Gazebo GUI Performance Profiling Report

profiled on gz-sim + PR #3447 (EnTT ECM)

Carlos Agüero

caguero@honurobotics.com

July 2026

Contents

1 Introduction

Gazebo runs as two cooperating processes: a headless *server* (`gz-sim-main`) that advances physics, sensors, and the Entity Component System (ECS), and a *GUI client* (`gz-sim-gui-client`) that renders the 3D scene with OgreNext, hosts the Qt/QML interface, and mirrors the server's world state into a client side ECS. A companion report profiled the server. This report profiles the **GUI process**, the part of Gazebo the user watches, whose responsiveness defines the perceived quality of the tool.

The GUI is profiled on **gz-sim with PR #3447 applied** ("ECM Implementation with EnTT"), which replaces the ECS storage backend with the EnTT library. This is the code baseline for the entire study: every capture below is of a GUI built on top of PR #3447. Choosing this baseline matters, because EnTT makes ECS iteration nearly free, which, as we will see, strips away the ECS sync cost and leaves the GUI's true bottleneck starkly exposed.

Prior GUI profiling relied on the built in Remotery instrumentation (`GZ_PROFILE`). Remotery does not instrument the OgreNext render pipeline, the gz-rendering scene abstraction, Qt/QML, or the GPU driver, which is where the GUI's cost lives. We therefore use **CPU flamegraph sampling** (Linux `perf`) attached to the GUI process, and, because the GUI is genuinely multithreaded, we split every capture **per thread**.

1.1 Goals

- Understand where CPU time goes inside the Gazebo GUI process (on the PR #3447 baseline), separating Gazebo owned code from Ogre/Qt/driver code.
- Attribute cost to the correct **thread** (render vs. Qt main vs. transport).
- Measure GUI startup/scene construction time.
- Identify concrete, actionable optimization opportunities in the GUI code path.
- Establish a reproducible GUI profiling methodology and benchmark suite.

1.2 Approach

We profile the GUI client process (`gz-sim-gui-client`, thread name `gz-sim-gui`) while a server feeds it world state over `/world/<name>/state`. `perf` attaches to the GUI process only. The server runs **playing** (`gz sim -r`) at maximum speed (`real_time_factor = 0`), so dynamic worlds actually move and the pose streaming / ECS sync path is exercised. We collect steady state **runtime** and GUI **loading** captures, then split every capture per thread. Four worlds isolate scene graph size, per frame pose streaming, and mesh complexity.

---

2 Methodology and Reproduction Procedure

2.1 Profiling Technique

CPU flamegraphs are generated from Linux `perf` sampling: `perf` interrupts the process at a fixed frequency and records each thread's call stack; samples are collapsed and rendered as an interactive SVG with Brendan Gregg's FlameGraph scripts [1]. The x axis is the proportion of CPU time (wider = more time), the y axis is stack depth (bottom = thread entry, top = leaf where time is spent). Flamegraphs capture *every* frame: OgreNext, Qt, the NVIDIA driver, and glibc.

Sampling event, `task-clock`. The test machine has a hybrid CPU (Intel performance + efficiency cores). With the default `cycles` event, `perf` creates two PMU events and `perf script` emits only one by default, silently dropping roughly half the samples. We therefore sample the `task-clock` software event (`perf record -e task-clock -F 997`), a single CPU agnostic wall clock on CPU timer that samples every thread on either core type uniformly. Stacks are unwound with `--call-graph dwarf` (the gz-sim, gz-gui, and gz-rendering libraries carry DWARF debug info; Qt and the driver do not, so a bounded fraction of samples land in `[unknown]` or unlabeled library frames).

Per thread flamegraphs. A merged flamegraph mixes the render thread, the Qt main thread, and idle transport threads. We split every capture into one flamegraph per thread (`stackcollapse-perf.pl --tid`) and report the per thread CPU distribution, the primary lens of this study.

2.2 GUI Process Architecture

When a user runs `gz sim <world>`, the launcher spawns the GUI as a separate child process, `gz-sim-gui-client`. Its work is split across:

- **Qt main thread** (`comm` = `gz-sim-gui`). Receives each server state message, deserializes it into the client ECS (`GuiRunner::OnStateQt` -> `EntityComponentManager::SetState`), then runs every GUI plugin's `Update()`. `GzSceneManager::Update` -> `RenderUtil::UpdateFromECM` extracts render data from the ECS each frame. **On this PR #3447 baseline, EnTT makes these ECS scans very cheap, so this thread is light.**
- **OgreNext render thread** (`comm` = `gz::gui::plugin...`). Runs the render loop: `RenderThread::RenderNext` -> `GzRenderer::Render` -> `Camera::Update` -> `Ogre2Scene::PreRender` (scene graph traversal) and the OgreNext draw/cull/ light passes. `RenderUtil::Update` (applying ECS changes to Ogre nodes) also runs here.
- **Qt Quick scene graph thread** (`QSGRenderThread`): composites the rendered texture into the window.
- **Transport threads** (`ZMQbg/IO`, discovery): deliver state messages.

