Gazebo GUI Performance Profiling Report

gz-sim PR #3447 (EnTT ECM) with gz-transport Zenoh 1.8 + shared memory (PRs #868, #842)

Carlos Agüero

caguero@honurobotics.com

September 2026

Contents

1 Introduction

This is the GUI half of the September 2026 profiling run. The companion server report (same date, same workspace) profiles `gz-sim-main`; this one profiles the GUI client process `gz-sim-gui-client` (thread `comm gz-sim-gui`) while a server feeds it world state. It follows the July 2026 GUI study, which was already taken on PR #3447, and repeats it on a newer stack:

- **gz-sim PR #3447** (ECM on EnTT) at `eec8c3fdb`, two months of upstream `main` merged in since July.
- **gz-transport PR #868 + #842** (Zenoh 1.8.0 with shared memory), used by both the server and the GUI (`GZ_TRANSPORT_IMPLEMENTATION=zenoh`). The `/world/<name>/state` and `/world/<name>/pose/info` streams that the GUI consumes are the first real exercise of the SHM path on the GUI side.
- gz-gui, gz-rendering, gz-common and the rest at the tip of `main` on 2026-09-01.

The world set is July's four (shapes, 3k_shapes_static, 3k_shapes dynamic, jetty) plus `moving_robots_and_sensors` from gz-sim PR #3846, with its vehicles driven and its sensors subscribed exactly as in the server run.

1.1 Goals

- Re-measure the GUI on the current stack and check whether July's conclusions (render thread bound, per frame scene graph walk, `shared_ptr` churn) still hold.
- Measure what the Zenoh shared memory transport changes on the GUI's state ingestion path (Qt main thread, transport threads).
- Characterize the GUI cost of the mixed robot world, where the scene is small but everything moves and sensors stream on the side.

1.2 Approach

Every flamegraph in this PDF is a static rendering; the caption of each figure links to the interactive SVG on the site (click frames to zoom, Ctrl+F to search), and the run page https://caguero.github.io/gz-profiling/2026-09-01/ lists all of them, including the per thread splits.

Same as July: the simulation is launched with `gz sim -r` (server and GUI, sim playing) and rendered on the NVIDIA GPU via PRIME offload. After a 45 s settle, `perf` attaches to the GUI process only and samples 30 s. Loading captures wrap the GUI launch (Qt init, render engine init, initial scene build) for 20 s with the server already running. Every capture is split per thread, because the GUI is a multi thread process with one dominant thread.

2 Methodology and Reproduction Procedure

2.1 Profiling Technique

`perf record -e task-clock -F 997 --call-graph dwarf` on the GUI process, collapsed with FlameGraph. Weights are nanoseconds of CPU time. The GUI process is found by thread count (the `gz` launcher spawns it behind an `sh -c` wrapper). The world is playing during runtime captures so the pose streaming and ECS synchronization paths are exercised; the July report's original captures were taken paused and its comparison run playing, this run is playing throughout.

2.2 GUI Process Architecture

The GUI is a Qt application with three threads that matter:

- **Qt main thread** (`comm gz-sim-gui`): runs `GuiRunner`, which receives the serialized ECM state over transport, deserializes it into the GUI side ECM, and runs each GUI plugin's `Update`. `RenderUtil` collects the entities and poses that changed.
- **Render thread** (`comm gz::gui::plugin...`): the OgreNext render loop owned by the `MinimalScene` plugin. Each frame it applies the collected changes (`RenderUtil::Update`), runs `Ogre2Scene::PreRender` (a walk over all visuals), renders, and swaps.
- **Qt Quick compositor** and the Zenoh runtime threads (`tx-0`, `rx-*`, `net-0`, `app-0`, `acc-0`, SHM watchdogs).

2.3 Benchmark Worlds

World	What it stresses on the GUI
`shapes`	idle floor: six primitives, nothing moves
`3k_shapes_static`	3000 visuals in the scene graph, nothing moves
`3k_shapes` (dynamic)	3000 visuals, all moving: pose streaming + scene graph
`jetty`	complex meshes and textures, GL submission cost
`moving_robots_and_sensors`	small scene, everything moves, sensors streaming to subscribers on the server side

All worlds run at `<real_time_factor>0</real_time_factor>` on the server; the GUI throttles its own state subscription, so server speed changes how often the state message changes, not the GUI frame rate.

2.4 Hardware and Software

Identical to the server report of the same date: Intel Core Ultra 9 285HX, NVIDIA RTX PRO 3000 (driver 580.159.03, GUI rendering through `libnvidia-glcore` via PRIME offload, verified from the process maps), kernel 6.17.0-35, perf 6.17.13, GCC 13.3, OgreNext 2.3.1, Zenoh 1.8.0. Workspace `~/rotary_bench_ws` built `RelWithDebInfo` with frame pointers and `ENABLE_PROFILER=OFF`; gz-sim `eec8c3fdb` (PR #3447), gz-transport `438ddec1` (PR #868 + #842), gz-gui `64b98b3`, gz-rendering `9cd640c`.

