How 2 draw lines in WebGL

that look like canvas 2d

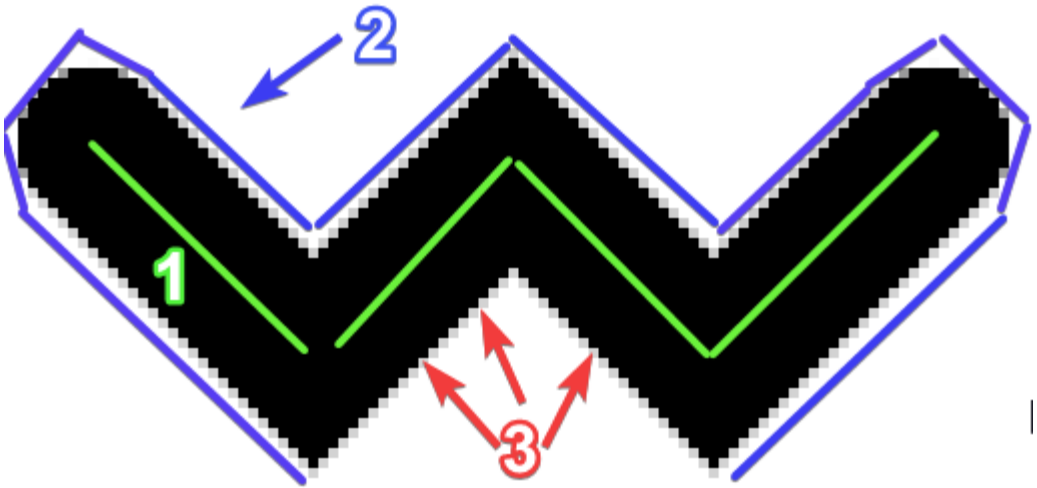
what took us so long?

by Ivan from PixiJS

and CRAZY PANDA

Plan

1. The problem
2. Prepare geometry
3. Coverage! <--- !!!magic!!!
4. Possible improvements

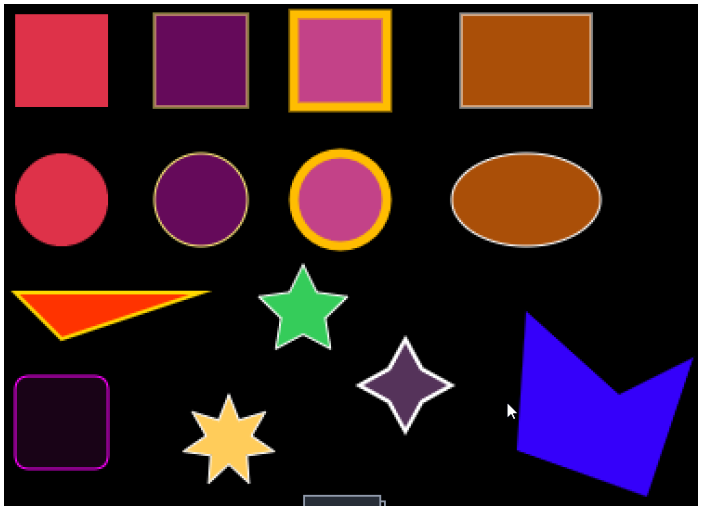


Part 1. The problem: no OSS solution!

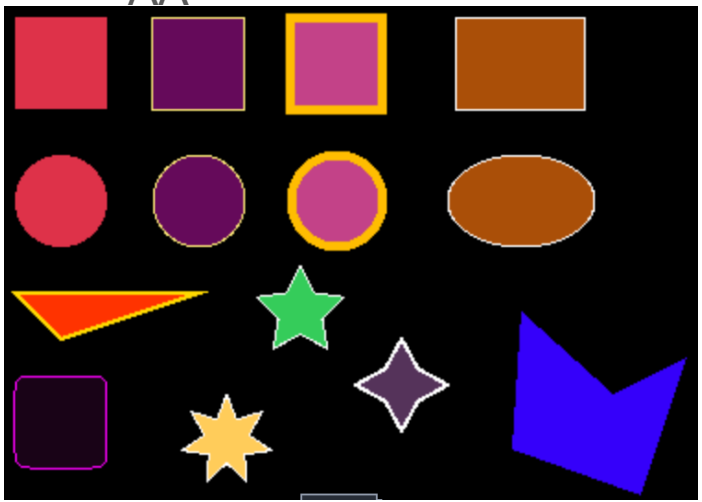
maybe even no commercial

getContext('2d') **vs** getContext('webgl')

PixiJS canvas 2d



PixiJS WebGL , no AA



How to draw line in WebGL

"Native" line in WebGL API

- drawArrays , LINE_STRIP
- lineWidth should be our solution

> As of January 2017 most implementations of WebGL only support a minimum of 1 and a maximum of 1 as the technology they are based on has these same limits.

