BenchLines: Benchmarking Interactive Web Graphics for Parallel Coordinates

Information Architecture and Web Usability (WS 2025/2026)

Group 4
Michael Anderson
Jyothish Atheendran
Filip Ljubotina

BenchLines - Overview

- BenchLines benchmarks polyline rendering in parallel coordinates.
- Compares rendering speed across dataset sizes.
- Compares interactive features across technologies.

Parallel Coordinates - Overview

- A visualization technique for exploring high-dimensional data.
- Each vertical axis represents a variable (dimension).
- Each polyline represents one data item across all dimensions.

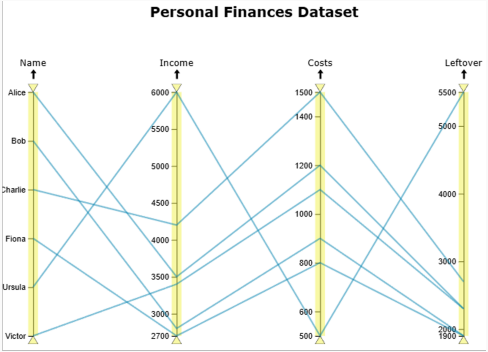


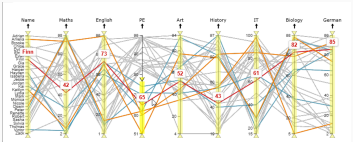
Image taken from https://tugraz-isds.github.io/pcee/ under MIT license.

BenchLines - Benchmarking

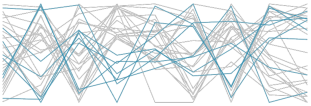
- Select technology, dataset, and iterations to run a benchmark.
- BenchLines re-renders continuously and records performance.
- Average render time is shown, with past results below the plot.

BenchLines - Plot Layers

3) Interactivity

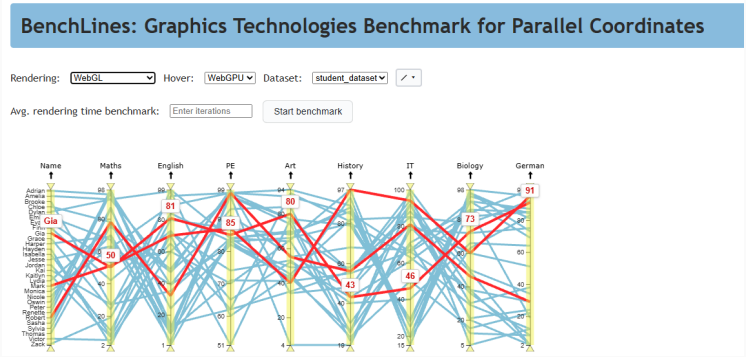


2) Rendering



1) Background

Demo

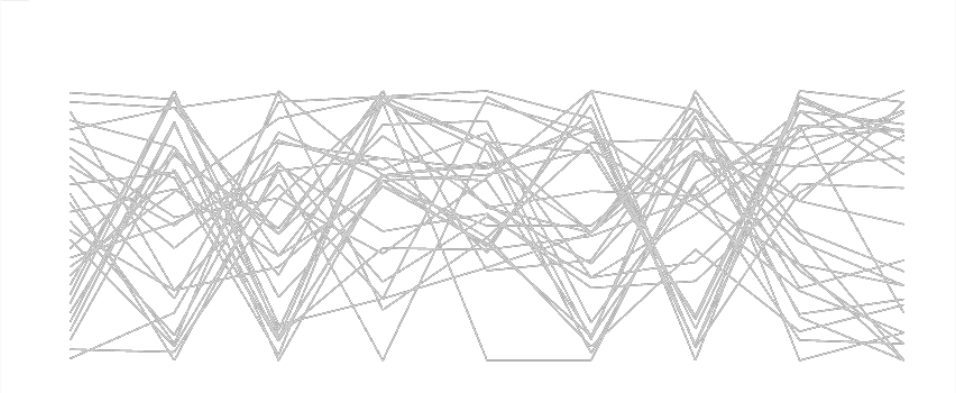


Live Site: https://filip-ljubotina.github.io/benchlines/

Github Repo: https://github.com/filip-ljubotina/BenchLines

Showcase Video: https://youtu.be/TX7kLXGYXA4

1) Background



Background Raster Image

- Charts can have thousands of lines.
- Rendering all lines can be slow if done every frame.
- Separate inactive lines from the active interactive lines.
- Rasterization is the process of converting shapes into pixels on a 2D grid.
- Using WebGL for fast rasterization and 2D canvas for the background.
- If WebGL is unavailable renders lines using Canvas2D.