2.5 Running Captures

```
source ~/rotary_bench_ws/install/setup.bash
export GZ_TRANSPORT_IMPLEMENTATION=zenoh
export GZ_INSTALL=~/rotary_bench_ws/install FLAMEGRAPH_DIR=~/FlameGraph
export __NV_PRIME_RENDER_OFFLOAD=1 __GLX_VENDOR_LIBRARY_NAME=nvidia
# Runtime: attach perf to the settled GUI for 30 s (sim playing)
STARTUP_WAIT=45 OUTPUT_DIR=captures_gui/runtime \
  ./scripts/gz_gui_flamegraph.sh worlds/3k_shapes.sdf 3k_shapes_dynamic 30 runtime
# Loading: wrap the GUI launch for 20 s
OUTPUT_DIR=captures_gui/loading \
  ./scripts/gz_gui_flamegraph.sh worlds/jetty.sdf jetty 20 loading
# Per thread split and GUI subsystem breakdown
OUTPUT_DIR=captures_gui/threads ./scripts/gz_per_thread_flamegraph.sh \
  captures_gui/runtime/perf_3k_shapes_dynamic.data 3k_shapes_dynamic
./scripts/gz_gui_analyze.sh captures_gui/runtime/3k_shapes_dynamic.folded
```

`gz_gui_flamegraph.sh` now honours the same `<world>.topics` and `<world>.setup.sh` companions as the server script, so for `moving_robots_and_sensors` the twelve sensor subscribers are started and the vehicles and arm are commanded once the GUI is up.

3 Executive Summary

Five worlds, 30 s runtime captures of the GUI process with the simulation playing, plus three loading captures, all split per thread.

World	GUI cores busy	Render thread	Qt main	Qt Quick	Zenoh rx/net	July 2026 cores
shapes	0.15	66%	10%	20%	3%	0.16
3k_shapes_static	1.04	89%	6%	4%	1%	1.05
3k_shapes_dynamic	0.81	88%	7%	5%	1%	1.04
jetty	0.49	75%	10%	8%	5%	0.67
moving_robots_and_sensors	0.44	27%	34%	12%	26%	new

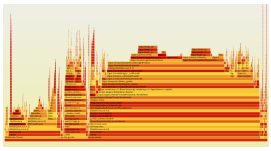
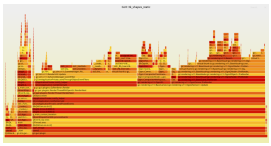
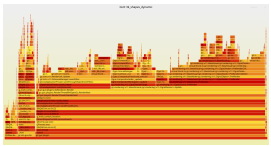
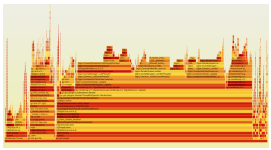
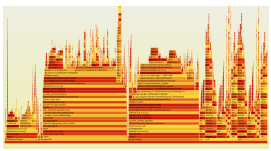
Top findings:

1. **The static 3000 entity world costs a full core in the GUI, and half of it is Gazebo code re-applying poses that did not change.** `RenderUtil::Update` is 38.7% of GUI CPU in `3k_shapes_static`. Every GUI update, `RenderUtilPrivate::UpdateRenderingEntities` runs an unconditional `Each<Model, Pose>` (and the Link, Visual, Light and Actor variants) over the GUI side ECM and copies every pose into `entityPoses`; the render thread then walks that map and, for each of the 3000 entities, does a `SceneManager::NodeById` lookup (five hash maps probed in sequence, 14.6% of GUI CPU), a `dynamic_pointer_cast<Visual>` (4.9%), two string keyed `UserData` variant lookups (9.6%) and `SetLocalPose` (4.2%). The server sends nothing for these entities (its state stream carries 13 entities per message at 47 Hz in this world); the cost is entirely GUI side.
2. **The per frame scene graph walk from July is unchanged.** `Ogre2Scene::PreRender` -> `BaseVisual::PreRenderChildren` -> `Ogre2Node::Children` (a by value copy of the child container) is 35.7% (static) and 41.5% (dynamic) of GUI CPU; `shared_ptr` refcount atomics are 6 to 8%. Together with item 1 the render thread is 88 to 89% of the GUI in the 3k worlds, exactly July's picture, now with the second half named.
3. **jetty's GUI got cheaper (0.67 -> 0.49 cores) and is GPU submission bound.** OgreNext `RenderQueue::renderGL3` and Forward Clustered light assignment are 32% and 21%, the NVIDIA driver 20%, and 12% is `pthread` mutex traffic inside the driver's submission path. `Ogre2Scene::PreRender` is 10% and `RenderUtil::Update` 5%: the scene graph walk matters less when each visual is expensive to draw.
4. **The mixed robot world turns the GUI into a state ingestion benchmark.** With the vehicles driving, the server publishes `/world/.../state` at **2,443 messages per second** (measured over 5 s: 12,216 messages, 24 entities and 30 components each). The cause is on the server: DiffDrive and AckermannSteering call `SetComponentData<JointVelocityCmd>` on every step, which marks a one time change, and `SceneBroadcaster` publishes immediately on any one time change instead of waiting for the 60 Hz period. On the GUI this shows up as 26% of CPU on two Zenoh receive threads (protobuf parse of `SerializedStepMap`, `GuiRunner::OnState` running on the receive thread, 37% of those threads in `malloc`), and a Qt main thread (34%) that spends a third of its time in `EntityComponentManager::SetState` deserializing text encoded components (`gz::math::operator>>`, `strtod`, 16% of the thread) and another 14% in protobuf. Rendering is only 27% of this GUI.
5. **Zenoh with shared memory is invisible in the entity heavy worlds and only visible where the server floods it.** In the 3k worlds and jetty the receive threads are 1 to 5% of GUI CPU; the state messages in the 3k dynamic world are 28 KB (below the 128 KB SHM threshold, so they go through the heap path), and the SHM path is exercised in the mixed world (`withPayloadView` frames on the receive threads) where the cost is the message rate, not the transport.
6. **The 3k GUI is memory bound** (IPC 0.69 static, 0.91 dynamic, 70% and 43% LLC load misses): the scene graph walk and the pose loop chase `shared_ptr` nodes and hash buckets across 3000 visuals. The mixed world's GUI is compute bound (IPC 1.88, 1.7% LLC misses) on deserialization.

4 Runtime Analysis Per World

4.1 Runtime Flamegraphs

Click any thumbnail to open the full interactive flamegraph in a browser (click to zoom, hover tooltips, Ctrl+F search; the link pre-highlights the frames discussed in the text).

World	GUI CPU	Flamegraph
shapes	0.15 cores	
3k_shapes_static	1.04 cores	
3k_shapes_dynamic	0.81 cores	
jetty	0.49 cores	
moving_robots_and_sensors	0.44 cores	

Percentages are of the GUI process's total CPU during the 30 s capture unless a thread is named.

4.2 shapes (idle baseline, 0.15 cores)

Function	Self %	Category
`Ogre::Node::updateFromParentImpl`	15.2	OgreNext scene graph
`[libnvidia-glcore]`	12.5	NVIDIA driver
`Ogre::ForwardClustered::collectLightForSlice`	8.2	OgreNext lighting
`Ogre::Camera::isViewOutOfDate`	5.7	OgreNext
`Ogre::Frustum::updateFrustumPlanesImpl`	5.3	OgreNext
`___pthread_mutex_lock`	2.0	driver / Qt

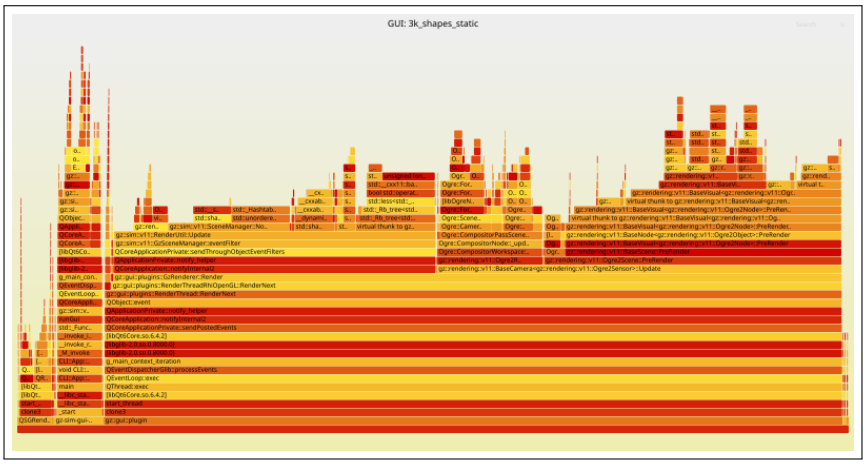
The idle floor is a render loop that keeps rendering six primitives: 61% of the process is under `gz::rendering::`, 20% in the driver, 7% in `pthread` synchronisation. `Ogre2Scene::PreRender` is 1.5% and `RenderUtil::Update` 1.1%; the ECM sync is 4.7%. July measured 0.16 cores; nothing changed here.

