

A PRACTICAL GUIDE FOR GODOT 4, 2D

Many Things on Screen, No Lag

Ten tips for games with hundreds of enemies, bullets and pickups. Each one explains what makes the game slow, why, and how to fix it, with before/after code you can lift into your own project.

All of it in plain GDScript, plus Godot's own shader language. No C#, no C++, no GDEXTension.



Why hordes get slow

Survivors-likes, bullet heavens and tower defences share one shape: a **few kinds** of things, each repeated **hundreds of times**, so every small per-object cost is multiplied by the size of the horde.

ONE FRAME AT 60 FPS = 16.7 MS, SHARED BY EVERYTHING



Every block grows with the number of objects. Run past the deadline and the frame is late: a stutter.

Shrink any block and the time freed is **headroom**: more enemies, more physics steps, bigger effects.

With a 60 fps cap you won't see a higher fps number. You'll see the stutters stop.

TIP	IN ONE SENTENCE
1. Measure in real play	Log timings while people actually play; fix the biggest measured cost.
2. Pool, don't instantiate	Make it all while loading, then reuse it; nothing is created mid-fight.
3. One owner per kind	A manager updates a list of objects; objects don't each get an engine callback.
4. Fake most lights	Use glowing sprites instead of real lights on bullets and pickups.
5. Collisions as maths	"Who is near this point or line?" is a grid lookup, not a physics job.
6. Share textures	Neighbours batch only if they share a texture and material: use atlases.
7. Draw copies as one	Hundreds of identical small things become one MultiMesh, with no nodes.
8. Only redraw what changed	Redraw only what moved, and render at your art's size.
9. Let the GPU move things	Send start, velocity and time; the shader moves it, no script.
10. Load early, and don't change the gameplay	Warm up shaders and pools while loading; keep the gameplay the same.

880 lights

on pickups once cost about 5 ms a frame
on their own

241 → 2

draw calls for 240 walking monsters, once
their frames shared one atlas

94 %

of enemy cost was checking enemies
against the player

ALL OF THIS IS GDSCRIPT

Every technique here is written in **plain GDScript**, with Godot's shader language for the GPU parts. There's no C#, no C++ module and no GDExtension. That was partly a constraint (we ship a web build, and C# web export wasn't an option for us). In our logs the slow part was rarely the GDScript itself. It was asking the engine to do one small thing per object, hundreds of times a frame. Changing the shape of that work is what moved our numbers.

WORDS WE USE

Draw call: one "draw these" instruction from the CPU to the GPU.

Tick: one physics step (60 a second by default).

Uniform: a value your game hands to a shader.

Canvas item: anything Godot 2D draws: a Sprite2D, a `_draw()`.

MultiMesh: one mesh drawn many times in one call.

Batch: neighbouring sprites merged into one draw call.

Headless: Godot running with no window, so nothing is drawn.

Culling: skipping anything outside the camera's view.

Interpolation: drawing an object between its last two ticks.

Atlas: many images packed into one sheet.

Measure in real play

On night 11 our frame rate fell from 48 to 32 fps as the horde passed about 120 enemies. The log showed why: past that point every frame ran two physics ticks instead of one, so all per-enemy work doubled. We would not have guessed that.

DO THIS

- **Build a tiny in-game logger.** Once per second, write a CSV row: fps, worst frame, physics ms, draw calls, node count, how many enemies, bullets and pickups are alive, and a timer around each big system.
- **Let people play normally,** then read the file. Lag shows up where it really happens: wave 11, 200 enemies, one particular spell.
- **Build a benchmark scene too:** a scripted fight you can re-run the same way every time. Make it answer one question, with switches to turn hits, trails or visuals off one at a time.

WATCH OUT

- **A busy machine lies.** We saw one benchmark swing 3× (9.4 vs 3.0 ms) from background load alone. Run A, then B, then A again.
- **Headless runs measure scripts only.** Draw calls, lights and batching need a real window.
- **Averages hide stutters.** Always log the worst frame too.
- **F5 is slower than your game.** The editor's debugger made GDScript 2–3× slower: 31–35 ms a frame against 11–19 ms.
- **Test the build players get:** a release export, with an opt-in log flag (`-- --perflog`).