Conclusion: only for debug

(wire mesh)

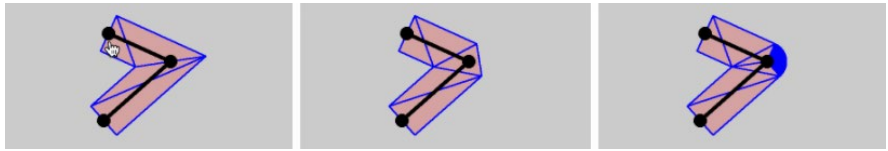
Make mesh look like line

Compute all points on CPU (PIXI.Graphics) based on path and

- lineStyle { width, color, texture }
- lineStyle { lineCap, lineJoin, miterLimit }
- lineStyle { alignment } ??? - to be explained

Vertex gets position, color, uv fragment color, uv

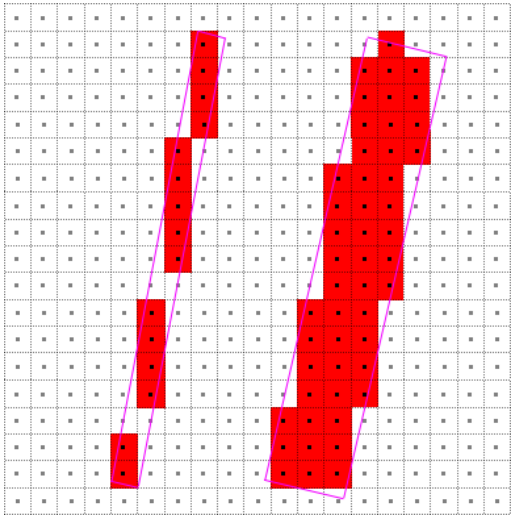
Supports joins because of our awesome community!



WebGL MultiSampling Anti-Aliasing (MSAA)

No MSAA

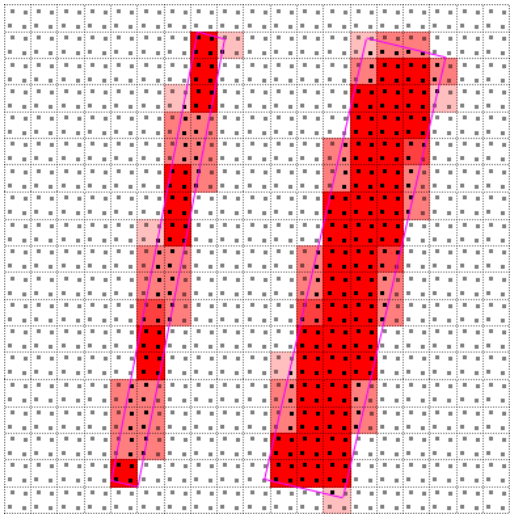
For every pixel that has center inside triangle, call fragment shader. Looks good only on vertical / horizontal lines.



MSAA x16

For every pixel store 16 samples, if at least one is covered - call fragment shader and blend the result to covered samples. BLIT to output.

- x6 memory Actually i do not remember
- slows down **everything**, not only lines
- For framebuffers - only in WebGL2
- renderbufferStorageMultisample on every framebuffer



MSAA x4 in example

How to draw a line in Canvas 2d

1. Simple beginPath, moveTo, lineTo, closePath
2. Store svg-like Path2D
3. Usual props: lineWidth, strokeStyle (color or pattern)
4. For experts: lineJoin, lineCap, miterLimit
5. Stroke the path
6. PROFIT!

Cannot change it:

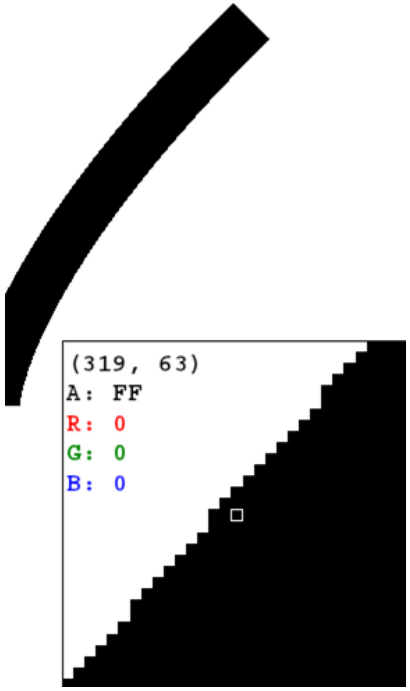
- Line is smoothed
- Width scales with matrix transform

JavaScript

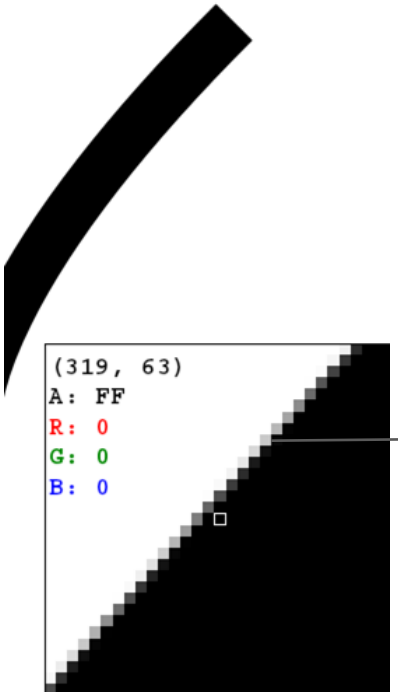
```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

ctx.moveTo(90, 130);
ctx.lineTo(95, 25);
ctx.lineTo(150, 80);
ctx.lineTo(205, 25);
ctx.lineTo(210, 130);
ctx.lineWidth = 15;
ctx.stroke();
```