4.3 3k_shapes_static (3000 visuals, nothing moves, 1.04 cores)

Function	Self %	Category
`gz::rendering::Ogre2Node::Children`	8.5	scene graph walk (by value copy)

Function	Self %	Category
`std::_Hashtable<Entity, shared_ptr<Visual>>::find`	7.1	`SceneManager::NodeById`
`BaseVisual::PreRenderChildren`	6.5	scene graph walk
`std::min<unsigned long>` (hash policy)	6.3	`NodeById`
`__gnu_cxx::__atomic_add`	4.5	`shared_ptr` refcount
`std::__shared_ptr<Node>` copy	4.4	`shared_ptr`
`__cxxabiv1::__vmi_class_type_info::__do_dyncast`	4.1	`dynamic_pointer_cast<Visual>`
`SceneManager::NodeById`	3.0	gz-sim GUI
`[libnvidia-glcore]`	2.7	driver

Inclusive: `gz::rendering::` 79.7%; **`RenderUtil::Update` 38.7%** (of which `NodeById` 37.9%, `UserData` 24.9%, `dynamic_pointer_cast` 12.6%, `SetLocalPose` 10.8%, the `entityPoses` map 5.6%); **`Ogre2Scene::PreRender` 35.7%** (`PreRenderChildren` 33.2%, `Ogre2Node::Children` 8.6%); all `dynamic_cast` 8.2%; refcount atomics 6.4%; `Ogre::` 18.1%; driver 4.1%; `GuiRunner/SetState` 4.8%; `UpdateRenderingEntities` 3.8%.



3k_shapes_static GUI runtime. Two towers on the render thread: `RenderUtil::Update` (pose loop over all entities) on the left and `Ogre2Scene::PreRender` (scene graph walk) on the right.

Click to open interactive flamegraph

Findings: see summary items 1 and 2. Both towers scale with the number of visuals and run every frame whether or not anything moved. The server's state stream for this world is 13 entities per message at 47 Hz (each of the 3004 entities is resent about once every 5 s), so `SetState` is cheap; the pose loop is fed by the GUI's own `Each<Pose>` scan, not by the server.

4.4 3k_shapes_dynamic (3000 visuals, all moving, 0.81 cores)

Function	Self %	Category
`Ogre2Node::Children`	9.0	scene graph walk
`BaseVisual::PreRenderChildren`	6.1	scene graph walk
`__gnu_cxx::__atomic_add`	5.3	`shared_ptr`
`_Hashtable<Entity, shared_ptr<Visual>>::find`	4.6	`NodeById`
`std::min<unsigned long>`	4.6	`NodeById`
`Ogre::Node::updateAllTransforms`	4.3	OgreNext transforms
`Ogre::Node::updateFromParentImpl`	3.4	OgreNext transforms
`__do_dyncast`	2.8	`dynamic_pointer_cast`

Inclusive: `Ogre2Scene::PreRender` 41.5%, `RenderUtil::Update` 27.1%, OgreNext transform update 8.5%, Forward Clustered 8.2%, refcount 8.1%, `dynamic_cast` 6.6%, `SetState` 5.5%, driver 4.7%.

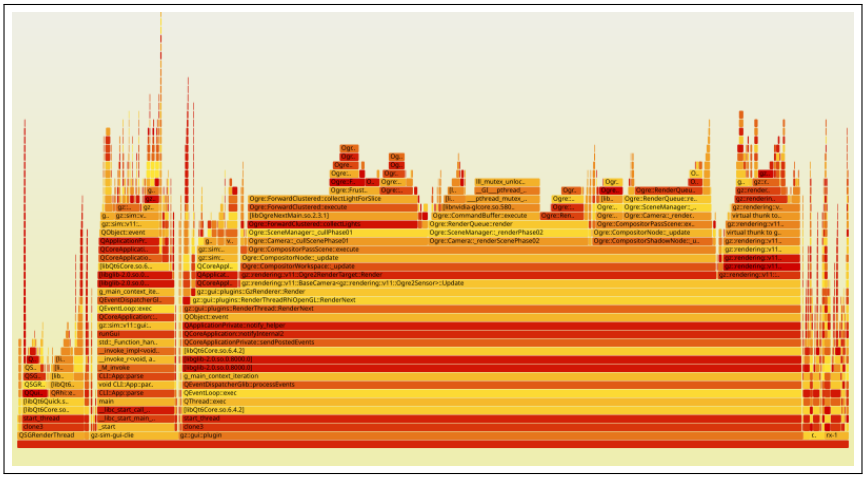
3k_shapes_dynamic GUI runtime.
Click to open interactive flamegraph

Findings: the dynamic world costs *less* GUI CPU than the static one (0.81 vs 1.04 cores; July had them equal at 1.04/1.05). By the time the capture starts the shapes have settled, the server sends 500 entities per message at 11 Hz, and the render thread has slightly less to do because OgreNext's transform update replaces part of the pose loop cost. The ordering of the two towers is the same as in the static world.

