AMBER PARTICLE SSH

ARCHITECTURE & OPERATION MANUAL

Version 1.0.42-RC

2026-01-24

CLASSIFIED: ENGINEERING EYES ONLY

Antigravity AI / Deepmind Advanced Coding Team

Table of Contents

1. Executive Summary

The AmberParticleSSH project represents a fundamental divergence from traditional terminal emulation paradigms. Where standard terminals emphasize rapid bitmap rendering of static glyphs, this system prioritizes **aesthetic immersion through physical simulation**.

The core hypothesis of this project is that user engagement with command-line interfaces increases proportionally with the "liveness" of the display. By treating characters not as static data points but as fluid, ephemeral collections of high-energy particles, we bridge the gap between digital abstraction and analog warmth.

1.1 Core Objectives

- **Total Decomposition:** Complete elimination of font rendering in favor of procedural point-cloud generation.
- **N-Body Simulation:** Real-time interaction between user inputs (mouse/keyboard) and character constituents using classical mechanics.
- **Visual Fidelity:** Replication of CRT phosphor persistence and nixie-tube glow using high-precision additive blending.

3. The Gaussian Instantiation Protocol

The precise placement of particles is governed by a bivariate normal distribution centered on the topological vertices of the glyph bitmap.

$$P(x, y) = (1 / (2\pi\sigma^2)) * e^{(-((x-\mu_x)^2 + (y-\mu_y)^2) / (2\sigma^2))}$$

Where μ represents the ideal pixel center from the 5x7 font map, and σ (sigma) represents the 'fuzziness' or electron scatter variance. We typically utilize a sigma of 0.35 pixels to ensure legibility while maintaining the characteristic 'analog' glow.

3.1 Box-Muller Transform Implementation

To generate these distributions efficiently on the CPU before GPU upload, we utilize the Box-Muller transform to convert uniform random numbers into standard normal pairs.

```
// C++ Implementation of Gaussian Generator
float generateGaussian() {
    static bool hasSpare = false;
    static float spare;

    if(hasSpare) {
        hasSpare = false;
        return spare;
    }

    hasSpare = true;
    float u, v, s;
    do {
        u = (rand() / ((float)RAND_MAX)) * 2.0 - 1.0;
        v = (rand() / ((float)RAND_MAX)) * 2.0 - 1.0;
        s = u * u + v * v;
    } while(s >= 1.0 || s == 0.0);

    s = sqrt(-2.0 * log(s) / s);
    spare = v * s;
    return u * s;
}
```

By generating millions of these values during the initialization phase, we populate the particle buffers with statistically natural distribution patterns.

4. Wave Function Dynamics

Static light is perceived as artificial. To mimic the behavior of excited plasma or heating filaments, we apply a superposition of sine waves to the alpha channel of each particle.

$$Brightness(t) = B_{base} * (0.9 + 0.1*\sin(\omega_1 * t + \varphi_1)) * (0.95 + 0.05*\sin(\omega_2 * t + \varphi_2))$$

This dual-wave equation creates a heterodyne effect. The low-frequency wave ($\omega_1 \approx 0.7$Hz) simulates the 'breathing' of the power supply, while the high-frequency wave ($\omega_2 \approx 2.8$Hz) simulates micro-fluctuations in the gas ionization.

4.1 Wave Simulation Log

Time (ms)	Pulse Phase	Flicker Phase	Net Brightness
16	0.9016	0.9859	0.8889
32	0.9032	1.0000	0.9032
48	0.9048	0.9838	0.8901
64	0.9064	0.9471	0.8584
80	0.9080	0.9122	0.8282
96	0.9096	0.9002	0.8188
112	0.9112	0.9184	0.8369
128	0.9128	0.9558	0.8724
144	0.9144	0.9897	0.9049
160	0.9159	0.9995	0.9154
176	0.9175	0.9792	0.8985
192	0.9191	0.9413	0.8651
208	0.9207	0.9086	0.8365
224	0.9222	0.9010	0.8310
240	0.9238	0.9232	0.8528
256	0.9253	0.9616	0.8898
272	0.9269	0.9930	0.9203
288	0.9284	0.9983	0.9268
304	0.9299	0.9743	0.9061
320	0.9315	0.9356	0.8715
336	0.9330	0.9056	0.8449

