Case Study: Diagnosing and Fixing Crowd Performance in a Unity/WebGL Third-Person Game

Context: *Cat Distribution Network* — a third-person game where the player controls a cat navigating a small town populated by 40+ NavMesh-driven NPCs. Target platform: WebGL.

Starting point: With NPCs in the scene, frame rate was well below target. With NPCs removed entirely, the same scene ran comfortably above 60 FPS — confirming the crowd was the bottleneck, but not yet why.

This case study walks through the investigation as it actually happened: several distinct problems, stacked on top of each other, each one only visible once the previous layer was peeled back. That iterative process — not just the final fix — is the point of writing this up.

Starting the investigation

The existing `CharacterPerson` NPC controller already had performance-conscious design: coroutine-based patrol logic instead of per-frame `Update()` polling, pooled `WaitForSeconds` instances to avoid steady GC churn, `Animator.cullingMode = CullCompletely` to skip animation evaluation off-screen, and a custom distance-based AI LOD that reduced NavMeshAgent obstacle-avoidance quality and patrol frequency for distant characters.

None of that explained the frame rate. The first step was pulling up the Unity Profiler's CPU Usage and Rendering modules to see where time was actually going, rather than assuming the existing LOD system was the problem.

First read of the profiler: Scripts, Physics, Animation, and GC were all negligible — sub-millisecond, flat. But the Rendering counters (Batches, SetPass Calls, Triangles, Vertices) were oscillating in a wave that tracked frame time almost exactly, swinging between roughly 600 and 1,800. That was the real signal: the bottleneck was rendering submission cost, not scripts or physics, despite the crowd being NavMeshAgent-driven and script-heavy on paper.

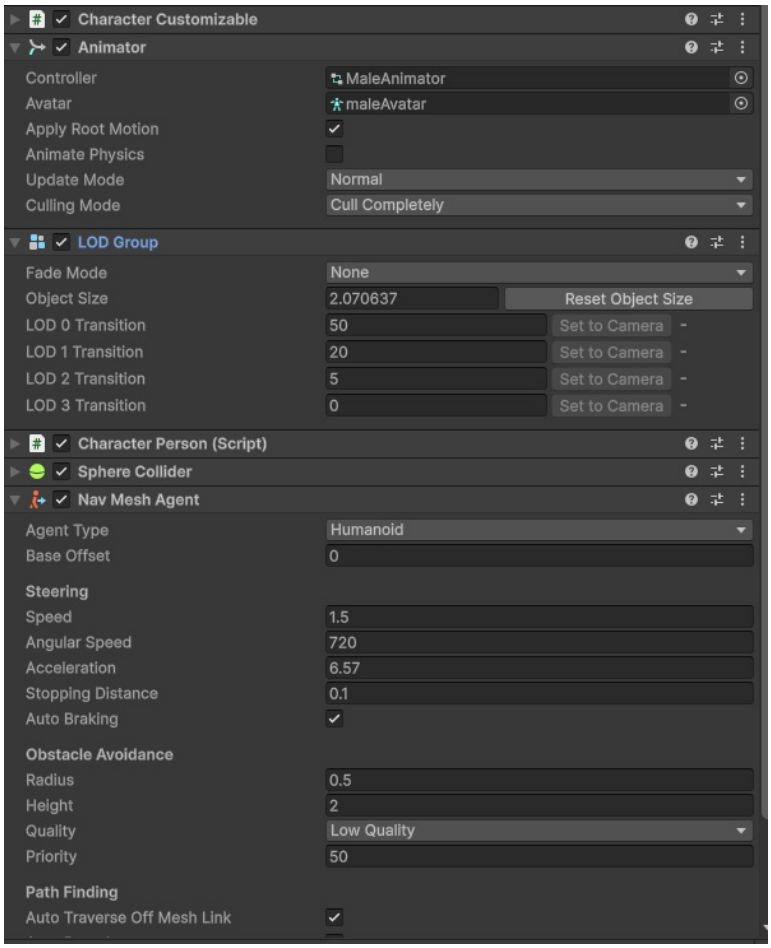
Initial profiler capture showing the Rendering counters oscillating in lockstep with frame time, while Scripts/Physics/Animation stay flat

Root cause #1: LOD culling wasn’t actually culling

Inspecting the `LODGroup` on the character prefab revealed the issue: LOD 3’s transition was set to **0%**, meaning there was no “Culled” region below it. In a normal LOD setup, once a character’s on-screen size drops below the last LOD threshold, Unity disables the renderer entirely — zero draw calls, zero cost. Here, because LOD 3 extended all the way to 0%, every character kept rendering its lowest-detail mesh forever, regardless of distance — even characters barely a few pixels tall near the camera’s far clip plane.

