Pushing HTML Canvas to the limit

Victor Fernandes

github | gmail | npm / @victorfern91

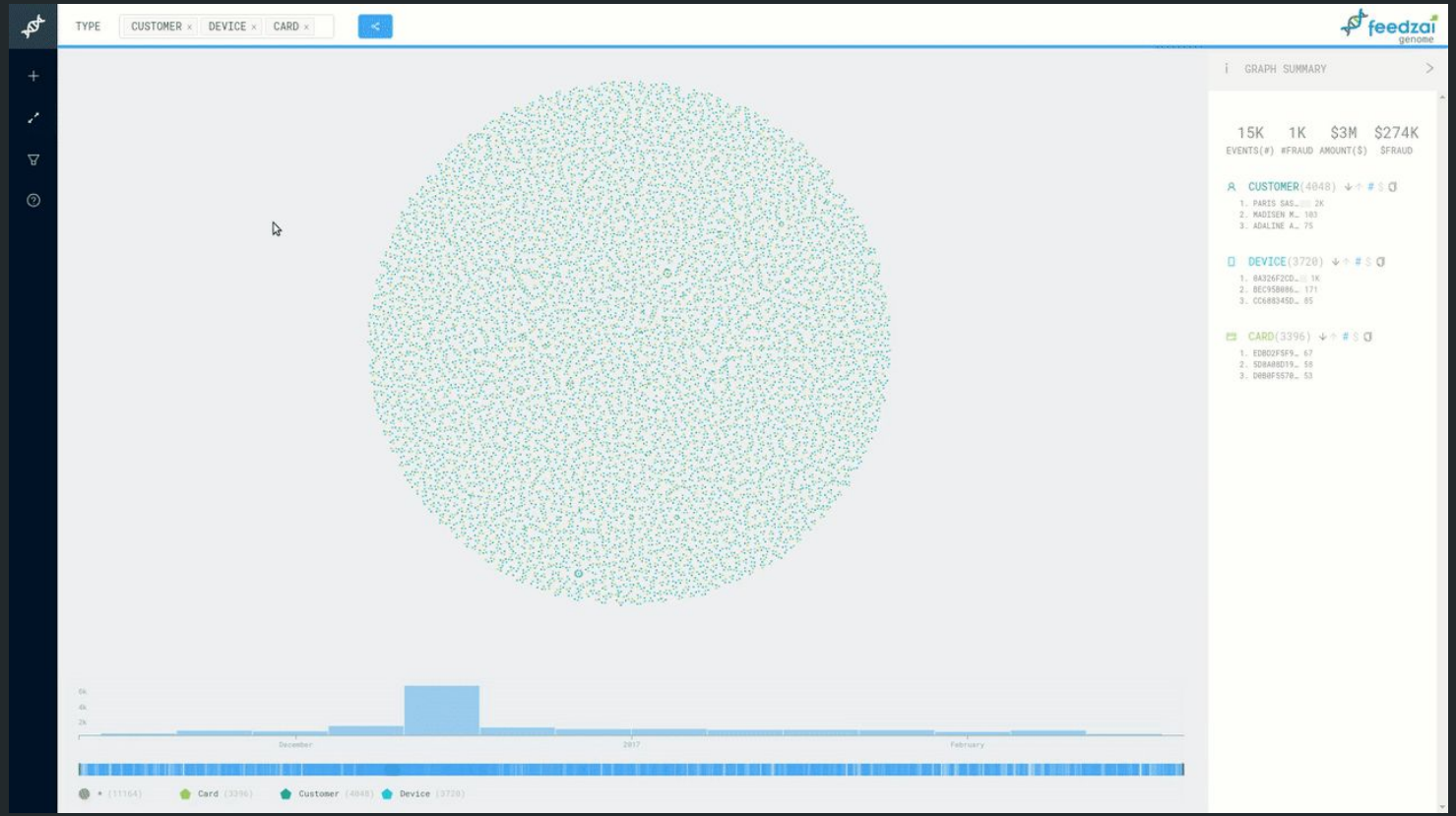
About me

- I was a C++ developer for fews months, and I found JavaScript @ MOG Technologies.
- Currently, I work as Front-end Engineer @ Feedzai
- DevOps enthusiast

About this talk

- At feedzai we’re developing a link-analysis tool called Genome.
- We’ve a lot of data to render in Genome.
- I’ll show the reason behind the decision of using canvas.
- Some tricks that you could use to improve your performance when using Canvas.
- I’ll show you some interactive examples.