The **render thread** and the **Qt main thread** are the only two worth optimizing; every recommendation targets one of them.

2.3 Benchmark World Selection

- **`shapes`**, three primitives. An idle GUI baseline: the fixed per frame render cost of an almost empty scene.
- **`3k_shapes_static`**, 3000 `<static>true</static>` models; nothing moves. Isolates per frame cost that scales with **scene size** from the cost of motion.
- **`3k_shapes` (dynamic)**, the same 3000 models, moving. Adds per frame pose streaming.
- **`jetty`**, a complex marine scene with detailed meshes and JPEG textures.

2.4 Hardware

- **CPU**: Intel Core Ultra 9 285HX, 24 cores (hybrid P+E, no HT), x86_64
- **GPU**: NVIDIA RTX PRO 3000 Blackwell Laptop GPU, driver 580.159.03, via PRIME render offload (`__NV_PRIME_RENDER_OFFLOAD=1 __GLX_VENDOR_LIBRARY_NAME=nvidia`)
- **Power profile**: `performance`
- **Kernel**: 6.17.0-35-generic

2.5 Software Versions

- **gz-sim / gz-gui / gz-rendering**: 11.0.0~pre1 + **PR #3447** (`luca/entt_playground` @ `bfbb174c6`, "ECM Implementation with EnTT")
- **OgreNext**: 2.3.1
- **Qt**: 6.4.2 (xcb platform plugin under XWayland)
- **perf**: 6.17.13
- **FlameGraph**: Brendan Gregg's scripts

gz-sim was built from source with:

```
-DCMAKE_BUILD_TYPE=RelWithDebInfo -DCMAKE_CXX_FLAGS=-fno-omit-frame-pointer -DENABLE_PROFILER=OFF
```

DWARF is present in the gz-sim / gz-gui / gz-rendering libraries. Rendering was verified with the GUI's `/gui/screenshot` service: both `shapes` (flat models) and `jetty` (nested models) render correctly on this PR #3447 build.

2.6 Setup and Running Captures

```
source ~/rotary_ws/install/setup.bash
export GZ_CONFIG_PATH=~/rotary_ws/install/share/gz:$GZ_CONFIG_PATH
export FLAMEGRAPH_DIR=~/rotary_ws/tools/FlameGraph
export __NV_PRIME_RENDER_OFFLOAD=1 __GLX_VENDOR_LIBRARY_NAME=nvidia
```

```
sudo sysctl kernel.perf_event_paranoid=1

# Runtime (attach perf to the settled GUI for 30 s; server playing via gz sim -r)
./scripts/gz_gui_flamegraph.sh worlds/3k_shapes.sdf 3k_shapes_dynamic 30 runtime
# Loading (wrap the GUI launch: Qt init + initial scene build)
./scripts/gz_gui_flamegraph.sh worlds/jetty.sdf jetty 20 loading
# GUI subsystem + hotspot breakdown; per thread split
./scripts/gz_gui_analyze.sh captures_gui/runtime/3k_shapes_dynamic.folded
./scripts/gz_per_thread_flamegraph.sh captures_gui/runtime/perf_3k_shapes_dynamic.data
↪  3k_shapes_dynamic
```

---

3 Executive Summary

This report profiles the Gazebo GUI process on the **PR #3447 (EnTT ECM)** baseline across four worlds (four runtime + two loading captures), every capture split per thread. The findings are consistent and point at a single subsystem.