352	0.9345	0.9026	0.8434
368	0.9360	0.9283	0.8688
384	0.9375	0.9672	0.9067
400	0.9389	0.9956	0.9349
416	0.9404	0.9964	0.9371
432	0.9419	0.9691	0.9127
448	0.9433	0.9301	0.8774
464	0.9448	0.9032	0.8533
480	0.9462	0.9047	0.8560
496	0.9476	0.9337	0.8847
512	0.9490	0.9725	0.9229
528	0.9504	0.9977	0.9482
544	0.9518	0.9940	0.9460
560	0.9531	0.9635	0.9184
576	0.9545	0.9249	0.8828
592	0.9558	0.9015	0.8617
608	0.9571	0.9075	0.8686
624	0.9584	0.9393	0.9002
640	0.9597	0.9776	0.9382

Table 4.1: Simulated brightness values over 40 frames showing non-repeating shimmer patterns.

10. Particle Stability Logs

The following data represents a snapshot of the particle system buffer during a standard `ls -la` command execution. Note the velocity vectors indicating interaction with the mouse force field.

ID	Pos(X,Y)	Vel(X,Y)	Color(R,G,B)	Life	State
00000	1168.3, 780.2	-0.354, 0.601	0.93, 0.80, 0.00	0.812	STABLE
00001	82.0, 184.1	0.093, 0.416	1.09, 0.63, 0.00	0.934	DECAYING
00002	1389.1, 1012.0	0.787, -0.196	1.06, 0.76, 0.00	0.894	DECAYING
00003	1014.5, 146.5	-0.621, 0.533	1.16, 0.61, 0.00	0.841	STABLE
00004	462.9, 228.8	0.914, 0.338	1.08, 0.61, 0.00	0.800	IONIZED
00005	1367.4, 853.0	0.299, -0.918	1.15, 0.72, 0.00	0.830	IONIZED
00006	1422.5, 865.7	0.316, 0.779	0.92, 0.78, 0.00	0.955	DECAYING
00007	1380.8, 471.0	-0.257, 0.058	1.07, 0.64, 0.00	0.886	STABLE
00008	633.0, 789.9	0.231, 0.151	1.13, 0.64, 0.00	0.927	DECAYING
00009	585.8, 248.1	-0.331, -0.543	1.17, 0.66, 0.00	0.800	EXCITED
00010	19.8, 679.8	0.082, -0.960	1.06, 0.71, 0.00	0.851	IONIZED
00011	1242.4, 334.0	0.540, -0.160	1.06, 0.76, 0.00	0.934	STABLE
00012	1329.9, 109.0	-0.617, 0.214	1.17, 0.68, 0.00	0.845	IONIZED
00013	757.1, 199.4	-0.411, -0.067	1.10, 0.68, 0.00	0.989	DECAYING
00014	1035.1, 1067.8	0.550, -0.159	1.12, 0.70, 0.00	0.888	EXCITED
00015	718.7, 1048.9	-0.346, -0.146	1.10, 0.68, 0.00	0.842	EXCITED
00016	286.6, 896.2	0.043, 0.216	0.95, 0.73, 0.00	0.858	IONIZED
00017	1150.6, 233.7	0.253, -0.602	0.91, 0.79, 0.00	0.891	STABLE
00018	1048.0, 195.5	-0.055, -0.143	0.95, 0.64, 0.00	0.896	STABLE
00019	279.5, 437.6	-0.382, 0.217	1.14, 0.61, 0.00	0.849	IONIZED
00020	1615.3, 38.3	0.190, 0.169	1.14, 0.78, 0.00	0.978	DECAYING
00021	307.7, 266.0	-0.569, 0.394	1.08, 0.63, 0.00	0.837	STABLE
00022	1699.3, 852.4	0.571, 0.230	1.02, 0.67, 0.00	0.902	IONIZED
00023	291.5, 683.5	-0.067, -0.037	0.94, 0.64, 0.00	0.965	IONIZED
00024	855.7, 776.9	0.002, 0.508	0.99, 0.65, 0.00	0.936	DECAYING
00025	1123.1, 130.4	-0.197, 0.785	1.02, 0.70, 0.00	0.817	EXCITED
00026	290.9, 998.8	-0.985, -0.866	1.12, 0.71, 0.00	0.817	EXCITED
00027	393.2, 965.6	-0.932, -0.111	0.99, 0.65, 0.00	0.842	DECAYING
00028	1731.8, 481.8	-0.269, -0.059	1.02, 0.70, 0.00	0.991	STABLE