About this talk

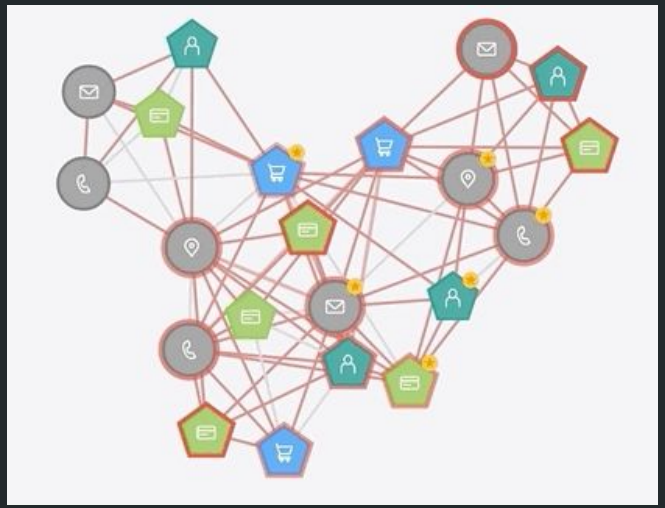


About this talk

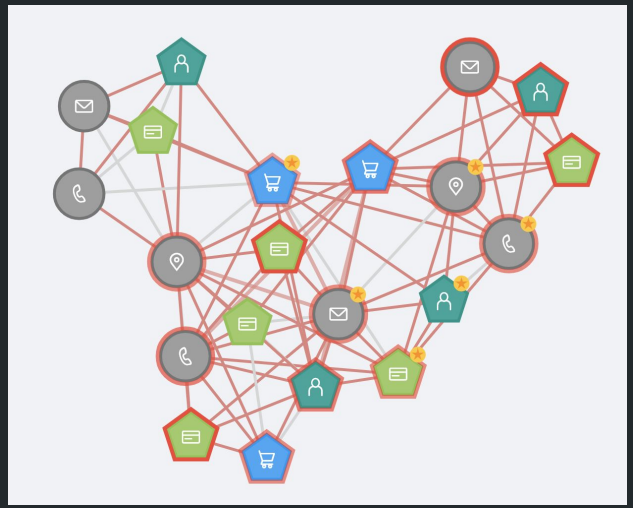
- At feedzai we’re developing a link-analysis tool called Genome.
- We’ve a lot of data to render in Genome.
- Some techniques that you could use to improve your performance when using Canvas.
- I’ll show you some interactive examples.

Some context about d3-force (graph engine)

Graph simulation (physics calculations)



Graph simulation end (static graph)



Why we choose canvas?

- Using SVG was a fiasco. (when we have > 200 elements)
- Canvas performance was good for our use case.
- WebGL?
  - The performance was very good.
  - Just 1 Front-end Developer in this product.
  - Raw WebGL it’s hard!
- d3-force simulation is the real bootle-neck.

The actual technology for our link analysis tool was: **Canvas** but we could migrate to **WebGL** in a near future.

What is HTML canvas?

“Added in HTML5, the HTML <canvas> element can be used to draw graphics via scripting in Javascript. [...] it can be used to draw graphs, make photo compositions, create animations, [...] video processing or rendering.”

API Canvas - MDN webpage

Problem:
It is really hard show more than 15k elements without improvements when you’re using Canvas