1. **The GUI is almost entirely bound to the render thread.** The single OgreNext **render thread** consumes **61% of GUI CPU on an idle scene and 90-91% on the entity heavy worlds**. The Qt main thread is a distant second (6-14%); QML, transport, and the Qt Quick compositor are minor. On this PR #3447 baseline the Qt thread is especially light (6-7% on the 3k worlds) because EnTT makes the ECS scans nearly free, which throws the render thread bottleneck into sharp relief.
2. **The dominant cost is a per frame scene graph walk that scales with scene size, not motion.** `gz::rendering::Ogre2Scene::PreRender` recursively visits every visual and node every frame (`BaseScene::PreRender` -> `BaseVisual::PreRenderChildren` -> `BaseNode::PreRender`), costing ~**52-56% of all GUI CPU** at 3000 entities. The static 3000 entity scene (1.05 cores) and the dynamic one (1.04 cores) cost essentially the same total CPU, the walk runs whether or not anything moves.
3. **That walk churns shared pointers.** `Ogre2Node::Children` (returning a by value copy of the child container) is the top self time leaf (~12.6%), and ~**9.6% of all GUI CPU is pure `shared_ptr` atomic reference counting** (`__atomic_add` + `__exchange_and_add_dispatch`). It is also memory bound (Section 8).
4. **Motion adds per entity `dynamic_cast` on the render thread.** `RenderUtil::Update`'s pose loop does a `dynamic_pointer_cast<Visual>` and a `NodeById` lookup per moved entity (~19% inclusive in `3k_shapes_dynamic`).
5. **Complex meshes cost at load time.** In `jetty`, GUI startup is dominated by `SceneManager::LoadGeometry` (22%) + `CreateVisual` (19%), mesh and material construction. At runtime, jetty's render thread (79%) includes the NVIDIA GL submission work (`RenderQueue::renderGL3`, `libnvidia-glcore`) inline.

Because the EnTT ECM has already removed the ECS sync cost, **every remaining GUI optimization target is in the render thread (gz-rendering)**, see Section 9.

---

4 Runtime Analysis Per World

All runtime captures sample the settled GUI process for 30 s at 997 Hz (`task-clock`), server playing at `real_time_factor = 0`. "Avg cores busy" is the total sampled CPU time over the 30 s window (the GUI is vsync capped, so it rarely saturates a full core).

World	Avg cores busy	Render thread	Qt main	Dominant cost
`shapes`	0.16	61%	21%	Ogre culling + GPU submit (fixed floor)
`3k_shapes_static`	1.05	91%	6%	`Ogre2Scene::PreRender` scene graph walk
`3k_shapes_dynamic`	1.04	90%	7%	PreRender walk + `RenderUtil::Update` pose apply
`jetty`	0.67	79%	14%	render + NVIDIA GL submit; PreRender walk