This meant LOD *transitions* were working (0→1→2→3 switched correctly at the right screen-size thresholds), but LOD *culling* — the actual cost-saving mechanism — never engaged.

Fix: raised LOD 3’s transition off 0%, giving it a real cutoff so distant characters stop rendering entirely instead of rendering forever at lowest detail.



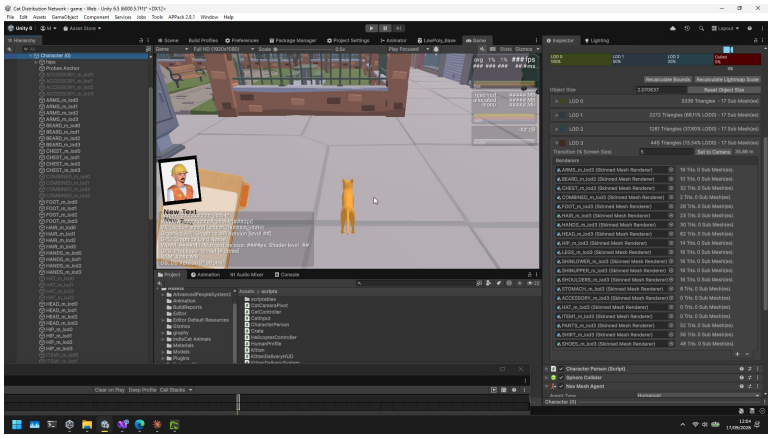
LODGroup inspector showing LOD 3’s transition set to 0%, meaning there is no culled region below the last LOD level

Root cause #2: Empty renderers paying full skinning cost

After the culling fix, frame time improved, but a new dominant cost appeared in the profiler: `MeshSkinning.CalcMatrices`, `MeshSkinning.GPUBatchedBlendShape`, and `UpdateRendererBoundingVolumes`. This pointed at mesh skinning overhead rather than draw-call submission.

Inspecting the character prefab explained why: each NPC was built from a **modular character system** — 17 separate `SkinnedMeshRenderer` components (arms, chest, hair, hands, head, hip, legs, shirt, pants, shoes, accessory, hat, item, etc.), all bound to a shared humanoid skeleton and swapped independently per LOD level. Several of these slots — accessory, hat, item — were empty (0 triangles) on characters with nothing equipped, but the renderer components were still enabled. Each one was still paying full per-frame cost: bone matrix computation, bounding volume updates, and a skinning dispatch — for geometry that didn't exist.

This cost was also structural, not just about wasted empty renderers: even at LOD 3 (445 total triangles), the character still carried **17 separate renderer instances**, each with its own per-renderer overhead, since Unity's skinning/bounds cost is driven largely by *renderer count*, not triangle count. LOD was reducing geometry correctly, but not reducing the number of things paying per-renderer tax.



Character hierarchy showing 17 independent SkinnedMeshRenderer components per character, including empty accessory/hat/item slots with 0 triangles still enabled