I will show you some improvements that you could use in canvas

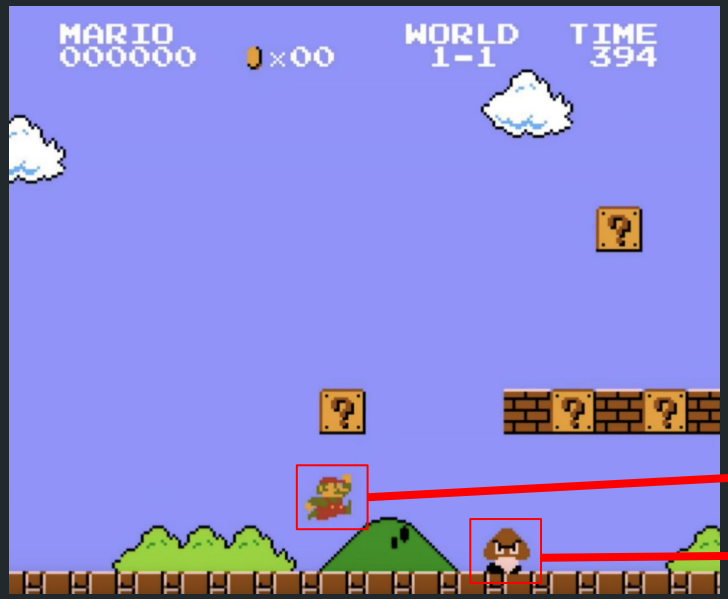
Render only the screen differences

Frame 1

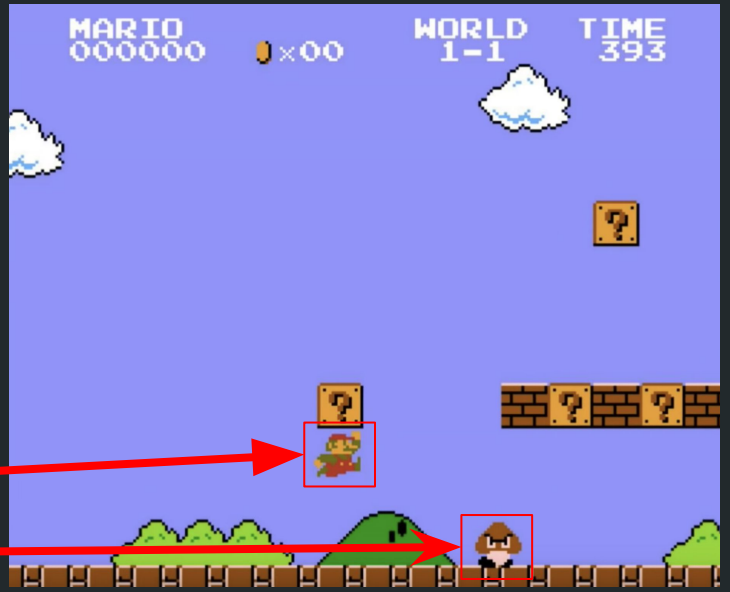
Frame 2

Render only the screen differences

Frame 1



Frame 2



Render only the screen differences

Frame 2 - t0

Frame 2 - t1

clearRect vs. fillRect

- This methods are used to clear drawn elements in canvas.
- https://jsperf.com/clearrect-vs-fillrect/9
- The clearRect is the faster and safe way to clear canvas.

Batch canvas calls together

- Drawing is an expensive operation.
- It’s more efficient to create one path with all the lines and draw it with a single draw call.

```
for (let i = 0, length = points.length -1; i < length; ++i) {
    const p1 = points[i];
    const p2 = points[i+1];
    context.beginPath();
    context.moveTo(p1.x, p1.y);
    context.lineTo(p2.x, p2.y);
    context.stroke();
}
```

Bad

```
context.beginPath();
for (let i = 0, length = points.length -1; i < length; ++i) {
    const p1 = points[i];
    const p2 = points[i+1];
    context.moveTo(p1.x, p1.y);
    context.lineTo(p2.x, p2.y);
}
context.stroke();
```

Good

- https://jsperf.com/batching-line-drawing-calls/47

setInterval vs. requestAnimationFrame

- setInterval

```
// 60 fps = 1/60 = 16.666..7
setInterval(() => {
    console.log(`I'm running at 60 fps!`);
}, 16);
```

  - Wildly inconsistent
  - Must track time manually
  - Runs on background

- requestAnimationFrame

```
const loop = () => {
  console.log(`I'm running at 60fps!`);

  requestAnimationFrame(loop);
}

requestAnimationFrame(loop);
```

  - Strongly consistent
  - Optimized for smooth animations
  - Doesn’t run on background

setInterval vs. requestAnimationFrame

Demo

http://localhost:8080/examples/time

setInterval vs. requestAnimationFrame

Demo

http://localhost:8080/examples/time

Advice:
For better results you should use the **requestAnimationFrame**

Memoization

- Should be used to improve performance in heavy computations.
- In order to memoize a function it should be pure.
- Memoization it’s always a trade-off between the speed improvements and the memory used.
- Use it with caution!

At genome we’ve used memoization in heavy computations like:

- The edges color through fraud ratio.
- sin/cos values for a specific angle.
- etc..

Memoization example

```
// A simple memoize implementation
const memoize = (fn) => {
    // Memoize cache
    const cache = new Map();

    return function () {
        // key to store/search function output in cache
        const key = JSON.stringify(arguments);

        if (cache.has(key)) {
            return cache.get(key);
        } else {
            // calculate function output
            const output = fn.apply(this, arguments);

            // setting output in cache
            cache.set(key, output);

            return output;
        }
    };
};
```

Memoization

- Should be used to improve performance in heavy computations.
- In order to memoize a function it should be pure.
- Memoization it’s always a trade-off between the speed improvements and the memory used.
- Use it with caution!

At genome we’ve used memoization in heavy computations like:

- The edges color through fraud ratio.
- sin/cos values for a specific angle.
- etc..

Data structures

- Whenever is possible change the usage of arrays by sets, if you're using the .includes method that will reduce the complexity from O(n) to O(1)
  https://jsperf.com/set-vs-array-find-values/1

- It's a common practice use objects in order to have a hash table. You could use Map or Set to be faster.

  Map version:
  https://jsperf.com/map-vs-object-hash-table/1

Data structures

```
const coolGuys = [
    // id: 0
    // ...
    { id: "99999", name: "Victor", age: 27, mood: "😎" }
    // ...
]

// converted  to Array => Object

// Hash Table (community way)
const coolGuysObj = {
    "99999": { name: "Victor", age: 27, mood: "😎" }
};

if (coolGuysObj.hasOwnProperty("99999")) {
    console.log(`Hey! We've found ${coolGuysObj["99999"].name} 👸 !`);
}

// The better way (Map)
const coolGuysMap = new Map();

coolGuysMap.add("99999", { name: "Victor", age: 27, mood: "😎" });

if (coolGuysMap.has("99999")) {
    console.log(`Hey! We've found ${coolGuysMap.get("99999").name} 👸 !`);
}
```