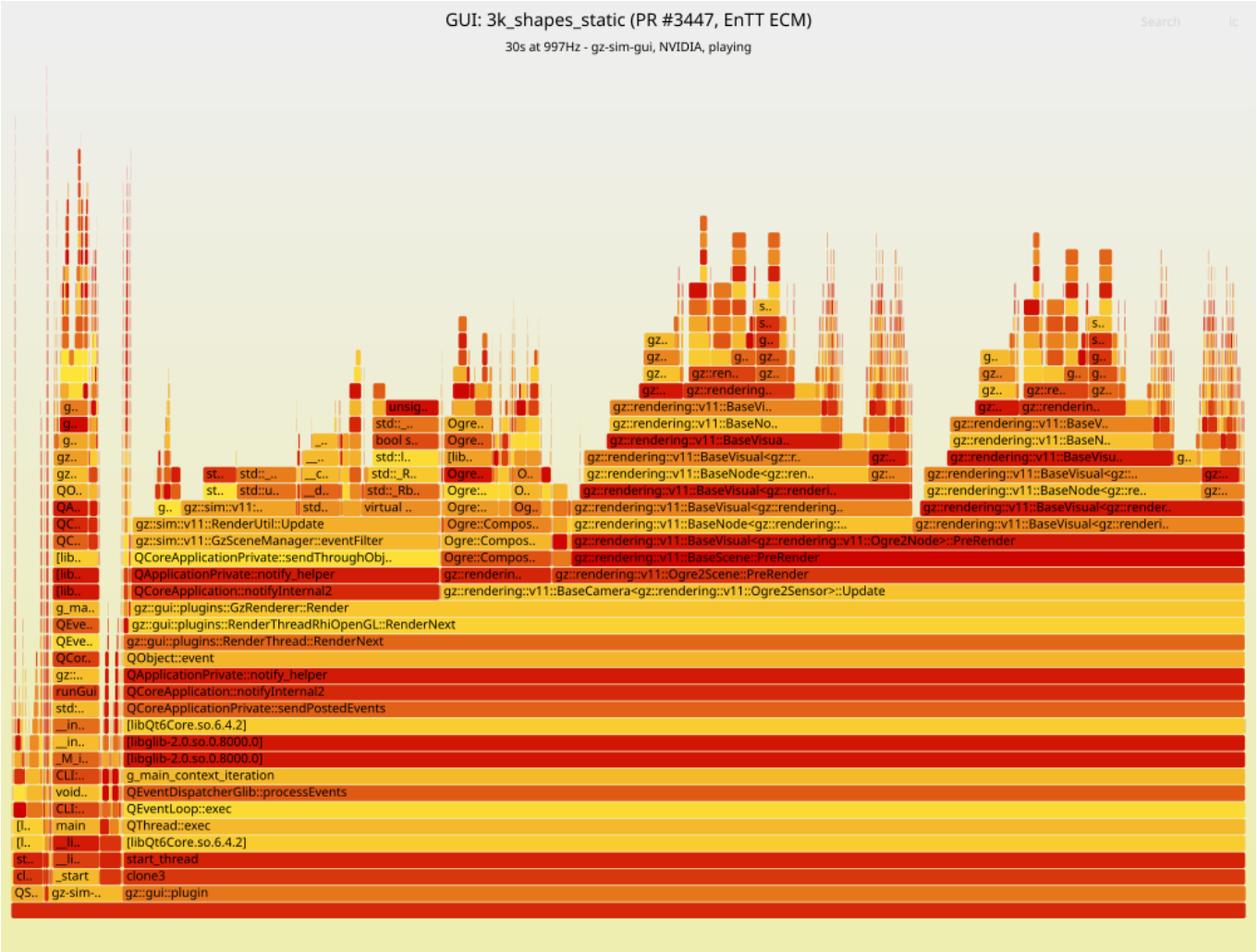

Figure 1: Merged flamegraph, `3k_shapes_static` (all threads, PR #3447). Almost the entire width is the render thread's `RenderThread::RenderNext` -> `Ogre2Scene::PreRender` walk; the Qt main thread (left) is a thin sliver.

4.1 shapes (idle GUI baseline)

Three primitives. The render thread is 61% of CPU; the leaf breakdown is Ogre scene graph transforms (18.4%, `Node::updateFromParentImpl` 14.6%), the NVIDIA driver (`libnvidia-glcore` 10.5%), Ogre culling/frustum (13.4%), and forward+ lighting (8.3%). The `PreRender` walk is negligible (few visuals). **Takeaway:** even a trivial scene pays a constant per frame bill for camera/frustum setup, forward+ light clustering, and GPU command submission, the floor every world builds on.

4.2 3k_shapes_static (3000 static entities)

3000 static models; nothing moves. The render thread jumps to **91% of CPU**, and its work is almost entirely the recursive scene graph traversal:

Frame (inclusive)	% of GUI CPU
`RenderThread::RenderNext` (render thread total)	90.9
`GzRenderer::Render`	90.4
`BaseCamera::Update`	65.3
`Ogre2Scene::PreRender`	56.1
`BaseScene::PreRender` -> `BaseVisual::PreRenderChildren`	52.1
`BaseNode::PreRender` (recursive)	48.7
`BaseVisual::PreRenderGeometries`	24.7
`RenderUtil::Update` (via `GzSceneManager::eventFilter`)	19.0

Every frame, OgreNext's scene manager calls `PreRender` on the whole scene, and gz-rendering's `BaseScene::PreRender` recurses through all 3000 visuals and their child nodes. The self time leaves are `Ogre2Node::Children` (**12.6%**, a by value copy of the child container), `BaseVisual::PreRenderChildren` (10.3%), and, strikingly, `__atomic_add` + `__exchange_and_add_dispatch` (`shared_ptr` refcounting) at a combined ~**9.6%**, plus `_Hashtable::find` (4.8%). By subsystem, gz-rendering's scene store/`PreRender` path is **46.9%** and `shared_ptr` alloc/refcount memory operations are **14.8%**. This world is memory bound (Section 8).

4.3 3k_shapes_dynamic (3000 moving entities)

The same 3000 models, moving. Total GUI CPU (1.04 cores) is essentially identical to the static case (1.05 cores), the per frame `PreRender` walk (~52%), not motion, is the base cost. Motion adds work on the **render thread**: `RenderUtil::Update` (~19% inclusive) applies the streamed pose changes to Ogre nodes, and its inner loop does a `dynamic_pointer_cast<Visual>` + `NodeById` lookup per moved entity. The render thread reaches **90%** of CPU; the Qt main thread, even with real state streaming, is only **7%**, because EnTT makes `SetState` deserialization and the `Each<>` scans cheap.

