Texel Density Balancer Prototype — Day 3 (Part B)

Automatic Bias Governor Based on Runtime Telemetry

Rameel Malik

Abstract

This document explains the *automatic* texel density adjustment (Day 3, Part B). Unlike earlier days where level-of-detail (LOD) bias was manually controlled, Part B introduces a closed-loop controller that measures runtime load and automatically increases or decreases mipmap bias to stabilize performance. The report highlights **only the new code chunks** added on top of Day 3 (Part A), details how they work, and provides practical guidance for tuning and testing.

Contents

1 Context and Goal

Texel density determines how many texture texels are sampled per screen pixel. Sampling higher mip levels (negative/low bias) yields sharper detail but costs more bandwidth/cache; sampling lower mip levels (positive/high bias) reduces cost but looks blurrier.

Goal (Part B). Make texel density *self-adjust* at runtime: when the system detects stress (e.g., frame rate drop), increase LOD bias (lower texel density) to regain headroom; when healthy, decrease bias (raise texel density) to improve quality.

2 Design Overview

We built a lightweight **feedback loop**:
1. Measure a performance signal each frame (FPS).
2. Compare it to a target (e.g., 60 FPS ± tolerance).
3. Adjust texture LOD bias up/down in small steps.
4. Clamp bias to a safe hardware range and apply.

3 What's New in Part B (Code Chunks)

Below are the **new code chunks** added on top of Day 3 (Part A).

3.1 Runtime Telemetry Variables

Listing 1: Telemetry state for FPS and smoothing

```
double lastTime = glfwGetTime();
double deltaTime = 0.0;
double fps = 0.0;
double avgFPS = 0.0;
const double smoothing = 0.9; // exponential moving average factor
```

3.2 Control Targets and Bias State

Listing 2: Controller configuration and state

```
float targetFPS = 60.0f; // Desired frame rate
float biasStep = 0.05f; // Bias delta per update (tune!)
float lodBias = 0.0f; // Current global LOD bias (starts sharp-ish)
```

3.3 FPS Measurement (Per Frame)

Listing 3: Compute instantaneous and smoothed FPS

```
double now = glfwGetTime();
deltaTime = now - lastTime;
lastTime = now;

fps = (deltaTime > 0.0) ? (1.0 / deltaTime) : fps;
avgFPS = smoothing * avgFPS + (1.0 - smoothing) * fps; // EMA
```

3.4 Automatic LOD-Bias Governor

Listing 4: Bias control based on FPS error

```
const float deadband = 5.0f; // +/- FPS tolerance around target

if (avgFPS < targetFPS - deadband) {
    // Below target: increase bias => push to lower mips (cheaper)
    lodBias += biasStep;
} else if (avgFPS > targetFPS + deadband) {
    // Above target: decrease bias => sharper (more expensive)
    lodBias -= biasStep;
```

```
}

// Safety clamp (hardware-reasonable range)
lodBias = std::clamp(lodBias, -0.25f, 3.0f);

// Apply to the texture sampler in use
glSamplerParameterf(samp, GL_TEXTURE_LOD_BIAS, lodBias);
```

3.5 Lightweight Console HUD (Optional)

Listing 5: Periodic debug print

```
static double t0 = glfwGetTime();
if (now - t0 > 1.0) {
    std::cout << "avgFPS=" << avgFPS
              << " bias=" << lodBias << "\n";
    t0 = now;
}
```

4 How It Works (Step-by-Step)

Step 1. Render the scene as usual (your cube, textures, transforms).

Step 2. Measure FPS each frame and compute a smoothed average.

Step 3. Compare to target: if below, raise `lodBias`; if above, lower it.

Step 4. Clamp and apply the bias via `glSamplerParameterf`.

Step 5. Repeat next frame, gently converging to a performance "sweet spot".

5 Feedback Loop Diagram

Figure 1 visualizes the controller used in Part B.

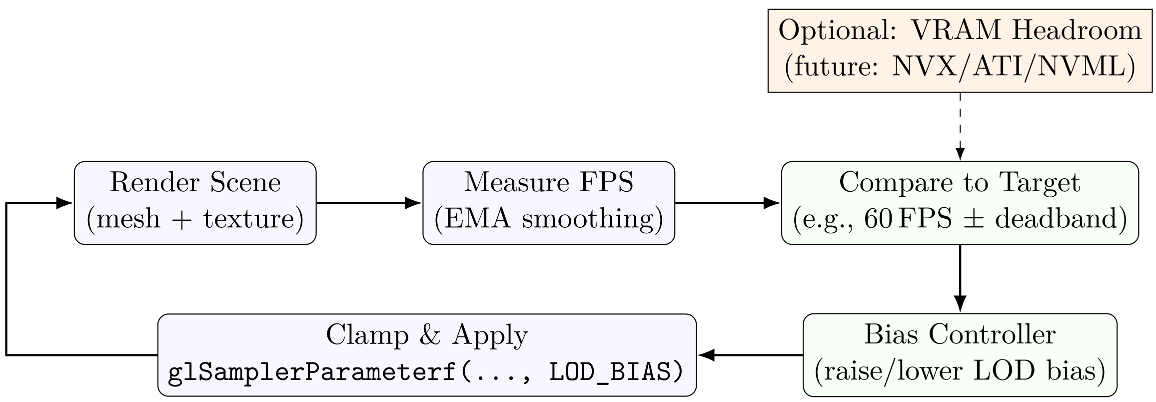

Figure 1: Automatic texel-density feedback loop used in Day 3 Part B.

6 Why Blur "Stops Increasing" After a Point

Textures have a finite mipmap chain. For a 4096×4096 texture:

$$\text{maxMip} \approx \lfloor \log_2(4096) \rfloor = 12.$$

Once the bias drives sampling to the smallest level (e.g., 1×1), further increases won't change the result: the entire surface samples that single texel. Visually, that looks like a solid tint or very heavy blur. This is normal and not a bug.

7 Testing and Forcing Load

A single cube may not stress the GPU enough to showcase the governor. To test:

- Load multiple large textures (4K/8K); draw more objects.
- Or add a *dummy texture harness* (e.g., keys `P`/`O` to spawn/free batches of 4K textures) to simulate VRAM pressure.
- Watch `avgFPS` and `bias` in the console; you should see bias rise when load increases and fall when load drops.

8 Tuning Cheatsheet

- **`targetFPS`**: Set your desired smoothness (60/90/120).
- **`deadband`**: Wider deadband = fewer small corrections.
- **`biasStep`**: Larger steps react faster but may overshoot; start at 0.03–0.07.
- **Clamp range**: Keep within `[-0.25, 3.0]` unless you have a reason (and textures) to widen it.

Appendix A: Minimal EMA Derivation

Let instantaneous FPS at frame t be f_t. The EMA with factor $\alpha \in (0, 1)$ is:

$$\text{avgFPS}_t = \alpha \cdot \text{avgFPS}_{t-1} + (1 - \alpha) \cdot f_t.$$

Higher α biases toward older values (smoother, slower to react). Here we use $\alpha = 0.9$.

Appendix B: Safety Clamp Rationale

Clamping `lodBias` ensures calls like `texture(...)` don't request mip levels outside the texture's valid range. Beyond a certain bias, the GPU will clamp internally anyway, but clamping in the app avoids unnecessary overshoot and makes the controller's behavior predictable.