PERF_LOG.GD: A PER-SECOND CSV ROW

GDSCRIPT

```
extends Node
var _file: FileAccess
var _acc := 0.0
var _worst := 0.0

func _ready() -> void:
    var path := "user://perf_%d.csv" % Time.get_unix_time_from_system()
    _file = FileAccess.open(path, FileAccess.WRITE)
    _file.store_line("t,fps,worst_ms,physics_ms,draw_calls,nodes,enemies,bullets")

func _process(delta: float) -> void:
    _worst = maxf(_worst, delta * 1000.0)
    _acc += delta
    if _acc < 1.0: return
    _file.store_line("%d,%d,%.1f,%.2f,%d,%d,%d,%d" % [
        Time.get_ticks_msec() / 1000,
        Performance.get_monitor(Performance.TIME_FPS), _worst,
        Performance.get_monitor(Performance.TIME_PHYSICS_PROCESS) * 1000.0,
        Performance.get_monitor(Performance.RENDER_TOTAL_DRAW_CALLS_IN_FRAME),
        Performance.get_monitor(Performance.OBJECT_NODE_COUNT),
        get_tree().get_node_count_in_group(&"enemies"),
        get_tree().get_node_count_in_group(&"bullets")])
    _acc = 0.0; _worst = 0.0
```

TIMING ONE SYSTEM

GDSCRIPT

```
var t0 := Time.get_ticks_usec()
_update_all_enemies(delta)
enemy_ms_total += (Time.get_ticks_usec() - t0) / 1000.0 # average it into the next CSV row
```

Fix the biggest measured number first. Then measure again.

2

TIP TWO

Pool, don't instantiate

Creating scenes mid-fight is one of the most common causes of hitches. Make what you'll need before the fight, then reuse it.

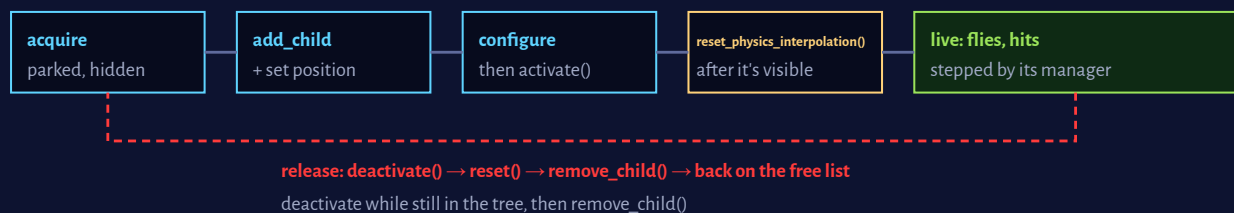
THE PROBLEM

`instantiate()` builds the node tree, runs `_ready()` and registers with physics and rendering; `queue_free()` tears it down again. Fifty of those in one frame (a shotgun spell, a wave dying) is a visible spike.

THE FIX

Keep a **free list per scene**, filled while loading ("prewarm"). In the fight, **acquire** a parked node and **release** it when done. Nothing is created or freed mid-fight, and a counter proves it.

ONE LIFE OF A POOLED BULLET, IN ORDER (THE ORDER MATTERS)



POOL.GD: ONE FREE LIST PER SCENE, PREWARMED, MISSES COUNTED

GDSCRIPT

```

class_name Pool extends RefCounted
var _free := {} # scene path -> Array of parked nodes (out of the tree)
var instantiations := 0 # after prewarm this must NOT move during a fight: test it

func prewarm(scene: PackedScene, count: int) -> void: # behind the loading screen
    for i in count:
        instantiations += 1
        _free.get_or_add(scene.resource_path, []).append(scene.instantiate())
func acquire(scene: PackedScene) -> Node:
    var list: Array = _free.get_or_add(scene.resource_path, [])
    if list.is_empty(): # a miss: counted, logged, never a crash
        instantiations += 1
        return scene.instantiate()
    return list.pop_back()
func release(node: Node) -> void:
    node.deactivate() # YOUR method on the bullet, not built-in: hide, collision off, still in the tree
    node.reset() # YOUR method too: clear this life's hit list, timers, tint...
    node.get_parent().remove_child(node)
    _free[node.scene_file_path].append(node) # the scene it was made from

```

RULES WE LEARNED THE HARD WAY

- **Author the scene inert** (hidden, collision off) so fresh and recycled nodes start identical; `activate()` only after parenting.
- **Write every visual setting on every reuse**, defaults included: a recycled fireball otherwise keeps its glow.
- **Reset physics interpolation after showing the node**: Godot ignores it on a hidden node, and the bullet slides in from where it last died.
- **Size the prewarm from the design** (projectiles per cast × casts per second × seconds each one lives). Log a miss instead of capping the spell, and free parked nodes on exit.

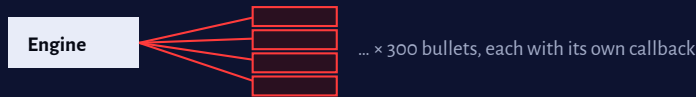
3

TIP THREE

One owner per kind of thing

Instead of giving every bullet its own `_process`, let one manager node update them all in one loop.