Result

Source: https://skia.org/docs/dev/design/aaa/

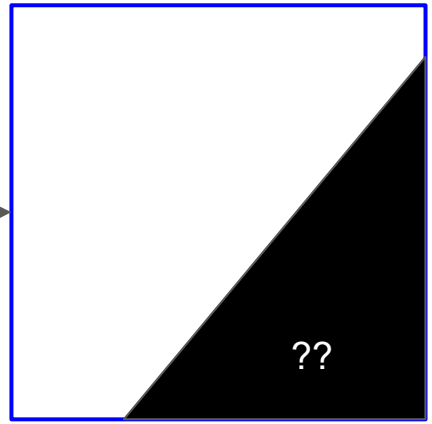


No AA



AA

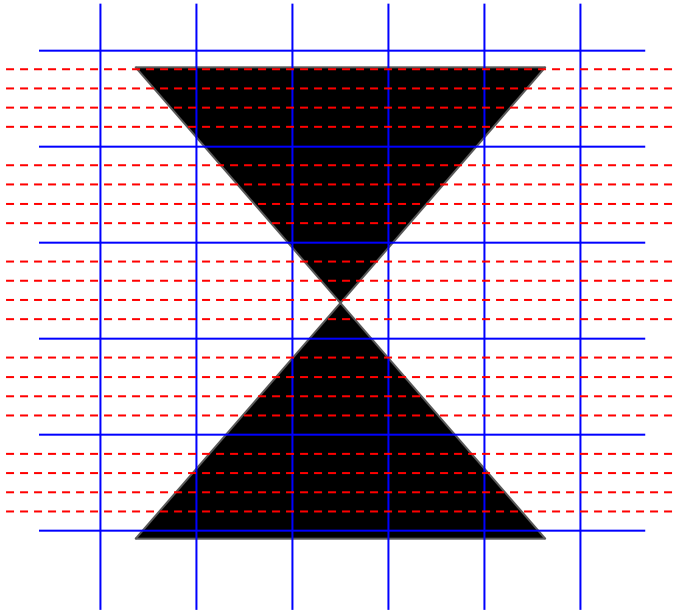
The number to compute:
coverage per pixel
(Area of the Intersection)



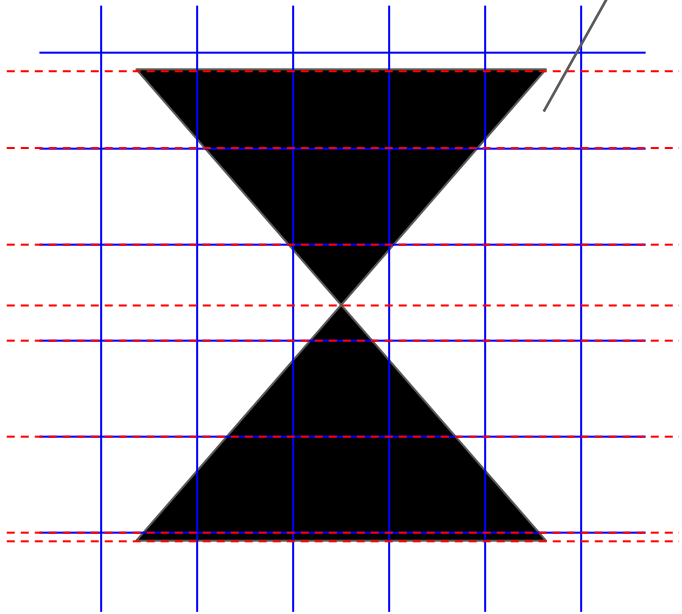
A single pixel
(unit area = 1.0 = 0xFF = 255)

Source: https://skia.org/docs/dev/design/aaa/

CPU Scanline: old vs. new



16x supersampling:
4 scan-lines per pixel



Analytic AA:
1 scan-line per pixel, per edge endpoint, per intersection

There are only trapezoids (triangles, rects) between scanlines

(similar to Cairo and Direct2D, except we never tesselate explicitly)

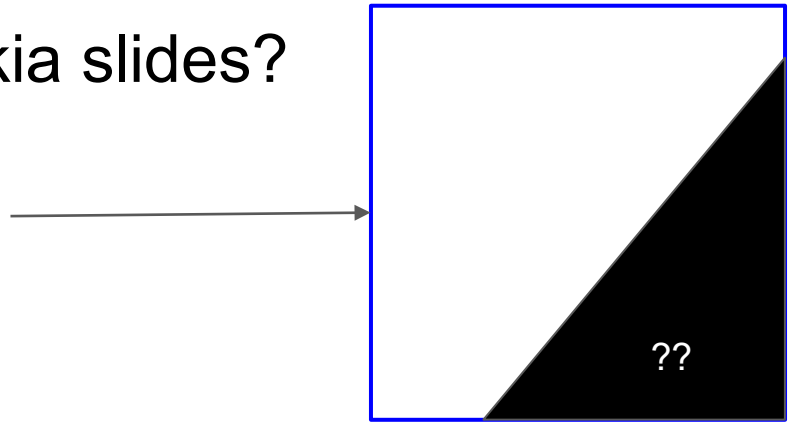
Conclusion: why not do the same for WebGL?

- Main part of algo - Scanline - relies on CPU. Javascript port would be slow.
- Even if we compute it with wasm, we have to texImage2D the result
- If it was possible, Skia guys would already have transferred everything to GPU
- At the least, it requires compute shaders (WebGPU)

What is important to us from skia slides?

Forget everything!

Calculate the coverage!



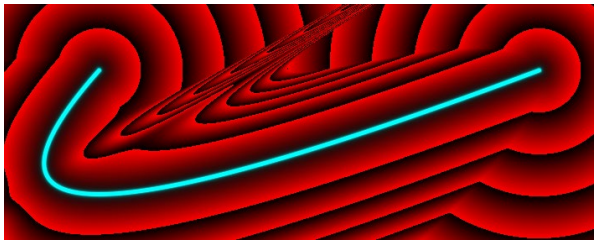
Bezier curves? SDF? More smart words?

Pretty things

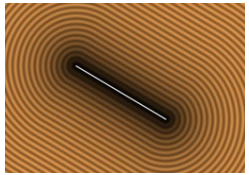
Shadertoy bezier curve by Taylor Holliday.