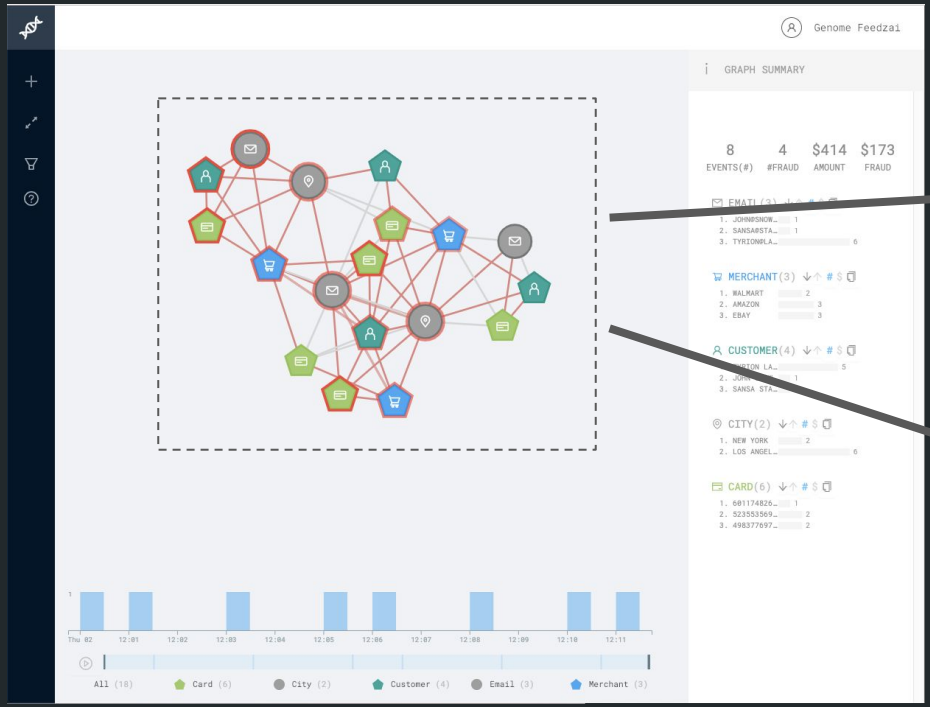
The avoidables

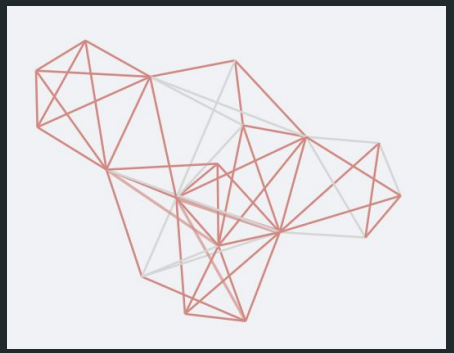
- Floating-points coordinates (interpolation/sub-pixeling)
  http://localhost:8080/examples/pixel

- Avoid declaring variables/objects inside of loops

- Text rendering (is a very expensive operation)