4.5 jetty (complex meshes, 0.49 cores)

Function	Self %	Category
`Ogre::ForwardClustered::collectLightForSlice`	9.7	OgreNext lighting
`lll_mutex_unlock_optimized`	8.2	driver submission lock
`Ogre::RenderQueue::renderGL3`	7.7	OgreNext GL submission
`[libnvidia-glcore]`	7.6	driver
`Ogre::Node::updateFromParentImpl`	4.3	OgreNext transforms
`Ogre::RenderQueue::addRenderableV2`	2.8	OgreNext
`Ogre::HlmsPbsDatablock::getCubemapProbe`	2.4	OgreNext PBS

Inclusive: `Ogre::` 59.2% (`RenderQueue::render/renderGL3` 31.6%, Forward Clustered 20.6%), driver 20.0%, `pthread` 11.7%, `Ogre2Scene::PreRender` 9.9%, `SetState` 8.7%, Qt Quick 8.6%, Zenoh 5.4%, `RenderUtil::Update` 4.9%.



jetty GUI runtime. GL submission and lighting dominate the render thread; the scene graph walk is the small tower on the right.
Click to open interactive flamegraph

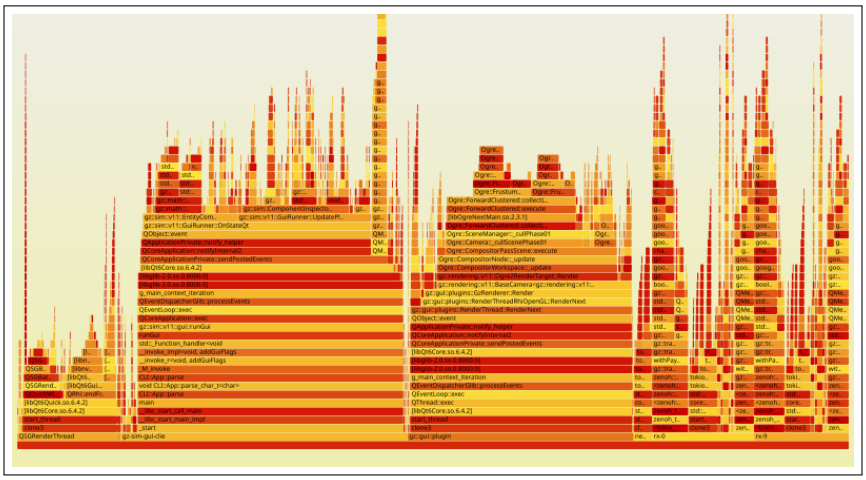
Findings: jetty is the one world where the GUI got cheaper since July (0.67 -> 0.49 cores). Its cost is per draw call (PBS datablocks, cubemap probe lookups, GL submission) rather than per visual, and the driver lock traffic suggests OgreNext's GL3+ render system serializes on the NVIDIA context. The ECM sync is 9%, the largest share of the four July worlds, because jetty has moving models whose poses stream continuously.

4.6 moving_robots_and_sensors (PR #3846 world, 0.44 cores)

Thread	Share	What it does
Qt main (`gz-sim-gui`)	34.2%	`GuiRunner::OnStateQt` -> `ECM::SetState` (33% of the thread), text deserialization (`strtod`, 16%), protobuf (14%), plugin updates
Render (`gz::gui::plugin`)	27.1%	OgreNext transforms 20%, Forward Clustered 16%, driver
Zenoh `rx-0` + `rx-9`	23.5%	protobuf parse of the state (54%), `GuiRunner::OnState` (33%), `malloc` 37%
Qt Quick render (`QSGRenderThread`)	12.0%	driver, compositor
Zenoh `net-0`	2.2%	

