Drawing Huge Plots on the Web

Mikola Lysenko and Perouz Taslakian
plot.ly

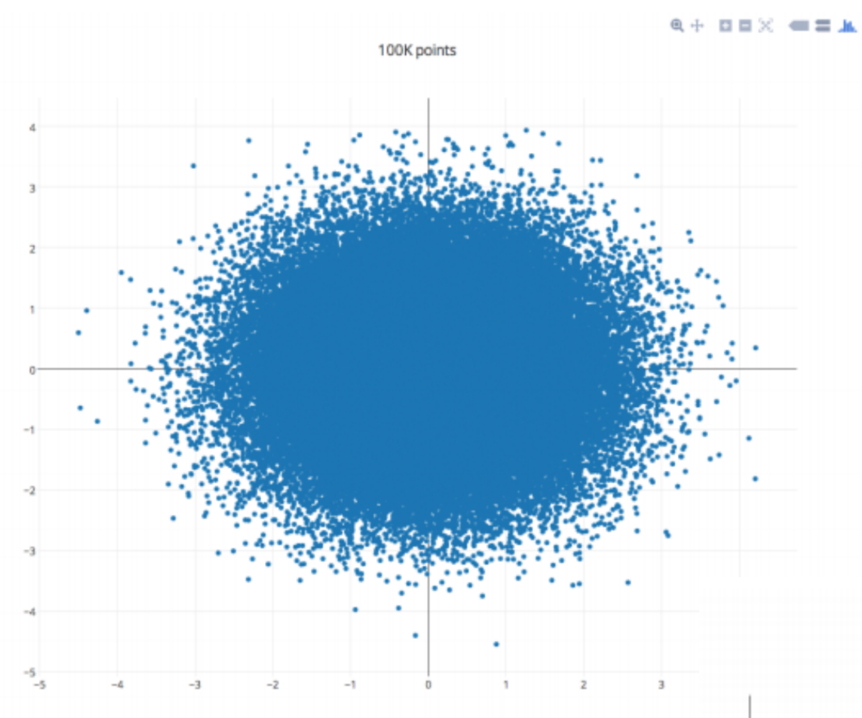
plotly

Requirements

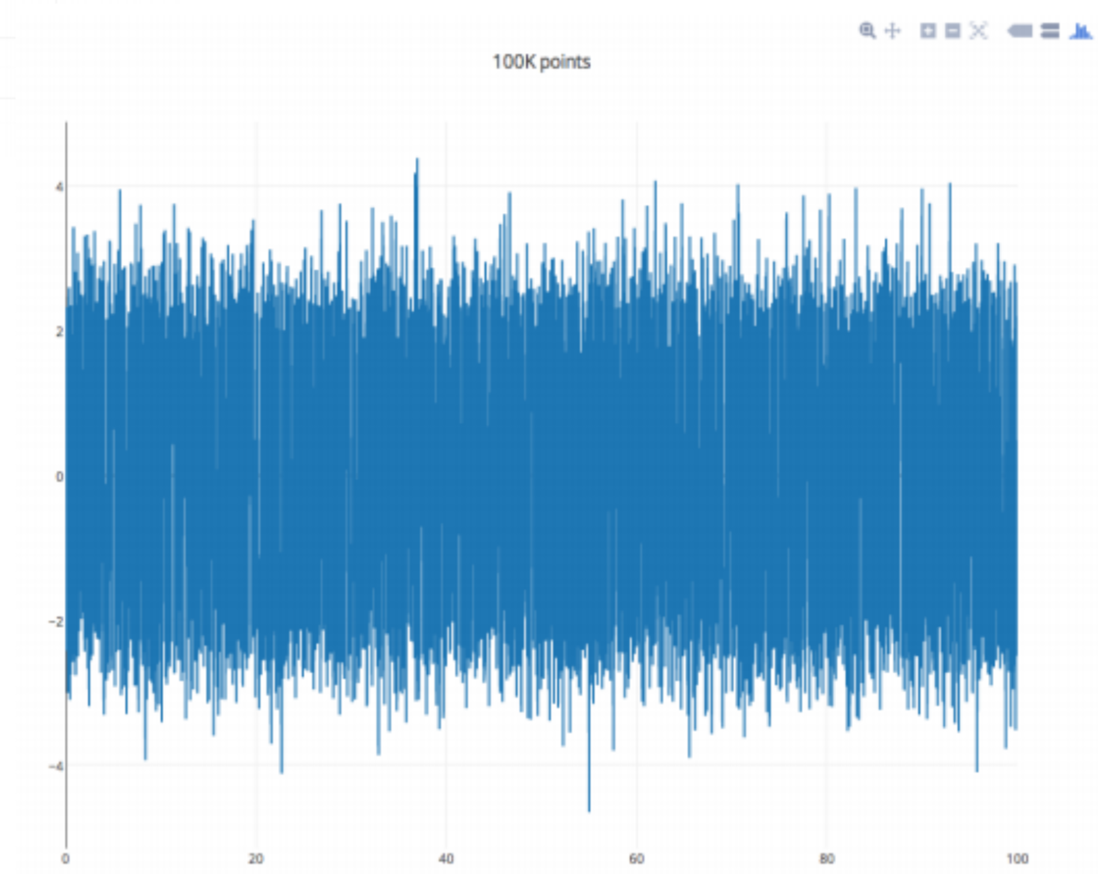
- > 1 million data points
- Exact rendering
- Interactive
- Important for plot.ly's business!