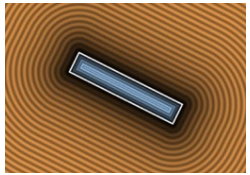
Note the interesting part:

```
if(d < thickness) {
  a = 1.0;
} else {
  // Anti-alias the edge.
  a = 1.0 - smoothstep(d, thickness, thickness+1.0);
}
```



Inigo Quilez collection of 2D shapes SDF





In production

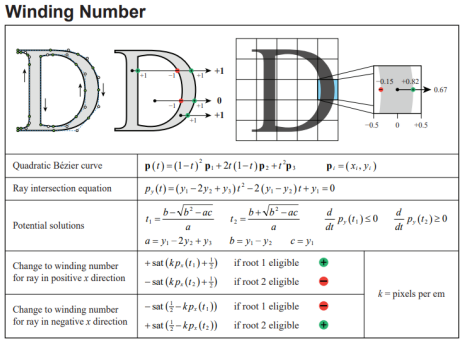
Slug library for font rendering

Pros:

- + Not just lines, any shapes!
- + Awesome!
- + Best quality!

Cons:

- - Commercial
- - Everything in fragment-shader



Summary

If everything is in fragment - it might be slow.

How much information can we read from data textures on slow devices?

How difficult is this math?

Actually, we might use for small parts.

Part 2. Prepare the geometry

No magic here.

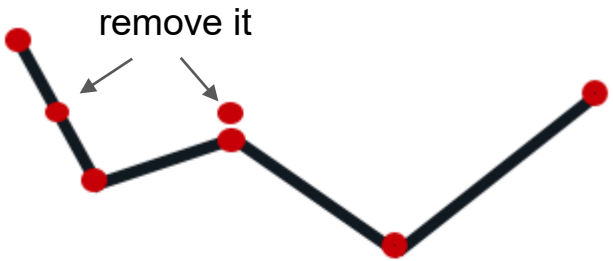
Many other renderers have geometry calculation with lineCap / lineJoin, for example AwayJS and Next2D . ThreeJS has fat lines

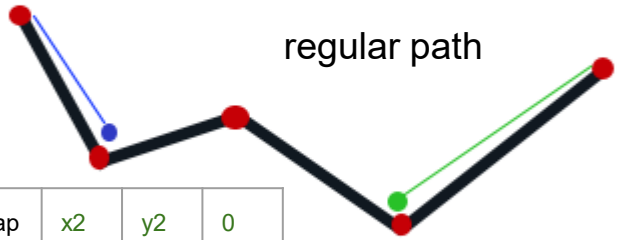
Those articles contain ideas for improvements, for example, how to move parts of geometry calculation to GPU:

- 2013 april, Robust polyline rendering with WebGL by Dan Bagnell
- 2015 march, Drawing Lines is Hard by Matt Deslauriers
- 2019 nov, Drawing lines in WebGL by Matt Stobbs
- 2019 nov, Instanced line rendering by Rye Terrell

Convert paths & points to joints

1. Remove duplicate points and unnecessary edges
2. Add extra end points
3. Assemble all into joint data
4. Concat joint data for all paths into one array





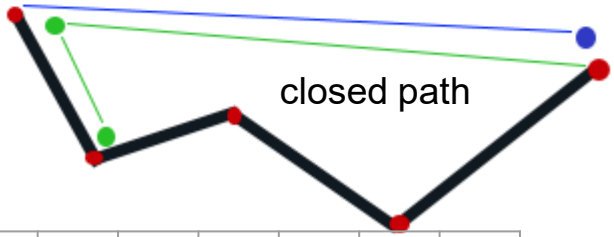
x1	y1	0	x0	y0	cap	x1	y1	join	x2	y2	join	x3	y3	cap	x2	y2	0