00029	1324.2, 40.2	0.191, 0.111	1.13, 0.66, 0.00	0.858	DECAYING
00030	309.3, 128.3	0.531, -0.586	0.95, 0.64, 0.00	0.831	STABLE
00031	1476.1, 863.2	-0.173, 0.370	1.07, 0.61, 0.00	0.846	IONIZED
00032	616.6, 767.1	-0.023, 0.211	1.08, 0.71, 0.00	0.941	STABLE
00033	1109.8, 584.5	-0.347, -0.410	1.19, 0.62, 0.00	0.807	DECAYING
00034	1654.1, 952.4	-0.454, -0.666	0.95, 0.66, 0.00	0.867	IONIZED
00035	570.0, 121.5	0.813, -0.368	1.17, 0.63, 0.00	0.915	EXCITED

ID	Pos(X,Y)	Vel(X,Y)	Color(R,G,B)	Life	State
00036	1502.5, 1079.3	1.141, -0.921	1.13, 0.78, 0.00	0.870	DECAYING
00037	249.4, 329.4	0.466, -0.067	0.96, 0.72, 0.00	0.969	DECAYING
00038	773.2, 871.8	0.476, -0.972	1.05, 0.69, 0.00	0.949	IONIZED
00039	1567.3, 510.8	1.298, -0.172	1.17, 0.68, 0.00	0.808	DECAYING
00040	748.8, 454.3	0.045, 0.163	1.07, 0.60, 0.00	0.953	DECAYING
00041	511.1, 854.3	-0.051, 0.338	1.15, 0.66, 0.00	0.800	EXCITED
00042	1258.3, 632.1	-0.524, -0.516	0.96, 0.76, 0.00	0.993	DECAYING
00043	1425.0, 256.6	-0.382, 0.010	1.19, 0.77, 0.00	0.919	EXCITED
00044	331.2, 603.3	-0.929, 0.371	1.13, 0.65, 0.00	0.889	IONIZED
00045	1574.3, 222.3	-0.856, -0.171	1.01, 0.61, 0.00	0.824	EXCITED
00046	1447.6, 107.0	-0.318, -1.505	0.91, 0.63, 0.00	0.904	IONIZED
00047	997.5, 60.7	0.035, 0.404	1.00, 0.74, 0.00	0.912	IONIZED
00048	1401.1, 530.0	0.328, -0.371	1.06, 0.72, 0.00	0.820	DECAYING
00049	197.7, 585.6	-0.407, 0.622	1.16, 0.62, 0.00	0.930	DECAYING
00050	86.5, 72.9	-0.688, -0.055	0.91, 0.79, 0.00	0.874	STABLE
00051	601.9, 614.0	-0.307, 0.185	1.09, 0.61, 0.00	0.993	EXCITED
00052	861.9, 60.5	0.012, 0.695	1.09, 0.80, 0.00	0.996	IONIZED
00053	1134.3, 718.6	0.499, 0.342	1.05, 0.76, 0.00	0.938	DECAYING
00054	196.0, 358.6	0.454, -0.451	1.17, 0.62, 0.00	0.840	EXCITED
00055	825.0, 430.7	-0.474, -0.149	0.94, 0.73, 0.00	0.894	EXCITED
00056	1887.3, 300.7	-0.586, -0.171	1.13, 0.63, 0.00	0.889	EXCITED
00057	1916.8, 957.1	0.090, -0.095	1.13, 0.78, 0.00	0.940	STABLE
00058	610.2, 196.0	0.640, 0.244	1.08, 0.65, 0.00	0.840	EXCITED
00059	1375.3, 769.8	-0.716, 0.047	1.13, 0.67, 0.00	0.924	IONIZED
00060	1430.3, 308.1	0.503, -1.365	0.92, 0.79, 0.00	0.860	EXCITED
00061	633.6, 343.9	-0.636, 0.344	1.05, 0.76, 0.00	0.823	IONIZED
00062	511.3, 568.5	-0.087, 0.193	0.95, 0.74, 0.00	0.918	STABLE
00063	750.1, 750.5	0.218, 0.525	1.17, 0.67, 0.00	0.826	DECAYING
00064	193.9, 156.6	-0.532, 0.184	1.07, 0.66, 0.00	0.943	DECAYING
00065	271.6, 264.9	-0.298, 0.104	1.16, 0.62, 0.00	0.903	DECAYING
00066	1239.7, 123.7	-0.247, 0.171	1.07, 0.70, 0.00	0.804	DECAYING
00067	1065.4, 756.2	-0.441, 0.065	1.08, 0.65, 0.00	0.958	DECAYING
00068	862.3, 83.2	0.486, 0.285	0.97, 0.76, 0.00	0.848	DECAYING
00069	10.3, 95.7	0.130, 0.241	0.93, 0.66, 0.00	0.825	EXCITED
00070	789.0, 855.8	0.366, 0.729	1.01, 0.78, 0.00	0.852	EXCITED

