Texel Density Balancer Prototype – Day 2

Rameel Malik

August 12, 2025

Contents

1 Overview

This report documents the Day 1 and Day 2 prototype for a *Texel Density Balancer* implemented with OpenGL. The goal is to control perceived texture detail ("texel density") at runtime by nudging the GPU's mipmap level selection via a **LOD (Level of Detail) bias**. This is the core building block for a memory-aware governor that can automatically reduce texture detail when VRAM pressure increases.

What this prototype demonstrates

- Creating a window and OpenGL context using **GLFW**.
- Loading modern OpenGL function pointers with **GLEW**.
- Rendering a textured quad with a minimal vertex/fragment shader pair.
- Generating mipmaps and creating a **sampler object** to control filtering.
- Adjusting **LOD bias** in real time to lower or raise texel density.

2 Core Concepts (Plain English)

2.1 What is OpenGL (GL)?

OpenGL is a cross-platform API for drawing 2D/3D graphics on the GPU. You issue `gl*` commands to upload data (textures, vertex buffers) and to render.

2.2 Why GLFW?

GLFW = *Graphics Library Framework*. OpenGL cannot create a window or handle input by itself; GLFW provides a portable way to:

- Create a desktop window.
- Create an OpenGL *context* (the GPU state machine where GL commands operate).
- Handle keyboard/mouse and window events.

2.3 Why GLEW?

GLEW = *OpenGL Extension Wrangler*. On Windows, the system headers only expose OpenGL 1.1. GLEW loads the pointers for modern GL functions (OpenGL 3.3) at runtime so we can call them.

2.4 Render Loop

An interactive graphics app runs a *render loop*:

1. Poll input/events.
2. Update scene state.
3. Issue draw calls.
4. Swap the back buffer to the screen.

Without this loop, a window would appear and exit immediately.

2.5 Clip Space (and -1..1 coordinates)

The vertex shader outputs positions in **clip space**. After perspective division, coordinates map to **NDC** (Normalized Device Coordinates) in $[-1, 1]$ on x/y, which OpenGL then maps to window pixels. Setting positions to the corners $(-1, -1)$, $(1, -1)$, $(1, 1)$, $(-1, 1)$ draws a full-screen quad.

2.6 Shaders

Vertex shader runs per-vertex; it sets the output position (`gl_Position`) and passes interpolated data (like UVs) to the fragment shader. **Fragment shader** runs per-pixel; it computes the final color, often by sampling a texture at the interpolated UV.

2.7 VBO/EBO and "elements"

A **VBO** (Vertex Buffer Object) holds vertex attributes (positions, UVs). An **EBO** (Element Buffer Object) holds *indices* ("elements") that reference vertices, so triangles can share vertices without duplication.

2.8 Mipmaps and LOD Bias

A texture's mipmap chain stores half-resolution versions: level 0 (full), 1 (1/2), 2 (1/4), ... The GPU chooses which level to sample based on how large the texture appears on screen. **LOD bias** adds an offset to that choice: positive bias selects lower-resolution mips sooner (lower texel density), negative bias selects higher-resolution mips (sharper).

3 Day 1: Minimal OpenGL App

3.1 Final Working Code (Day 1)

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.20)
project(VramGovernor LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

find_package(glfw3 CONFIG REQUIRED)
find_package(GLEW  CONFIG REQUIRED)

add_executable(vram_app
    src/main.cpp
)

if (WIN32)
  target_link_libraries(vram_app PRIVATE opengl32)
endif()

target_link_libraries(vram_app PRIVATE glfw GLEW::GLEW)

