Grid Snapping in Force-Directed Layouts

Status: active **Category**: research **Created**: 2026-01-30 **Tags**: force-layout, grid, d3, layout-enhancement, value-add

Summary

Force-directed layouts discover topology organically but produce floating-point positions that can feel "messy". Grid snapping offers a middle ground: let forces discover structure, then impose order. This is an underexplored area with potential for significant UX improvement.

The Problem with Pure Force Layouts

Force-directed layouts excel at:

- Discovering cluster structure
- Minimizing edge crossings (to a degree)
- Handling unknown topologies
- Producing visually "organic" results

But they struggle with:

- Alignment (nodes rarely line up)
- Consistent spacing (distances vary)
- Professional/polished appearance
- Reproducibility (different runs → different layouts)

The result often looks like a sketch when users want a diagram.

Grid Snapping Approaches

1. Post-Snap (Simplest)

Run simulation to completion, then snap all positions to nearest grid point.

```
postSnap :: Number -> Array Node -> Array Node
postSnap gridSize = map \node ->
  node { x = roundTo gridSize node.x
       , y = roundTo gridSize node.y
       }

roundTo :: Number -> Number -> Number
roundTo grid val = grid * round (val / grid)
```

Pros: Simple, predictable **Cons**: Can introduce overlaps, may undo careful force balancing, feels abrupt

2. Grid Force (Continuous)

Add a custom force that pulls nodes toward grid points. Competes with other forces.

```
// D3 custom force
function forceGrid(gridSize) {
  let nodes;
  let strength = 0.1;

  function force(alpha) {
    for (const node of nodes) {
      const targetX = Math.round(node.x / gridSize) * gridSize;
      const targetY = Math.round(node.y / gridSize) * gridSize;

      node.vx += (targetX - node.x) * strength * alpha;
      node.vy += (targetY - node.y) * strength * alpha;
    }
  }

  force.initialize = (n) => nodes = n;
  force.strength = (s) => { strength = s; return force; };

  return force;
}
```

Pros: Smooth, integrates with simulation dynamics **Cons**: May fight other forces, can create oscillation, nodes may not actually land on grid

3. Annealing Snap (Recommended)

Gradually increase grid force strength as simulation cools. Early: topology discovery. Late: alignment.

```
-- Grid strength as function of alpha
gridStrength :: Number -> Number
gridStrength alpha
  | alpha > 0.5 = 0.0           -- Early: no grid
  | alpha > 0.1 = (0.5 - alpha) -- Middle: gradual
  | otherwise   = 1.0           -- Late: strong grid

-- In tick handler
onTick :: Number -> Effect Unit
onTick alpha = do
  simulation # setForceStrength "grid" (gridStrength alpha)
```

Pros: Best of both worlds - organic discovery, then alignment **Cons**: More complex, requires tuning the annealing curve

4. Quantized Velocity

Instead of snapping positions, quantize velocity deltas. Nodes move in discrete steps.

```
quantizedTick :: Number -> Node -> Node
quantizedTick quantum node =
  let
    dx = roundTo quantum node.vx
    dy = roundTo quantum node.vy
  in node { x = node.x + dx, y = node.y + dy, vx = 0.0, vy = 0.0 }
```

Pros: Creates distinctive "mechanical" movement aesthetic **Cons**: Can feel jerky, may not converge well

5. Constraint-Based Hybrid

Run force simulation, then solve a constrained optimization:

- Minimize deviation from force-computed positions
- Subject to: all positions on grid

```
minimize: Σ (x_i - x_force_i)² + (y_i - y_force_i)²
subject to: x_i ∈ {0, gridSize, 2*gridSize, ...}
            y_i ∈ {0, gridSize, 2*gridSize, ...}
            no overlaps
```

Pros: Principled, can add arbitrary constraints **Cons**: Requires optimization solver, more complex implementation

Design Considerations

Grid Size Selection

- **Semantic**: Grid = node diameter (ensures no overlap)
- **Aesthetic**: Grid = 2× node diameter (breathing room)
- **Data-driven**: Grid derived from data extent / node count
- **User-controlled**: Slider to adjust grid density

Edge Treatment

When nodes snap to grid, edges need consideration:

- **Straight edges**: May overlap with snapped nodes
- **Orthogonal routing**: Edges follow grid lines (Manhattan)
- **Curved edges**: Beziers that avoid intermediate grid points
- **Edge bundling**: Group parallel edges after snapping

Animation

The transition from "floating" to "snapped" is a design opportunity:

- **Instant**: Jarring but honest
- **Interpolated**: Smooth LERP to grid positions

- **Staggered**: Nodes snap one-by-one (ripple effect)
- **Elastic**: Overshoot then settle (playful)

Value-Add Opportunities

Grid snapping is one of several "layout enhancement" techniques:

Enhancement	Effect
Grid snapping	Alignment, order
Edge bundling	Reduce clutter
Label placement	Avoid overlaps
Symmetry detection	Reveal structure
Cluster boxing	Group semantics
Level assignment	Hierarchy clarity
Aspect ratio fitting	Use available space

These could compose into a "layout polish" pipeline:

```
Raw data
  → Force layout (discover topology)
  → Grid snap (impose order)
  → Edge bundle (reduce clutter)
  → Label place (avoid overlaps)
  → Render
```

Implementation Plan for Hylograph

Phase 1: Post-Snap Utility

Add to hylograph-simulation:

```
module Hylograph.Simulation.Snap where

snapToGrid :: Number -> Array (SimNode d) -> Array (SimNode d)

-- Also snap-aware bounds calculation
gridAlignedExtent :: Number -> Array (SimNode d) -> Extent
```

Phase 2: Grid Force

Custom force for D3 simulation:

```
module Hylograph.Simulation.Force.Grid where

gridForce :: Number -> Force d
gridForce gridSize = ...

-- With strength control
gridForceWithStrength :: Number -> (Number -> Number) -> Force d
```

Phase 3: Annealing Integration

Hook into simulation lifecycle:

```
type SimulationConfig d =
  { forces :: Array (Force d)
  , onTick :: Maybe (Number -> Effect Unit)  -- alpha passed
  , gridAnnealing :: Maybe GridAnnealingConfig
  }

type GridAnnealingConfig =
  { gridSize :: Number
  , curve :: AnnealingCurve
  }

data AnnealingCurve
  = Linear
  | Exponential Number
  | StepAt Number  -- sharp transition at alpha value
  | Custom (Number -> Number)
```

Relation to Sugiyama

Sugiyama (layered) layouts are inherently grid-aligned:

- Nodes assigned to discrete layers (Y grid)
- Nodes ordered within layers (X grid)

Grid snapping for force layouts is essentially "Sugiyama-lite":

- Topology from forces
- Order from grid
- No explicit layer assignment

Could offer as alternative when Sugiyama's strict layering is too rigid.

Research Questions

1. **Optimal annealing curve?** Linear? Exponential? Step function?
2. **Grid size heuristics?** How to auto-compute good grid from data?
3. **Overlap resolution**: When snapping creates collisions, how to resolve?

4. **Edge routing**: Best approach for snapped layouts?
5. **Perceptual evaluation**: Do users actually prefer snapped layouts?

References

- D3 force simulation: https://d3js.org/d3-force
- Constraint-based graph layout (Dwyer, Marriott)
- Grid-based diagram editors (draw.io, Lucidchart)
- Snap-to-grid in CAD systems

Related

- L-Systems Visualization - could use grid for turtle segments
- Sugiyama Layered Graph - related layout concerns
- hylograph-simulation force module