SLOW: 300 CALLBACKS A TICK, ONE PER BULLET



FAST: ONE CALLBACK, THEN A PLAIN LOOP



BULLET.GD (BEFORE)

GDSCRIPT

```
func _physics_process(delta):
    position += velocity * delta
    life -= delta
    if life <= 0.0:
        pool.release(self)
# x 300 bullets = 300 callbacks a tick
```

BULLET.GD (AFTER)

GDSCRIPT

```
var slot := -1 # index in the manager's list

# no _physics_process any more
func step(delta) -> void:
    position += velocity * delta
    life -= delta
    if life <= 0.0:
        manager.release(self)
```

BULLET_MANAGER.GD: SWAP-REMOVE LIST, WALKED BACKWARDS

GDSCRIPT

```
var live: Array[Bullet] = []

func add(b: Bullet) -> void:
    b.set_physics_process(false)
    b.slot = live.size()
    live.append(b)

func release(b: Bullet) -> void:
    var last := live[live.size() - 1]
    live[b.slot] = last # the last one fills the hole: O(1), no shifting
    last.slot = b.slot
    live.pop_back()
    b.slot = -1
    pool.release(b)

func _physics_process(delta: float) -> void:
    if delta <= 0.0: return # paused (time_scale 0): costs nothing
    var i := live.size() - 1
    # backwards: a bullet that dies is replaced by one already updated, never skipped or doubled
    while i >= 0:
        if i < live.size(): # a step can release more than one bullet, so the list may have shrunk
            live[i].step(delta)
        i -= 1
```

- **Share drawing the same way.** If 200 enemies each show a status icon, one drawer can loop over them in a single `_draw()` instead of 200 canvas items.
- **No allocation per tick.** Reuse packed arrays; build no dictionaries or lambdas in hot loops.
- **A pause should cost nothing.** If you pause with `time_scale = 0` (we do, for menus and level-up choices), managers should return at once rather than move 400 objects by zero distance. `get_tree().paused` already skips pausable nodes for you. **Check every kind of object:** our bullets and enemies stopped, but area spells and beams kept sweeping at zero speed and cost 6–14 ms a frame on the level-up screen.

4

TIP FOUR

Fake most lights

We once measured 880 pickup lights costing about 5 ms on their own. A glowing bullet looks great; a real light on every one is a very expensive way to get it.

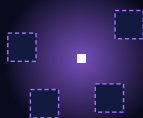
THE PROBLEM

In Godot's 2D renderer a light adds drawing work to **every sprite it touches**, and each sprite has a cap on how many lights can reach it. Fifty lit bullets flying through a crowd multiply the cost of the crowd.

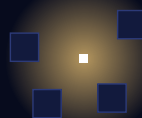
THE FIX

We used real lights for two jobs. **For the floor glow**, use a soft round sprite with additive blending; every glow shares one texture and material, so they batch. **To keep sprites readable** when night darkens the screen (a `CanvasModulate`), have their shader multiply their colour by the opposite amount: if night is 50 % dark, they brighten $\times 2$, so they end up looking normal.

A REAL LIGHT REDRAWs ALL IT TOUCHES



A GLOW SPRITE IS ONE MORE QUAD



Keep real lights for the few sources that matter (the player, a boss, a campfire). Fake the rest.

BULLET.TSCN (BEFORE)

SCENE

```
Bullet (Area2D)
├ Sprite2D
├ PointLight2D # floor light
└ PointLight2D # night readability
# 2 lights × 50 bullets = 100 lights
```

BULLET.TSCN (AFTER)

SCENE

```
Bullet (Area2D)
├ Sprite2D # lifted.gdshader
└ Glow # soft circle, floor_glow
# 0 lights; all glows share 1 material
```

FLOOR_GLOW.GDSHADER

SHADER

```
shader_type canvas_item;
render_mode blend_add, unshaded;
uniform float gain = 0.05; // glow brightness
void fragment() {
    // COLOR already = modulate × texel
    COLOR = vec4(COLOR.rgb * COLOR.a * gain, 1.0);
}
```

LIFTED.GDSHADER

SHADER

```
shader_type canvas_item;
// declared in Project Settings → Shader Globals
global uniform vec4 night_lift;
void fragment() {
    COLOR.rgb *= night_lift.rgb;
}
```

THE GAME SIDE, ONCE PER FRAME WHEREVER NIGHT IS DRIVEN

GDSCRIPT

```
# ambient = your night CanvasModulate colour; the 0.4 caps the boost at 2.5×
var k := 1.0 / maxf(ambient.r, 0.4)
var lift := Vector4(k, k, k, 1.0)
RenderingServer.global_shader_parameter_set(&"night_lift", lift)
```

GOTCHAS

- In a Godot 4 canvas shader `COLOR` already holds the texel; sampling `TEXTURE` again darkens the sprite.
- `render_mode unshaded` ignores lights, but **not** a `CanvasModulate` tint.
- Shader colour can go above 1.0, so a lifted sprite survives the night's darkening.

