Orthographic bp-per-pixel — Reference (Three.js + ParametricLine)

Scope & assumptions
- Camera: `THREE.OrthographicCamera` (no perspective foreshortening).
- Geometry: one or more polylines representing graph “superedges” (degree-2 runs between junctions).
- Each superedge carries base-pair accounting per segment (Δbp) and cumulative bp (`Cbp[i]`).
- Your `ParametricLine` exposes `t∈[0,1] → xyz` and `xyz → t` along each superedge.

Goal
Define and compute **local** and **aggregate** bp-per-pixel for a wiggly, embedded (non-linear) rendering, suitable for LOD and picking.

---

1) Definitions

For segment `i` between vertices `i→i+1` (after model transforms): - `Δbp_i` = basepairs on that segment (from node lengths or per-edge labels).
- `Δpx_i` = on-screen length of that segment in **pixels** under the current camera + viewport.
- **Local scale per segment:**
`bpPerPixel_i = Δbp_i / max(Δpx_i, ε)` (ε ≈ 1e-6 px to avoid divide-by-zero).

Cumulative arrays (per superedge): - `Cbp[i] = Σ_{k< i} Δbp_k`
- `Cpx[i] = Σ_{k< i} Δpx_k`

At an arbitrary parameter `t` along a superedge, a centered estimate:

```
bpPerPixel(t) ≈ ( Cbp(t+δ) − Cbp(t−δ) ) / ( Cpx(t+δ) − Cpx(t−δ) )
```

Choose `δ` so the pixel span `Cpx(t+δ) − Cpx(t−δ)` ≈ 4–8 px for stability.

---

2) Pixels-per-world for an orthographic camera

Let canvas drawing-buffer size be `Wpx × Hpx` (physical pixels). You can obtain this via `renderer.getDrawingBufferSize(v)`.

Effective view extents in world units:

```
Wworld = (camera.right  - camera.left ) / camera.zoom
Hworld = (camera.top    - camera.bottom) / camera.zoom
```

Pixels per world unit (constant for the whole view):

```
Sx = Wpx / Wworld = Wpx * camera.zoom / (camera.right - camera.left)
Sy = Hpx / Hworld = Hpx * camera.zoom / (camera.top   - camera.bottom)
```

> **Notes**
> • These scales change only when the **zoom or viewport** changes. Panning does **not** affect them.
> • If you use `renderer.setPixelRatio(dpr)`, `getDrawingBufferSize` already includes `dpr`.

---

3) Project geometry to view space (orthographic)

Distances must be measured in **camera view axes** (x,y), not world axes (the camera may be rotated). For each vertex `P_world`:

```
P_view = camera.matrixWorldInverse * object.matrixWorld * P_world
```

For segment `i→i+1` in view space:

```
dx = P_view[i+1].x - P_view[i].x
dy = P_view[i+1].y - P_view[i].y
Δpx_i = sqrt( (dx*Sx)^2 + (dy*Sy)^2 )
```

Accumulate `Cpx` with `Δpx_i`.
(You can skip the matrix multiply if your geometry is already authored in camera-aligned space. In general, do the proper transform.)

---

4) Local and aggregate bp-per-pixel

- **Per-segment:** `bpPerPixel_i = Δbp_i / max(Δpx_i, ε)`
- **At parameter t:** use the centered finite-difference form above with `δ` chosen to cover a few pixels.
- **For a visible range [a,b]:**
  `bpPerPixel_avg([a,b]) = (Cbp(b) - Cbp(a)) / max(Cpx(b) - Cpx(a), 1)`

These quantities update only when **zoom/resize** (or camera rotation) changes.

---

5) Pixel-based LOD binning (recommended)

Pick a minimum on-screen thickness `p_min` in pixels (e.g., 1–2 px). For each superedge: 1) Walk segments accumulating `Δpx` until the bin reaches `p_min`.
2) Merge everything inside the bin into a **run** (for coloring, labeling, or instancing).
3) Also accumulate `Δbp` in the same loop to keep bp summaries for the run.

This is equivalent to a local bp threshold `L_min_bp = p_min × bpPerPixel(t)` but is numerically more stable because it works directly in pixels.

Output per run: `{u0,u1, pxSpan, bpSpan, color/type, idsSummary}` where `u` is your along-superedge parameter (0..1) if you want shader-friendly sampling.

---

6) Integrating with `ParametricLine`

- If your `ParametricLine` is already arc-length parameterized, you can map a pixel target directly to `t` via a lookup table `t ↔️ Cpx`.
- Otherwise, build a small table per superedge: `[{t_i, Cpx_i, Cbp_i}]` at each vertex (or at resampled breakpoints).
- For picking: raycast → `(superedgeId, segmentIndex, α)` → derive `t` and then read local `bpPerPixel(t)` or the current **run** that contains `t`.