Using multiple layers for complex scenes

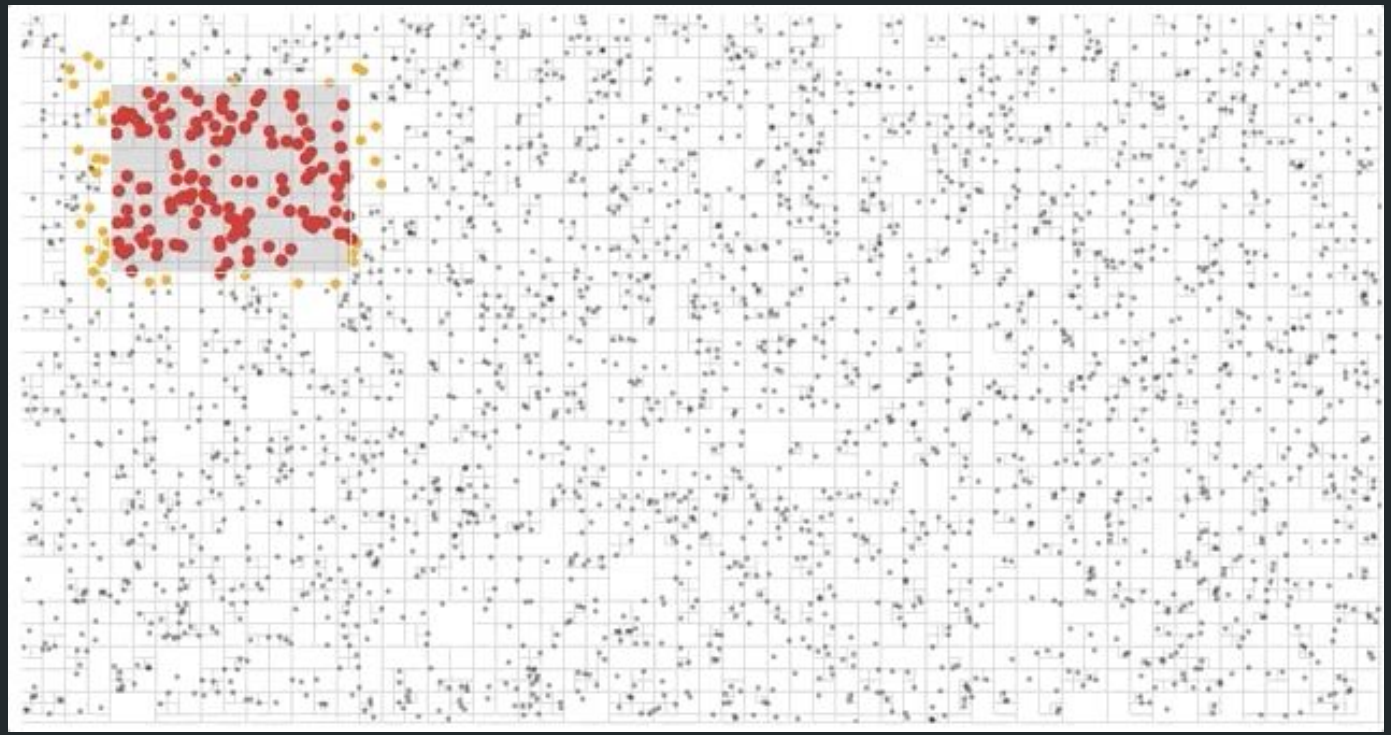




Lazy rendering

- Render only the visible elements. That could be done by using:
  - Interval filters - inefficient
  - quadtree - Most efficient (for our use case), but is very intensive during d3-force graph simulation.
  - We’re using an Hybrid mode
- Render graph details through the scrolling value