Plots

1. Scatter plots



2. Line plots



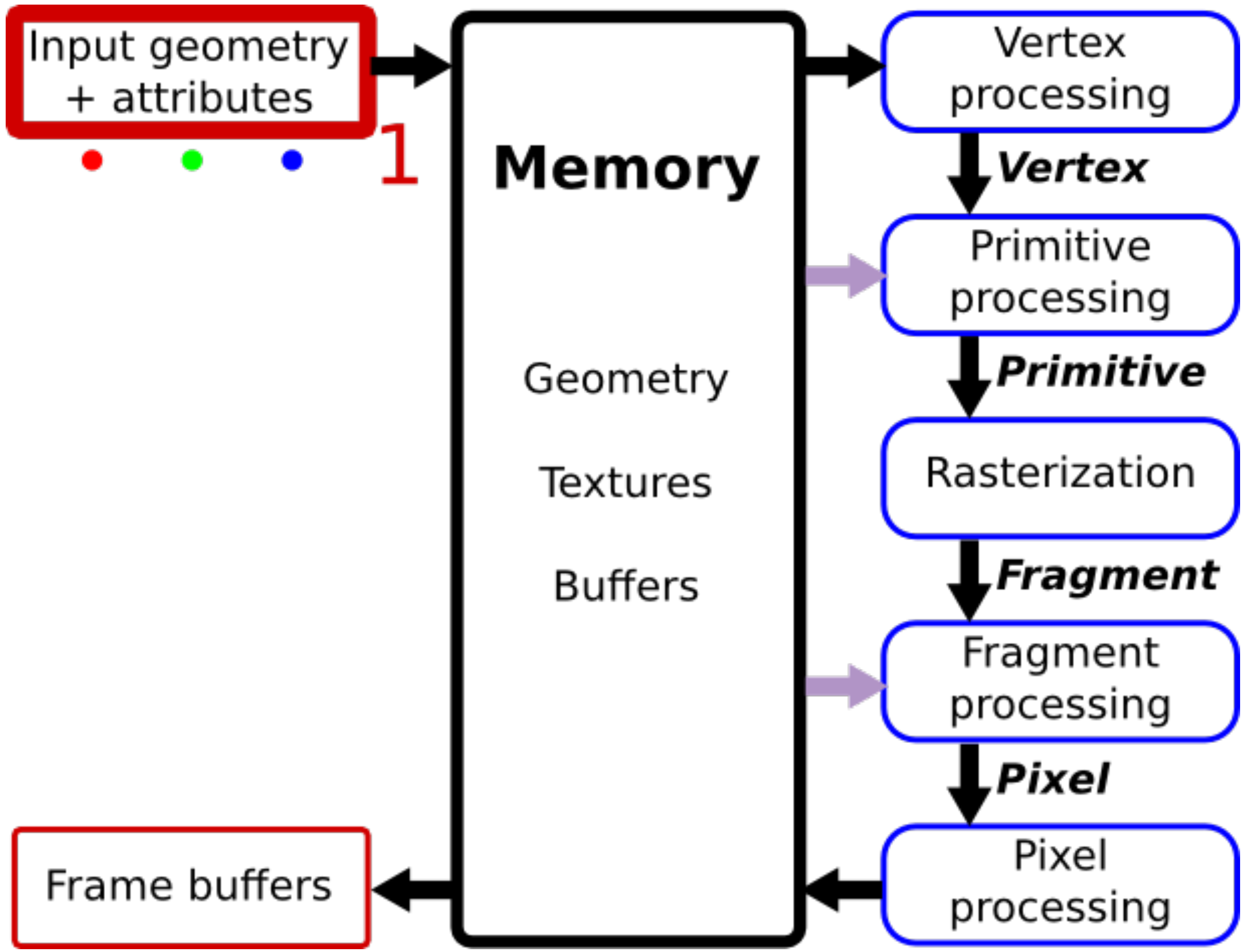
Main Question

Is it possible to render > 10 million points/lines?

WebGL

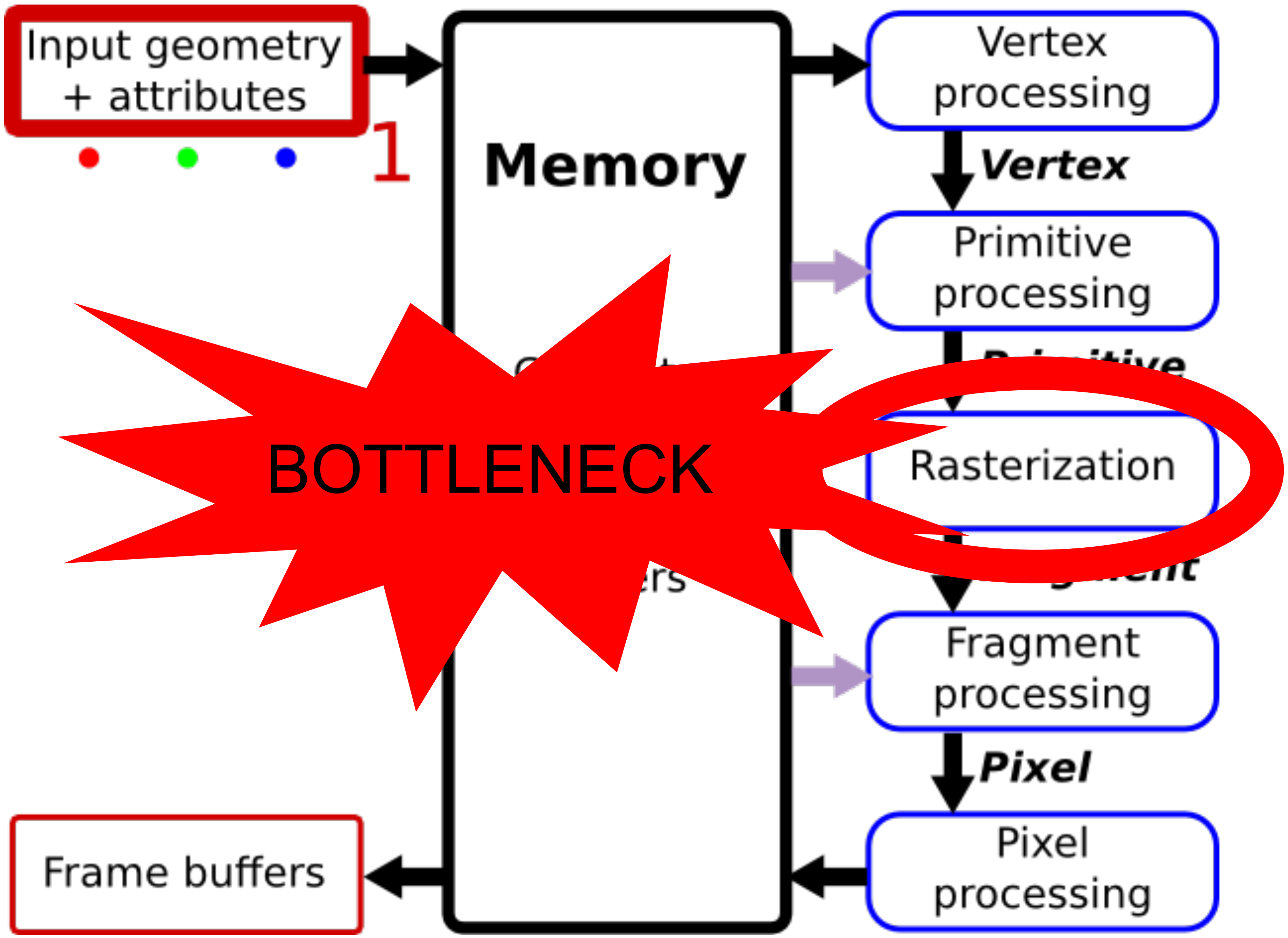
- Low level rendering API for drawing triangles, uses GPU
- Executes asynchronously and in parallel
- Performance factors:
  - Draw calls
  - Fragment processing
  - Vertex processing
  - Bandwidth
- Extremely fast performance possible