if (MSVC)
  target_compile_options(vram_app PRIVATE /W4 /permissive-)
else()
  target_compile_options(vram_app PRIVATE -Wall -Wextra -
     Wpedantic)
endif()
```

src/main.cpp

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <iostream>

static void APIENTRY glDebugCallback(GLenum source, GLenum type,
   GLuint id,
    GLenum severity, GLsizei length, const GLchar* message, const
       void* userParam) {
     std::cerr << "[GL] " << message << "\n";
}

int main() {
    if (!glfwInit()) { std::cerr << "Failed to init GLFW\n";
       return -1; }
```

```
12
13      glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
14      glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
15      glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE)
            ;
16
17      GLFWwindow* window =
18          glfwCreateWindow(1280, 720, "Day 1     VramGovernor",
                nullptr, nullptr);
19      if (!window) { std::cerr << "Failed to create window\n";
            glfwTerminate(); return -1; }
20
21      glfwMakeContextCurrent(window);
22      glfwSwapInterval(1); // vsync
23
24      glewExperimental = GL_TRUE; // expose all core functions
25      GLenum glewErr = glewInit();
26      glGetError(); // clear harmless flag
27
28      if (glewErr != GLEW_OK) {
29          std::cerr << "GLEW init error: " << glewGetErrorString(
                glewErr) << "\n";
30          glfwTerminate(); return -1;
31      }
32
33  #ifdef GL_DEBUG_OUTPUT
34      glEnable(GL_DEBUG_OUTPUT);
35      glDebugMessageCallback(glDebugCallback, nullptr);
36  #endif
37
38      std::cout << "Vendor   : " << glGetString(GL_VENDOR) << "\n";
39      std::cout << "Renderer: " << glGetString(GL_RENDERER) << "\n"
            ;
40      std::cout << "Version : " << glGetString(GL_VERSION)  << "\n"
            ;
41
42      while (!glfwWindowShouldClose(window)) {
43          glfwPollEvents();
44          if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
45              glfwSetWindowShouldClose(window, 1);
46
```

```
        int w, h; glfwGetFramebufferSize(window, &w, &h);
        glViewport(0, 0, w, h);
        glClearColor(0.07f, 0.10f, 0.15f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glfwSwapBuffers(window);
    }
    glfwDestroyWindow(window);
    glfwTerminate();
    return 0;
}
```

3.2 What Day 1 Proved

- GLFW successfully created a window + OpenGL 3.3 Core context.
- GLEW loaded modern GL functions so we can use them.
- The render loop cleared the screen and swapped buffers each frame.
- GPU information printed via `glGetString`.

4 Day 2: Textured Quad + LOD Bias (Texel Density)

4.1 Shaders

Vertex Shader (GLSL)

```
#version 330 core
layout(location = 0) in vec2 aPos;  // vertex position in clip-
   space
layout(location = 1) in vec2 aUV;   // vertex texture coordinate

out vec2 vUV; // interpolated UV for the fragment shader

void main() {
    vUV = aUV;
    gl_Position = vec4(aPos, 0.0, 1.0);
}
```

Fragment Shader (GLSL)

```
#version 330 core
in vec2 vUV;
out vec4 fragColor;
uniform sampler2D uTex;

void main() {
    fragColor = texture(uTex, vUV);
}
```

