many paths stacked on top of one another (for example the ghostscript tiger) by leveraging its tiling scheme to discard all content under a tile that is fully covered by a path.

In our example, the opaque tiles would be:



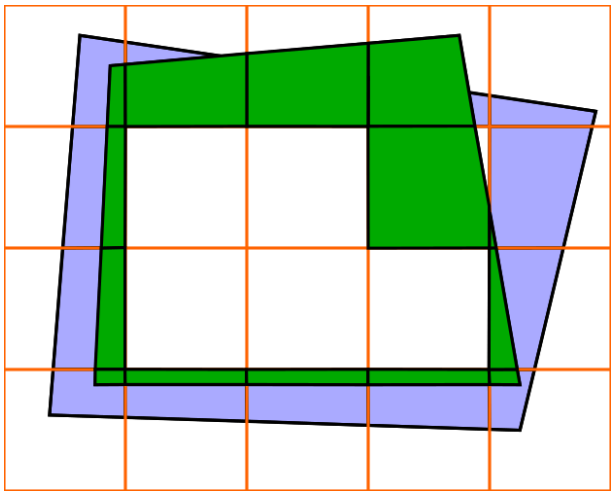
From this we can:

- Discard all blue content that is below an opaque green tile since we know it is occluded

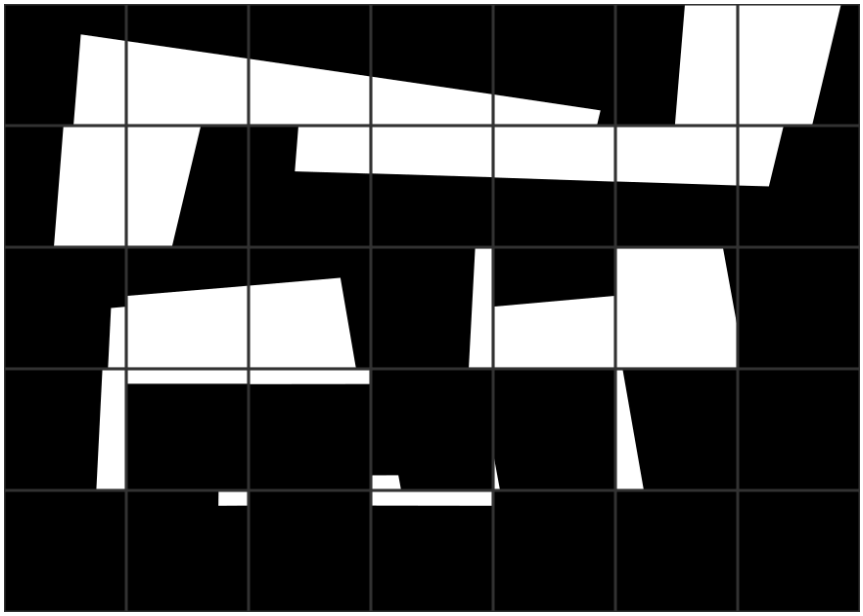
- Draw these opaque tiles using a fast path that simply render a quad without a mask.

Mask tiles

This leaves us with the partially covered tiles to deal with:



Partially covered tiles of each path are rendered into a portion of a floating point mask texture. In our example this mask texture might look like:

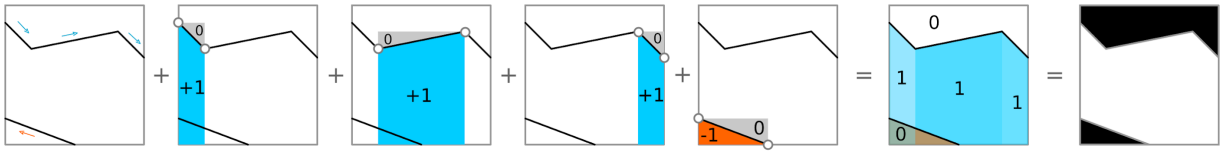


Rendering the mask tiles

Pathfinder can render tiles independently on the GPU as long as it knows the list of edges that are in the tile and whether the top-left corner of the tile is inside or outside.

At the moment, both are computed during the sweep-line pass in tiles.rs.

The work happening on the GPU looks like this:



On the figure above we can see quads used to render a mask tile. The bottom edge of each quad is snapped down to the bottom of the tile, while the top, and right edges correspond to the min and max of the edge being rendered. The code that positions the quad is in resources/shaders/fill.vs.glsl.

The fragment shader determines on which side of the edge the fragment is, and assigns either:

- +1 or -1 (depending to the edge's winding number) if the fragment is below.
- 0 if the fragment is above.
- A value in between for partially covered fragments, by computing the fragment coverage (see resources/shaders/fill.vs.glsl).

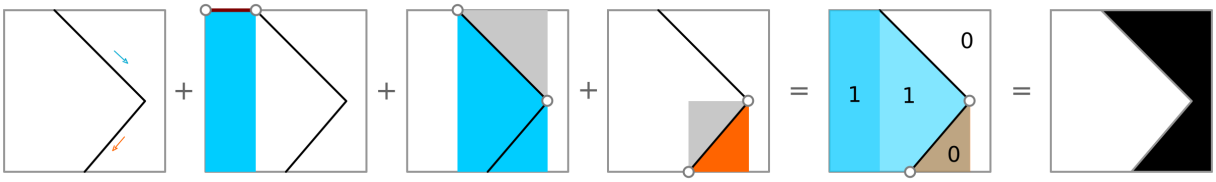
These quads are rendered with additive blending. In the resulting mask, each fragment is set to its own winding number and the rendering phase will be able to decide whether the fragment is in or out based on that number and a fill rule.

