OpenStrandStudio3D

Rendering Pipeline V3

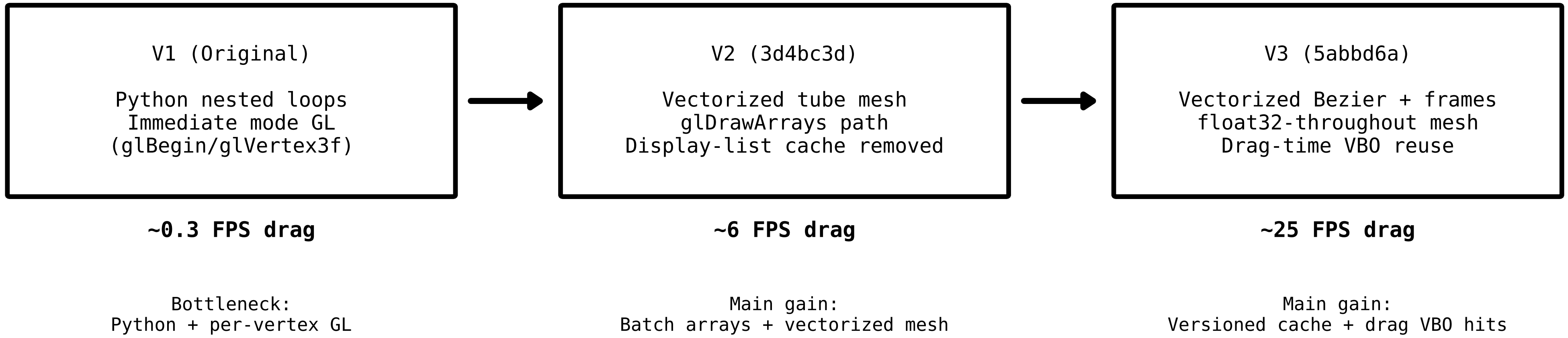
VBO Cache + Numpy Vectorisation

This document explains how OpenStrandStudio3D renders strand tubes to the screen. It covers the scene structure, how a single strand becomes a 3D tube mesh, the three-layer caching system that avoids redundant work, and the numpy broadcasting tricks that replaced slow Python loops with fast C-level array operations.

The result: drag interactions went from ~0.3 FPS to ~25 FPS.

Old vs New Approaches (V1 → V2 → V3)

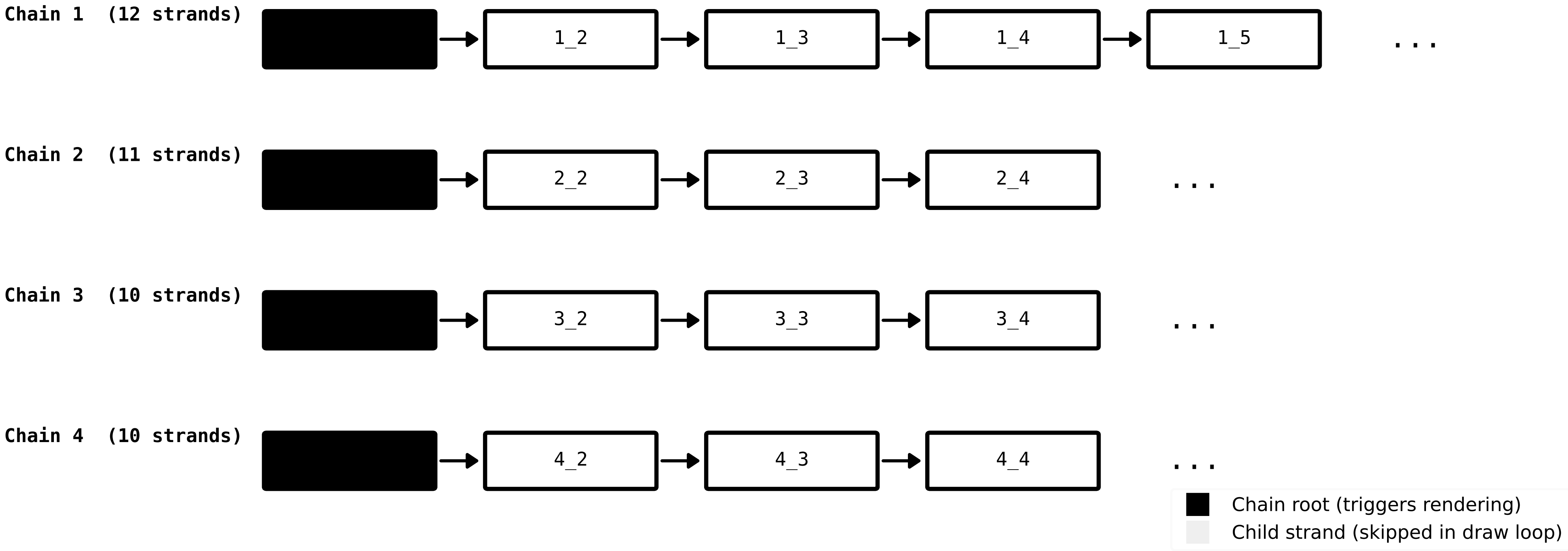
Explicit evolution of rendering strategy



This page separates OLD vs NEW approaches clearly:

V1 used Python loops and immediate-mode GL calls for lots of tiny operations. V2 moved to vectorized mesh construction and batched draw-arrays rendering, and removed display-list replay that caused stale-geometry edge cases. V3 adds drag-time VBO reuse for non-affected chains, vectorized Bezier generation, optimized sequential frame transport, and float32 data flow end-to-end. The result is much better interactive drag performance.