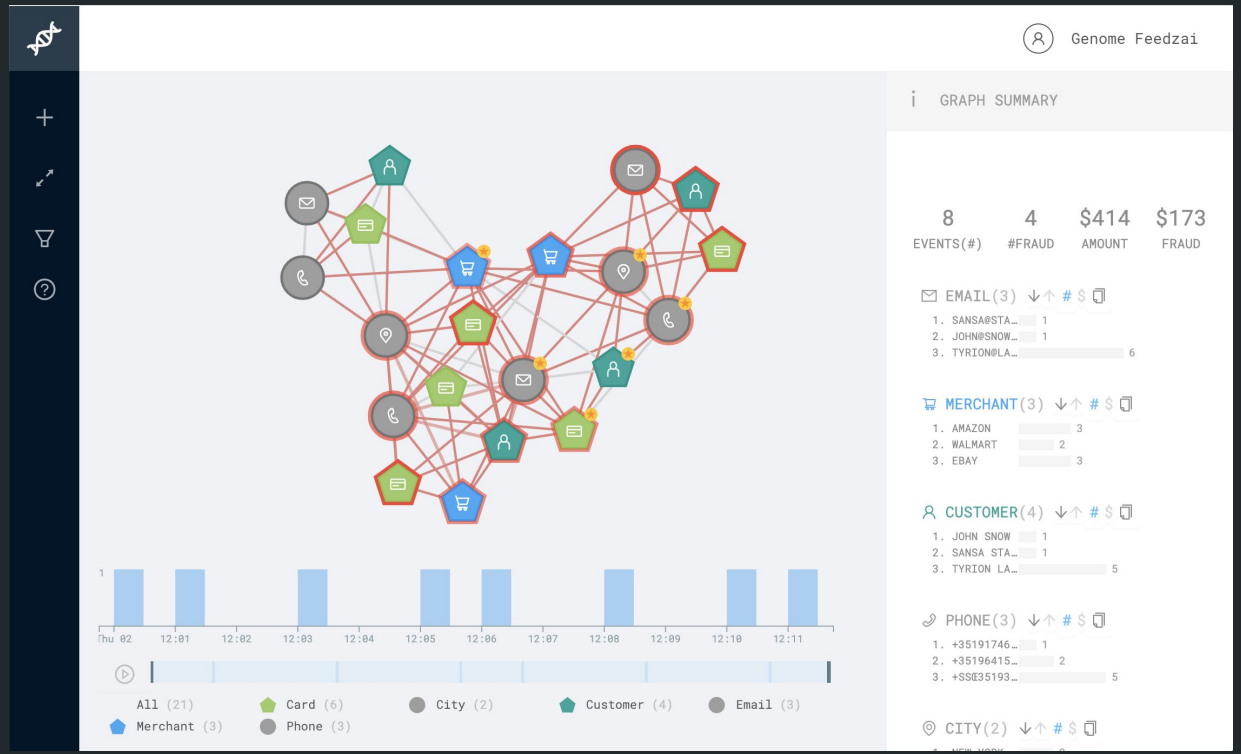
Lazy rendering - quadtree



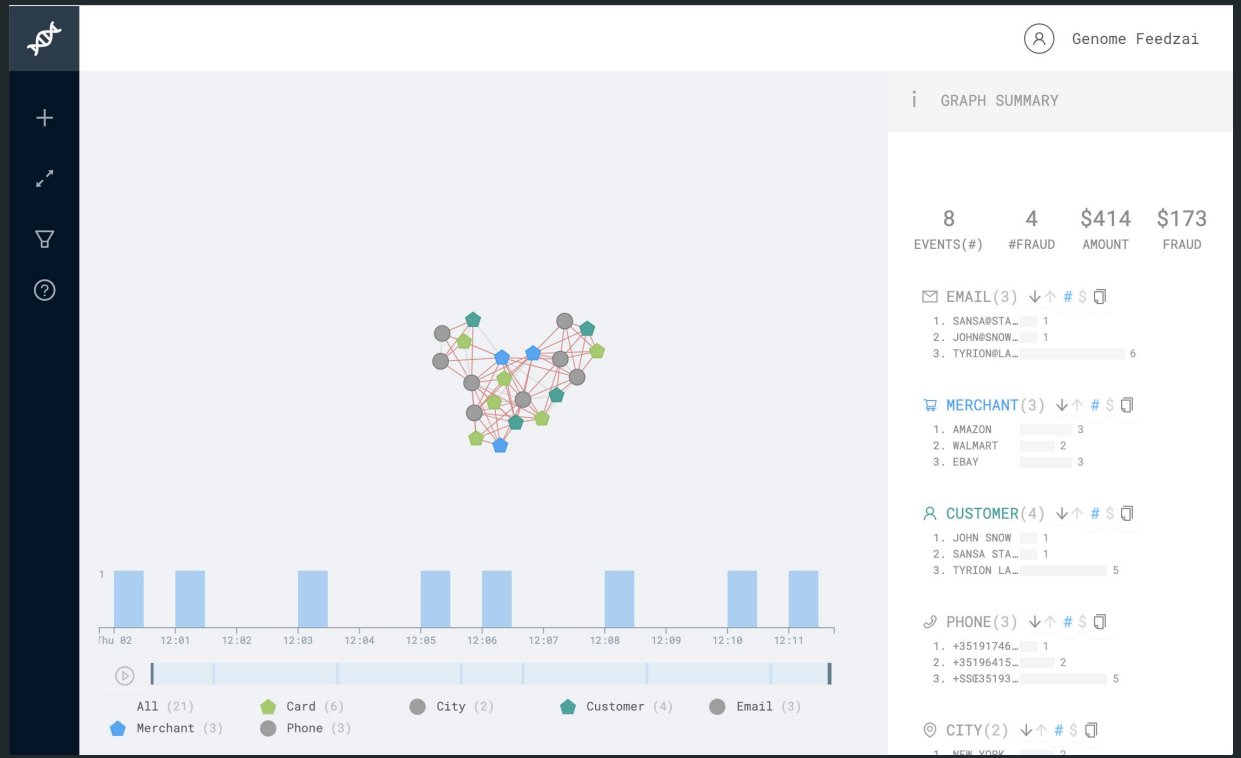
Lazy rendering

- Render only the visible elements. That could be done by using:
  - Interval filters - inefficient
  - quadtree - Most efficient, but is very intensive during d3-force graph simulation.
  - We’re using an Hybrid mode
- Render graph details through the scrolling value

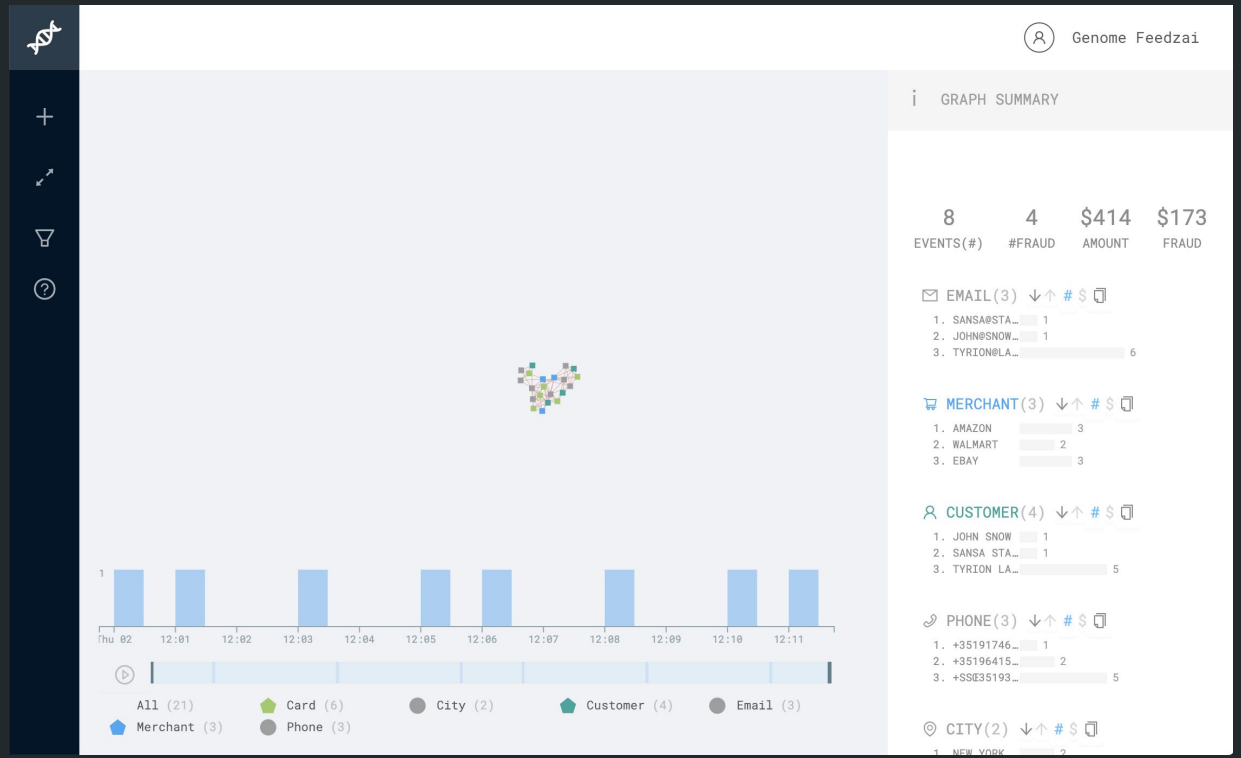
Lazy rendering - Using to show/hide details