4.2 Full Day 2 C++ (key parts)

Geometry (quad), shaders, texture load, sampler, and LOD bias control.

```
// --- Quad geometry (positions + UVs) and buffers ---
// Each vertex: [x, y, u, v]; positions are clip space corners
float verts[] = {
    -1, -1,   0, 0,  // bottom-left
     1, -1,   1, 0,  // bottom-right
     1,  1,   1, 1,  // top-right
    -1,  1,   0, 1   // top-left
};
unsigned int idx[] = { 0,1,2,  2,3,0 };

GLuint vao, vbo, ebo;
glGenVertexArrays(1, &vao);
glGenBuffers(1, &vbo);
glGenBuffers(1, &ebo);

glBindVertexArray(vao);

glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(verts), verts,
   GL_STATIC_DRAW);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(idx), idx,
   GL_STATIC_DRAW);

// layout(location=0) -> position (2 floats), stride=4 floats,
   offset=0
```

```
25 glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 4*sizeof(float),
       (void*)0);
26 glEnableVertexAttribArray(0);
27
28 // layout(location=1) -> UV (2 floats), stride=4 floats, offset=2
       floats
29 glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 4*sizeof(float),
       (void*)(2*sizeof(float)));
30 glEnableVertexAttribArray(1);
31
32 // --- Load image with stb_image ---
33 #define STB_IMAGE_IMPLEMENTATION
34 #include "stb_image.h"
35 int tw, th, ch;
36 unsigned char* data = stbi_load("assets/checker.png", &tw, &th, &
       ch, STBI_rgb_alpha);
37 if(!data) { std::cerr << "Failed to load texture\n"; /* handle
       error */ }
38
39 // --- Create GL texture + mipmaps ---
40 GLuint tex;
41 glGenTextures(1, &tex);
42 glBindTexture(GL_TEXTURE_2D, tex);
43 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, tw, th, 0, GL_RGBA,
       GL_UNSIGNED_BYTE, data);
44 glGenerateMipmap(GL_TEXTURE_2D);
45 stbi_image_free(data);
46
47 // --- Create a sampler object (filtering rules) ---
48 GLuint samp;
49 glGenSamplers(1, &samp);
50 glSamplerParameteri(samp, GL_TEXTURE_MIN_FILTER,
       GL_LINEAR_MIPMAP_LINEAR);
51 glSamplerParameteri(samp, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
52 // optional but helpful for angled surfaces:
53 glSamplerParameterf(samp, GL_TEXTURE_MAX_ANISOTROPY, 4.0f);
54
55 // --- Compile/link shaders (helper not shown) and set sampler
       uniform ---
56 GLuint prog = createProgram(); // attaches vertex+fragment GLSL,
       links program
```

```
glUseProgram(prog);
glUniform1i(glGetUniformLocation(prog, "uTex"), 0); // texture unit 0

// --- Runtime LOD bias control (texel density knob) ---
float lodBias = 0.0f; // 0 = default; + makes blurrier; - makes sharper

while (!glfwWindowShouldClose(window)) {
    glfwPollEvents();

    // Press ']' to increase LOD bias (cheaper, blurrier)
    if (glfwGetKey(window, GLFW_KEY_RIGHT_BRACKET) == GLFW_PRESS)
        lodBias += 0.01f;
    // Press '[' to decrease LOD bias (sharper, costlier)
    if (glfwGetKey(window, GLFW_KEY_LEFT_BRACKET)  == GLFW_PRESS)
        lodBias -= 0.01f;

    // Clamp to a safe range to avoid extreme blur/aliasing
    lodBias = std::max(-0.25f, std::min(1.6f, lodBias));

    // Apply sampling rules to texture unit 0
    glBindSampler(0, samp);
    glSamplerParameterf(samp, GL_TEXTURE_LOD_BIAS, lodBias);

    glClearColor(0.07f, 0.10f, 0.15f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, tex);

    glUseProgram(prog);
    glBindVertexArray(vao);
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);

    glfwSwapBuffers(window);
}
```

4.3 Line-by-Line Explanations (Selected Hotspots)

VAO/VBO/EBO setup

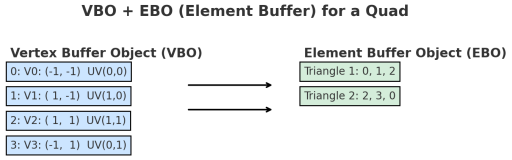

Figure 1: VBO/EBO representation of a quad using triangle indices. This shows how vertex data is stored once in the Vertex Buffer Object (VBO), while the Element Buffer Object (EBO) stores the indices to form triangles without duplication.

- `glGenVertexArrays/BindVertexArray`: create/bind a VAO to record attribute state.
- `glBindBuffer(GL_ARRAY_BUFFER, vbo)` then `glBufferData`: upload vertex data (positions+UVs).
- `glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo)`: upload index data; binding is stored in the VAO.
- `glVertexAttribPointer/glEnableVertexAttribArray`: declare attribute 0 = position (2 floats, offset 0), attribute 1 = UV (2 floats, offset 2 floats), with stride 4 floats per vertex.

Texture creation

- `glTexImage2D`: copies CPU pixels (`data`) into GPU storage as `GL_RGBA8`.
- `glGenerateMipmap`: builds the full mip chain (1/2, 1/4, ...).

Sampler rules

- `GL_LINEAR_MIPMAP_LINEAR` (minification): trilinear filtering (smooth transitions between mips).
- `GL_LINEAR` (magnification): bilinear filter when upscaling.
- `GL_TEXTURE_MAX_ANISOTROPY` $\approx$ sharper textures at glancing angles (driver extension).

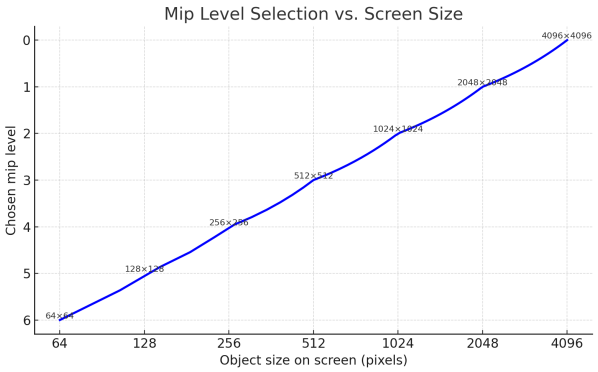

Figure 2: Relationship between object size on screen and selected mipmap level. Larger on-screen sizes use higher-resolution mip levels, while smaller sizes use lower mip levels to save memory bandwidth and avoid aliasing.