Scene Structure



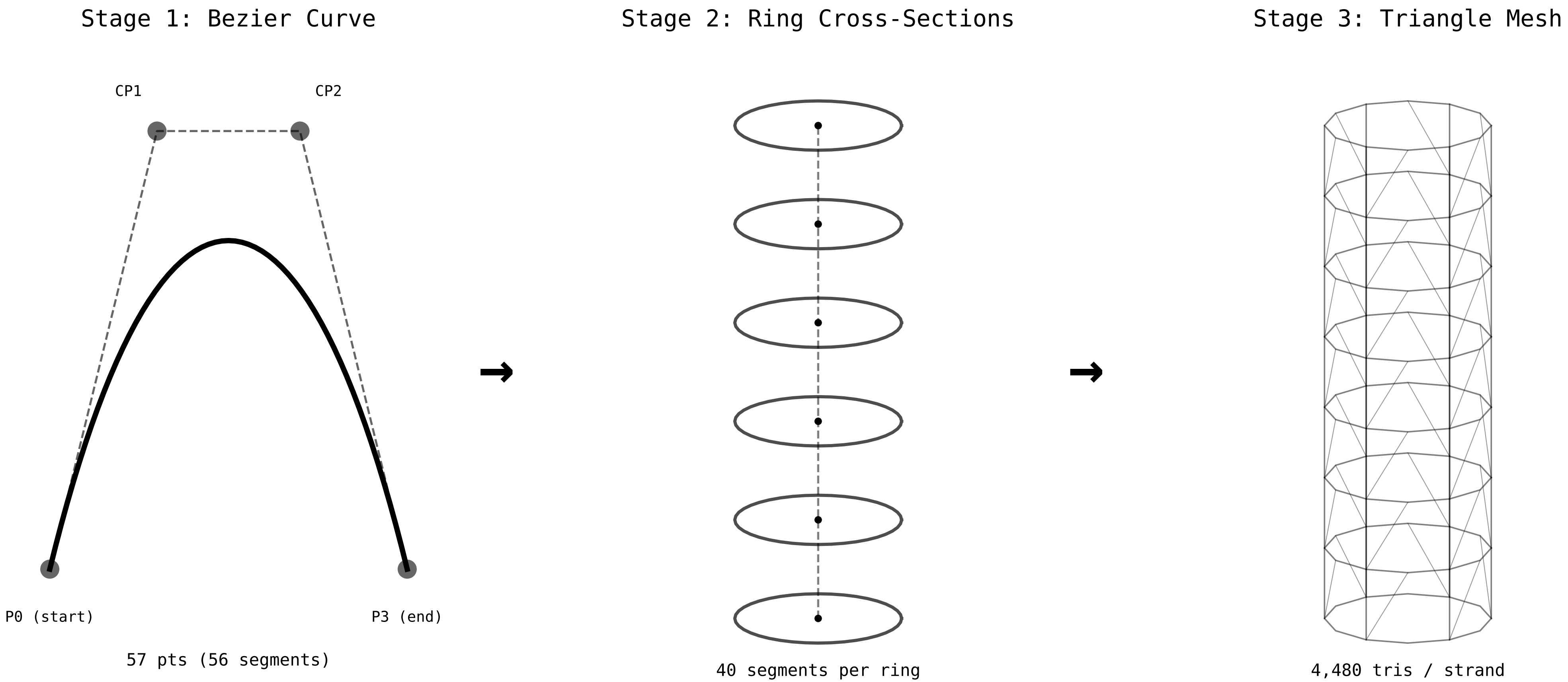
Strands are grouped into chains. Each chain is rendered as one continuous tube.
The first strand in a chain is the chain root -- it owns all caches and is the
only strand whose draw() method actually does rendering work.

When paintGL() loops over all 43 strands, 39 non-root strands return
immediately.
Only the 4 chain roots call _draw_chain_as_spline(), which collects geometry
from
every strand in the chain, builds the tube mesh, and sends it to the GPU.

This means the rendering cost scales with the number of chains (4), not the
total
number of strands (43). Each chain root checks its caches before doing any work.