---

7) Recompute triggers & performance tips

Recompute `Δpx/Cpx` **and any LOD runs only when:** - `camera.zoom` changes or the canvas is resized.
- The camera rotates (if you allow non-upright ortho).
- The object's model matrix changes.

Keep it fast: - Work on decimated polylines (screen-space error target ~0.3–0.7 px).
- Batch many superedges into one geometry; carry `pathId/breakFlag` vertex attributes to reset joins.
- Store `Cbp/Cpx` in Float32Arrays; reuse buffers across frames.
- Use `renderer.getDrawingBufferSize()` and avoid recomputing `Sx,Sy` unless zoom/viewport changes.

---

8) Robustness & pitfalls

- **Zero-length segments:** if consecutive vertices coincide after transform, set `Δpx_i=0` and let ε guard the division.
- **Anisotropic pixels:** non-square pixels are rare on the web; if present, `Sx≠Sy` handles it.
- **Huge Δbp with tiny Δpx:** cap `bpPerPixel` in tooltips to a max for UI sanity; run-binning will merge them anyway.

- **Crossings/overlaps:** irrelevant to LOD because binning is along path arc-length, not image density.

---

9) Utility snippets (JS)

```
// Get drawing buffer size (physical pixels)
function getBufferSize(renderer){
  const v = new THREE.Vector2();
  renderer.getDrawingBufferSize(v);
  return { Wpx: v.x, Hpx: v.y };
}

// Ortho pixel scales
function orthoScales(camera, renderer){
  const { Wpx, Hpx } = getBufferSize(renderer);
  const Wworld = (camera.right - camera.left) / camera.zoom;
  const Hworld = (camera.top - camera.bottom) / camera.zoom;
  return {
    Sx: Wpx / Wworld,
    Sy: Hpx / Hworld,
    Wpx, Hpx
  };
}

// Cumulative pixel arc length for one superedge
function computeCumPx(superedge, object3D, camera, renderer){
  const { Sx, Sy } = orthoScales(camera, renderer);
  const mv = new THREE.Matrix4().multiplyMatrices(camera.matrixWorldInverse,
object3D.matrixWorld);
  const P = superedge.positions; // Float32Array or array of Vector3
  const n = P.length;
  const Cpx = new Float32Array(n);
  const tmpA = new THREE.Vector3();
  const tmpB = new THREE.Vector3();

  tmpA.copy(P[0]).applyMatrix4(mv);
  for (let i=1; i<n; i++){
    tmpB.copy(P[i]).applyMatrix4(mv);
    const dx = (tmpB.x - tmpA.x) * Sx;
    const dy = (tmpB.y - tmpA.y) * Sy;
    const d  = Math.hypot(dx, dy);
    Cpx[i] = Cpx[i-1] + d;
    tmpA.copy(tmpB);
  }
  return Cpx; // paired with superedge.Cbp
}
```

```
// Pixel-based LOD runs (per superedge)
function makePixelRuns(superedge, Cpx, pMinPx=1){
  const runs = [];
  const Cbp = superedge.Cbp;
  const n = Cpx.length;
  let i0 = 0;
  while (i0 < n-1){
    const px0 = Cpx[i0];
    let i1 = i0 + 1;
    while (i1 < n && (Cpx[i1] - px0) < pMinPx) i1++;
    const u0 = superedge.tAtIndex(i0); // or i0/(n-1) if uniformly sampled
    const u1 = superedge.tAtIndex(i1);
    runs.push({
      i0, i1,
      u0, u1,
      pxSpan: Cpx[i1] - Cpx[i0],
      bpSpan: Cbp[i1] - Cbp[i0]
    });
    i0 = i1;
  }
  return runs;
}

// Local bp/px near param t via a few-pixel window
function bpPerPixelAtT(superedge, Cpx, t, targetPxWindow=6){
  // map t→nearest index; expand until ~targetPxWindow
  const idx = superedge.indexNearT(t);
  let i0 = idx, i1 = idx;
  while (i0>0 && (Cpx[i1]-Cpx[i0]) < targetPxWindow) i0--;
  while (i1<Cpx.length-1 && (Cpx[i1]-Cpx[i0]) < targetPxWindow) i1++;
  const dBp = superedge.Cbp[i1] - superedge.Cbp[i0];
  const dPx = Math.max(Cpx[i1] - Cpx[i0], 1e-6);
  return dBp / dPx;
}
```

Summary

- With an orthographic camera, `Sx,Sy` give a **global, constant** pixel scale.
- Measure segment lengths in **view space** (x,y), multiply by `Sx,Sy` to get `Δpx`.
- Use `Cpx/Cbp` to compute local `bpPerPixel`, and drive LOD via **pixel-based** binning for numeric stability and crisp visuals.
- Recompute only on zoom/resize/rotation/model changes; keep arrays and buffers persistent for speed.