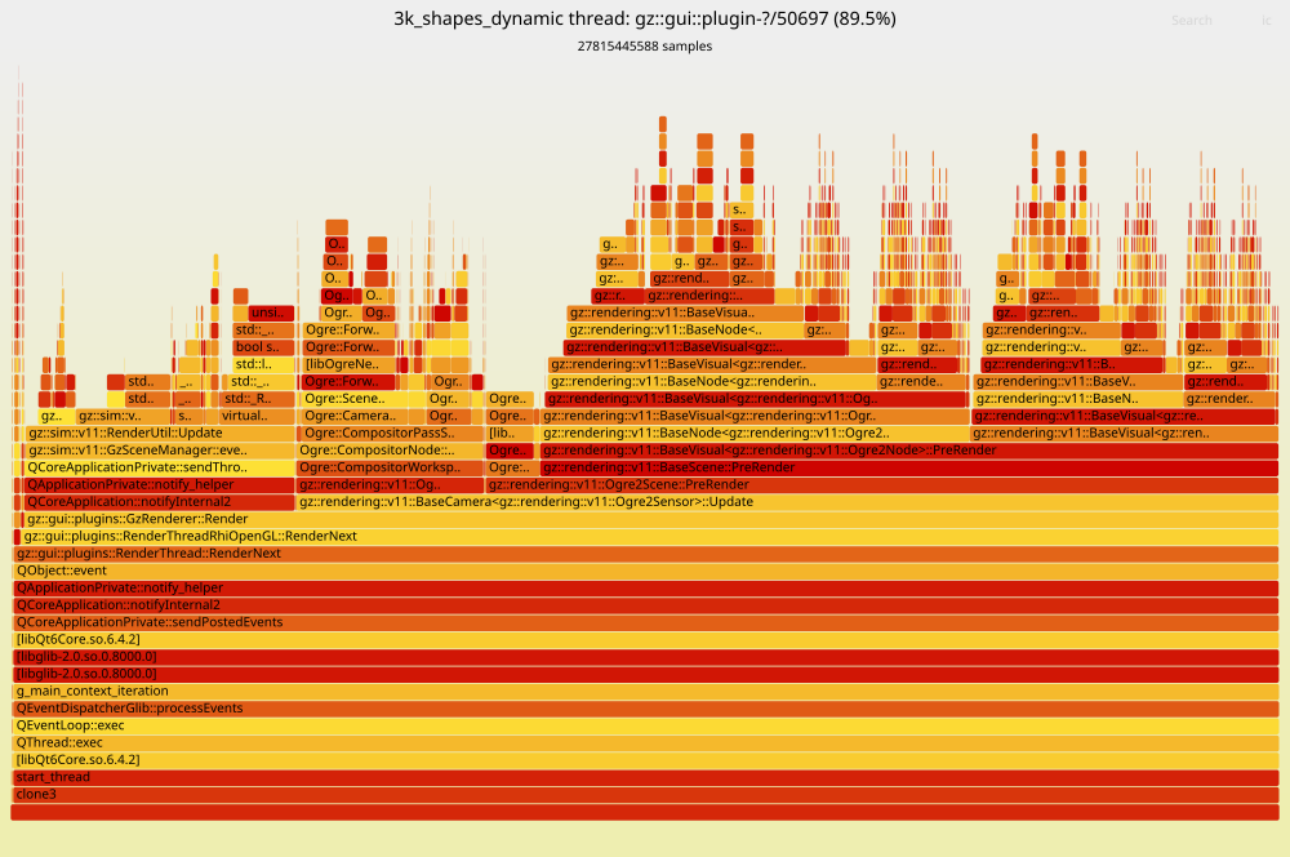

Figure 2: Render thread, `3k_shapes_dynamic` (PR #3447). `GzRenderer::Render` splits into a `RenderUtil::Update` pose apply tower (left) and the large recursive `Ogre2Scene::PreRender` -> `BaseVisual::PreRenderChildren` -> `BaseNode::PreRender` scene graph tower (right).

4.4 jetty (complex meshes and textures)

A detailed marine scene with real meshes and JPEG textured materials. The render thread is **79%** of CPU. On the NVIDIA GPU the driver's submission work is inline in the render thread (`Ogre::RenderQueue::renderGL3` 6.9%, `libnvidia-glcore` 6.6%, forward+ lighting `collectLightForSlice` 7.2%) rather than in a separate driver thread. With far fewer visuals than the 3k worlds, the `PreRender` walk is smaller (~13% subsystem). jetty's real Gazebo owned cost is at **load time** (Section 7): building the meshes and materials.

---

5 Per Thread Analysis

Splitting each capture by thread ID gives the true picture. The table is the share of total GUI process CPU per thread (aggregated over threads sharing a `comm`).