Uniform `uTex` and texture unit

- `uniform sampler2D uTex;` in GLSL holds the *texture unit index* (0,1,...).
- `glUniform1i(...,"uTex"), 0`: tell the shader to read from texture unit 0.
- `glActiveTexture(GL_TEXTURE0)` and `glBindTexture`: bind the actual texture object to that unit.

LOD bias (texel density)

- `glSamplerParameterf(samp, GL_TEXTURE_LOD_BIAS, lodBias)` shifts the GPU's chosen mip level.
- Positive bias ⇒ lower-resolution mips sooner ⇒ lower texel density (cheaper).
- Negative bias ⇒ sharper mips (costlier, risk of aliasing if extreme).

5 Theory: Why Lowering Texel Density Helps

5.1 Mipmaps Prevent Aliasing and Save Bandwidth

If a texture appears very small on screen, sampling the full-res image causes many texels to map to a single pixel (aliasing, shimmering). Mipmaps use smaller prefiltered versions so each on-screen pixel samples fewer texels, improving quality and reducing memory bandwidth.

5.2 How the GPU Chooses a Mip (intuition)

The hardware estimates how fast UVs change across neighboring pixels. A common shorthand is a value $\lambda \approx \log_2(\rho)$ (where ρ relates to UV gradients $\times$ texture size). The

chosen mip is roughly level λ. We apply:

$$\lambda_{\text{effective}} = \lambda_{\text{hardware}} + \text{lodBias}$$

so a lodBias of +1 nudges sampling about one mip lower (half-resolution in each dimension).

5.3 Performance Effects

- **Lower bandwidth**: reading from a smaller mip avoids large texture fetches.
- **Better cache locality**: smaller working sets fit into GPU texture caches.
- **Smoother under pressure**: lowering texel density when VRAM is tight helps avoid paging and stalls.

6 Expected Runtime Behavior

- A textured quad fills the screen.
- Press `[` to decrease LOD bias (sharper, higher texel density).
- Press `]` to increase LOD bias (blurrier, lower texel density).
- The effect is smooth because the sampler uses trilinear filtering.

7 Appendix A: Detailed Q&A (Consolidated Explanations)

What are `aUV` and `vUV`? `aUV` is the per-vertex UV attribute (from the VBO). The vertex shader copies it into `vUV`, which the GPU interpolates per-pixel for the fragment shader to sample.

What are vertex and fragment shaders? Vertex shader runs per vertex (sets `gl_Position`, forwards attributes). Fragment shader runs per pixel (computes final color, typically by sampling a texture with `texture(uTex, vUV)`).

What is `GLuint`? An OpenGL-defined typedef for an unsigned integer used as an object ID (textures, buffers, shaders, etc.).

What is an "element"? A single index into the vertex buffer. The EBO stores indices so triangles can *reuse* vertices (0,1,2) and (2,3,0) for our quad.

Why does OpenGL need a window and a context? OpenGL commands must target a valid context (driver-managed GPU state). GLFW creates the window and context so subsequent `gl*` calls have a destination.

What is `glewExperimental = GL_TRUE`? It tells GLEW to expose core/modern function pointers even if marked "experimental" in its DB. Required for core profiles like GL 3.3.

What is `uTex`? A GLSL `uniform sampler2D` that stores the texture unit index. We set it from C++ with `glUniform1i(...,0)` and then bind our texture to unit 0.

How do we clamp LOD bias? After adjusting it with keys, restrict it: `lodBias = clamp(lodBias, -0.25f, 1.6f)` to avoid extreme blur/sharpening.

How does changing LOD bias change texel density? It shifts the chosen mip level: positive bias ⇒ lower mips sooner (⇒ fewer texels per pixel); negative bias ⇒ higher mips (⇒ more texels per pixel).

8 Appendix B: Minimal Asset Note

At runtime the program loads `assets/checker.png`. Ensure your executable can find the file (e.g., place a copy next to the executable inside an `assets/` folder, or adjust the path).

9 Roadmap (Next Steps)

- Add an on-screen ROI so the center remains sharp while the periphery uses higher LOD bias.
- Add VRAM monitoring and switch bias automatically using thresholds + hysteresis.
- Abstract the quality control as a reusable "governor" that later plugs into a volume renderer.