Strand Tube Construction

Three-stage pipeline: Control Points → Curve → Rings → Mesh



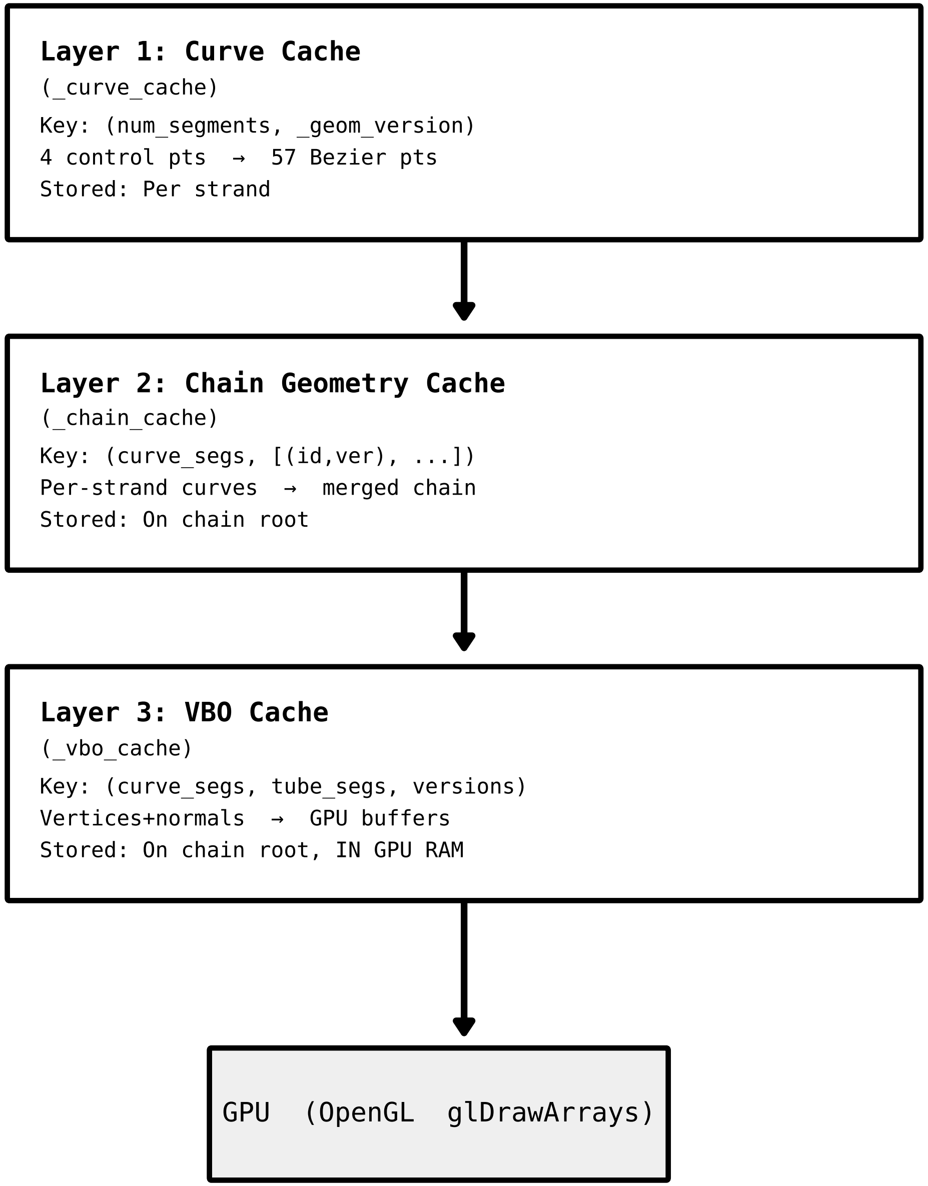
Each strand is defined by 4 control points (start, CP1, CP2, end). A cubic Bezier curve interpolates these into 57 evenly spaced points (56 segments). At each curve point, a ring of 40 vertices is placed perpendicular to the curve direction using parallel-transport frames (right/up vectors). Adjacent rings are connected into quads, each split into 2 triangles = 4,480 triangles per strand.

For a 12-strand chain: 12 x 56 = 672 curve points, 672 x 40 x 2 = 53,760 triangles.

Tube Construction: Vectorised Steps

```
STEP A: BEZIER POINTS (vectorised with numpy)

  Old way:  for i in range(57): point = (1-t)^3*P0 + ...   (684 Python calls for 12 strands)
  New way:  t = np.linspace(0, 1, 57)   then broadcast  t[:,None] * P0[None,:]
            All 57 points computed in ~1 numpy call per strand.  12 calls total, not 684.


STEP B: PARALLEL TRANSPORT FRAMES (scalar math)

  Each curve point needs a right/up coordinate frame to orient its ring. Each frame
  depends on the previous one (sequential), so full vectorisation is impossible.

  But numpy has ~20us overhead PER CALL for tiny 3-element arrays:
    np.dot([a,b,c], [d,e,f])  =  parse args + check types + allocate + compute + wrap  = ~20us
    px*tx + py*ty + pz*tz     =  3 float multiplies + 2 adds                           = ~0.1us

  So we pre-compute all tangents in one batch:  tangents = pts[1:] - pts[:-1]
  Then loop with plain Python scalars:  672 iters x 0.5us = 0.3ms  (vs 50ms with numpy)


STEP C: BUILD TRIANGLE MESH (numpy broadcasting -- the big win)

  vertex = center + width * x_cs * right + height * y_cs * up

  We reshape arrays so numpy broadcasts the entire mesh in one operation:
    centers[:, None, :]   (671, 1, 3)  -- one center per curve segment
    x_cross[None, :, None] (1, 40, 1)  -- one x per ring position
    rights[:, None, :]    (671, 1, 3)  -- one right vector per segment
                                Result: (671, 40, 3) = 80,520 floats in ONE C loop.

  All arrays are float32 (4 bytes), not float64 (8 bytes), halving memory and matching
  OpenGL's native format -- no conversion needed.


STEP D: SEND TO GPU

  For cached chains (VBO):     glBindBuffer + glDrawArrays  (GPU reads its own memory)
  For the affected chain:      glVertexPointer + glDrawArrays  (GPU reads CPU numpy array)
  No VBO is created for the affected chain because the data changes every frame anyway.
```

Three-Layer Cache System



The rendering pipeline has three caching layers stacked on top of each other. Each layer stores the output of a computation stage, keyed by a version tuple.

Layer 1 (Curve Cache) is per-strand. Each strand caches its own 57-point Bezier curve. The key includes _geom_version, an integer that increments whenever a control point moves. If the version matches, the cached curve is returned (a simple dict lookup).