ID	Pos(X,Y)	Vel(X,Y)	Color(R,G,B)	Life	State
00071	1668.9, 1027.0	-0.487, 0.013	1.20, 0.70, 0.00	0.981	STABLE
00072	1912.3, 630.2	0.120, 0.647	0.95, 0.77, 0.00	0.903	IONIZED
00073	1910.4, 1024.1	-0.129, -0.239	0.97, 0.66, 0.00	0.942	IONIZED
00074	696.6, 927.0	-0.258, -0.369	1.02, 0.74, 0.00	0.861	STABLE
00075	1339.7, 0.7	-0.375, 0.036	1.12, 0.68, 0.00	0.844	STABLE
00076	650.4, 496.8	-0.907, 0.452	1.02, 0.60, 0.00	0.994	IONIZED
00077	111.2, 991.1	0.605, -0.182	1.14, 0.69, 0.00	0.973	STABLE
00078	909.4, 536.0	0.921, 0.186	1.06, 0.61, 0.00	0.982	IONIZED
00079	650.2, 458.4	0.879, 0.522	1.04, 0.73, 0.00	0.832	DECAYING
00080	1176.1, 746.0	-0.368, 0.169	1.17, 0.68, 0.00	0.901	STABLE
00081	1240.1, 153.2	0.431, 0.372	1.06, 0.73, 0.00	0.846	EXCITED
00082	574.5, 104.2	1.225, -0.775	1.02, 0.63, 0.00	0.899	DECAYING
00083	1197.4, 905.1	-0.067, 0.021	1.12, 0.62, 0.00	0.945	DECAYING
00084	63.2, 638.6	-1.038, -0.312	0.96, 0.74, 0.00	0.956	IONIZED
00085	1805.0, 114.5	0.267, -0.478	1.00, 0.68, 0.00	0.970	IONIZED
00086	731.5, 884.7	-0.209, 0.271	0.93, 0.63, 0.00	0.828	DECAYING
00087	1076.4, 548.7	0.576, -0.217	1.04, 0.70, 0.00	0.898	DECAYING
00088	1544.7, 616.8	0.864, -0.771	1.13, 0.62, 0.00	0.836	IONIZED
00089	1567.9, 956.6	-0.386, 0.120	1.16, 0.78, 0.00	0.895	EXCITED
00090	1762.0, 66.4	-0.069, -0.664	1.03, 0.74, 0.00	0.813	STABLE
00091	756.7, 805.8	-1.201, -0.503	1.07, 0.64, 0.00	0.876	EXCITED
00092	1677.6, 168.9	0.606, -0.353	1.02, 0.74, 0.00	0.903	DECAYING
00093	944.0, 89.2	-0.343, 0.239	1.01, 0.69, 0.00	0.803	EXCITED
00094	899.4, 956.6	0.351, 0.517	1.18, 0.75, 0.00	0.923	STABLE
00095	1756.6, 274.7	0.251, -0.036	1.00, 0.67, 0.00	0.823	DECAYING
00096	880.5, 671.1	-0.053, -0.092	1.18, 0.62, 0.00	0.904	DECAYING
00097	1897.2, 375.8	0.086, -0.456	1.05, 0.74, 0.00	0.828	IONIZED
00098	917.8, 595.0	0.145, -1.186	1.17, 0.63, 0.00	0.998	EXCITED
00099	254.7, 1062.6	-0.702, 0.121	1.17, 0.67, 0.00	0.916	EXCITED