GPU Pipeline



Source: http://romain.vergne.free.fr/teaching/IS/SI03-pipeline.html

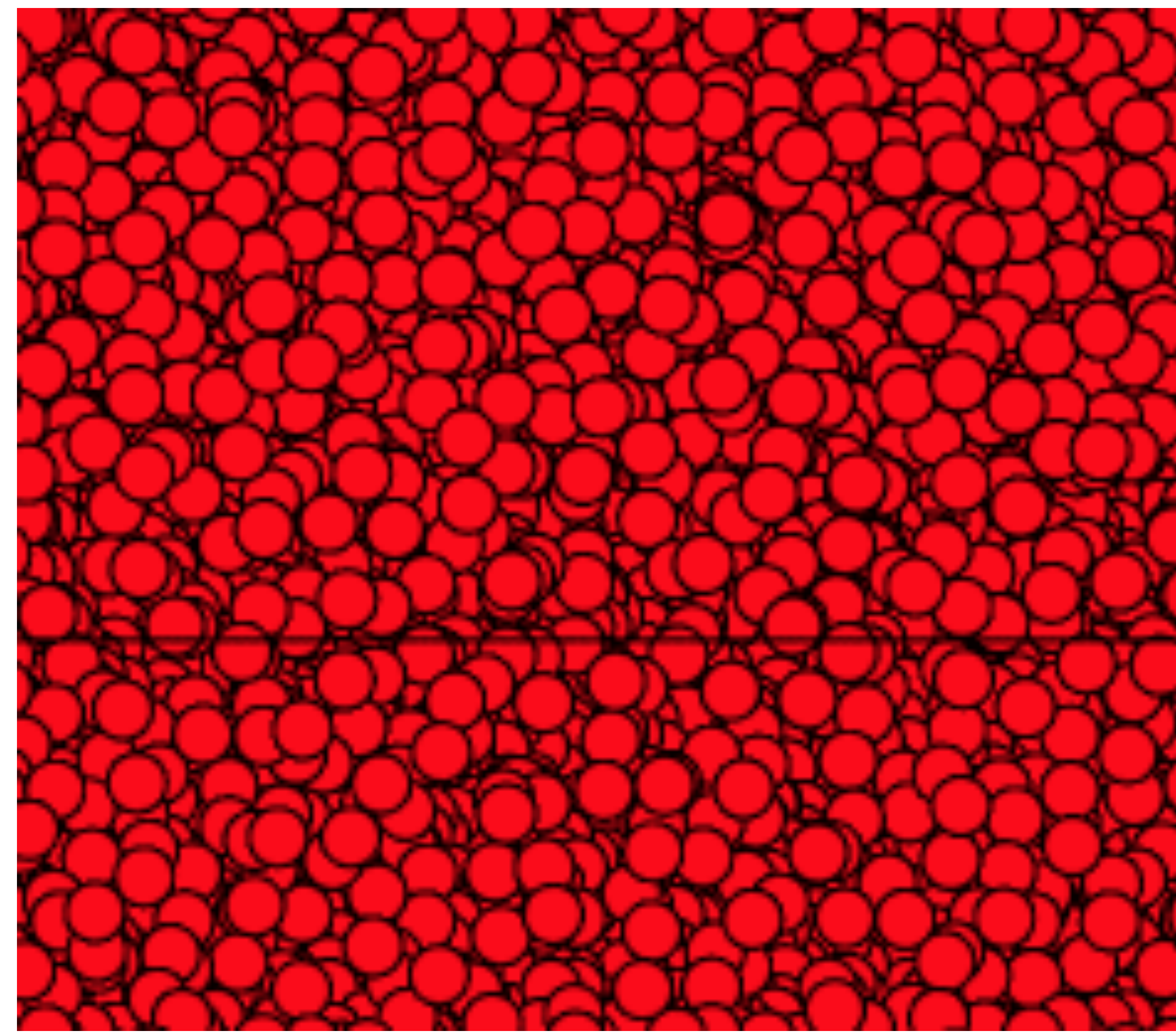
GPU Pipeline



Source: http://romain.vergne.free.fr/teaching/IS/SI03-pipeline.html

Fill rate

- Observation:
  Large scatter plots are fill rate limited.
  - Can process many vertices easily, as long as most end up outside viewport.
- For huge plots, many points end up on same pixel, so it is enough to draw just one sprite for whole group.



Cover Order

- Let $\mathcal{P} = \{p_0, p_1, \dots p_n\}$ be a set of primitives with $p_i \in R^2$
- $s_0 > s_1 > \dots > s_k = 0$ be a set of scales (pixel sizes)
- The *cover order* of $\mathcal{P}$ is the filtration