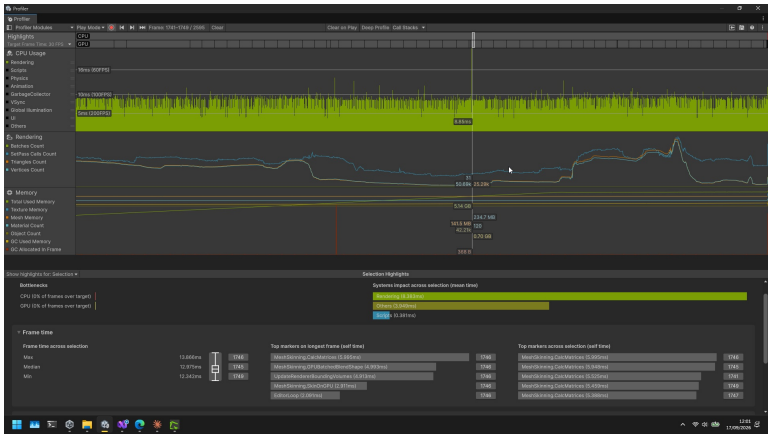
Fix (part 1): added a one-time pass in `CharacterPerson.Awake()` that disables any `SkinnedMeshRenderer` with an empty/zero-vertex mesh, so unequipped slots stop costing anything:

```
        void Awake()
        {
            _agent = GetComponent<NavMeshAgent>();
            _anim = GetComponent<Animator>();
            if (_anim == null) _anim = GetComponentInChildren<Animator>();

            if (_anim != null) _anim.cullingMode =
AnimatorCullingMode.CullCompletely;

            // Disable renderer slots with no geometry (unequipped
accessory/hat/item),
            // so they don't cost bone-matrix/bounds updates every frame.
            foreach (var smr in GetComponentsInChildren<SkinnedMeshRenderer>
())
            {
                if (smr.sharedMesh == null || smr.sharedMesh.vertexCount ==
0)
                    smr.enabled = false;
            }
```

```
}
```

Fix (part 2): merged the 17 modular skinned meshes into fewer combined renderers, cutting the fixed per-renderer overhead that every character was paying regardless of equipped items or LOD level.

After the LOD culling fix, the profiler's "Systems impact" breakdown shifted to show Rendering as the dominant cost, with `MeshSkinning.CalcMatrices`, `MeshSkinning.GPUBatchedBlendShape`, and `UpdateRendererBoundingVolumes` topping the self-time list — the signal that led to investigating the modular renderer structure above.



Profiler frame after the LOD culling fix, showing MeshSkinning.CalcMatrices and related skinning calls as the new dominant cost

A dead end worth including: chasing a GC hypothesis that wasn't the cause

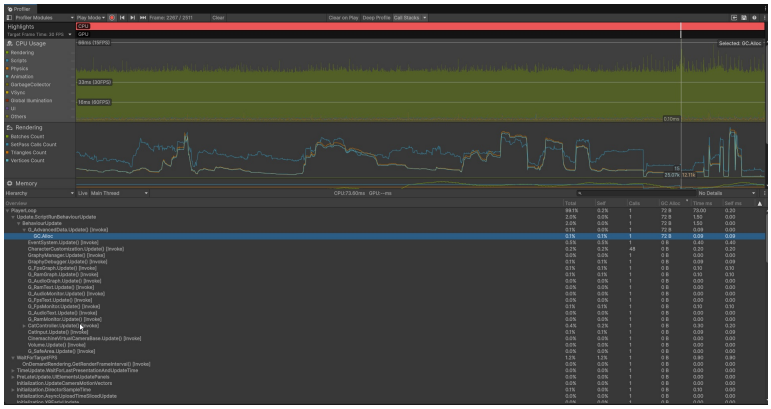
Later in the same play session, two clearly separated stretches of frames showed a different failure shape: sustained multi-frame CPU plateaus (30–66ms for many frames in a row) rather than isolated spikes, visually correlated with denser clusters of GC allocation activity in the profiler timeline. This looked, at a glance, like garbage collection pauses — a real risk on WebGL, where GC can't run incrementally the way it can on desktop.

Rather than assume, the investigation moved to confirming it directly:

1. Paused on a flagged frame and sorted the Hierarchy view by GC Alloc.
2. The first attempt showed the frame was still in **Editor Play Mode**, and the dominant cost was `EditorLoop` / `Profiler.WriteBuffer` / `UnityEngine.GUIUtility.BeginGUI()` — Editor-only tooling overhead, not gameplay cost. This is a known confound: profiling itself, and the Editor's own IMGUI repaint, can show up as false spikes that never occur in an actual build.
3. Switched to a **Development Build** (not Editor Play Mode) to eliminate that contamination, and re-profiled the same scenario.
4. In the clean build data, GC Alloc for the stalled frame was **72 bytes** — trivial, traced to a third-party FPS/debug overlay (Graphy), not gameplay code. This ruled out GC as the cause of this particular stall.