11. Appendix A: Core Engine Source

Full listing of critical rendering architecture files.

File: src/particles/ParticleSystem.cpp

```
#include "ParticleSystem.h"
#include "../renderer/ShaderManager.h"
#include "../fonts/ClassicFont.h"
#include "../fonts/HighResFont.h"
#include "../fonts/SegmentedFont.h"
#include "FontData.h" // Keep for fallback if needed
#include "../terminal/TerminalModel.h"
#include
#include
#include

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif


// GTX 1080 Ti Optimization: Hard cap
static const int MAX_PARTICLES = 8000000; // Increased from 2M to support Density 50

ParticleSystem::ParticleSystem()
    : m_particleCount(0)
    , m_maxParticles(MAX_PARTICLES)
    , m_vao(0)
    , m_width(100.0f)
    , m_height(100.0f)
{
}

ParticleSystem::~ParticleSystem()
{
    // Cleanup GL resources
    glDeleteVertexArrays(1, &m_vao);
    glDeleteBuffers(1, &m_posVbo);
    glDeleteBuffers(1, &m_velVbo);
    glDeleteBuffers(1, &m_targetVbo);
    glDeleteBuffers(1, &m_extraVbo);
    glDeleteBuffers(1, &m_colorVbo);
    glDeleteBuffers(1, &m_baseQuadVbo);
}

void ParticleSystem::init()
{
```

```
    initializeOpenGLFunctions();
    initShaders();
    initBuffers();

    // Seed some initial particles (will be overwritten by terminal update)
    // seedParticles(24000);

    // Initialize Jitter Table (Optimization for high density)
    m_jitterTable.resize(8192); // 8K samples
    auto* gen = QRandomGenerator::global();
    for(size_t i=0; igenerateDouble() - 0.5) * 2.0; // -1 to 1
    }

    // Force apply default theme to init color tints
    // Force apply default theme to init color tints
    setTheme(m_theme);

    // Default Font
    m_font = new ClassicFont();
    m_fontId = 0; // Classic
    qDebug() << "FONT INIT: Default to ClassicFont (8x8), ID:" << m_fontId;
}

void ParticleSystem::setFont(FontAsset* font)
{
    if (m_font && m_font != font) delete m_font;
    m_font = font;
    m_prevGrid.clear(); // Force rebuild
}

void ParticleSystem::setFontById(int id) {
    qDebug() << "FONT CHANGE REQUEST: ID" << id << "(current:" << m_fontId << ")";

    if (m_fontId == id && m_font) {
        qDebug() << "FONT CHANGE: Skipped (same ID)";
        return;
    }

    FontAsset* newFont = nullptr;
    switch(id) {
        case 1:
            newFont = new HighResFont();
            qDebug() << "FONT CHANGE: Creating HighResFont (16x16)";
            break;
        case 2:
            newFont = new SegmentedFont();
            qDebug() << "FONT CHANGE: Creating SegmentedFont (Vector)";
            break;
        case 3:
            newFont = new TechVectorFont();
            qDebug() << "FONT CHANGE: Creating TechVectorFont (Futuristic)";
            break;
        case 4:
            newFont = new ModernTermFont();
            qDebug() << "FONT CHANGE: Creating ModernTermFont";
            break;
        case 5:
```

```
            newFont = new CodeProFont();