data needed by adjacent segments

type

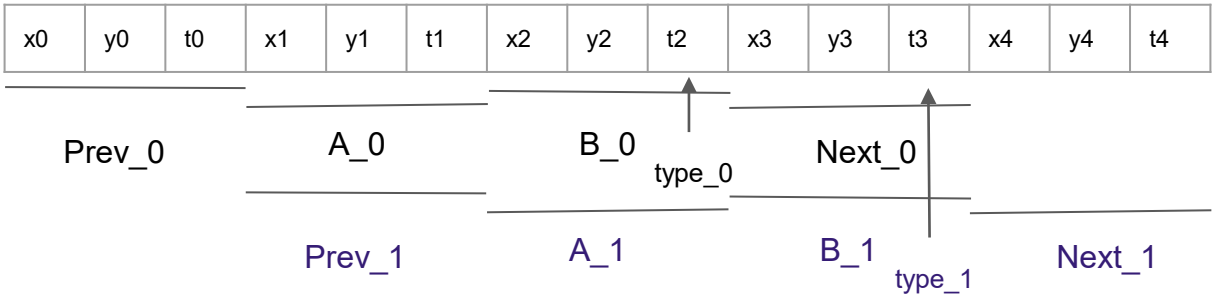
caps: square, butt, round

join: miter, bevel, round



x3	y3	0	x0	y0	0	x1	y1	join	x2	y2	join	x3	y3	join	x0	y0	join	x1	y1	0

Pack joints into instances



Vertex info has only vertex number 1-9

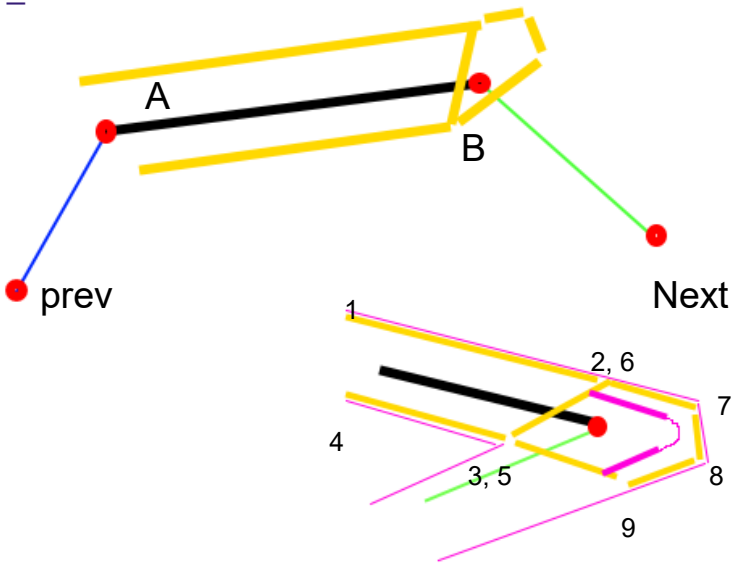
Two implementations:

- Instanced packer
- Per-vertex packer

Are basically the same, packer differs

Instance info:

- Each instance draws 1 segment and 1 joint
- Instances can intersect by data, use it
- 4 vertices per segment, 5 per joint (BEVEL) = 9 vertices per instance!
- Add style (width, color, texture)
- Add extra style (miterLimit, bevelLimit, alignment)
- In case ANGLE_instanced_arrays is not available, use per-vertex packer.
- You might need add extra joint in start or end of buffer, for padding



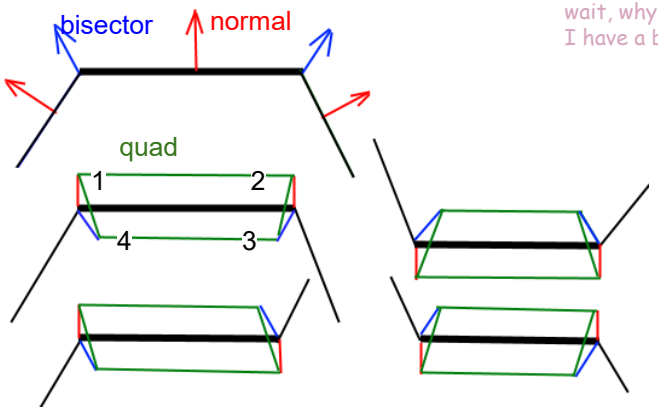
Vertex shader calculates vertex coords and something else

- The full formulas are out-of-scope of those slides, read referenced articles for more info
- There are a number of vectors we have to know about, everything else is usual dot() +- *scalar
- I'm trying to explain it without blowing up someone's brain, because this is not the main part of slides!

Vertex shader structure

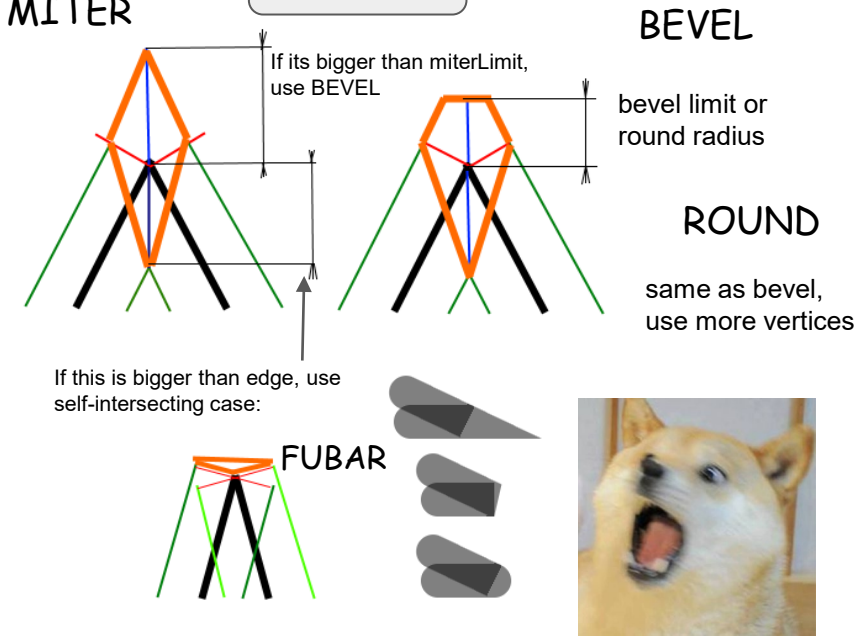
1. transform (A, B, prev, next, width) to **pixel coords**
2. calculate normals and bisectors
3. based on vertex number, calculate its position
4. pass some of style elements to fragment [varying floats]
5. pass distance to lines and width to [varying floats], just in case ;)