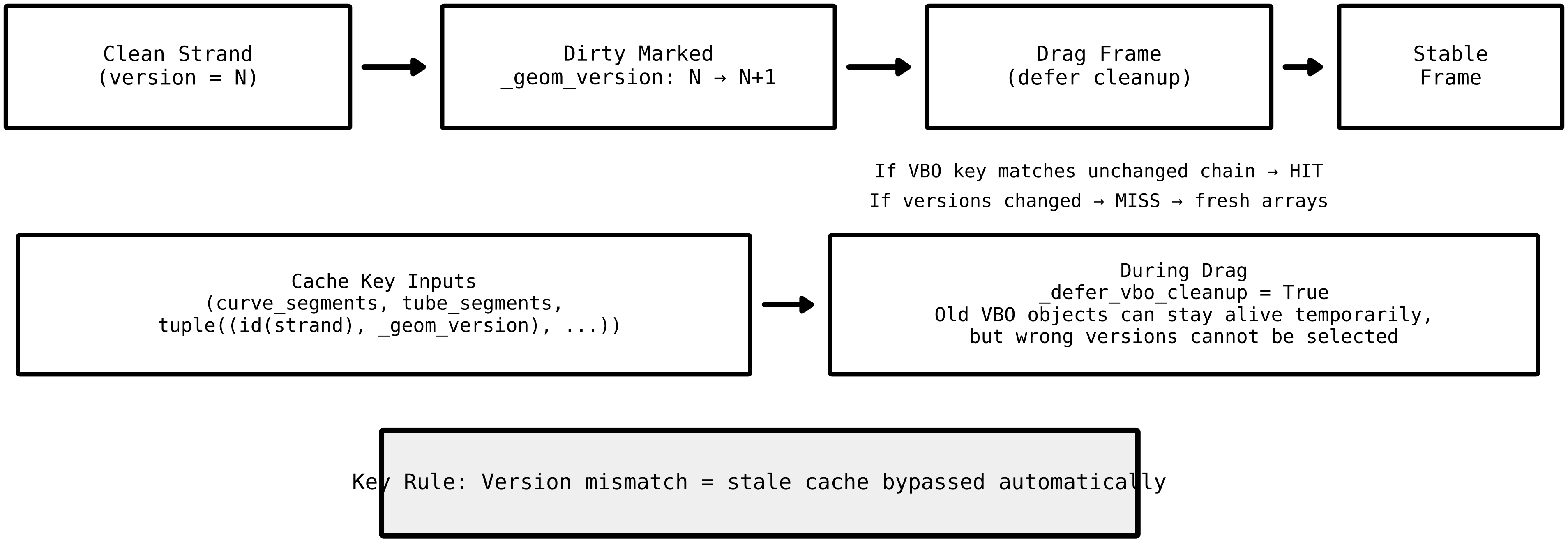
Layer 2 (Chain Geometry Cache) merges all strand curves in a chain into one array of points + parallel-transport frames. The key includes every strand's (id, version) tuple. If ANY strand in the chain changed, this cache misses and the chain is recomputed.

Layer 3 (VBO Cache) holds the final triangle mesh as GPU vertex buffer objects. A VBO hit means the geometry already sits in GPU memory -- rendering costs just 5 GL calls (~0.1ms).

During a drag, only the affected chain's versions change. The other 3 chains hit all 3 cache layers and render almost for free.

