Textual Density Balancer — Day 3 (P1)

Rameel Malik

August 13, 2025

1 Goal (Day 3, Part 1)

Move from the Day 2 full-screen textured quad to a **true 3D scene** (a textured, rotating cube) and keep the manual **texel density control** (LOD bias on `[` and `]`). Add a simple **FPS monitor** to observe performance. ROI/auto-control comes in Part 2.

2 What changed compared to Day 2

- **Geometry:** 2D full-screen quad → **3D indexed cube** (36 indices) with per-face UVs.
- **Camera & transforms:** We now compute an **MVP** matrix (Perspective × View × Model) and transform vertices in the vertex shader.
- **Depth testing:** We request a window depth buffer and enable depth testing to render correct front/back face visibility.
- **Sampler control:** Same trilinear + mipmap sampler as Day 2; LOD bias still mapped to `[ / ]`, but behavior differs in 3D (explained later).
- **FPS monitor:** A tiny timer that prints FPS and current bias once per second.

3 How the cube is constructed

Unlike a quad, a cube needs **positions (x,y,z)** and **UVs** for each face. We duplicate logical corners so each face can have its own clean UV mapping. We draw with an **index buffer** to reuse vertices.

Vertex & index data

```
// Each vertex: position (x,y,z) + texcoords (u,v)
struct V { float x,y,z,u,v; };
static const V cubeVerts[] = {
    // Front (+Z)
    {-1,-1, 1, 0,0}, { 1,-1, 1, 1,0}, { 1, 1, 1, 1,1}, {-1, 1, 1,
        0,1},
    // Back (-Z)
```

```
    { 1,-1,-1, 0,0}, {-1,-1,-1, 1,0}, {-1, 1,-1, 1,1}, { 1, 1,-1,
        0,1},
    // Left (-X)
    {-1,-1,-1, 0,0}, {-1,-1, 1, 1,0}, {-1, 1, 1, 1,1}, {-1, 1,-1,
        0,1},
    // Right (+X)
    { 1,-1, 1, 0,0}, { 1,-1,-1, 1,0}, { 1, 1,-1, 1,1}, { 1, 1, 1,
        0,1},
    // Top (+Y)
    {-1, 1, 1, 0,0}, { 1, 1, 1, 1,0}, { 1, 1,-1, 1,1}, {-1, 1,-1,
        0,1},
    // Bottom (-Y)
    {-1,-1,-1, 0,0}, { 1,-1,-1, 1,0}, { 1,-1, 1, 1,1}, {-1,-1, 1,
        0,1},
};

static const unsigned cubeIdx[] = {
    // front
     0, 1, 2,  2, 3, 0,
    // back
     4, 5, 6,  6, 7, 4,
    // left
     8, 9,10, 10,11, 8,
    // right
    12,13,14, 14,15,12,
    // top
    16,17,18, 18,19,16,
    // bottom
    20,21,22, 22,23,20
};
```

GPU buffers & layout

```
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(cubeVerts), cubeVerts,
   GL_STATIC_DRAW);

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

// layout(location=0) => position (x,y,z)
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(V), (void
   *)0);
```

```
glEnableVertexAttribArray(0);
// layout(location=1) => UV (u,v)
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, sizeof(V), (void
   *)(3*sizeof(float)));
glEnableVertexAttribArray(1);

// draw call
glBindVertexArray(vao);
glDrawElements(GL_TRIANGLES,
    (GLsizei)(sizeof(cubeIdx)/sizeof(unsigned)), GL_UNSIGNED_INT,
        0);
```

4 MVP transforms (column-major) and depth test

Day 2 bypassed transforms (positions were already in clip space). Day 3 introduces a real camera and projection:

- **Perspective** creates the frustum (FOV, aspect, near/far).
- **View (LookAt)** positions the camera.
- **Model** rotates the cube over time.
- Vertex shader applies: `gl_Position = uMVP * vec4(aPos,1)`.

Matrix upload: We use **column-major** math and pass with `GL_FALSE` (no transpose).

```
// Column-major multiply: out = a * b
static void mul44(const float a[16], const float b[16], float out
   [16]) {
    for (int c = 0; c < 4; ++c)
      for (int r = 0; r < 4; ++r)
        out[c*4 + r] =
            a[0*4 + r]*b[c*4 + 0] +
            a[1*4 + r]*b[c*4 + 1] +
            a[2*4 + r]*b[c*4 + 2] +
            a[3*4 + r]*b[c*4 + 3];
}

float proj[16], view[16], model[16], pv[16], mvp[16];
makePerspective(fovRad, aspect, 0.1f, 100.0f, proj);
makeLookAt(eyeX,eyeY,eyeZ, 0,0,0, 0,1,0, view);
// simple Y-rotation for model
float c=cos(t*0.8f), s=sin(t*0.8f);
float model[16] = { c,0,s,0,  0,1,0,0,  -s,0,c,0,  0,0,0,1 };

mul44(proj, view, pv);
mul44(pv, model, mvp);

glUniformMatrix4fv(uMVP, 1, GL_FALSE, mvp);
```