wait, why do we need it?
I have a bad feeling about this



Vertices 1-4

Vertices 5-9

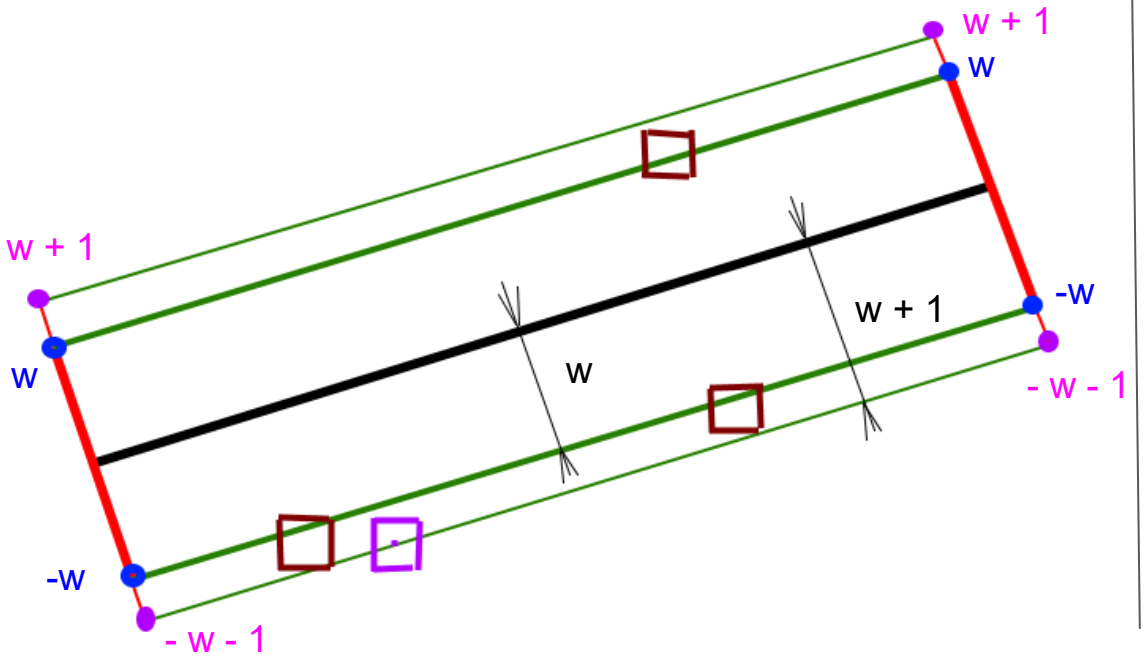


Part 3. Calculate the coverage with **HHAA**

Hatiko waited 10 years while we solve it.

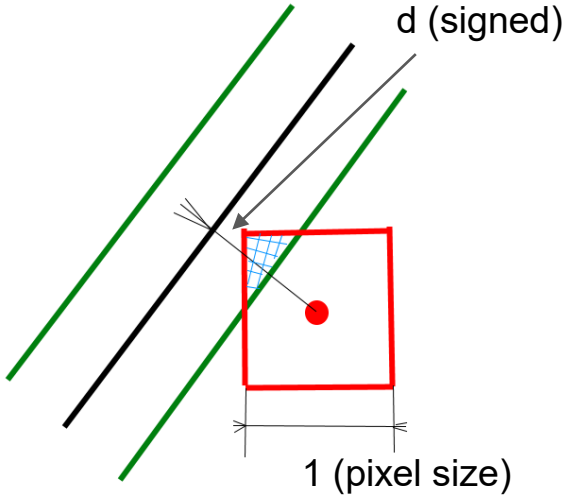
Suppose we have a single bar

- w = lineWidth/2
- For old vertex position calculation use "w+1" instead of "w", getting new position
- All pixels that were **covered only partially** (without center), now are covered
- Make sure that signed distance to line is passed to fragment accordingly



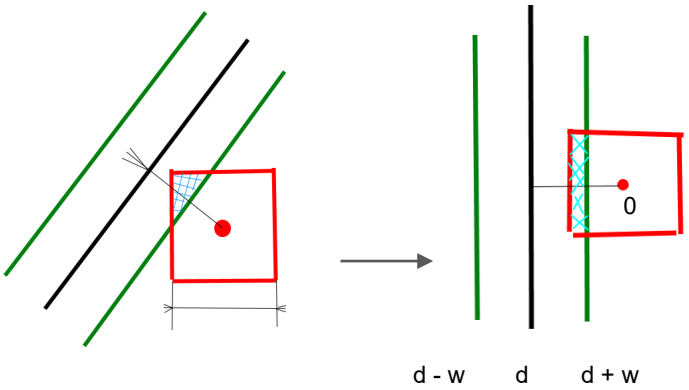
What happens if we pass signed distance from vertex shader to fragment through "varying"?

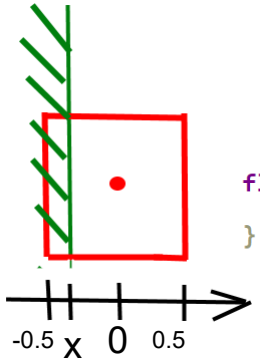
The answer is easy: in fragment shader we'll get signed distance from center of pixel to the line. Also pass the width, and we get enough info to calculate coverage with good precision.



Rotation? Ignore it for now.

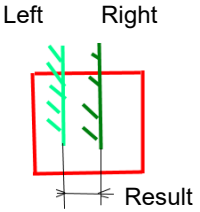
- Forget about rotation, error is about +-0.04 alpha
- Make a function that intersects single half-plane with a pixel
- Coverage of pixel is the difference of intersection with two half-planes





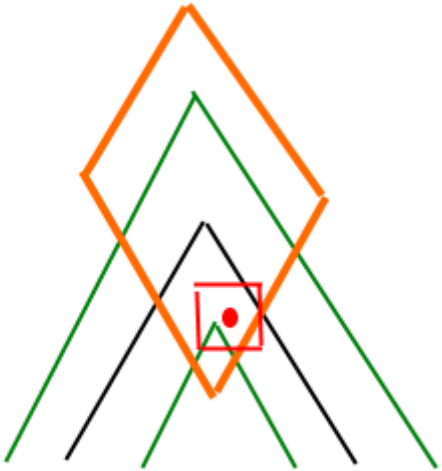

```
float pixelLine(float x) {
    return clamp(x + 0.5, 0.0, 1.0);
}
```

```
in vec4 vColor; //color style
in float vType; //type of segment/joint
in float d; //varying signed distance
in float w; //varying half-width

void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    float result = 0.0;
    if (vType == 0.0) {
        float left = pixelLine(d - w);
        float right = pixelLine(d + w);
        result = right - left;
    }
    fragColor = vColor * result;
}
```