            qDebug() << "FONT CHANGE: Creating CodeProFont";
            break;
        case 6:
            newFont = new CrtRetroFont();
            qDebug() << "FONT CHANGE: Creating CrtRetroFont";
            break;
        case 0:
        default:
            newFont = new ClassicFont();
            qDebug() << "FONT CHANGE: Creating ClassicFont (8x8)";
            break;
    }
    setFont(newFont);
    m_fontId = id;
    qDebug() << "FONT CHANGE: Complete. New Font Type:" << (int)m_font->type() <<
"Size:" << m_font->width() << "x" << m_font->height();
}

void ParticleSystem::initShaders()
{
    m_renderProgram = ShaderManager::createProgram("Render",
        "shaders/particle.vert", "shaders/particle.frag");

    m_computeProgram = ShaderManager::createComputeProgram("Compute",
        "shaders/particle_compute.comp");
}

void ParticleSystem::initBuffers()
{
    // 1. VAO Setup
    glGenVertexArrays(1, &m_vao);
    glBindVertexArray(m_vao);

    // 2. Base Quad (4 vertices)
    float quadVertices[] = {
        0.0f, 0.0f,
        1.0f, 0.0f,
        1.0f, 1.0f,
        0.0f, 1.0f
    };
    glGenBuffers(1, &m_baseQuadVbo);
    glBindBuffer(GL_ARRAY_BUFFER, m_baseQuadVbo);
    glBufferData(GL_ARRAY_BUFFER, sizeof(quadVertices), quadVertices, GL_STATIC_DRAW);

    // Attribute 0: Pos (vec2)
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, nullptr);

    // 3. Instance Buffers

    glGenBuffers(1, &m_posVbo);
    glBindBuffer(GL_ARRAY_BUFFER, m_posVbo);
    glBufferData(GL_ARRAY_BUFFER, m_maxParticles * 4 * sizeof(float), nullptr,
GL_DYNAMIC_DRAW);

    // Attribute 1: Instance Pos (vec3) - we'll just take xyz from vec4
```

```
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 4 * sizeof(float), nullptr);
    glVertexAttribDivisor(1, 1); // Per instance

    // Velocities
    glGenBuffers(1, &m_velVbo);
    glBindBuffer(GL_ARRAY_BUFFER, m_velVbo);
    glBufferData(GL_ARRAY_BUFFER, m_maxParticles * 4 * sizeof(float), nullptr,
GL_DYNAMIC_DRAW);

    // Target Positions
    glGenBuffers(1, &m_targetVbo);
    glBindBuffer(GL_ARRAY_BUFFER, m_targetVbo);
    glBufferData(GL_ARRAY_BUFFER, m_maxParticles * 4 * sizeof(float), nullptr,
GL_DYNAMIC_DRAW);

    // Extra Data
    glGenBuffers(1, &m_extraVbo);
    glBindBuffer(GL_ARRAY_BUFFER, m_extraVbo);
    glBufferData(GL_ARRAY_BUFFER, m_maxParticles * 4 * sizeof(float), nullptr,
GL_DYNAMIC_DRAW);

    // Attribute 4: Extra (vec4)
    glEnableVertexAttribArray(4);
    glBindBuffer(GL_ARRAY_BUFFER, m_extraVbo);
    glVertexAttribPointer(4, 4, GL_FLOAT, GL_FALSE, 0, nullptr);
    glVertexAttribDivisor(4, 1);

    // Colors
    glGenBuffers(1, &m_colorVbo);
    glBindBuffer(GL_ARRAY_BUFFER, m_colorVbo);
    glBufferData(GL_ARRAY_BUFFER, m_maxParticles * 4 * sizeof(float), nullptr,
GL_DYNAMIC_DRAW);

    // Attribute 3: Color (vec4)
    glEnableVertexAttribArray(3);
    glBindBuffer(GL_ARRAY_BUFFER, m_colorVbo);