Depth test: We request a depth buffer and enable it:

```
glfwWindowHint(GLFW_DEPTH_BITS, 24);
...
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glEnable(GL_DEPTH_TEST);
```

5 Shaders (Day 3)

Vertex shader

```
static const char* kVS = R"(
#version 330 core
layout(location=0) in vec3 aPos;
layout(location=1) in vec2 aUV;
uniform mat4 uMVP;
out vec2 vUV;
void main(){
    vUV = aUV;
    gl_Position = uMVP * vec4(aPos, 1.0);
}
)";
```

Fragment shader

```
static const char* kFS = R"(
#version 330 core
in vec2 vUV;
out vec4 fragColor;
uniform sampler2D uTex;
void main(){
    fragColor = texture(uTex, vUV);
}
)";
```

6 Sampler & LOD bias (manual control)

We keep trilinear filtering with mipmaps and expose **LOD bias** on keys.

```
// Trilinear + mipmaps
glSamplerParameteri(samp, GL_TEXTURE_MIN_FILTER,
   GL_LINEAR_MIPMAP_LINEAR);
glSamplerParameteri(samp, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

// Bias control: '[' decreases, ']' increases
if (glfwGetKey(window, GLFW_KEY_RIGHT_BRACKET)==GLFW_PRESS)
   lodBias += 0.01f;
```

```
if (glfwGetKey(window, GLFW_KEY_LEFT_BRACKET) ==GLFW_PRESS)
   lodBias -= 0.01f;
lodBias = std::clamp(lodBias, -0.25f, 3.0f);
glSamplerParameterf(samp, GL_TEXTURE_LOD_BIAS, lodBias);
```

Why blur "tops out" sooner than Day 2:

1. **Mip floor:** there are only $\lfloor \log_2(\max(\text{width}, \text{height})) \rfloor$ mip levels. Once a pixel samples the lowest mip (often 1x1), extra bias has no further effect.
2. **Magnification zones:** close/large pixels use the **MAG** filter (no mipmaps); bias doesn't apply there.
3. **3D perspective:** different pixels have different derivative scales; many won't reach the deepest mip simultaneously like the Day 2 full-screen quad did.

7 Why the blur looks smooth (not blocky)

Two reasons:

1. **Mipmap generation** (`glGenerateMipmap`) down-samples by averaging texels (box-like filter), producing progressively *pre-blurred* images.
2. **Trilinear filtering** (`GL_LINEAR_MIPMAP_LINEAR`) blends both *within* a mip (bi-linear) and *between* adjacent mips. The combined effect visually approximates a Gaussian blur instead of blocky pixels.

8 FPS monitor (what you added)

We measure wall-clock time once per second and print FPS + bias. Disable VSYNC to uncap FPS.

```
#include <chrono>
float frameCount = 0.0f, fps = 0.0f;
auto lastTime = std::chrono::high_resolution_clock::now();

...

// in the render loop:
frameCount++;
auto now = std::chrono::high_resolution_clock::now();
float dt = std::chrono::duration<float>(now - lastTime).count();
if (dt >= 1.0f) {
    fps = frameCount / dt;
    frameCount = 0.0f;
    lastTime = now;
    std::cout << "FPS: " << fps << " | LOD Bias: " << lodBias <<
       "\n";
}
```

Why you didn't see a big FPS boost yet

- **VSYNC cap:** with `glfwSwapInterval(1)` you were clamped to refresh (e.g., 240 Hz).
- **Scene is too cheap:** one cube, tiny shader, small texture ⇒ not texture-bound.
- **Magnification dominates:** bias only helps on minified texels.
- **Cache-friendly:** small textures stay in cache; bandwidth savings are tiny in this configuration.

When we move to a texture-heavy scene (bigger textures, more instances, higher resolution, anisotropy), the FPS/*GPU ms* impact becomes obvious.

9 Expected output

- A rotating textured cube, with correct front/back visibility (depth test).
- Press `]`: increase LOD bias (softer / lower texel density).
- Press `[`: decrease LOD bias (sharper / higher texel density).
- Console prints `FPS` and `LOD Bias`.

10 What's next (Day 3, Part 2 preview)

- Add ROI-based and/or automatic texel-density control (per-frame metric via derivatives + mipmapped reduction to a 1x1 average).
- Use the metric with a small controller (target density + hysteresis) to self-tune bias.
- Optional: GPU timer queries (ms/frame) for precise performance evidence.