Bet you can't do miter

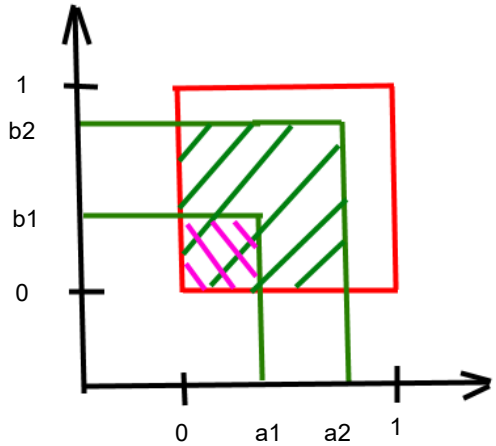
- Miter is a corner of two lines
- Lines are oriented
- Geometry is bigger because we use "w+1" instead of "w" in vertex
- Forget rotation, get that corner!



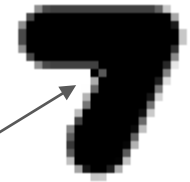
Using pixelLine() we can calculate a1,b1,a2,b2, those numbers are between 0 and 1.

Big square has area a2*b2, smaller has area is a1*b1

Result is a2*b2 - a1*b1



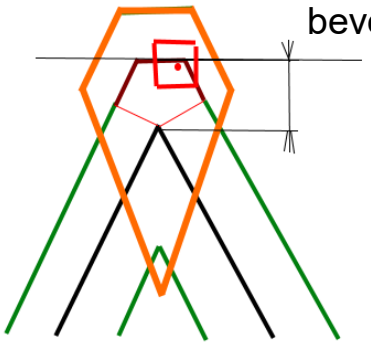

```
if (vType == 1.0) {
    float a1 = pixelLine(d1 - w);
    float a2 = pixelLine(d1 + w);
    float b1 = pixelLine(d2 - w);
    float b2 = pixelLine(d2 + w);
    result = a2 * b2 - a1 * b1;
}
```



If you dont subtract a1*b1
you'll get armpit hair

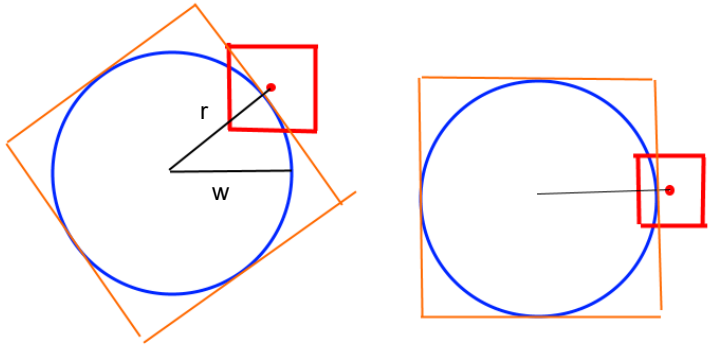
Bevel

- Basically MITER but with extra pixelLine
- To intersect MITER result with bevel half-plane use min() function
- Perfect result can be obtained with trigonometry, so let's not do that
- Suppose d3 is distance from center of pixel to bevel line

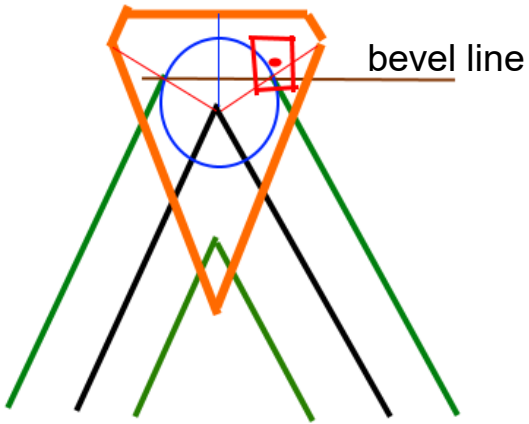



```
if (vType == 2.0) {
    float a1 = pixelLine(d1 - w);
    float a2 = pixelLine(d1 + w);
    float b1 = pixelLine(d2 - w);
    float b2 = pixelLine(d2 + w);
    float result_miter = a2 * b2 - a1 * b1;
    float result_bevel = pixelLine(d3);
    result = min(result_miter, result_bevel);
}
```

Round



- Join bevel half-plane with a circle! (take max)
- But how to intersect with circle?
- Suppose "circle" is vec2, vector relative to our joint point
- Calculate r, distance to center
- Imagine that circle is a square (Pi = 4, in wartime, its ok)
- Intersect two squares with pixelLine()




```
if (vType == 3.0) {
    float a1 = pixelLine(d1 - w);
    float a2 = pixelLine(d1 + w);
    float b1 = pixelLine(d2 - w);
    float b2 = pixelLine(d2 + w);
    float result_miter = a2 * b2 - a1 * b1;
    float result_bevel = pixelLine(d3);

    float r = length(circle.xy);
    float circle_hor = pixelLine(w + r) - pixelLine(-w + r);
    float circle_vert = min(w * 2.0, 1.0); //height of square
    float result_circle = circle_hor * circle_vert;

    result = min(result_miter, max(result_bevel, result_circle));
}
```

Pixel-perfect WIP

What if we actually want to handle rotation of pixel?