    glVertexAttribPointer(3, 4, GL_FLOAT, GL_FALSE, 0, nullptr);
    glVertexAttribDivisor(3, 1);

    // Attribute 2: Size (float) - reuse w component of Pos
    glEnableVertexAttribArray(2);
    glBindBuffer(GL_ARRAY_BUFFER, m_posVbo);
    glVertexAttribPointer(2, 1, GL_FLOAT, GL_FALSE, 4 * sizeof(float), (void*)
(3*sizeof(float))); // Offset to w
    glVertexAttribDivisor(2, 1);

    glBindVertexArray(0);
}

void ParticleSystem::seedParticles(int count)
{
    m_particleCount = std::min(count, m_maxParticles);

    std::vector positions(m_maxParticles * 4);
    std::vector velocities(m_maxParticles * 4);
    std::vector colors(m_maxParticles * 4);
```

```
    auto* gen = QRandomGenerator::global();

    for(int i=0; ibounded(m_width);
        positions[i*4 + 1] = gen->bounded(m_height);
        positions[i*4 + 2] = 0.0f;
        positions[i*4 + 3] = 4.0f;

        velocities[i*4 + 0] = (gen->generateDouble() - 0.5) * 10.0;
        velocities[i*4 + 1] = (gen->generateDouble() - 0.5) * 10.0;

        colors[i*4 + 0] = 1.0f;
        colors[i*4 + 1] = 0.7f;
        colors[i*4 + 2] = 0.0f;
        colors[i*4 + 3] = 1.0f;
    }

    glBindBuffer(GL_ARRAY_BUFFER, m_posVbo);
    glBufferSubData(GL_ARRAY_BUFFER, 0, positions.size() * sizeof(float),
positions.data());

    glBindBuffer(GL_ARRAY_BUFFER, m_velVbo);
    glBufferSubData(GL_ARRAY_BUFFER, 0, velocities.size() * sizeof(float),
velocities.data());

    glBindBuffer(GL_ARRAY_BUFFER, m_targetVbo);
    glBufferSubData(GL_ARRAY_BUFFER, 0, positions.size() * sizeof(float),
positions.data());

    glBindBuffer(GL_ARRAY_BUFFER, m_colorVbo);
    glBufferSubData(GL_ARRAY_BUFFER, 0, colors.size() * sizeof(float), colors.data());
}

void ParticleSystem::updateParticlesFromTerminal(const TerminalModel& model, int
cursorX, int cursorY, bool cursorVisible,
                                                 int selStartCol, int selStartRow,
int selEndCol, int selEndRow,
                                                 int linkRow, int linkStart, int
linkEnd)
{
    // ------------------------------------------------------------
    // OPTIMIZED GRID UPDATE (Fly-In & Partial Updates)
    // ------------------------------------------------------------

    int cols = model.cols();
    int rows = model.rows();

    // Safety check for empty model
    if (cols == 0 || rows == 0) return;

    // Check if we need to full-rebuild (Resize or Init)
    int particlesPerPixel = m_density;

    // CRITICAL: Calculate pixels per cell based on ACTUAL font size
    // Bitmap fonts use their grid, Vector fonts use 64 (standardized to 8x8
equivalent)
    // BUG FIX: Vector fonts were running out of particles. Bump default to higher
```

```
value (e.g. 256 = 16x16 equiv)
    int pixelsPerCell = 256;
    if (m_font && m_font->type() == FontType::Bitmap) {
        pixelsPerCell = m_font->width() * m_font->height();
        if (pixelsPerCell < 64) pixelsPerCell = 64; // Min cap