Thread (role)	shapes	3k_static	3k_dynamic	jetty
Render (OgreNext render loop)	61.3%	91.0%	89.6%	78.5%
Qt main (ECS sync + plugin update)	20.7%	5.9%	7.1%	13.9%
Qt Quick compositor	15.8%	2.6%	3.0%	5.6%
Transport / X events	<1%	<1%	<1%	1.6%

The render thread's `comm` is `gz::gui::plugin...`; the Qt main thread's is `gz-sim-gui-clie`. **On this PR #3447 baseline the Qt main thread is small on the entity heavy worlds (6-7%)**, EnTT's dense pool iteration removes the per frame `Each<>` scan and component lookup cost that would otherwise dominate it. The result is that the render thread accounts for essentially all GUI CPU (90-91% on the 3k worlds), and it is the sole optimization target.

Two code paths run on the render thread:

- `Ogre2Scene::PreRender`, the per frame scene graph walk. `BaseScene::PreRender` recurses over all visuals (`BaseVisual::PreRenderChildren`) and their nodes (`BaseNode::PreRender`), calling `Ogre2Node::Children` (a by value copy) at each level. Self time is dominated by that copy and by `shared_ptr` atomic refcounting. This is scene size work that runs every frame whether or not anything changed.
- `RenderUtil::Update`, applies streamed ECS changes to Ogre nodes. In dynamic worlds its pose loop does a `dynamic_pointer_cast<Visual>` + `NodeById` lookup per moved entity.

---

6 Cross Reference: Gazebo Owned Hotspots

The functions in Gazebo maintained code (gz-rendering, gz-sim) with the most inclusive CPU time, the optimization targets. Every one is on the render thread.

Function (file location in Section 9)	Peak %	Actionable fix
`BaseVisual::PreRender` / `PreRenderChildren`	52 (3k)	Skip unchanged visuals (dirty flag)
`Ogre2Node::Children`	12.6 (3k)	Return by `const&`, avoid copy
`shared_ptr` refcount	9.6 (3k)	Pass nodes by `const&` in the walk
`RenderUtil::Update` (pose loop)	19 (3k_dyn)	Cache node type; drop `dynamic_cast`
`SceneManager::NodeById`	n/a	Single typed lookup, not 5 maps
`SceneManager::LoadGeometry` + `CreateVisual`	41 (jetty load)	Async / cached mesh+texture load

External context (not directly optimizable): OgreNext scene graph/culling/forward+ lighting (`Node::updateFromParentImpl`, `Frustum::*`, `collectLightForSlice`), the NVIDIA driver (`libnvidia-glcore`, `RenderQueue::renderGL3`), and glibc.

---

7 Loading Analysis (GUI startup and scene construction)

GUI "loading" captures wrap the GUI process from launch, covering Qt/QML init, render engine init, and construction of the initial scene from the first full state snapshot. The cost depends on the kind of geometry the world contains.

World	Dominant loading cost	Evidence
`3k_shapes_static`	The per frame `PreRender` walk (render 78%)	`Ogre2Scene::PreRender` 41%, `BaseVisual::PreRenderChildren` 40%
`jetty`	**Mesh + material construction**	`SceneManager::LoadGeometry` 22%, `CreateVisual` 19%, `RenderUtil::Update` 16%

For **primitive only** worlds (`3k_shapes`), creating 3000 box/sphere visuals is cheap, so even the loading window is dominated by the same per frame scene graph walk as runtime. For **mesh heavy** worlds (`jetty`), GUI startup is dominated by `SceneManager::LoadGeometry -> loadMesh + scene->CreateMesh` and `SceneManager::CreateVisual -> LoadMaterial` (which decodes textures), on the render thread, blocking first paint.