Dirty Strand + Cache State Flow

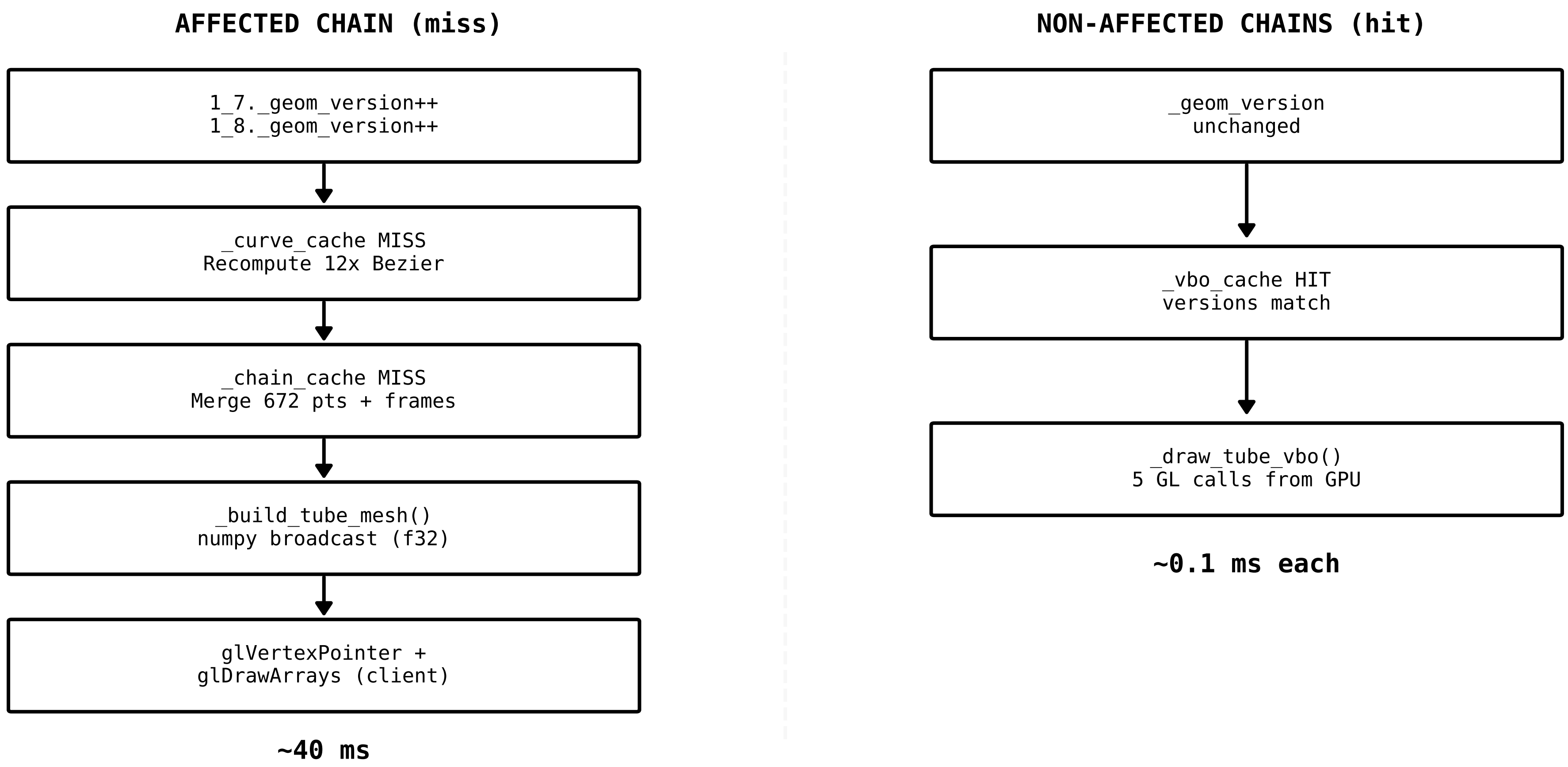
How _geom_version and cache keys prevent stale renders



A dirty strand is any strand whose geometry update increments _geom_version. Cache keys include that version for every strand in the chain, so stale entries are safe to keep temporarily: they simply do not match the new key. During drag, VBO deletion is deferred to avoid GPU sync stalls, while non-affected chains can still hit cached VBOs and render quickly.

Drag-Frame Rendering Flow

Example: dragging strand 1_7's end point

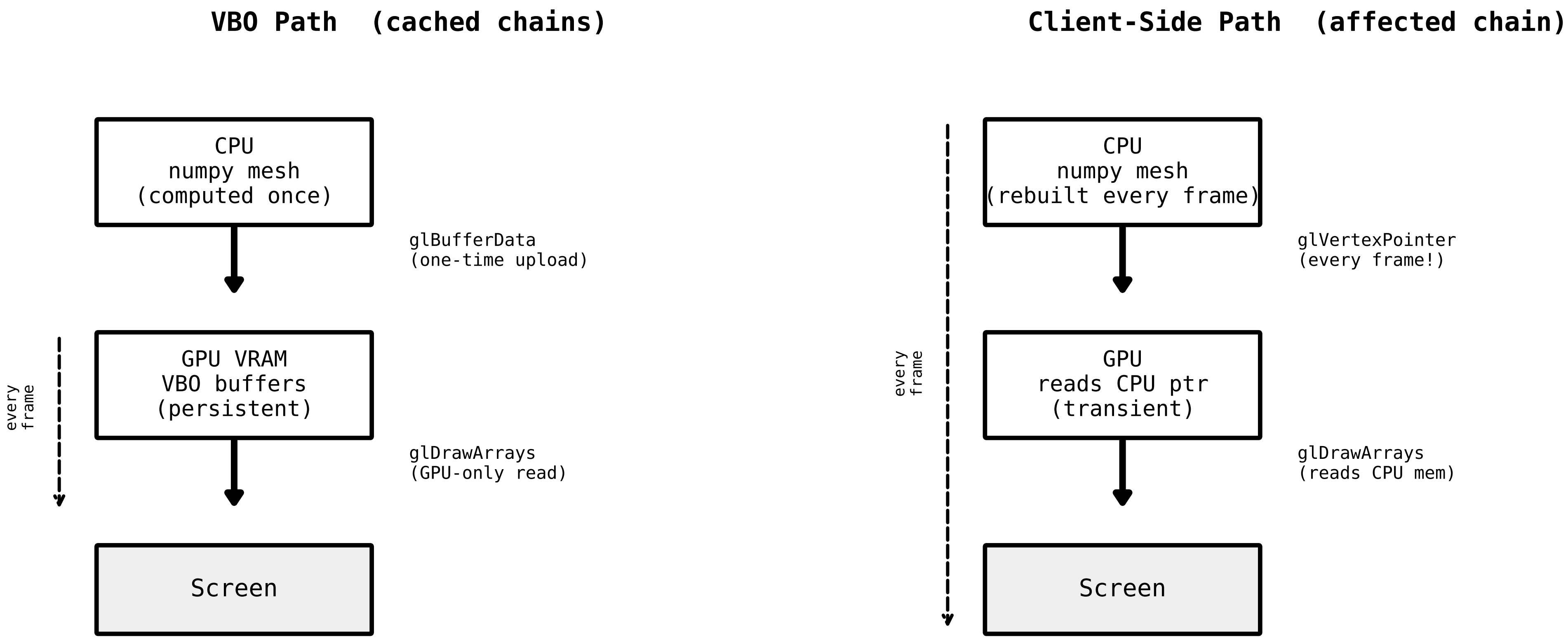


When you drag 1_7's end point, _update_move() shifts 1_7.end and 1_8.start, then calls _mark_geometry_dirty() on both. This increments their _geom_version and clears their curve caches. Chain 1's root (1_1) detects the version mismatch and rebuilds the entire chain. Chains 2, 3, 4 have unchanged versions -- they skip straight to the cached VBO in GPU memory. Total per frame: ~40ms (1 rebuild) + 3 x 0.1ms (3 hits) = ~40ms.