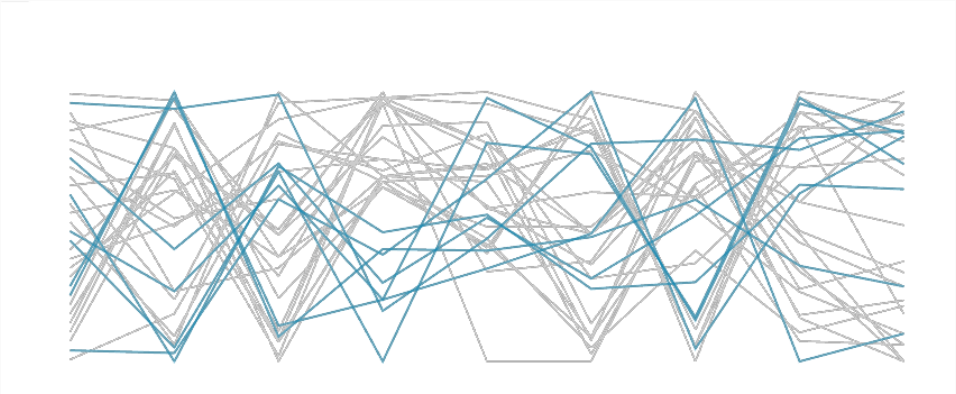
Background Rasterization

- Render background lines once in an off-screen WebGL canvas.
- Rasterized to a background image and interactive lines rendered above.
- Eliminates cost for rendering inactive lines.
- Background lines are re-rendered once after augmenting the data.

Rasterization Example

```
1  // Create offscreen WebGL canvas
2  const bg = document.createElement("canvas");
3  bg.width = mainCanvas.width;
4  bg.height = mainCanvas.height;
5
6  // Render lines offscreen
7  const bgGl = initLineTextureWebGL(bg);
8  drawInactiveLinesTexture(dataset, parcoords);
9
10 // Rasterize into 2D canvas
11 const ctx = inactiveLinesCanvas.getContext("2d")!;
12 const pixels = new Uint8Array(bg.width * bg.height * 4);
13 bgGl.readPixels(0,0,bg.width,bg.height,bgGl.RGBA,bgGl.UNSIGNED_BYTE,pixels);
14 ctx.putImageData(flipY(pixels,w,h),0,0);
```

2) Rendering



Rendering Technology

We utilize four core technologies to render graphics:

- SVG-DOM
- Canvas2D
- WebGL
- WebGPU (Successor to WebGL)

Rendering Libraries

Additionally we utilize two libraries:

- Three.js
- Pixi.js

SVG-DOM Implementation

```
1  const svg = document.createElementNS("http://www.w3.org/2000/svg", "svg");
2  svg.setAttribute("width", canvas.width);
3  svg.setAttribute("height", canvas.height);
4  document.body.appendChild(svg);
5
6  for (const d of dataset) {
7    const polyline = document.createElementNS("http://www.w3.org/2000/svg", "polyline");
8    polyline.setAttribute("points", getPolylinePoints(d));
9    polyline.setAttribute("stroke", "steelblue");
10   polyline.setAttribute("stroke-width", "2");
11   polyline.setAttribute("fill", "none");
12   svg.appendChild(polyline);
13 }
```

Canvas2D Implementation

```
function initCanvas2D(dpr: number) {
  ctx = canvasEl.getContext("2d")!;
  ctx.setTransform(dpr, 0, 0, dpr, 0, 0);
  return ctx;
}

function redrawCanvasLines(dataset, parcoords) {
  for (const d of dataset) {
    const pts = getPolylinePoints(d, parcoords);
    ctx.beginPath();
    ctx.moveTo(pts[0][0], pts[0][1]);
    pts.slice(1).forEach(p => ctx.lineTo(p[0], p[1]));
    ctx.lineWidth = 2;
    ctx.strokeStyle = (lineState[getLineName(d)]?.active ?? true)
    ? "rgba(0,129,175,0.5)" : "rgba(211,211,211,0.4)";
    ctx.stroke();
  }
}
```

WebGL Implementation

```
function redrawWebGLLines(dataset: any[], parcoords: any) {
  gl.useProgram(program);
  gl.clear(gl.COLOR_BUFFER_BIT);

  //Loop through dataset points creating an array of vertices...

  const vertexData = new Float32Array(vertices);

  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, vertexData, gl.DYNAMIC_DRAW);
  gl.vertexAttribPointer(posLoc, 2, gl.FLOAT, false, 0, 0);

  gl.drawArrays(gl.LINES, 0, vertexData.length / 2);
}
```

*Example WebGL code; not from the final implementation.