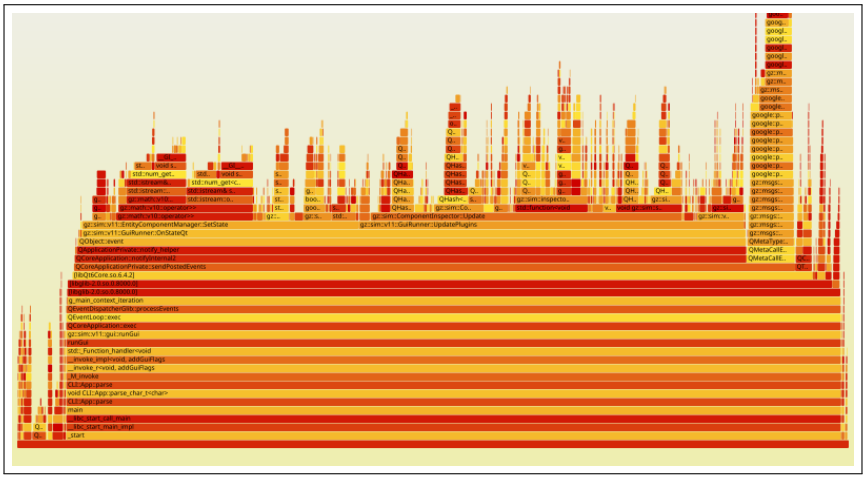
Function	Self %	Category
`[unknown]` (kernel, under recv/ioctl)	8.3	transport / driver
`[libnvidia-glcore]`	6.7	driver
`_int_malloc`	6.4	message and ECM allocations
`Ogre::Node::updateFromParentImpl`	5.6	OgreNext
`Ogre::ForwardClustered::collectLightForSlice`	3.1	OgreNext
`_int_free`	2.5	
`__GI_____strtod_l_internal`	1.5	text component deserialization

Inclusive: `GuiRunner/SetState` 35.5%, Zenoh 25.8%, protobuf 18.3%, `gz::transport::` 17.2%, `malloc/free` 15.5%, `gz::rendering::` 24.6%, Forward Clustered 15.8%, driver 10.6%, Qt Quick 12.6%, `sdf::` 1.2%.



moving_robots_and_sensors GUI runtime. Left to right: Qt main thread (state deserialization), render thread, two Zenoh receive threads, Qt Quick render thread.

Click to open interactive flamegraph



Qt main thread of the mixed world: `ECM::SetState` and text deserialization of components.

Click to open interactive flamegraph

Findings: the server sends 2,443 state messages per second in this world. Three component types are in every message: the `Pose` of the vehicle links (text encoded, `"x y z r p y"`), a binary encoded joint component on the arm and wheel joints, and `JointVelocityCmd` on the five driven wheel joints. The last one is the trigger: `DiffDrive` and `AckermannSteering` write it with `SetComponentData` on every `PreUpdate`, which the ECM records as a one time change, and `SceneBroadcaster::PostUpdate` publishes on any one time change (`changeEvent` in its `shouldPublish` condition) without waiting for the `state_hertz` period. At the world's steady 3,000 steps per second the GUI receives a message on eight of every ten steps. Each message is small (24 entities, under 1 KB) so the cost is per message overhead: Zenoh delivery, `SerializedStepMap` parse, the `OnState` callback on the receive thread, then the Qt side `SetState` with `istream` based deserialization of `Pose3d`. The render thread itself is cheap (0.12 cores) because the scene is small (a few dozen visuals).

5 Per Thread Analysis

Thread (role)	shapes	3k_static	3k_dynamic	jetty	moving_robots
Render (`gz::gui::plugin`, OgreNext loop)	65.6%	88.8%	87.5%	75.1%	27.1%
Qt main (`gz-sim-gui`, ECS sync + plugins)	9.7%	5.8%	6.6%	10.3%	34.2%
Qt Quick render (`QSGRenderThread`)	19.8%	4.1%	4.6%	8.0%	12.0%
Zenoh receive (`rx-*`)	2.2%	0.9%	0.7%	4.6%	23.5%
Zenoh `net-0`, `tx-0`, X events, other	2.7%	0.4%	0.6%	2.0%	3.2%

Two points differ from July. First, the Zenoh receive threads are a real consumer whenever the server publishes at high rate: the gz-transport subscription callback (`GuiRunner::OnState`) runs on the Zenoh receive thread, so message parsing and the first half of the state handling are off the Qt main thread but still cost CPU. Second, the Qt main thread's share is unchanged on the entity heavy worlds (6 to 7%), confirming that the EnTT ECM keeps the GUI side ECM cheap; where the Qt thread is large (mixed world) it is deserialization work proportional to message rate, not to entity count.