Per-Frame Breakdown During Drag

```
EACH FRAME WHEN DRAGGING 1_7's END:

  39 non-root strands:  strand.draw() -> return immediately          ~0 ms

  Chain 1 (1_1, AFFECTED):
    _get_chain_geometry  -> MISS -> recompute                         ~35 ms
      |-- 12x vectorised Bezier points (numpy broadcast)
      |-- 672-point scalar parallel transport (plain Python math)
      +-- chain twist application
    _build_tube_mesh    -> numpy broadcast f32                        ~5 ms
      |-- (671, 40, 3) vertices + normals in one operation
      +-- glVertexPointer + glDrawArrays (client-side)

  Chain 2 (2_1, not affected):
    _get_chain_geometry  -> HIT  -> cached                            ~0.01 ms
    VBO lookup           -> HIT  -> GPU render                        ~0.1 ms

  Chain 3 (3_1, not affected):    same as chain 2                     ~0.1 ms
  Chain 4 (4_1, not affected):    same as chain 2                     ~0.1 ms

  ------------------------------------------------------------------------
  TOTAL PER FRAME:                                                    ~40 ms = 25 FPS


WHY NOT CREATE A VBO FOR THE AFFECTED CHAIN?

  Creating a VBO means uploading data to GPU memory with glBufferData(). But the
  affected chain's geometry changes every single frame during drag. Uploading to a
  VBO that will be stale 16ms later is wasted work. Instead, we use client-side
  arrays: glVertexPointer points the GPU directly at our numpy array in CPU RAM.
  The GPU reads it once and draws. No upload, no cleanup.

  VBO cleanup is also deferred during drag (_defer_vbo_cleanup = True) so old
  cache entries aren't deleted mid-operation. After the drag ends, end_drag_operation()
  clears stale VBOs and the next static frame creates fresh ones.


COMPARISON:

  Without VBO cache (all chains rebuild):   4 x ~40ms = ~170ms = 6 FPS
  Without numpy (Python loops, per-vertex): all chains = ~3,000ms = 0.3 FPS
  Current (VBO + numpy):  1 rebuild + 3 hits = ~40ms = 25 FPS
```