$$F_0 \subseteq F_1 \subseteq \dots\dots \subseteq F_k = \mathcal{P}$$

such that $\cup \mathcal{P} \subseteq \cup F_j \oplus B_{sj}$

$$\text{level}\,(p_i) = \min_{p_i \in F_j} j$$

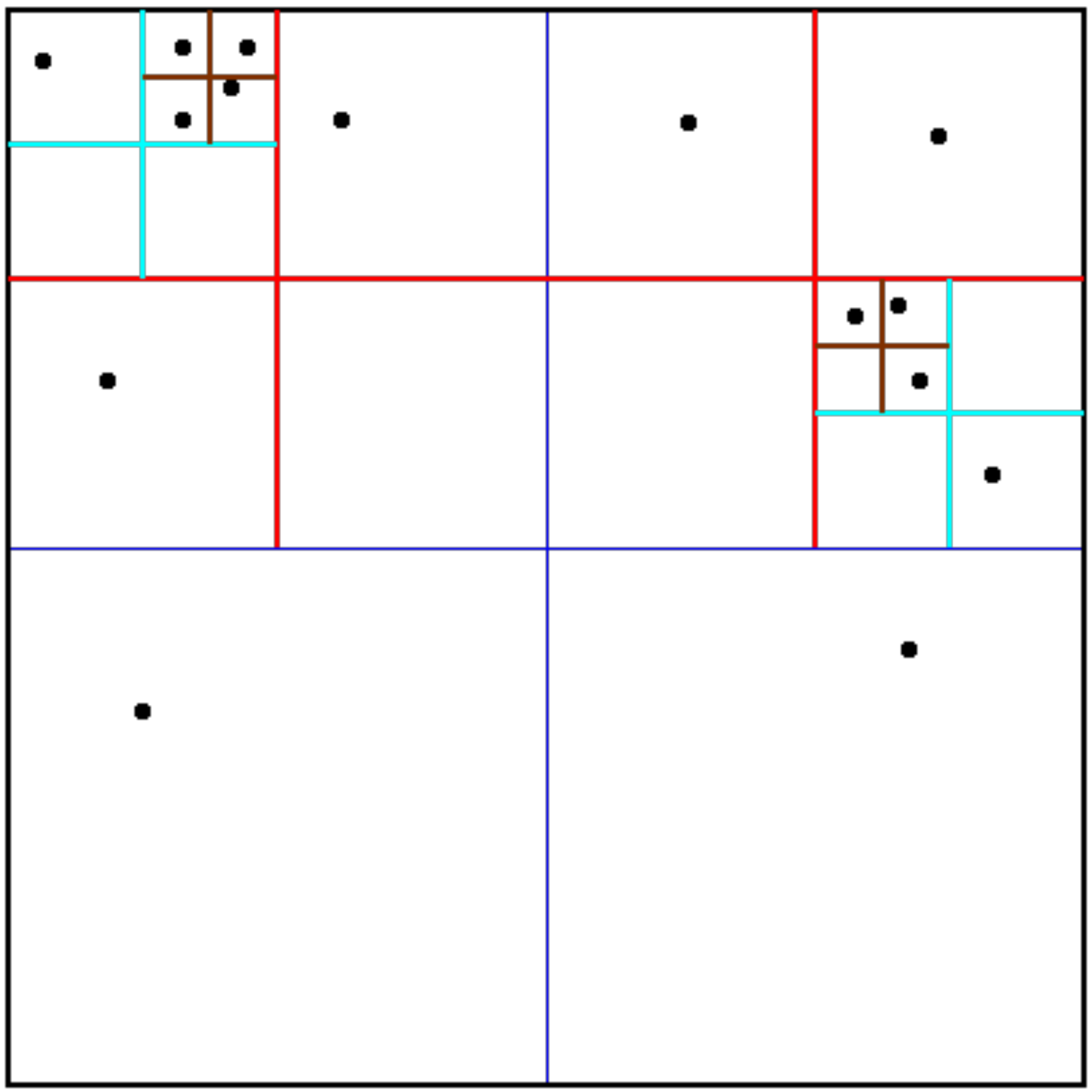
Cover Order

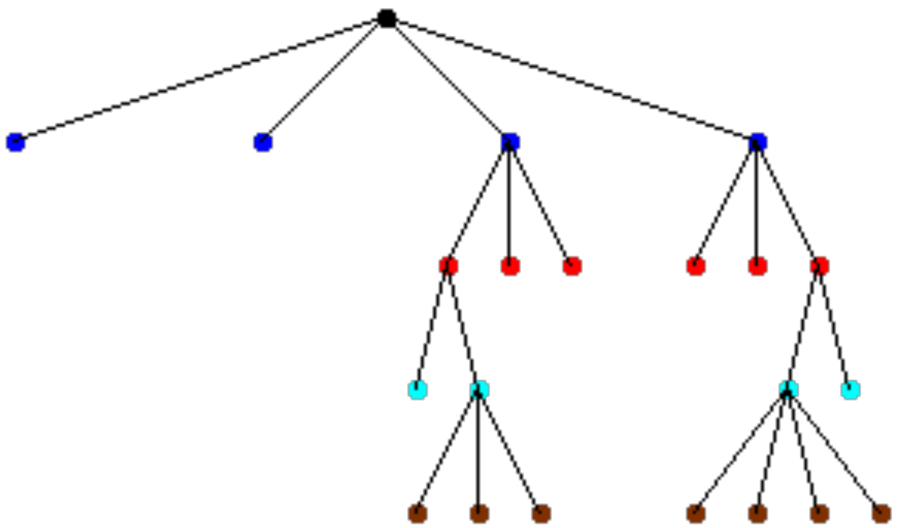
- We want a cover order of the points (for scatter plots) or line segments (for line plots) so that given a zoom level, we can quickly determine the points/segments that need to be rendered by the GPU (i.e. those that are not covered or hidden behind others).

100 million points

- We preprocess the points as follows:
  1. Construct a quadtree of the given pointset using a depth-first traversal of the points and keeping track of their level in the tree.
     Store the points in array Q.
  2. Sort the points of Q in increasing order of level, and of x-coordinate.
  3. Keep track of point levels using an array of indices of size equal to total number of levels.