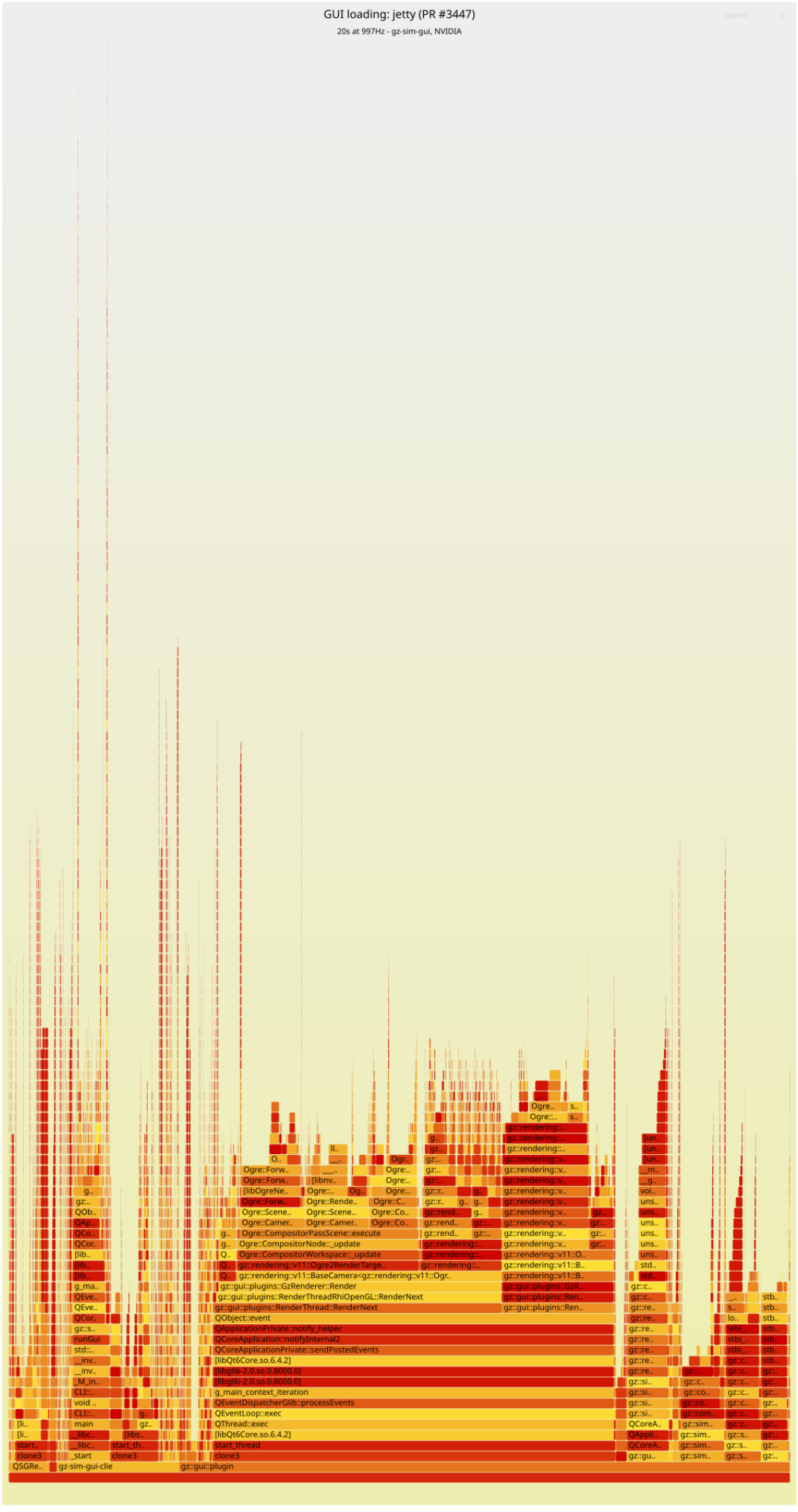

Figure 3: GUI loading flamegraph, `jetty` (PR #3447). `SceneManager::LoadGeometry` and `CreateVisual` (mesh + material construction) dominate.

8 CPU Cache Analysis

Hardware counters (`perf stat`, P-core PMU) measured Instructions Per Cycle (IPC) and cache miss rates for the GUI process on the PR #3447 build. IPC of 2-4 indicates compute bound code; IPC below 1 means the CPU is stalled most of the time waiting for RAM.

World	IPC	Cache miss	LLC miss	Assessment
`shapes`	2.38	12.7%	10.1%	Healthy, tiny working set
`3k_shapes_static`	**0.69**	**74.4%**	**70.7%**	**Severely memory bound**
`3k_shapes_dynamic`	1.09	43.4%	36.5%	Memory bound
`jetty`	1.57	31.1%	21.2%	Moderate

The static 3000 entity scene is **catastrophically cache hostile**: three quarters of cache accesses miss and 71% of last level cache accesses go to main memory, leaving the CPU stalled ~two thirds of the time (IPC 0.69). This is the signature of **pointer chasing through heap allocated `shared_ptr` node hierarchies**, the `PreRender` walk dereferences a different heap object at every node. The dynamic world is less memory bound (IPC 1.09) only because `RenderUtil::Update` adds compute bound pose apply work on top of the same walk.