Lazy rendering - Using to show/hide details



Lazy rendering - Using to show/hide details



Shapes matters

- Render squares is faster than circles

Demo

http://localhost:8080/examples/shapes

Speedup canvas interactions

- Use power of math - good for a small amount of elements

```
// Rectangle
if (clickPosition.x >= rect.x
    && clickPosition.x <= rect.x + rect.width
    && clickPosition.y >= rect.y
    && clickPosition.y <= rect.y + rect.height) {
    // The click was inside the rectangle
}
```

```
// Circle
if (Math.sqrt((circle.x - click.x) ** 2 + (circle.y - click.y) ** 2) < RADIUS) {
    // The click was inside the circle
}
```

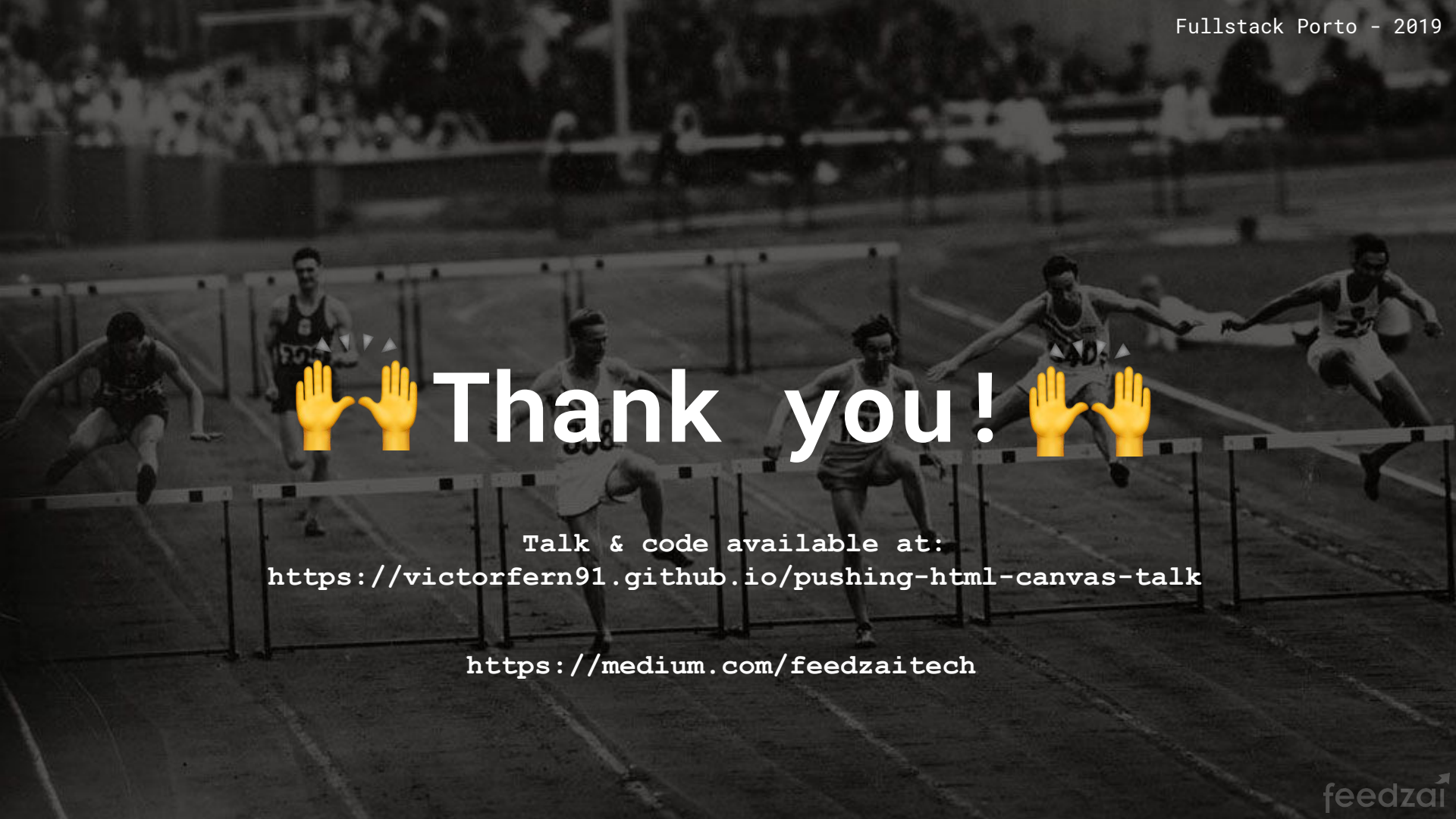
- quadtree - Best option
- By using a hidden canvas - Faster, but too expensive