Digging further into the same frame's Hierarchy, `MeshSkinning.CalcMatrices` call count was essentially unchanged from baseline (1825 vs. 1827 calls) — ruling out a second hypothesis (LOD

thrashing from characters flickering between LOD levels near a transition boundary), since a thrashing pattern would have shown call count spiking, not staying flat.



Development Build profiler capture of the stalled frame: GC Alloc is 0B and MeshSkinning.CalcMatrices call count (1827) is nearly identical to the baseline (1825), ruling out both GC pressure and LOD thrashing

What *had* changed was cost-per-call: `MeshSkinning.Update` went from ~5.4ms to ~54ms total, with `MeshSkinning.Skin` alone accounting for ~38ms self-time, despite the same number of renderer calls. Same call count, ~10x the cost per call — that ruled out “more objects” and pointed at “more expensive objects.”

Cross-referencing with what was actually happening in-game (camera panning close to a small cluster of 4–5 characters) confirmed the actual mechanism: LOD is driven by on-screen size, so as the camera approached those characters, each one’s screen coverage grew and pushed it from LOD3 (445 triangles) toward LOD0 (3,336 triangles) — nearly an 8x geometry jump per character, times several characters simultaneously, all while call count stayed flat because renderer count per character was unchanged.

Fix: tightened the LOD culling threshold to 5%, combined with the earlier mesh-merge fix — reducing both the number of characters ever reaching expensive LOD tiers and the per-character renderer overhead when they did.

Additional optimization pass: rendering setup and build size

Alongside the crowd-specific investigation above, a broader pass covered general rendering and build-size hygiene:

- **Enabled batching for materials** to reduce draw call overhead further, complementing the renderer-count reductions from the LOD/mesh-merge work above.
- **Marked static, non-moving assets as Static** in the scene, allowing Unity’s static batching and lighting systems to handle them more cheaply than dynamic objects.
- **Reduced texture sizes** across the project as part of a deliberate build-size budget.
- **Enabled mesh compression** for further size optimization.

Result: build size came in at **26MB**, against a target ceiling of under 50MB — comfortably under budget, which matters directly for a WebGL target where initial load size affects time-to-play.

Result

Frame rate recovered from the original sub-target performance to a stable 60+ FPS in the same test scenario (crowd of 40+ NPCs, free-roaming third-person camera), alongside a build size of 26MB against a 50MB target.

What this investigation demonstrates

- **Profiler-first debugging, not assumption-first.** Every fix in this case study came from reading Profiler data (CPU Usage, Rendering counters, Hierarchy self-time, GC Alloc) rather than guessing at the cause from the symptom. Two plausible-looking hypotheses (LOD thrashing, GC pressure) were actively tested and ruled out with data before moving on, rather than "fixed" speculatively.
- **Recognizing tooling artifacts.** Distinguishing genuine gameplay cost from Editor-only overhead (`EditorLoop`, `Profiler.WriteBuffer`, IMGUI repaint cost) by re-testing in a Development Build — a distinction that matters a lot when profiling in-Editor can otherwise send you chasing phantom problems.
- **Distinguishing call count from cost-per-call.** The final root cause (LOD-driven geometry/skinning cost scaling as the camera approaches characters) was only visible by comparing `Calls` and `Self ms` side by side across frames — a flat call count with a 10x time increase is a specific, diagnostic signature, not something visible from a single frame in isolation.
- **Layered problems, layered fixes.** Four distinct issues (broken LOD culling, empty-but-enabled renderers, high per-renderer overhead from a modular mesh system, and per-vertex/per-renderer cost scaling at close range) were each masking the next until the one before it was resolved — reflecting how real crowd-rendering performance problems in Unity typically don't have a single root cause.
- **WebGL-specific awareness.** Kept in mind throughout that WebGL's single-threaded execution and non-incremental GC make it less forgiving of both allocation patterns and CPU-side rendering submission cost than desktop targets — informing both what was investigated and how seriously transient stalls were taken.
- **Build-size discipline as part of platform readiness.** Runtime performance and delivery footprint were treated as related problems, not separate concerns — batching, static flagging, texture size reduction, and mesh compression brought the WebGL build in at 26MB against a 50MB target, relevant to load time and platform constraints in a way that's directly applicable to console porting budgets.