5

TIP FIVE

Collisions as maths

If all you need to know is "which enemies are inside this circle" or "what did this bullet pass through", a simple grid answers it far more cheaply than a physics engine.

THE PROBLEM

Every enemy and every bullet is an `Area2D`. The physics engine pairs them, sorts them and sends a signal per contact. In our game, enemies overlapping the player were **54 %** of all enemy cost, and 300 bullets through 240 enemies cost 7.9 ms with hits on against 4.9 ms with hits off.

THE FIX

Once per tick, drop each enemy into a **grid of 64 px cells**. To check a shape, visit only the cells it covers and do an exact distance test on those few. Enemies leave the physics engine entirely; their hit logic doesn't change.



A BULLET'S MOVE = A CAPSULE

From where it was to where it is now, so a fast bullet can never skip over an enemy between ticks.

ONLY SHADED CELLS ARE VISITED

10 enemies, 4 in the shaded cells, 3 touched. The other 6 are never looked at.

Cost grows with what's nearby, not with the size of the horde.

ENEMY_GRID.GD: BUILD ONCE PER TICK, QUERY ANYWHERE

GDSCRIPT

```
const CELL := 64.0
const MAX_R := 24.0 # biggest NORMAL enemy (bosses: see the gotcha below)
var cells := {} # Vector2i -> PackedInt32Array of enemy indices

# simplified for reading; in a hot loop, reuse the arrays instead of making new ones each tick
func build(pos: PackedVector2Array) -> void:
    cells.clear()
    for i in pos.size():
        var key := Vector2i(floori(pos[i].x / CELL), floori(pos[i].y / CELL))
        if not cells.has(key): cells[key] = PackedInt32Array()
        cells[key].append(i)

# every enemy touching the segment a->b swept with radius r (a moving bullet)
func query_capsule(a, b, r: float, pos, radius, out: Array) -> void:
    var box := Rect2(a, Vector2.ZERO).expand(b).grow(r + MAX_R)
    var lo := Vector2i((box.position / CELL).floor())
    var hi := Vector2i((box.end / CELL).floor())
    for cy in range(lo.y, hi.y + 1):
        for cx in range(lo.x, hi.x + 1):
            for i in cells.get(Vector2i(cx, cy), PackedInt32Array()):
                var c := Geometry2D.get_closest_point_to_segment(pos[i], a, b)
                if c.distance_to(pos[i]) <= r + radius[i]:
                    out.append(i)
```

When a boss shows up

Swapping physics for maths must not change how the game plays, and a grid has one trap that only shows up in boss fights.

IN THE BULLET: "HIT ONCE ON ENTRY" SURVIVES, AND THE OLD HIT HANDLER IS REUSED

GDSCRIPT

```
_now.clear()
grid.query_capsule(prev_position, position, radius, enemy_pos, enemy_radius, _now)
# remember enemies by a STABLE id, never by list index: with swap-remove, an index can
# point at a different enemy next tick (a missed or a double hit)
_now_ids.clear()
for i in _now:
    var id := enemy(i).get_instance_id()
    _now_ids[id] = true
    if not _inside_ids.has(id): on_hit(enemy(i)) # only NEW arrivals hit; same handler as before
_inside_ids = _now_ids.duplicate()
```

Gotcha we hit: one huge monster slows every query

The trap. A big monster centred in a cell **outside** your search can still overlap your shape, so every query has to look past its own reach by the radius of the **biggest** monster. The natural way is one global maximum, rebuilt each tick. Then a single boss joins the fight, and every bullet's query, even one on the far side of the screen, starts scanning extra cells.

How we found it. Our horde fights ran smoothly while the final boss stuttered. A boss has a ~45 px radius against 11–24 px for everyone else, and there were 600–700 bullets in flight against it.

The fix, measured: 600 bullets, 60 enemies and one boss went from 17.4–18.3 ms to 15.5–15.7 ms of physics per frame (about 2 ms, ~13 %). Real, but not the whole story: the rest of that fight's cost was hit processing.

ONE GLOBAL MAX RADIUS



Every bullet's search grows by the boss's size.

BIG MONSTERS IN THEIR OWN LIST



Normal-sized search, plus one direct check per boss.