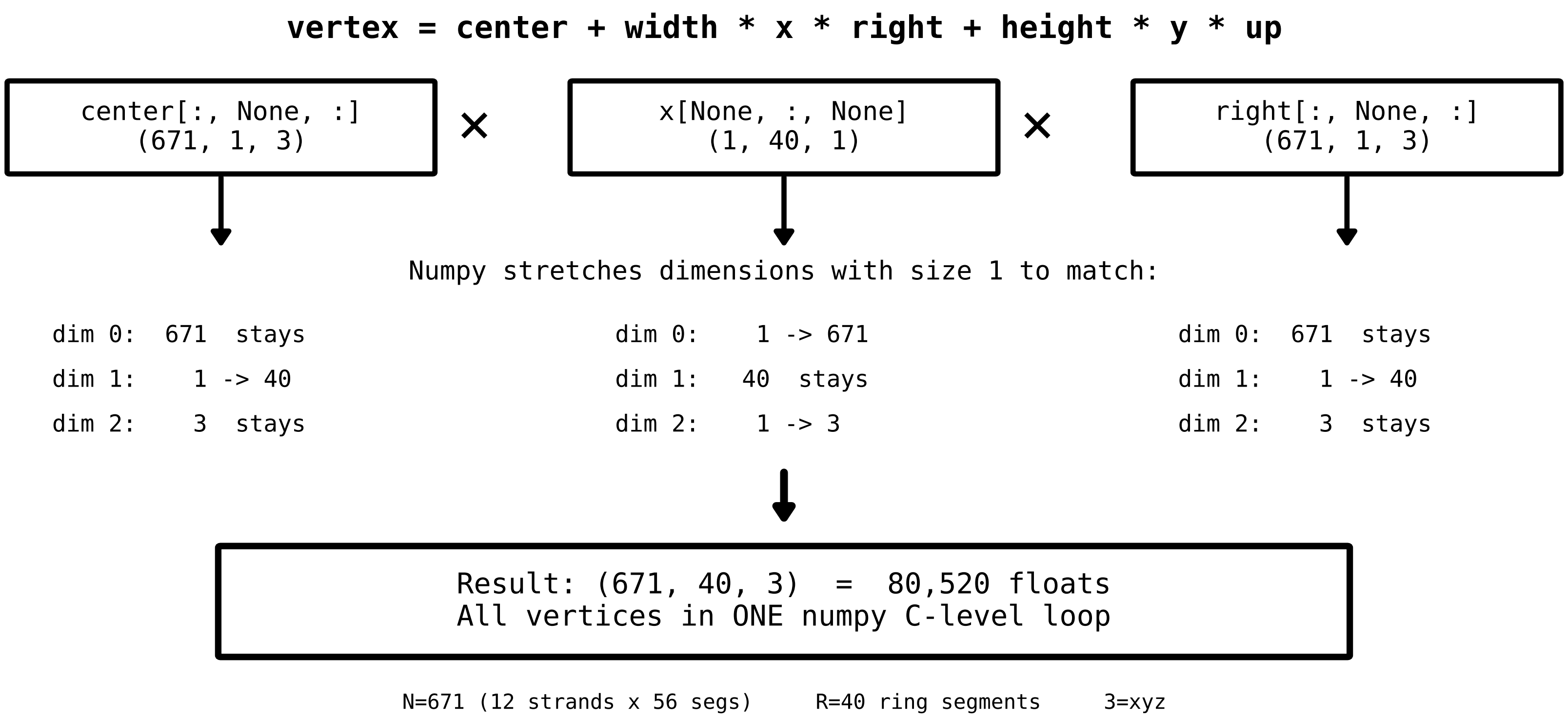
VBO vs Client-Side Arrays



VBO (Vertex Buffer Object) stores geometry in GPU memory. Once uploaded, the GPU reads from its own fast VRAM on every frame -- the CPU just says "draw what you have".
Cost: ~0.1ms per chain. This is used for non-affected chains whose data hasn't changed.

Client-side arrays keep data in CPU RAM. The GPU must reach across the bus to read it
each frame. This is slower, but makes sense for the affected chain: the geometry changes
every frame during drag, so uploading to a VBO would be wasted effort.

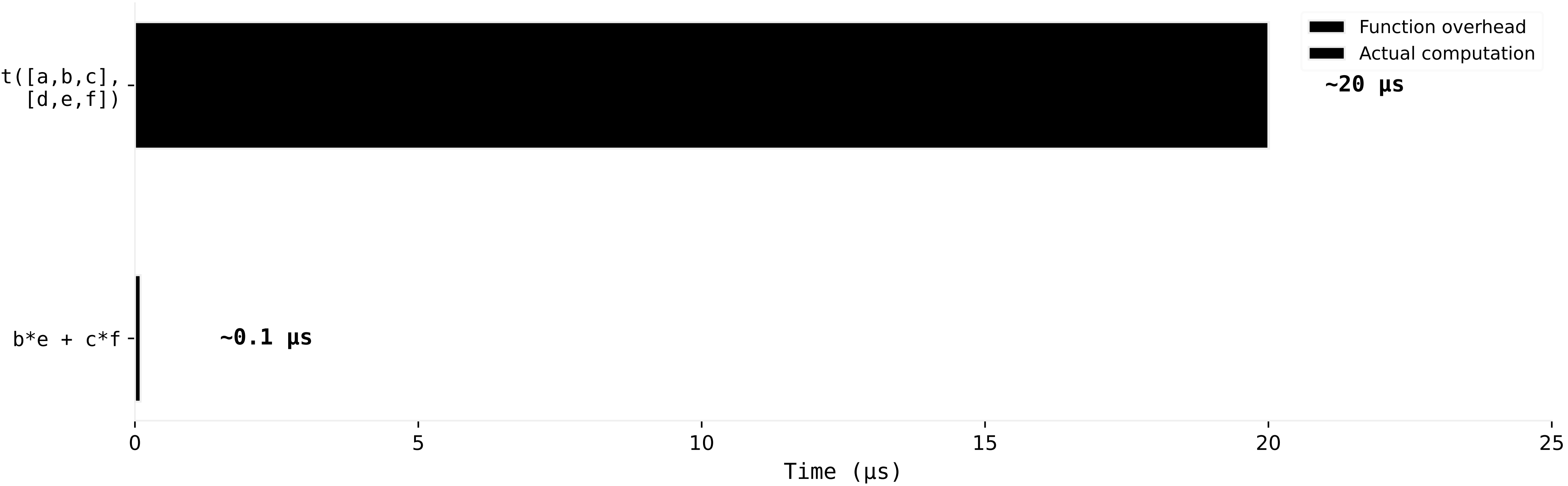
Numpy Broadcasting in _build_tube_mesh



Traditional approach: two nested Python for-loops (671 x 40 = 26,840 iterations), each computing one vertex with Python-level math. Numpy broadcasting replaces this with a single C-level operation over the entire (671, 40, 3) array. Same math, ~100x faster because the Python interpreter never touches individual floats.

Normals are computed the same way: cross products via broadcasting, then batch-normalised with np.linalg.norm(all_normals, axis=3, keepdims=True).

Why Scalar Math Beats Numpy for Tiny Arrays



Numpy is designed for large arrays. For a 3-element dot product, the actual
math (3 multiplies + 2 adds) takes ~0.01us. But numpy's function call overhead
dominates: parsing Python arguments (~5us), checking array types/shapes (~5us),
allocating a result array (~5us), wrapping the result as a Python object (~5us).
Total: ~20us for 3 multiplications.