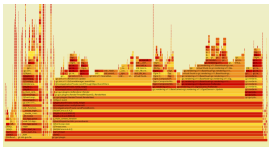
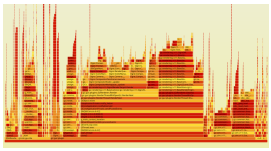
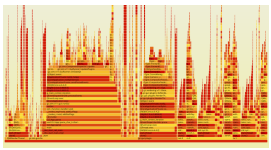
6 Cross Reference: Gazebo Owned Hotspots

Item	Owner	Where	Cost	Notes
Unconditional `Each<Pose>` + per entity pose apply every frame	gz-sim `RenderUtil`	3k_static, 3k_dynamic	39% / 27%	`NodeById` (5 hash maps), `dynamic_pointer_cast`, 2 string keyed `UserData` lookups, `SetLocalPose` per entity per frame
Per frame scene graph walk with by value child copies	gz-rendering `Ogre2Scene::PreRender`	3k worlds	36% / 41%	July's Priority 1 and 2, unchanged
State published on every one time change	gz-sim `SceneBroadcaster` + `DiffDrive/AckermannSteering`	moving_robots	2,443 msg/s, ~60% of GUI CPU	server side trigger, GUI side cost
Text (`istream`) deserialization of `Pose3d` and vector components	gz-sim components / gz-math	moving_robots Qt main	16% of thread	`strtod` per number, allocation per component
`GuiRunner::OnStat` on the transport receive thread	gz-sim GUI + gz-transport	moving_robots	33% of rx threads	copies state before posting to Qt
GL submission lock traffic	OgreNext GL3+ on NVIDIA	jetty	12%	not Gazebo code

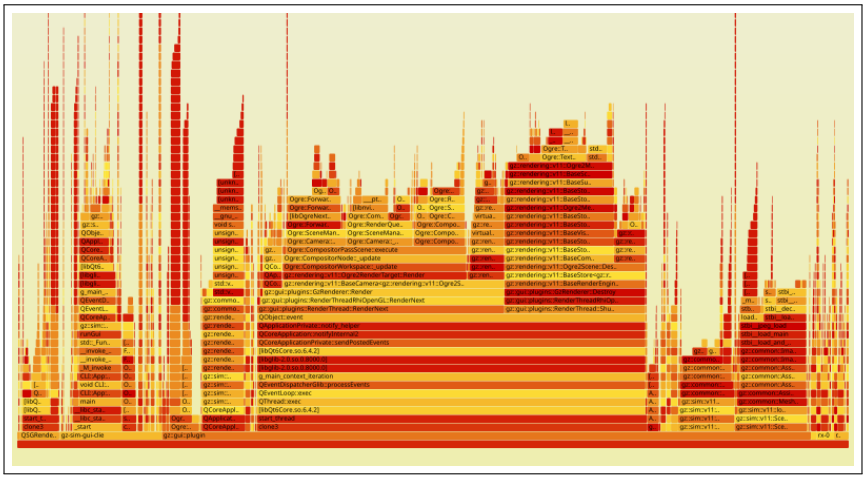
7 Loading Analysis (GUI startup and scene construction)

7.1 Loading Flamegraphs

Click any thumbnail to open the full interactive flamegraph in a browser (click to zoom, hover tooltips, Ctrl+F search; the link pre-highlights the frames discussed in the text).

World	CPU in 20 s	Flamegraph
3k_shapes_static	22.4 CPU s	
jetty	17.8 CPU s	
moving_robots_and_sensors	6.8 CPU s	

World	GUI CPU in the 20 s window	Dominant cost
3k_shapes_static	22.4 s	`RenderUtil::Update` 37%, `Ogre2Scene::PreRender` 26%, `CreateMaterial` 12%, `CreateVisual` 8%
jetty	17.8 s	`Ogre2Material/CreateMaterial` 38%, `LoadGeometry` 24%, `CreateVisual` 24%, image decoding (`stbi_*`, `Image::`) 19%, Assimp 18%, driver 16%
moving_robots_and_sensors	6.8 s	Zenoh 34%, `GuiRunner/SetState` 33%, protobuf 18%, driver 16%



jetty GUI loading: mesh and material construction.

Click to open interactive flamegraph

The 3k world's GUI start is dominated by the same two per frame towers as its runtime; the scene is built within the first seconds and the rest of the window is steady state. jetty's start is mesh and texture construction (`SceneManager::LoadGeometry`, `CreateVisual`, material creation with `stbi` decoding of textures on the GUI side, since the server's lazy texture loading does not apply to the GUI's own render engine). The mixed world's GUI start is already the state ingestion pattern.

8 Cache Analysis

`perf stat` on the GUI process for 10 s, unprofiled, sim playing.

World	IPC	L1d miss	LLC load miss
3k_shapes_static	0.69	2.4%	69.7%
3k_shapes_dynamic	0.91	2.2%	43.0%
moving_robots_and_sensors	1.88	1.0%	1.8%

The 3k GUI is memory bound, as in July (IPC 0.88 to 0.96 then). The pose loop and the scene graph walk both pointer chase through heap allocated `shared_ptr` nodes and hash buckets for 3000 visuals, and 70% of last level cache loads miss in the static case. The mixed world's GUI is compute bound on parsing and deserialization.