THE FIX, INSIDE BUILD()

GDSCRIPT

```
const BIG_RADIUS := 32.0 # ~half a cell
for i in count:
    if radius[i] > BIG_RADIUS:
        big.append(i) # bosses, giant elites
        continue
    bucket(i)
    max_r = maxf(max_r, radius[i])

# every query: its cells, then the few big ones
for i in big:
    if rect.grow(radius[i]).has_point(pos[i]):
        out.append(i)
```

BEFORE YOU SHIP THE SWAP

- **Test with your biggest monster.** 240 small enemies never show this.
- Call your **existing hit functions**; only the "who did I touch" source changes.
- Never add or remove physics objects **inside a physics callback**; queue them.
- Cap catch-up steps (Physics → Common → Max Physics Steps per Frame; we use 2) so one slow frame can't snowball.

6

TIP SIX

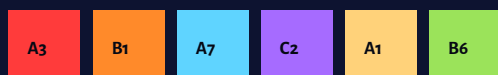
Share textures

Usually the cheapest change in 2D: no game code changes, only how the art is packed.

HOW DRAWING REALLY WORKS

Each frame the CPU sends the graphics card a list of instructions: "bind this texture, use this shader, draw these rectangles". Each new **batch** has a setup cost, like a painter switching brush and paint pot. Godot merges neighbouring sprites into one batch automatically, but **only while the texture and the material stay the same**.

ONE IMAGE PER ANIMATION FRAME



6 enemies on different frames → **6 batches**

Real example: 12 looks × 72 frames (8 standing directions + 8 directions × 8 walk steps) + 12 flats = 876 images. Draw calls rose about one per enemy (197 → 305 from 0 to 100 enemies), because y-sorted neighbours almost never matched.

ONE SHARED SHEET (ATLAS)



every sprite shows a rectangle of one sheet

→ **1 batch**, however many enemies

THORNLING_WALK.TRES (BEFORE)

RESOURCE

```
frames = [
    load("res://enemy/walk/0/east.png"),
    load("res://enemy/walk/0/south_east.png"),
    ... # 64 images: 8 directions × 8 steps
]
```

EACH FRAME BECOMES A REGION

GDSCRIPT

```
var f := AtlasTexture.new()
f.atlas = preload("res://art/enemy_atlas.png")
f.region = Rect2(687, 763, 32, 32)
sprite.texture = f # game code unchanged
```

PACK_ATLAS.PY: AN OPTIONAL BUILD STEP (ANY SPRITE-SHEET TOOL WORKS)

PYTHON + PILLOW

```
CANVAS, PAD = 32, 3 # transparent padding around every frame
CELL, COLS = CANVAS + 2 * PAD, 32
sheet = Image.new("RGBA", (COLS * CELL, rows * CELL), (0, 0, 0, 0))
for i, png in enumerate(sources):
    x, y = (i % COLS) * CELL + PAD, (i // COLS) * CELL + PAD
    sheet.paste(Image.open(png), (x, y))
    regions[png] = (x, y, CANVAS, CANVAS) # then rewrite the .tres files to AtlasTexture regions
sheet.save("art/enemy_atlas.png")
```

Measured: 240 walking monsters, 241 draw calls → 2. Frame time stayed ~2.2 ms on our fast GPU: it's headroom, not a visible fps jump.

- **Pad every frame** so smoothing or outline shaders never pick up pixels from the neighbouring frame.
- **Check shaders that assume "UV 0..1 is my whole sprite"**. On an atlas, UV is a small rectangle of the sheet; an outline shader must find its cell first.
- **Make the sheet a build step**, with a test that fails, naming the command to run, when someone re-exports art and forgets it.
- **Materials split batches too**. A special material on 3 of 200 enemies costs 3 extra batches; a unique material per enemy costs 200.

7

TIP SEVEN

Draw copies as one

Got hundreds of the same small thing (pickups, sparks, trail segments) that don't need to sort in front of or behind other objects? Stop giving each one a node.

3 NODES FOR EVERY PICKUP

Pickup (logic)

Sprite

Floor glow

× 400

= 1,200 nodes; 800 drawn items walked, culled, sorted and positioned every frame

ONE MULTIMESH FOR ALL OF THEM

one quad + one buffer

```
#0 transform colour custom(a,b,c,d)
#1 transform colour custom(a,b,c,d)
... × 400 → ONE draw call
```

Logic stays a light node or plain data; drawing becomes a single item.