Its handy when the line is moving, its animated, +-0.04 alpha is bad precision

Consider **A** = max(abs(**n**.x), (**n**.y)), **B** = min(abs(**n**.x), abs(**n**.y)) where **n** is normal

```
float pixelLine(float x, float A, float B) {
    float y = abs(x), s = sign(x);
    if (y * 2.0 < A - B) {
        return 0.5 + s * y / A;
    }
    y -= (A - B) * 0.5;
    y = max(1.0 - y / B, 0.0);
    return (1.0 + s * (1.0 - y * y)) * 0.5;
    //return clamp(x + 0.5, 0.0, 1.0);
}
```

HHAA

Scanline

Part 4. We have solution. What is next?

PixiJS Graphics Smooth

PixiJS library graphics-smooth is drop-in replacement for "PIXI.Graphics". All users can adopt it in their project and specify "antialias:false" for WebGL.

pixi-candles is light version that draws a single line - it should be easy to port the code to other renderers.

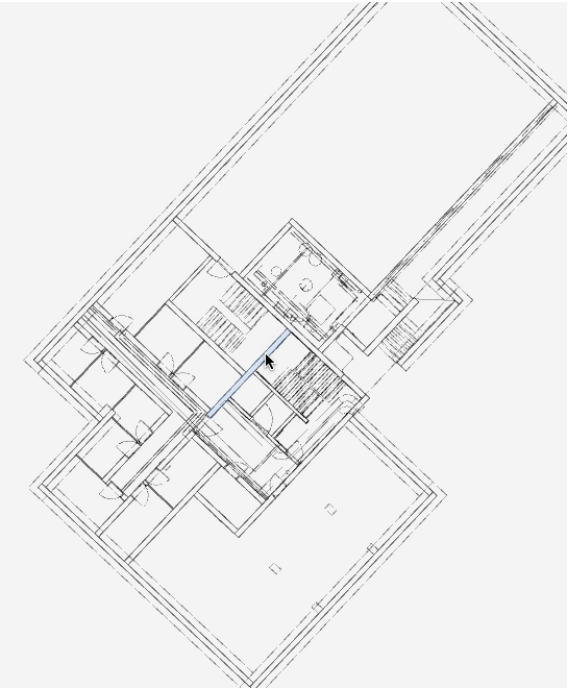
Those two libs are first implementations of **HHAA**

Shader code: 350 LoC for vertex 60 for fragment

FoundryVTT , one of biggest contributors to PixiJS are the first adopters, they pushed it in their dev build in the first week after library was published.

Thin lines problem

It all started in the issue for CAD software



Extra style parameters

Line Scale Mode

- Adobe Flash Graphics
- Specifies how to scale line width depending on transform
- Line width is the same vertically and horizontally
- Very useful for scaling plots!
- Many requests in PixiJS issues

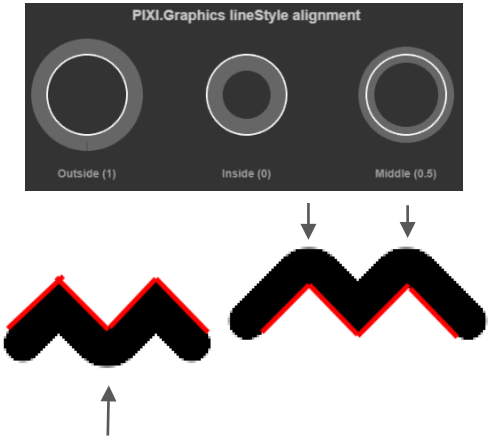
vertical horizontal

$$\begin{bmatrix} a & c & t_x \\ b & d & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

```
float avgScale = 1.0;
if (scaleMode > 2.5) {
    avgScale = length(translationMatrix * vec3(1.0, 0.0, 0.0));
} else if (scaleMode > 1.5) {
    avgScale = length(translationMatrix * vec3(0.0, 1.0, 0.0));
} else if (scaleMode > 0.5) {
    vec2 avgDiag = (translationMatrix * vec3(1.0, 1.0, 0.0)).xy;
    avgScale = sqrt(dot(avgDiag, avgDiag) * 0.5);
}
lineWidth *= 0.5 * avgScale;
```

Line Alignment

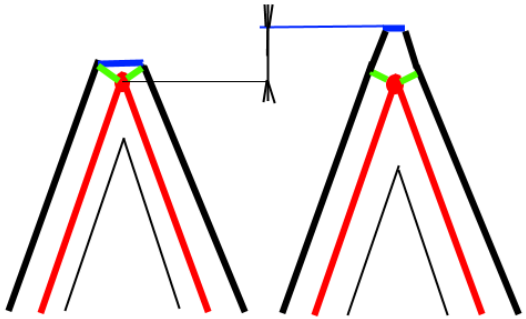
- Pixi Graphics
- Shifts the line
- Added by Mat Groves
- Cannot implement in canvas2d because it affects joins!
- Outline shapes inside or outside, maybe with gradient



Bevel Limit WIP

- Bevel sometimes looks bad on plots
- Bevel looks bad with alignment=0
- If equal to miter limit, smooth transition between miter and bevel
- Extra shift of bevel line

bevelLimit * lineWidth/2



Batching & Performance

Attributes:

- Color

Style:

- Width
- Texture
- Gradient WIP
- Dash WIP

Extra:

- MiterLimit WIP
- BevelLimit WIP
- Alignment

My general approach on batching:

How to separate style into attributes, style and extra style? Its up to you.

Style & Extra can be stored in UBO or Data Textures, just include their ID in attribute to reference it in vertex shader.

Animated props are better in styles, that way you can animate single uniform or UBO without re-uploading the buffer.

Props that are different for many objects should go in attributes, because UBO is limited.

HHAA successfully works in production, I did not do any benchmarks yet, it's just looks fast.

If you want a benchmark, please help me by posting a demo :)

Thank you for watching!

Just post PixiJS issue somewhere and I will find you!

[Html5 gamedevs forum]
[PixiJS discussions]