9 Optimization Recommendations (Gazebo owned, priority ranked)

9.1 Priority 1: Only apply poses that changed

`RenderUtilPrivate::UpdateRenderingEntities` should collect poses with `EachChanged` (or from the `ChangedState` the GUI just applied) instead of `Each<..., Pose>`, so `entityPoses` holds only entities whose pose moved since the last frame. In `RenderUtil::Update`, resolve the entity once to a `VisualPtr` when the visual is created and keep it in the `entityPoses` value (or in a parallel vector), which removes the five map probes of `NodeById`, the `dynamic_pointer_cast` and the two `UserData` string lookups per entity per frame; the `pause-update` and `gazebo-entity` user data can be checked once for the selected entity rather than for all 3000. Expected: up to 39% of the GUI in static scenes and 27% in moving ones.

9.2 Priority 2: Stop traversing unchanged visuals in `Ogre2Scene::PreRender`

Unchanged from July: `BaseScene::PreRender` walks every visual every frame, and `Ogre2Node::Children` returns the child container by value. Returning a `const` reference and keeping a dirty list of visuals that need `PreRender` would remove most of the 36 to 41%.

9.3 Priority 3: Do not publish state on every one time change

`SceneBroadcaster` publishes as soon as any one time change is pending. A controller that writes a command component every step (`DiffDrive`, `AckermannSteering` with `SetComponentData<JointVelocityCmd>`) therefore forces a publish every step or two. Either the command components should be marked as periodic changes (they are overwritten every step by design), or `SceneBroadcaster` should coalesce one time changes into the next periodic publish unless an entity was created or removed. This removes the 2,443 messages per second and with it roughly 60% of the mixed world's GUI CPU, and the same server side cost of serializing those messages.

9.4 Priority 4: Binary serialization for math components

`Pose3d` and the vector valued components are serialized as text (`operator<<` / `operator>>`), so every state message pays `strtod` on both ends and produces a temporary string per component. Serializing `Pose3d` as a `gz::msgs::Pose` (or raw doubles) would remove the 16% `istream` cost from the Qt main thread and shrink the messages.

9.5 Priority 5: Keep the state callback off the transport receive thread

`GuiRunner::OnState` runs inside the Zenoh receive callback and does the protobuf copy there before handing the state to the Qt thread. Handing the raw buffer to the Qt thread and parsing there (or parsing directly from the SHM buffer) would keep the receive threads at their 1% floor even under a flooding publisher and avoid blocking Zenoh's delivery.

10 Comparison with July 2026

Item	July 2026	September 2026
shapes	0.16 cores	0.15 cores
3k_shapes_static	1.05 cores, render 91%	1.04 cores, render 89%
3k_shapes_dynamic	1.04 cores, render 90%	0.81 cores, render 88%
jetty	0.67 cores, render 79%	0.49 cores, render 75%
`Ogre2Scene::PreRender` share (3k)	52 to 56%	36 to 41%
`RenderUtil::Update` share (3k)	19% (dynamic)	39% static, 27% dynamic
Transport	ZeroMQ, <1%	Zenoh + SHM, 1 to 5% (26% under a 2.4 kHz publisher)
Qt main thread (3k)	6 to 7%	6 to 7%

The July study attributed the render thread to the scene graph walk alone; with the pose loop now separated, the static world's render thread is two comparable halves. The absolute totals for the 3k worlds are within noise of July for static and lower for dynamic and jetty.

11 Validating Optimizations

Use `gz_diff_flamegraph.sh` on the render thread folded files (`captures_gui/threads/<world>_thread_gz::gui::pl` for Priorities 1 and 2, the Qt main and `rx-*` thread files for 3 to 5, and the state message rate (`gz topic -e -t /world/<name>/state` for 5 s, count messages) as the direct metric for Priority 3.

12 References

1. B. Gregg, FlameGraph, https://github.com/brendangregg/FlameGraph
2. Gazebo profiling repository, https://github.com/caguero/gz-profiling
3. Gazebo GUI profiling report, July 2026, https://caguero.github.io/gz-profiling/2026-07-03/
4. Gazebo server profiling report, September 2026 (same run), https://caguero.github.io/gz-profiling/2026-09-01/
5. gz-sim PR #3447, ECM implementation with EnTT, https://github.com/gazebosim/gz-sim/pull/3447
6. gz-sim PR #3846, Benchmark for world with moving robots and sensors, https://github.com/gazebosim/gz-sim/pull/3846
7. gz-transport PR #868 and #842, Zenoh 1.8.0 and shared memory, https://github.com/gazebosim/gz-transport/pull/868, https://github.com/gazebosim/gz-transport/pull/842