A MULTIMESH STRAIGHT ON THE RENDERINGSERVER (NO NODES)

GDSCRIPT

```
var mm := MultiMesh.new()
mm.transform_format = MultiMesh.TRANSFORM_2D
mm.use_colors = true
mm.use_custom_data = true           # 4 free floats per instance for the shader
mm.mesh = quad_mesh_built_from_2d_vertices # see gotchas
mm.instance_count = 400

var ci := RenderingServer.canvas_item_create()
RenderingServer.canvas_item_set_parent(ci, get_canvas_item())
RenderingServer.canvas_item_set_material(ci, pickup_material.get_rid())
RenderingServer.canvas_item_add_multimesh(ci, mm.get_rid(), pickup_texture.get_rid())
RenderingServer.canvas_item_set_custom_rect(ci, true, Rect2(-1e7, -1e7, 2e7, 2e7))
RenderingServer.canvas_item_set_interpolated(ci, false)

# write a slot only when that pickup CHANGES (lands, gets collected), never every frame
mm.set_instance_transform_2d(slot, Transform2D(0.0, landed_at))
mm.set_instance_custom_data(slot, Color(land_time, bob_phase, 1.0, 0.0))
```

THE IDLE BOB, ON THE GPU

SHADER

```
uniform float now; // game clock, set on physics ticks
void vertex() {
    float age = now - INSTANCE_CUSTOM.x;
    float bob = round(sin(TAU * (1.2 * age
        + INSTANCE_CUSTOM.y)) * 2.0);
    VERTEX.y -= bob * INSTANCE_CUSTOM.z;
} // resting pickups cost zero script
```

TRAILS, THE SAME WAY

A **Line2D** per bullet is a draw call per bullet. Instead, keep a **ring of segment slots** in one MultiMesh: each tick a bullet writes one segment stamped with the time, and the shader tapers and fades segments by age. **Whole frame in our 300-arrow benchmark: 13.7 → 5 ms, and 316 → 28 draw calls.**

GOTCHA

CAUSE → FIX

Everything vanishes off screen

The visibility box was measured while every instance sat at (0,0) → **custom world-sized cull rect** (above).

Streaks on reused slots

Interpolation blended a slot from its previous owner → **interpolation off**, blend by hand if needed.

Upside-down sprites

A 3D **QuadMesh** has y-up UVs → **build the quad from 2D vertices** with explicit UVs.

Tests read nonsense

Headless Godot stores no MultiMesh data → **keep a CPU copy** of what you wrote and test that.

Only redraw what changed

Godot keeps a canvas item's draw commands until you ask it to redraw. A `_draw()` that runs every frame throws that away and pays for the whole picture again, even when almost nothing moved.

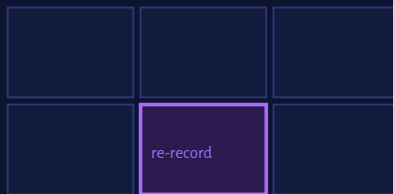
THE PROBLEM

Our skill-tree screen is a big web of hexes, threads, level notches, icons, prices and names. A few nodes hold an animated liquid, so the whole web called `queue_redraw()` every frame. Fully upgraded, that cost **49.9 ms a frame** (36 ms of it GDScript re-issuing the same commands), and **116 ms** at 4× the nodes, for a screen that barely moves.

THE FIX

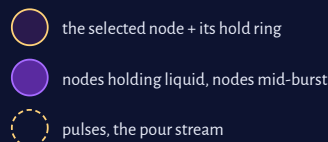
Cut the still part into **tiles**, one canvas item each, and re-record a tile only when something *in it* changes. Pan and zoom just move or scale the tiles. A thin **live layer** redraws every frame, but only what really animates: the selection, the liquid, a burst, a pulse.

TILES: RECORDED ONCE



a purchase dirties ONE tile; pan/zoom dirties none

LIVE LAYER: EVERY FRAME, BUT TINY



live nodes are left OUT of their tile, so nothing draws twice

THE SHAPE OF IT

GDSCRIPT

```
class Tile extends Node2D:           # one canvas item = one retained command list
    var ref_zoom := 1.0; var ref_centre := Vector2.ZERO # the camera it was recorded with
    func _draw() -> void: web.draw_tile(self) # runs ONLY after queue_redraw()

    func _process(delta) -> void:
        for t in tiles:
            if t.dirty: t.queue_redraw() # data changed in this tile
            var k := camera.zoom / t.ref_zoom # camera = your own pan/zoom state (zoom is a float here)
            t.scale = Vector2(k, k)
            t.position = screen / 2 - k * screen / 2 + (t.ref_centre - camera.centre) * camera.zoom
            live_layer.queue_redraw() # the few things that really animate
```