The way these edge quads are rendered is equivalent to casting a ray from the top and considering only the contribution of a single edge at a time for each quad. The additive blending lets us obtain the contribution for each edge.

A lot of renderers usually "cast the rays from the left" in the sense that edges are walked from left to right when counting coverage. This is purely a matter of convention and pathfinder could also do the latter by generating quads that extend to the right side of the tile instead of the bottom.

In the previous example the top of the tile was outside the path. It could have been inside the path due to contributions from edges above the tile. In order to take this into account, pathfinder

inserts extra edges at the top of the tile as necessary. The sweep line of the CPU tiling phase needs to keep track of the edges to insert at the to of the tile to get correct winding numbers.



Rendering strokes

Currently strokes are converted to fills on the CPU for convenience, There may be a stroke-specific mask tile generation implementation at some point (if need be).

Comparison with “traditional” stencil-and-cover

Stencil-and-cover is a broad family of path rendering approaches which consists in separating the shape and the shading two steps:

- In the first step (stencil), a mask is generated.
- In the second step (cover) the fragments of the final image are shaded using the previously generated mask to determine which fragments belong to the path.

In general, the stencil step consists in rendering a fan of triangles (one triangle per edge of the path) in the stencil buffer. In contrast, pathfinder uses a quad per edge and uses a float texture instead of the stencil buffer.
Traditional stencil-and-cover approaches requires rendering each path on by one, flipping back and forth between the stencil buffer and color target, while pathfinder can generate a mask tiles for all paths at once and then render all tiles in the color buffer in a single pass.

In addition, pathfinder’s tiling scheme provides coarse occlusion culling which helps with reducing memory bandwidth, avoiding work in occluded areas and reduce the memory needed to store the mask tiles.

NVPath style stencil-and-cover is probably the simplest approach to implement. However I’m not very enthused by that approach because:

- Overhead from switching back and forth between the stencil and the color target
- No occlusion culling to save memory bandwidth
- No great antialiasing solution (nvpath uses msaa which is expensive at high quality settings and very slow on intel GPUs).

Nvidia can paper over the driver overhead of switching between stencil and color targets by working inside the driver, and can muscle through Okay-ish anti-aliasing with large msaa resolutions (16x which is still a lot worse than pathfinder) on reasonably powerful discrete GPUs, but the approach is hard to make performant on other GPUs, especially intel.

Comparison with tessellation based approaches

Generating triangle meshes for paths using path tessellation is an appealing alternative. The main advantages would be:

- Rendering can happen directly in the color target without building a mask.
- The mesh doesn't depend on translation and rotation, so once tessellated it is cheap to animate.
    - If the tessellator handles curves, the geometry could also be made scale-independent.

It has, however, some important drawbacks:

- It is *very* hard to build a 100% robust tessellator supporting self-intersections because of floating point precision issues when a lot of paths self-intersect at the same location.
- Handling “hair-line” curvy shapes is hard. It either forces sub-dividing edges into very thin segments or some non-trivial way to render overlapping curve hulls. In contrast pathfinder handles this case by construction the same way it handles the easier configurations.
- There isn't a very satisfying solution for anti-aliasing as far as I know
    - Vertex-aa is hard to do correctly when with edges that are about the same size as a pixel or less
    - Post-processing approaches such as fxaa don't yield satisfying results
    - Msaa could be enough in *some* cases but it is very slow on intel GPUs, and it doesn't provide top-notch quality either.

Pathfinder's main strengths, in my opinion, are:

- The high quality anti-aliasing.
- An implementation that is simpler, less prone to bad edge cases, so less risky to ship.

Comparison with "Random-access of general vector graphics"

The paper by Diego Nehab and Hugues Hoppe describes a similar approach with a tiled representation of the paths. The same approach is also described in the Massively parallel vector graphics paper.

The main difference being that while WebRender renders masks tiles, the RAGVG paper proposes a scheme to render directly from the path description during shading.

Their solution is significantly more complicated but it does not suffer from the seams on coincident edges issue caused by anti-aliasing per path separately, which the majority of path renderers suffer from (pathfinder included).

The similar tiling setup is also good to keep in mind as a reference. It may (or may not) be faster than pathfinder's tiling stage.

What's interesting with their tiler is that it does not rely on a sweep-line pass. Paths are binned into tiles in parallel, and each edge that goes crosses the right side of a tile is assumed to then go upwards. This is not necessarily correct, but is easy to detect in a second step. When an error is introduced by this assumption, it is sufficient to update the winding number of all tiles on the left of the tile that introduced the error, which is cheap.

Skia

TODO: look at skia CCPR ("coverage counting path rendering")

How the tile decomposition is performed in more details

TODO

Sweep-line algorithm
renderer/src/tiles.rs

Populating the cpu z-buffer and the mask vertices is done in a single sweep-line phase per path. Curves are flattened while they are assigned to tiles so shaders only see line segments.

Observations

TODO

Parallel operations on the CPU

Pathfinder can generate the opaque tiles and mask vertices of each path in parallel using rayon. Each worker gets a path to render to. The "CPU z-buffer" (the 2D array of top-most opaque tiles) uses atomics and is written to in parallel.

On Patrick's MBP, this provides a 2x performance speedup, on my 5 years old laptop (2 cores), the performance is exactly the same as running the decomposition on a single thread without rayon.