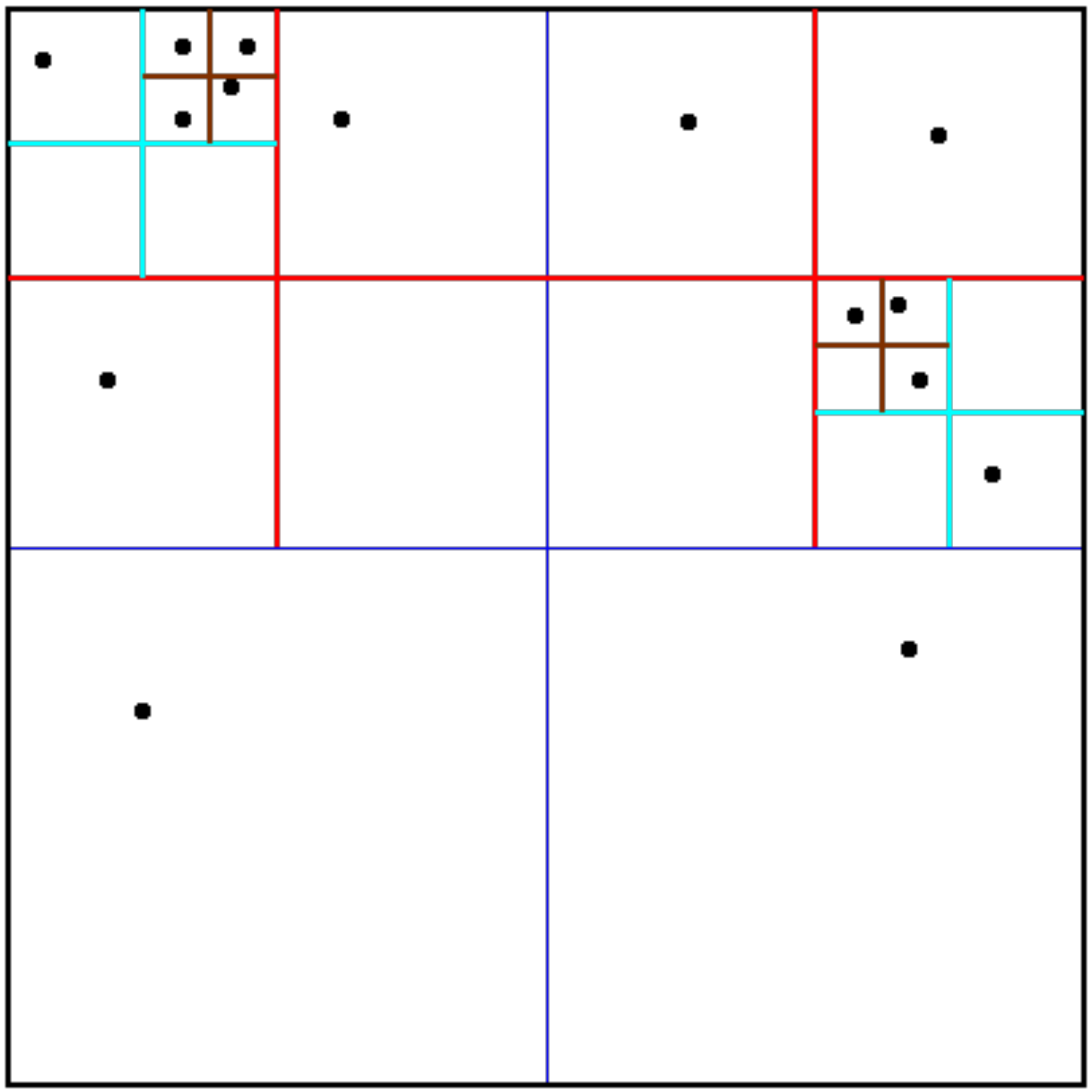
Quadtrees

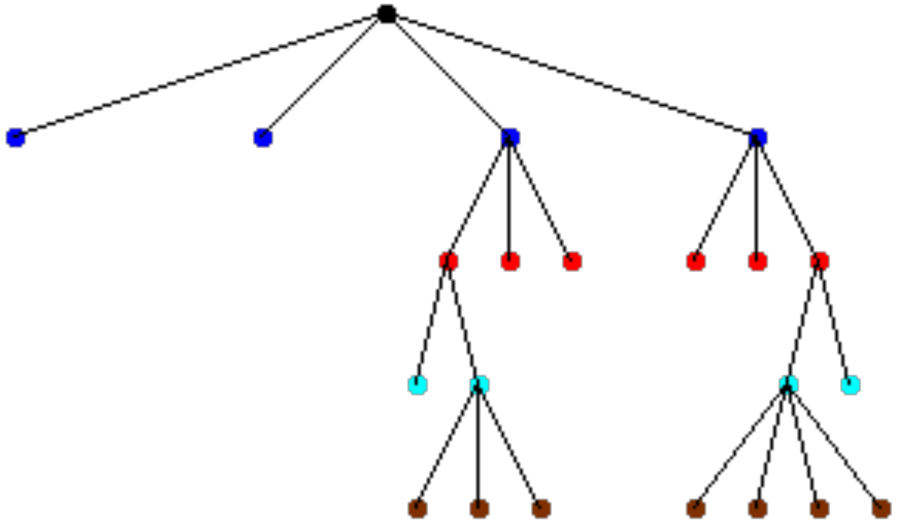


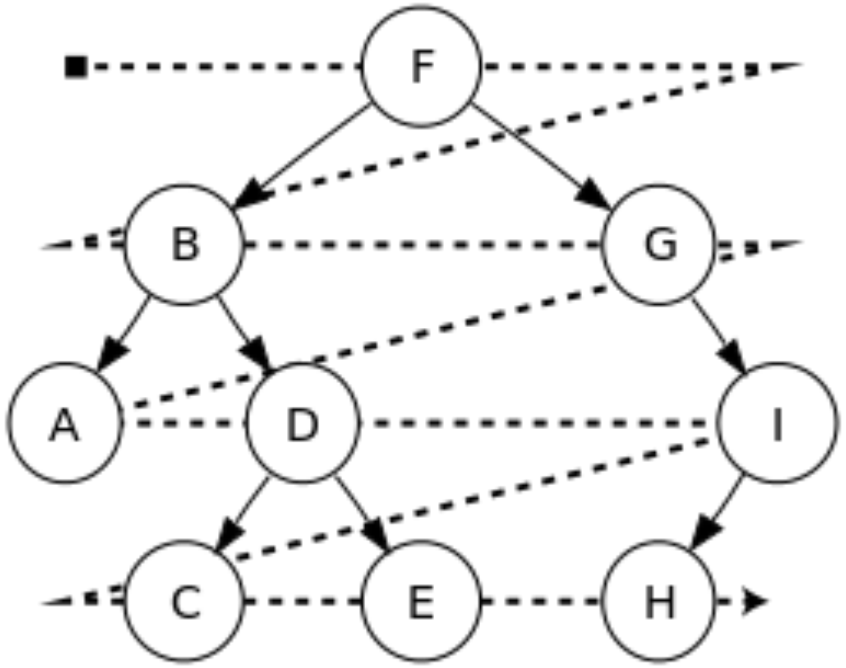


Source: http://www.cs.berkeley.edu/~demmel/cs267/lecture26/lecture26.html

Quadtrees







Source: http://www.cs.berkeley.edu/~demmel/cs267/lecture26/lecture26.html

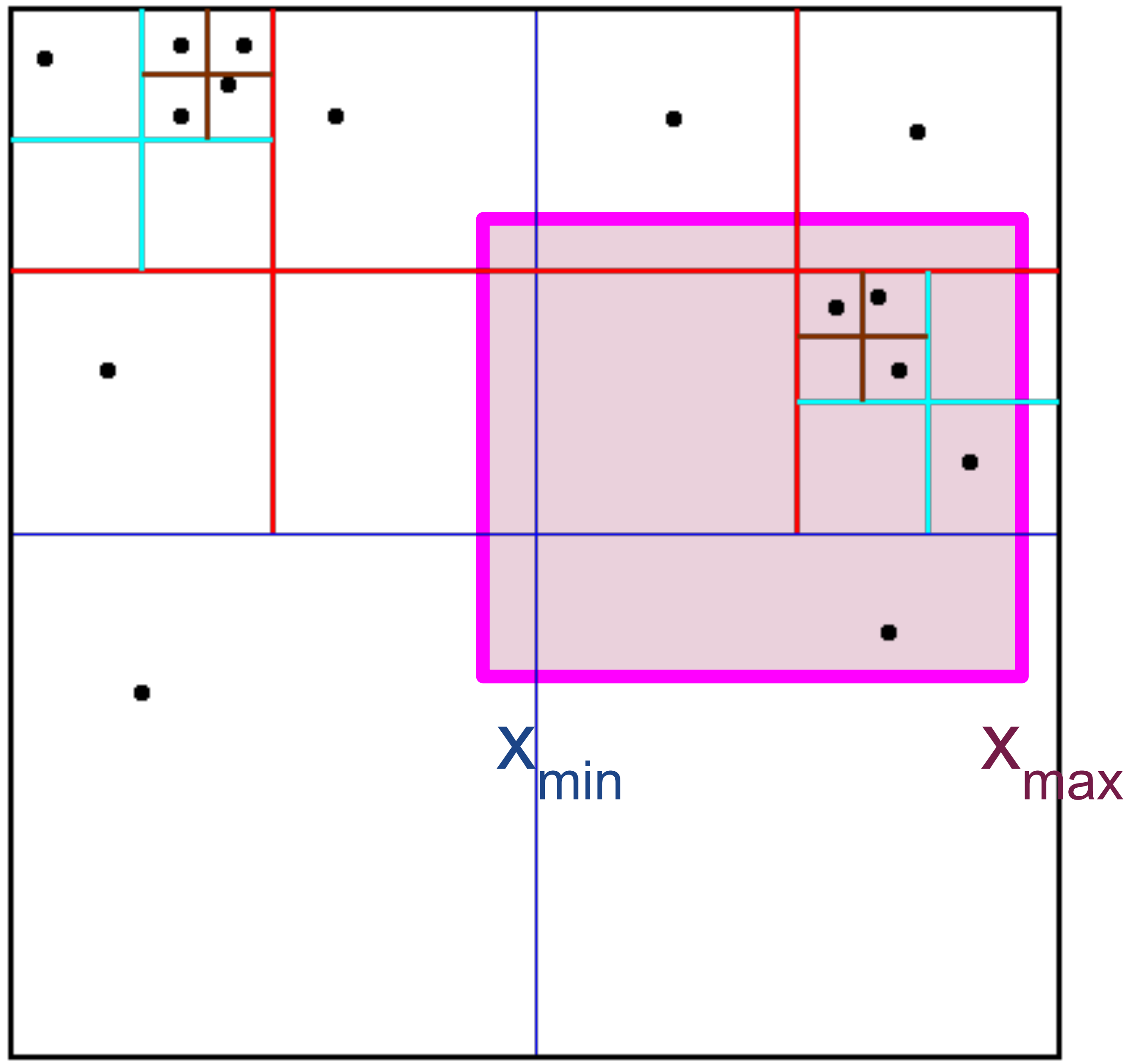
Rendering

- We send all the points (array Q) to the GPU.
- To render for a given level, we ask the GPU to draw some superset of the points that are visible on the screen and are not hidden behind others.

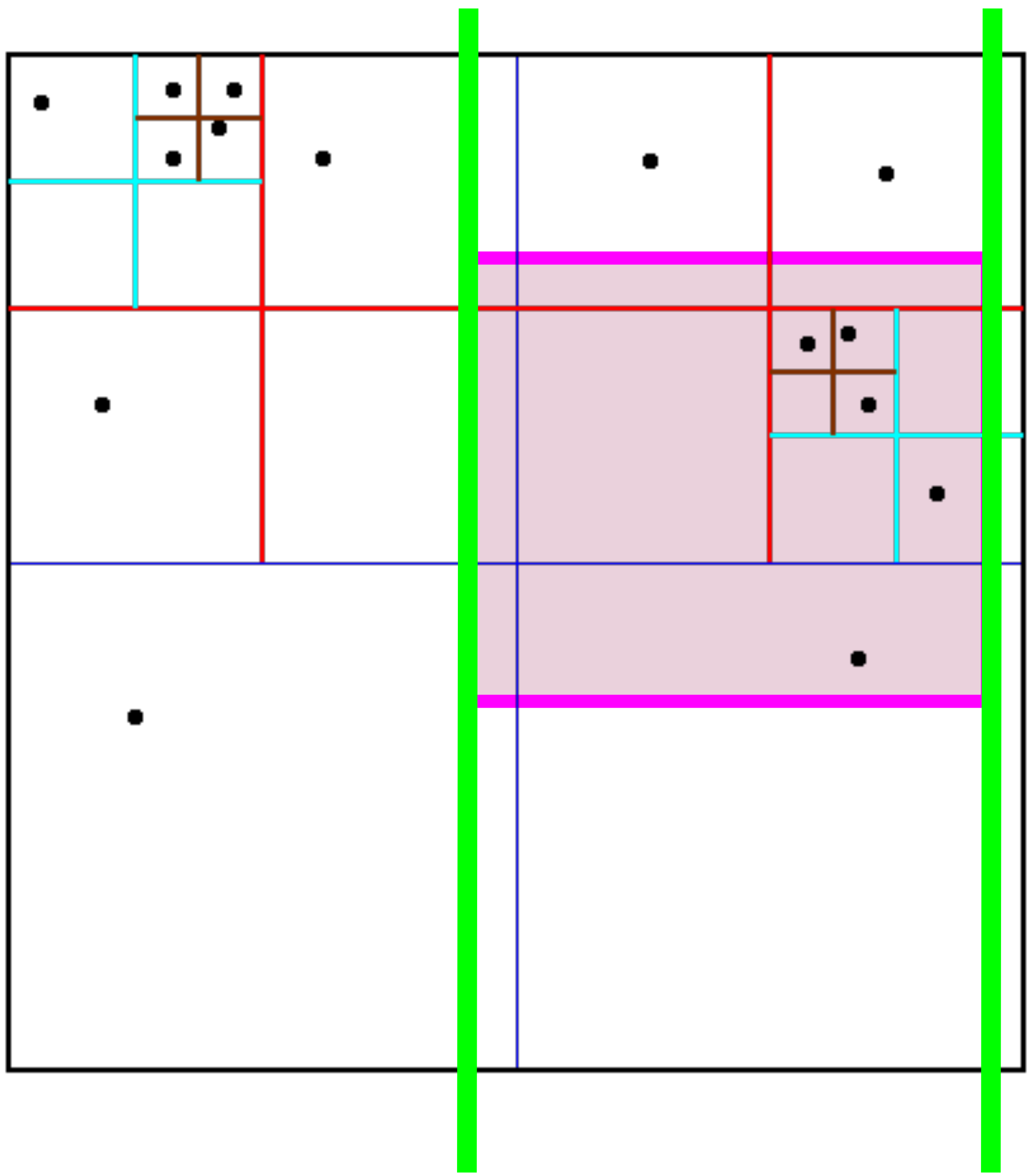
Rendering

1. Compute the size of a pixel in the data coordinate to get the current zoom level Z.
2. Compute the x_{min} and x_{max} of the screen.
3. Starting at level Z, for each level above Z
   i. Find the predecessor p of x_{min} and successor s of x_{max}
   ii. draw the points whose x-coordinates are between p and s.

Quadtrees



Quadtrees



Quadtree Bad Cases

- Preprocessing may take a long time if the quadtree ends up having height O(n).
- This happens when the input consists of a big cluster of points that are far from the rest of the points.
- To avoid such a bad case, we introduce a small change to the way we construct the quadtree.