FULLY UPGRADED SKILL TREE, MEASURED

BEFORE

AFTER

Today's 87 nodes	49.9 ms/frame	10.7 ms
4× the nodes (a bigger tree later)	116 ms/frame	15.2 ms
Panning across it	a full redraw every frame	0 redraws

The same idea, one level up: render at your art's size

With the `canvas_items` stretch mode, Godot draws at the **screen's** resolution. A 4K monitor is about 9× the pixels of a 1280×720 design, and every glow and full-screen shader pays for each one (on our test PC, ~8 ms of GPU work per frame, with the fans to match). The `viewport` mode renders at the design size and scales the finished frame up. That's crisp for pixel art, though text gets chunkier. We made it a setting, so players choose:

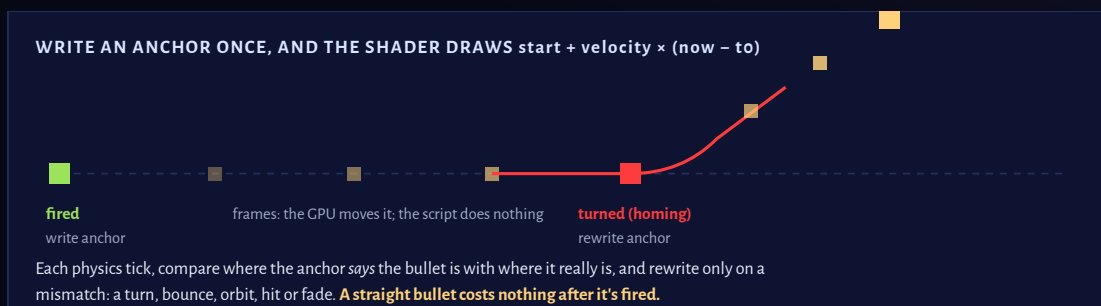
RENDER QUALITY, LIVE, NO RESTART

GDSCRIPT

```
get_tree().root.content_scale_mode = Window.CONTENT_SCALE_MODE_VIEWPORT # "Performance (720p)"
get_tree().root.content_scale_mode = Window.CONTENT_SCALE_MODE_CANVAS_ITEMS # "Sharp (native)"
```

Let the GPU move things

Most bullets fly in straight lines. If the graphics card knows where one started, how fast it's going and when it left, it can draw it in the right place every frame without any help from your scripts.



FLIGHT.GDShaderINC SUPPLIES FLY(); EACH BULLET SHADER SUPPLIES ITS OWN VERTEX()

SHADER

```
// INSTANCE_CUSTOM = (velocity.x, velocity.y, anchor_time, is_live)
uniform float flight_now; // set once per frame by the manager
vec2 fly(vec2 v, vec4 c, mat4 model) {
    if (c.w < 0.5) return v; // ordinary sprites using this shader: untouched
    float t = max(flight_now - c.z, 0.0);
    mat2 lin = mat2(model[0].xy, model[1].xy); // VERTEX is still in mesh space here,
    return v + inverse(lin) * (c.xy * t); // so turn the world offset back into it
}
// --- in each bullet shader, not in the include ---
#include "res://flight.gdshaderinc"
void vertex() { VERTEX = fly(VERTEX, INSTANCE_CUSTOM, MODEL_MATRIX); }
```

THE MANAGER SIDE: RE-ANCHOR ONLY ON A MISMATCH

GDSCRIPT

```
func after_step(b, prev_pos: Vector2, dt: float) -> void:
    var predicted := b.anchor_pos + b.anchor_vel * (clock - b.anchor_t0)
    if b.look_changed or predicted.distance_squared_to(b.position) > 0.0025: # = (0.05 px)2
        # lay the anchor on the PREVIOUS tick, with this step's real velocity
        write_anchor(b, prev_pos, (b.position - prev_pos) / dt, clock - dt)

func _process(_delta) -> void: # one tick behind + the fraction = Godot's own smoothing
    material.set_shader_parameter("&flight_now",
        clock - last_dt + Engine.get_physics_interpolation_fraction() * last_dt)
```

WHY "ONE TICK BEHIND"

With physics interpolation on, Godot draws a moving object between its **previous** and **current** tick. Running the shader clock the same way keeps GPU-moved bullets glued to everything the engine draws, like their trails.

STAY SAFE

Guard the code with a per-instance flag so other sprites sharing the shader are unaffected, and let anything the fast path doesn't understand keep drawing the old way.

WHAT WE MEASURED, AND WHERE IT BACKFIRED

- **~800 glowing bullets** (body + halo + floor glow each): draw calls fell from **~1,150 to ~480**.
- **Frame time** stayed within our benchmark's noise, and at **300 plain arrows nothing changed**: identical sprites with one material *already* batch in Godot.
- **Then real play bit us**: most of our bullets home, orbit or boomerang, so they bent every tick and were rewritten every tick: **4–6 ms a frame**, worse than plain nodes. Now only straight-flying bullets go through it.