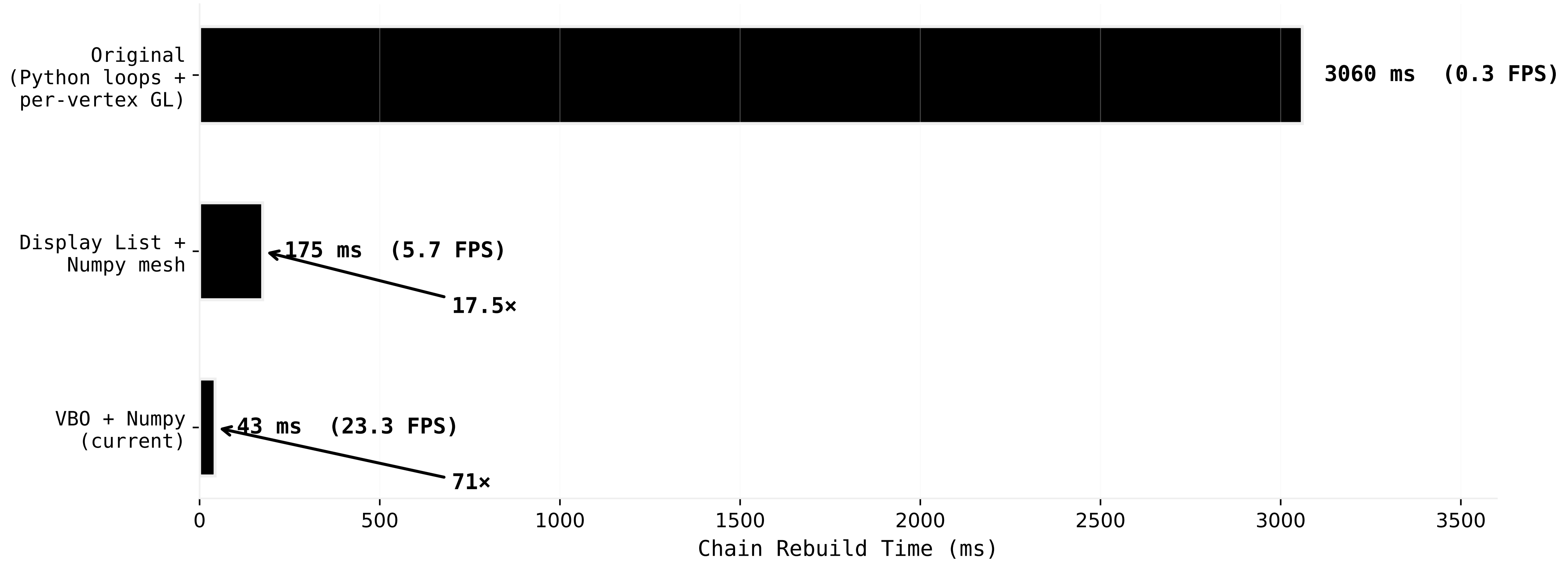
Plain Python scalar math (a*d + b*e + c*f) takes ~0.1us -- 200x faster.

This matters for parallel transport frames, where each of 672 iterations
computes
a dot product, cross product, and rotation. With numpy: 672 x 75us = 50ms
overhead.
With scalars: 672 x 0.5us = 0.3ms. The fix: pre-compute tangents in one numpy
batch,
then loop with plain Python floats.

Rule of thumb: numpy is 100x FASTER for arrays with 1000+ elements (C loop
amortises
the overhead). But 200x SLOWER for tiny arrays (3 elements) where overhead
dominates.

Performance Comparison

Chain rebuild time for 43 strands across 4 chains



ORIGINAL: Each strand's tube was built with nested Python for-loops, and each vertex was sent to the GPU individually with glVertex3f(). For 43 strands, this meant millions of Python-level operations per frame. Result: ~3,060ms per chain rebuild, ~0.3 FPS.

DISPLAY LIST + NUMPY: Replaced Python loops with numpy broadcasting for mesh construction, and per-vertex GL calls with glVertexPointer + glDrawArrays. OpenGL display lists cached the draw commands. Result: 17.5x speedup to ~175ms, but display lists are deprecated.

VBO + NUMPY (current): Replaced display lists with VBOs stored in GPU memory. Only the affected chain rebuilds during drag; others render from cached GPU buffers in

Summary: Key Optimisation Techniques

1. CHAIN-BASED RENDERING
   Only 4 chain roots do rendering work, not all 43 strands. Non-root strands return immediately from draw(). Cost scales with chain count, not strand count.

2. THREE-LAYER CACHE with VERSION TRACKING
   Each strand has a _geom_version integer. Cache keys include version tuples. When a control point moves, only that strand's version increments, causing cache misses only in the affected chain. All other chains hit every layer.

3. NUMPY BROADCASTING for MESH CONSTRUCTION
   The (N, R, 3) vertex array is computed in one C-level operation instead of 26,840 Python iterations. Uses float32 arrays that OpenGL reads directly.

4. SCALAR MATH for SEQUENTIAL OPERATIONS
   Parallel transport frames must be computed sequentially. Plain Python float math avoids numpy's ~20us per-call overhead on 3-element arrays. Result: 672 iterations in 0.3ms instead of 50ms.

5. VBO for STATIC CHAINS, CLIENT-SIDE ARRAYS for DYNAMIC
   Non-affected chains keep geometry in GPU VRAM (VBO) -- rendering costs ~0.1ms. The affected chain uses client-side arrays because its data changes every frame. VBO cleanup is deferred during drag to avoid GPU sync stalls.

6. DEFERRED VBO CLEANUP
   During drag, _defer_vbo_cleanup prevents glDeleteBuffers from running. Stale VBO entries accumulate (max 30) but aren't freed until the drag ends. This avoids expensive GPU pipeline flushes during interactive operations.

RESULT: 3,060ms -> 43ms (71x faster) | 0.3 FPS -> 25 FPS