Trick with Quadtrees

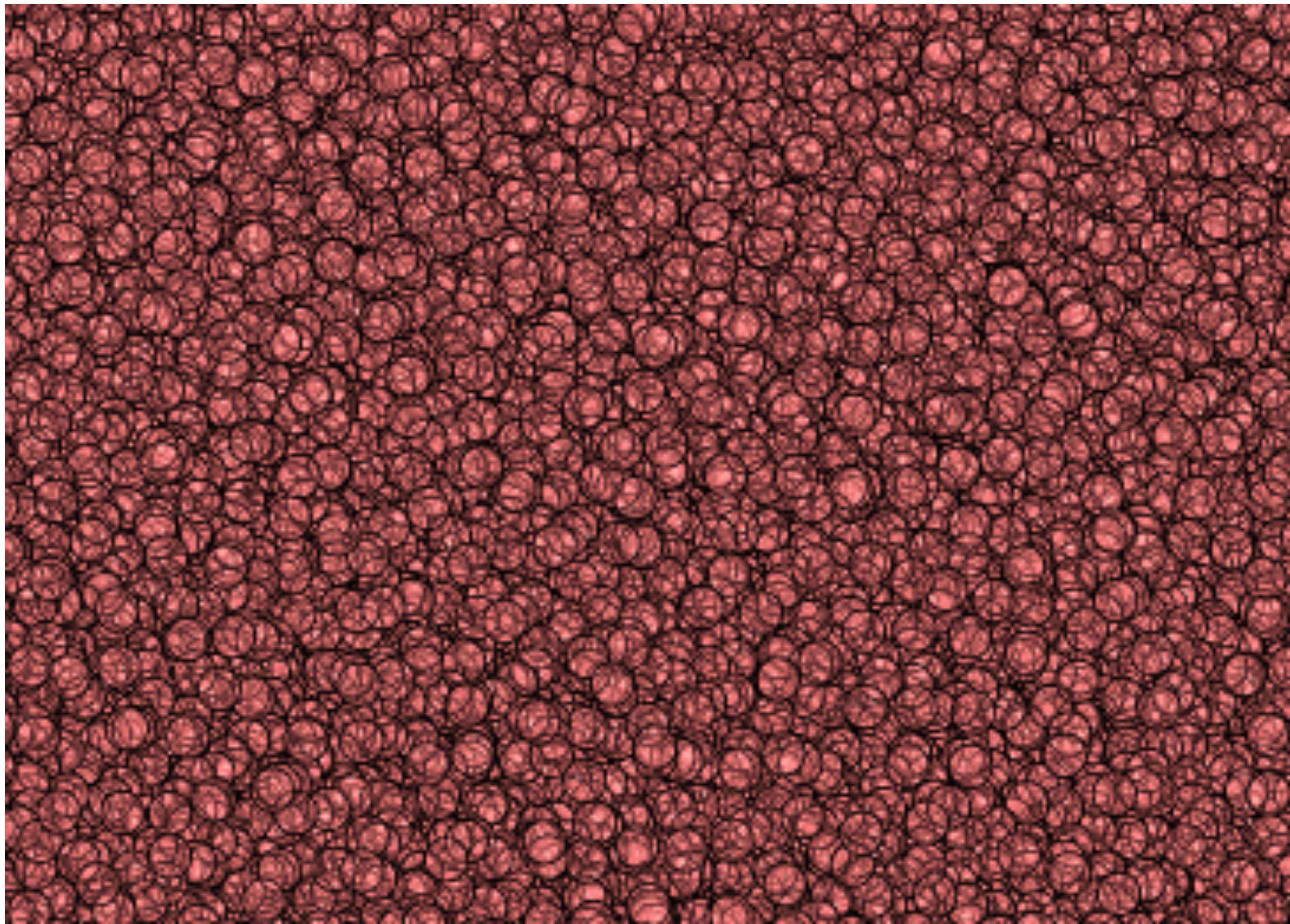
- The idea is as follows.
  1. After every split of the area into four quadrants, check each quadrant to see if they contain greater than 90% of the points.
  2. If such a quadrant is found, then split the point cluster arbitrarily into two equal sets and construct a new quadtree for each one separately.

- Doing this will preserve the level information for each node, which is all that we need to render effectively.

Transparent Points

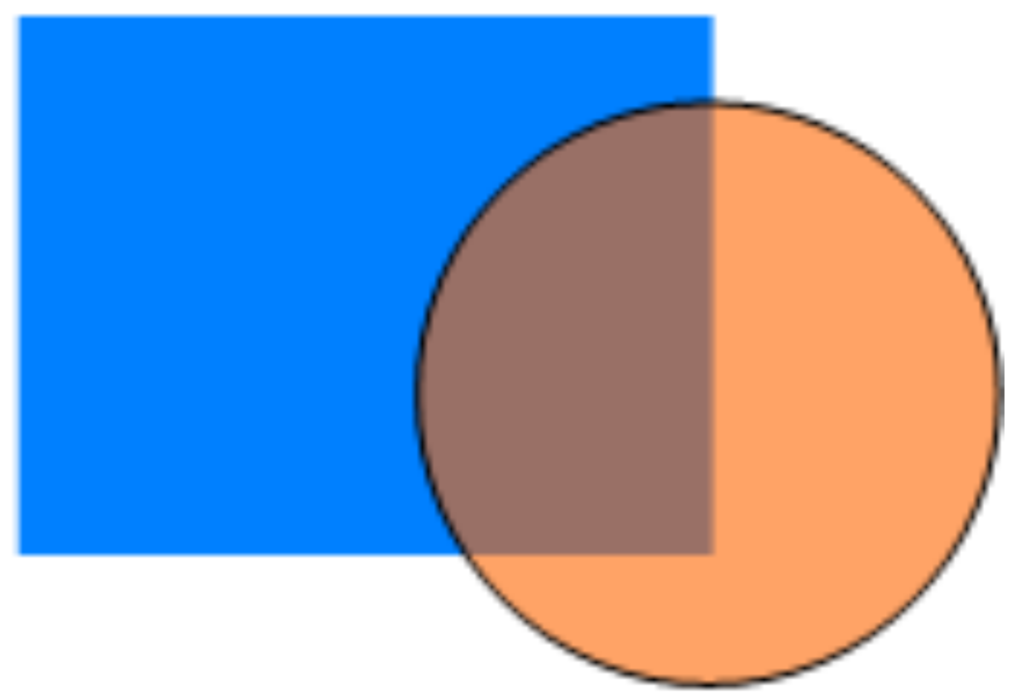
- For points that are semi-transparent, “hiding” the ones that are behind others will not give the desired result.

- Instead we need to “blend” colors for the points that are stacked.