WebGPU Implementation

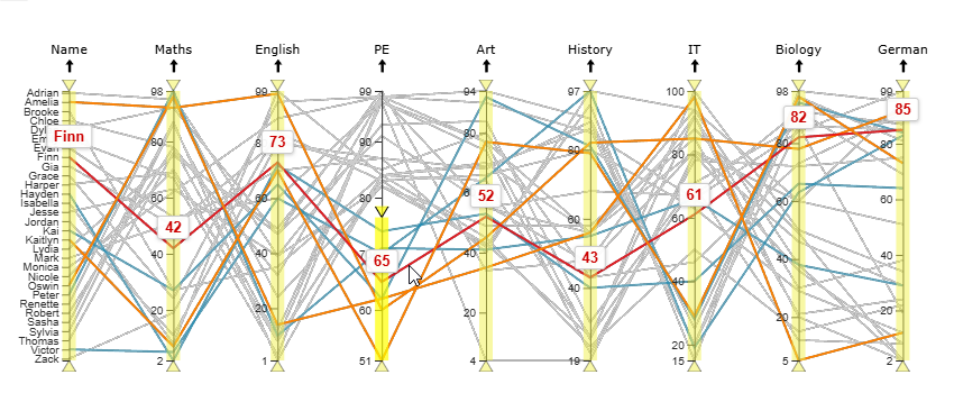
```
function redrawWebGPULines(dataset: any[], parcoords: any) {
  // Setup render pass (WebGPU specific)
  const pass = encoder.beginRenderPass({ /* attachment config */ });

  // Batch all vertices into a single Float32Array
  const allVerts = new Float32Array(totalVertexCount * 2);
  let offset = 0;
  for (const line of hoveredLines) {
    // convert segments to triangles with HOVER_LINE_WIDTH
    for (let i=0;i<line.pts.length-1;i++){ ... fill allVerts ... }}
  // ... fill with normalized coordinates
  // Convert vertices to two triangles and single buffer upload
  const vertexBuffer = device.createBuffer({
    size: totalBufferSize,
    usage: GPUBufferUsage.VERTEX | GPUBufferUsage.COPY_DST,
  });
```

WebGPU Implementation (contd.)

```
17    device.queue.writeBuffer(vertexBuffer, 0, allVerts);
18    pass.setPipeline(pipeline);
19    pass.setVertexBuffer(0, vertexBuffer);
20
21    let vertexOffset = 0;
22    for (const line of allLines) {
23      pass.setBindGroup(0, getColorBindGroup(line));
24      pass.draw(line.vertexCount, 1, vertexOffset, 0);
25      vertexOffset += line.vertexCount;
26    }
27
28    pass.end();
29    device.queue.submit([encoder.finish()]);
30  }
```

3) Interactivity



Interactions on Dimensions

- Reordering
- Inversion
- Filtering

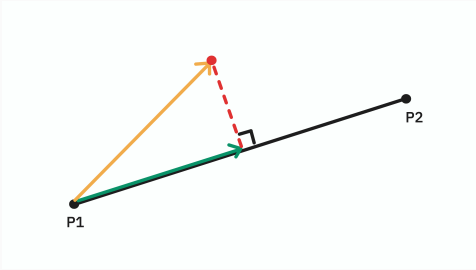
*Cascade effect: change requires update to lower layers

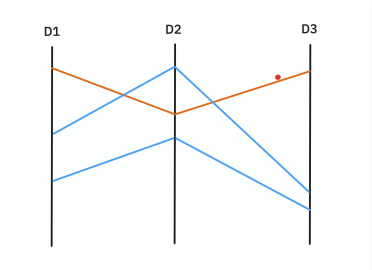
Hover and Selection Algorithms

- From-scratch implementation equivalent to library raycasting solutions.
- Raycasting logic decoupled from rendering technology.
- Two implementations: JS and GPU.
- GPU compute shaders via WebGPU API for parallel processing.
- 256 threads per workgroup, one thread per polyline.

Hover

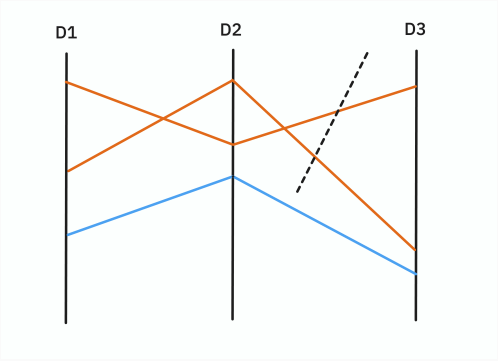
- Point-to-line distance detection on mouse move.
- GPU calculates distance from cursor to all polyline segments in parallel.
- JS only considers segments between the two dimensions surrounding the cursor.
- Configurable threshold (Fuzzy selection).

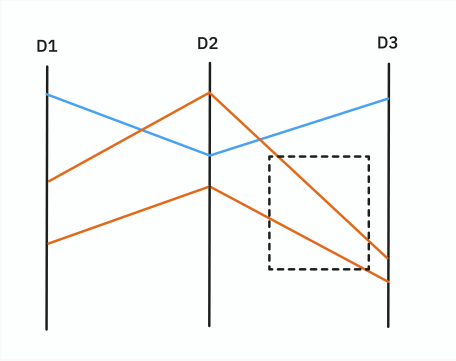




Selection Algorithms

- Line selection: detects all polylines intersecting a drawn line.
- Box selection: detects polylines inside or crossing a rectangle.





Conclusion

- Use WebGPU if available as its the fastest (requires GPU programming).
- Use WebGL if WebGPU is not available.
- Use Pixi.js or Three.js to avoid GPU programming.
- Use Canvas2D to work with just the CPU.

Thank You!

Questions?