  http://localhost:8080/examples/hitbox

  http://localhost:8080/examples/hitbox-hidden

Conclusions

- In our use case if we need to render more than 25k elements in browser we must consider use WebGL
- Be careful with heavy physics libraries
- Try use different data structures in order to reduce the time complexity
- Be creative and do performance tests in your code



🙌 Thank you! 🙌

Talk & code available at:
https://victorfern91.github.io/pushing-html-canvas-talk

https://medium.com/feedzaitech

Map

	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Android webview	Chrome for Android	Edge Mobile	Firefox for Android	Opera for Android	Safari on iOS	Samsung Internet	Node.js
Basic support	38	12	13	11	25	8	38	38	12	14	25	8	Yes	0.12
`new Map(iterable)`	38	12	13	No	25	9	38	38	12	14	25	9	Yes	0.12
`new Map(null)`	Yes	12	37	11	Yes	9	Yes	Yes	12	37	Yes	9	Yes	0.12
`Map()` without `new` throws	Yes	12	42	11	Yes	9	Yes	Yes	12	42	Yes	9	Yes	0.12
Key equality for -0 and 0	38	12	29	No	25	9	38	38	12	29	25	9	Yes	4.0.0
`clear`	38	12	19	11	25	8	38	38	12	19	25	8	Yes	0.12
`delete`	38	12	13	11	25	8	38	38	12	14	25	8	Yes	0.12
`entries`	38	12	20	No	25	8	38	38	12	20	25	8	Yes	0.12
`forEach`	38	12	25	11	25	8	38	38	12	25	25	8	Yes	0.12
`get`	38	12	13	11	25	8	38	38	12	14	25	8	Yes	Yes
`has`	38	12	13	11	25	8	38	38	12	14	25	8	Yes	Yes
`keys`	38	12	20	No	25	8	38	38	12	20	25	8	Yes	0.12
`prototype`	38	12	13	11	25	8	38	38	12	14	25	8	Yes	Yes
`set`	38	12	13	11 *	25	8	38	38	12	14	25	8	Yes	Yes
`size`	38	12	19 *	11	25	8	38	38	12	19 *	25	8	Yes	0.12
`values`	38	12	20	No	25	8	38	38	12	20	25	8	Yes	0.12

Set

	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Android webview	Chrome for Android	Edge Mobile	Firefox for Android	Opera for Android	Safari on iOS	Samsung Internet	Node.js
Basic support	38	12	13	11	25	8	38	38	12	14	25	8	Yes	0.12
`new Set(iterable)`	38	12	13	No	25	9	38	38	12	14	25	9	Yes	0.12
`new Set(null)`	Yes	12	37	11	Yes	9	Yes	Yes	12	37	Yes	9	Yes	0.12
`Set()` without `new` throws	Yes	12	42	11	Yes	9	Yes	Yes	12	42	Yes	9	Yes	0.12
Key equality for -0 and 0	38	12	29	No	25	9	38	38	12	29	25	9	Yes	4.0.0
`add`	38	12	13	11 *	25	8	38	38	12	14	25	8	Yes	Yes
`clear`	38	12	19	11	25	8	38	38	12	19	25	8	Yes	0.12
`delete`	38	12	13	11	25	8	38	38	12	14	25	8	Yes	0.12
`entries`	38	12	24	No	25	8	38	38	12	24	25	8	Yes	0.12
`forEach`	38	12	25	11	25	8	38	38	12	25	25	8	Yes	0.12
`has`	38	12	13	11	25	8	38	38	12	14	25	8	Yes	Yes
`prototype`	38	12	13	11	25	8	38	38	12	14	25	8	Yes	Yes
`size`	38	12	19 *	11	25	8	38	38	12	19 *	25	8	Yes	0.12
`values`	38	12	24	No	25	8	38	38	12	24	25	8	Yes	0.12