This adds a second optimization lever beyond "do less work": **data layout**. Storing visuals/nodes in contiguous arrays indexed by entity (structure of arrays) instead of chasing `shared_ptr`s would improve cache behaviour even where the per frame walk cannot be fully eliminated.

9 Optimization Recommendations

Because the EnTT ECM (PR #3447) has already removed the Qt thread ECS sync cost, **all remaining GUI optimization targets are on the render thread, in gz-rendering.** They are ordered by measured leverage.

9.1 Priority 1, Stop traversing unchanged visuals on every frame

- **Evidence:** `BaseVisual::PreRender/PreRenderChildren` is ~52% of all GUI CPU in the 3k worlds, and runs identically whether the scene is static or dynamic.
- **Impact:** Up to ~50% of GUI CPU in large scenes; the single biggest win.
- **Location:** `gz-rendering` `BaseScene::PreRender` -> `BaseVisual::PreRender`.
- **Fix:** Add a dirty flag visuals/nodes so `PreRender` only descends into subtrees that changed since the last frame. A static subtree should cost O(1) to skip, not O(nodes) to traverse.

9.2 Priority 2, Eliminate `shared_ptr` churn in the scene graph walk

- **Evidence:** ~9.6% of all GUI CPU is `shared_ptr` atomic refcounting, and `Ogre2Node::Children` (a by value copy of the child `shared_ptr` map) is the top self time leaf (~12.6%).
- **Location:** `Ogre2Node::Children` and the `BaseNode/BaseVisual` traversal.
- **Fix:** Have `Children()` and the traversal hand out `const&` / iterators instead of returning owning pointer copies; avoid a temporary `shared_ptr` per node per frame. Combined with Priority 1 this removes most of the refcount traffic.

9.3 Priority 3, Drop the per entity `dynamic_cast` in the pose apply loop

- **Evidence:** `RenderUtil::Update`'s pose loop shows a `dynamic_pointer_cast<Visual>` and `NodeById` lookup per moved entity (~19% inclusive in `3k_shapes_dynamic`).

- **Location:** `gz-sim` `RenderUtil::Update` pose loop (`RenderUtil.cc:1350`), `SceneManager::NodeById` (`SceneManager.cc:1789`).
- **Fix:** Store the already typed `VisualPtr` alongside the node id so the loop skips `dynamic_pointer_cast<Visual>`; make `NodeById` a single typed lookup rather than probing five separate maps.

9.4 Priority 4, Async / cached mesh and texture loading

- **Evidence:** `jetty` GUI startup is `SceneManager::LoadGeometry` (22%) + `CreateVisual` (19%), including texture decode, on the render thread.
- **Location:** `gz-sim` `SceneManager::LoadGeometry` / `CreateVisual` / `LoadMaterial`.
- **Fix:** Cache decoded textures so a shared texture is decoded once; decode / upload meshes and textures on a background thread so the render thread does not block on first use.

9.5 Priority 5, Render on demand for the idle floor

- **Evidence:** Even `shapes` spends its render thread time on per frame frustum culling, forward+ light clustering, and GPU submit (~0.16 cores).
- **Fix (investigate):** Skip or throttle redrawing when neither the camera nor any visual changed, so an idle viewport approaches zero CPU. Interacts with Priority 1's dirty tracking.

---

10 Validating Optimizations

For each optimization:

1. **Baseline**, capture the target world (`gz_gui_flamegraph.sh`), split per thread to record the render thread's share.
2. **Optimize**, apply the change, rebuild.
3. **Measure again**, capture the same world identically.
4. **Compare**, `gz_diff_flamegraph.sh baseline.folded optimized.folded` (blue = improvement, red = regression); confirm the render thread shrank without shifting cost elsewhere.

The most informative validation world is `3k_shapes_static` (large scene, no physics noise) for scene graph/`PreRender` changes, and `jetty` for mesh/material changes.

---

11 References

[1] B. Gregg, "Flame Graphs," *ACM Queue*, vol. 14, no. 2, 2016. Software: https://github.com/brendangregg/FlameGraph

[2] B. Gregg, "Off-CPU Flame Graphs," 2015. https://www.brendangregg.com/FlameGraphs/offcpuflamegraphs.html

[3] PR #3447, "ECM Implementation with EnTT," gazebosim/gz-sim. https://github.com/gazebosim/gz-sim/pull/3447

[4] Gazebo Sim GUI architecture, `GuiRunner`, `GzSceneManager`, `RenderUtil`, and the `minimal_scene` render thread (gz-sim / gz-gui sources).