Transparent Points

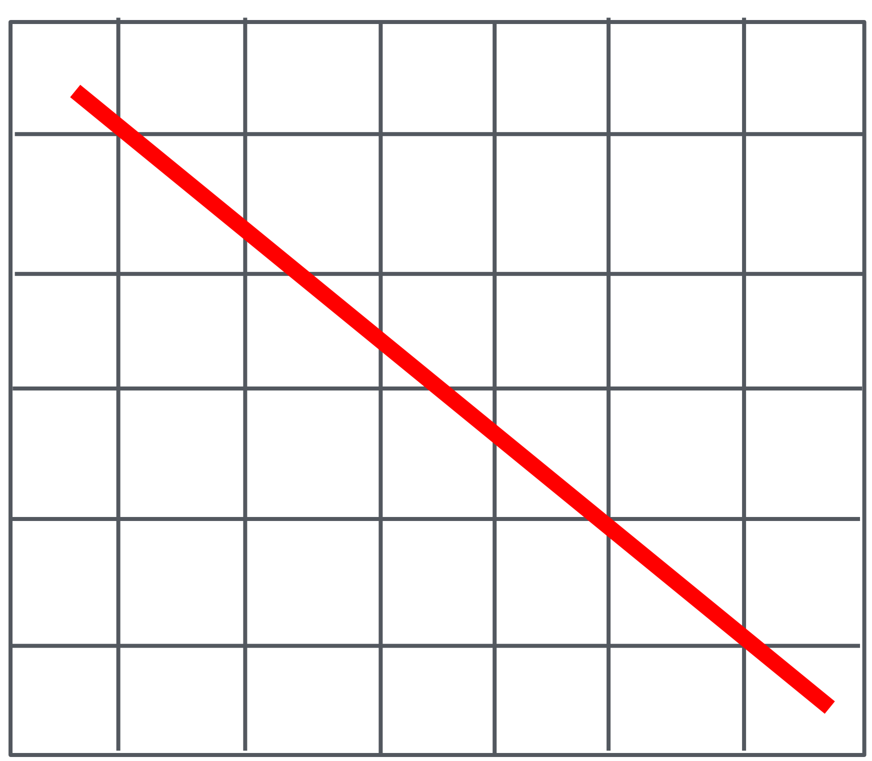
- For transparent points **having the same color and marker**, we need to know the number of points that a given point "hides".



- We can compute this information while building the quadtree -- for every node of the tree,
store the number of its descendants.

Line Plots

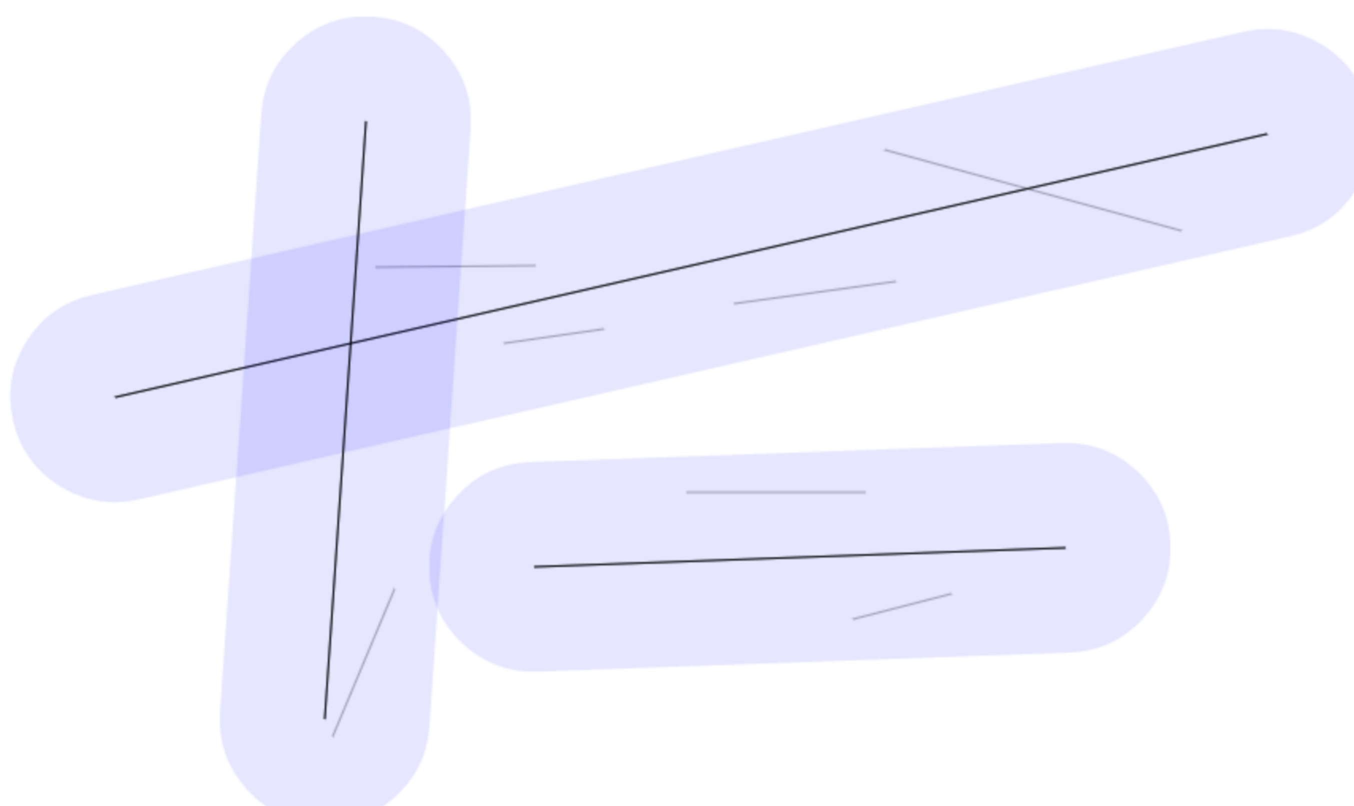
- Want to sort lines in cover order (reduce fragment processing cost on GPU).
- Quadtrees don't work.



Might hit $O(2^h)$ boxes in level h

Cover Order for Lines

- Build covers inductively, solve for $F_j \setminus F_{j-1}$:
- Want smallest set of line segments that cover $\mathcal{P} \setminus F_{j-1}$ at scale s_j



Line Plots: Ideas

1. Set Cover Approximation.
   - The cover ordering problem for line segments can be reduced to the problem of *Set Cover* (which is NP-complete).
   - This allows us to use the Set Cover approximation algorithm that has an approximation factor of $\theta(\log n)$.
     Running time: $O(n^3)$.
2. Greedy Longest Segment.
   - Sort the segments and process them by decreasing order of length. Running time: $O(n^2)$.
3. Divide-and-Conquer.
   - Recursively split the segments (arbitrarily) into two equal groups and process. Running time: $O(n \log n)$.

Summary of results

- Scaled from 10 million points to **100 million** interactively.
- Solved transparent rendering for a special case of scatter plots.
- New heuristics for line plots (to be investigated further).
- Beginnings of theoretical framework for rendering large data sets.

What needs work

- Preprocessing time
- Transparent lines, complex scatter plots
- SVG export
- Analysis and implementation of line cover

Thank you