Load early, and don't change the gameplay

NO FIRST-TIME STUTTERS

- **Warm up shaders.** A material compiles the first time it's drawn, causing a hitch on the first fireball. Draw each effect **in view** for a few frames while an opaque loading screen covers it. Off-screen items are culled and never compile.
- **Prewarm pools from the design.** Work out how many of each object a maxed-out build can have; create them while loading. Log any shortfall.

FASTER, BUT PLAYS THE SAME

- **Keep old objects as the source of truth** while other code reads them; move only the expensive part (e.g. the drawing).
- **Write the rules as tests:** "hits once on entry", "pickups freeze when paused".
- **Give players a renderer choice.** Our Windows build defaults to Direct3D 12 (`rendering_device/driver.windows`). Rare 100–600 ms stalls where neither our code nor rendering was busy usually mean a GPU pipeline compiling, and drivers differ in how they handle that. `--rendering-driver vulkan` switches renderer at launch; Steam launch options can offer it as a second "Play" entry.

SHADER WARM-UP BEHIND THE LOADING SCREEN

GDSCRIPT

```
# runs while the loading curtain (an opaque UI layer on top) still hides the world
func warm_up(effects: Array[PackedScene], at: Vector2) -> void:
    for scene in effects:
        var fx := scene.instantiate()
        fx.position = at          # IN VIEW (e.g. the player's spot): culled items never compile
        warm_root.add_child(fx)
    for i in 4: await get_tree().process_frame # let every pipeline compile
    warm_root.queue_free()                  # then lift the loading curtain
```

Think less often than you move

Movement must run every tick or the horde stutters; deciding where to go needn't. Two halves of the horde think on alternate ticks (30 Hz), each handed the time since its last think, so cooldowns keep real time. Bosses and monsters mid-wind-up still think every tick, where a lag would show.

IN EACH MONSTER'S TICK: MOVE EVERY TICK, THINK EVERY OTHER

GDSCRIPT

```
_think_dt += delta
_think_n += 1
if _think_n >= stride or busy or not brain.staggers(): # stride 2 = 30 Hz; bosses say no
    _think_n = 0
    brain.think(self, _think_dt) # the banked time, so cooldowns keep real time
    _think_dt = 0.0
# at spawn: _think_n = 0 or 1, alternating, so the two halves take turns
```

WHAT IT BOUGHT, AND TWO LESSONS NEXT TO IT

- **Worth it, not magic.** A think was 1.85 of a monster's 8.7 μ s: ~0.9 ms a tick saved at 1000.
- **A stagger must survive deaths.** Our separation checked half the horde per tick, but any death forced a full pass, and something dies every tick. Moving each saved result with its monster: 5.4 $\rightarrow$ 2.9 ms.
- **Browsers are slower** (1.4–1.7 $\times$ in Chrome): a `.web` setting override gives web a smaller horde.

Your checklist

IF YOU FIND...	TRY...
A light on every bullet or pickup	Additive glow sprite + a night-lift shader uniform
A <code>Line2D</code> trail per bullet	Trail segments in one MultiMesh, faded by age
<code>instantiate()</code> / <code>queue_free()</code> mid-fight	A pool, prewarmed while loading
<code>_process</code> on every pooled object	One manager looping over a list
<code>Area2D</code> on every enemy and bullet	A spatial grid + circle/capsule maths
One image per animation frame	A padded texture atlas
Hundreds of identical small sprites	A MultiMesh on the RenderingServer
Script moving every bullet every frame	An anchor in instance data; rewrite on change
A stutter the first time an effect plays	Shader warm-up while loading
<code>queue_redraw()</code> every frame on a big, mostly still screen	Retained tiles + a small live layer
A 4K screen drawing 720p art	A render-quality option (viewport stretch)
Every monster deciding every tick	Think at 30 Hz in alternating halves; move every tick
A web build with the desktop horde	A <code>.web</code> project-setting override with a smaller cap
Timings taken with F5	Re-measure without the debugger, or in a release build
"It just feels laggy"	A perf log from real play; fix the biggest number

Measure, find the thing you have hundreds of, and stop paying for it one at a time.

Written by the team at Naïo Games, an indie game studio, while optimising our Godot 4 game Hexwright. Three things here haven't had a full playtest yet: Tip 9's straight-bullets-only drawing, Tip 7's pickup MultiMesh and 30 Hz thinking. Everything else was measured on our own machine, so your numbers will differ.