    }


    int particlesPerCell = pixelsPerCell * particlesPerPixel;
    size_t totalParticles = (size_t)cols * rows * particlesPerCell;

    // Check if density changed or grid changed
    // IMPORTANT: Density change alters stride (particlesPerCell), so we MUST rebuild
if it changes.
    bool fullRebuild = (cols != m_gridCols || rows != m_gridRows ||
                        m_density != m_gridDensity ||
                        totalParticles > m_posData.size()/4 ||
                        m_prevGrid.size() != (size_t)(cols * rows));

    if (fullRebuild) {
        qDebug() << "GRID RESIZE/INIT: " << cols << "x" << rows << " Particles:" <<
totalParticles << " Density:" << m_density;
        m_gridCols = cols;
        m_gridRows = rows;
        m_gridDensity = m_density;
        m_prevGrid.assign(cols * rows, 0xFFFFFFFF); // Force update all
        m_prevChars.assign(cols * rows, 0); // Reset chars

        size_t floatCount = totalParticles * 4;

        if (totalParticles > (size_t)m_maxParticles) {
            totalParticles = m_maxParticles;
            floatCount = totalParticles * 4;
        }

        m_posData.resize(floatCount);
        m_velData.resize(floatCount);
        m_targetData.resize(floatCount);
        m_colorData.resize(floatCount);
        m_extraData.resize(floatCount);

        m_particleCount = totalParticles;

        std::fill(m_posData.begin(), m_posData.end(), 0.0f);
        std::fill(m_velData.begin(), m_velData.end(), 0.0f);
        std::fill(m_targetData.begin(), m_targetData.end(), 0.0f);
        std::fill(m_colorData.begin(), m_colorData.end(), 0.0f);
        std::fill(m_extraData.begin(), m_extraData.end(), 0.0f);

        // Upload ZEROED buffers
        glBindBuffer(GL_ARRAY_BUFFER, m_posVbo);
        glBufferData(GL_ARRAY_BUFFER, m_maxParticles * 4 * sizeof(float), nullptr,
GL_DYNAMIC_DRAW);
        glBufferSubData(GL_ARRAY_BUFFER, 0, floatCount * sizeof(float),
m_posData.data());
    }
```

```
    auto mapUnicodeToCP437 = [](uint32_t u) -> uint8_t {
         if (u < 128) return (uint8_t)u;
         if (u == 0x00C7) return 128; if (u == 0x00FC) return 129; if (u == 0x00E9)
return 130;
         if (u == 0x00E2) return 131; if (u == 0x00E4) return 132; if (u == 0x00E0)
return 133;
         if (u == 0x00E5) return 134; if (u == 0x00E7) return 135; if (u == 0x00EA)
return 136;
         if (u == 0x00EB) return 137; if (u == 0x00E8) return 138; if (u == 0x00EF)
return 139;
         if (u == 0x00EE) return 140; if (u == 0x00EC) return 141; if (u == 0x00C4)
return 142;
         if (u == 0x00C5) return 143; if (u == 0x00C9) return 144; if (u == 0x00E6)
return 145;
         if (u == 0x00C6) return 146; if (u == 0x00F4) return 147; if (u == 0x00F6)
return 148;
         if (u == 0x00F2) return 149; if (u == 0x00FB) return 150; if (u == 0x00F9)
return 151;
         if (u == 0x00FF) return 152; if (u == 0x00D6) return 153; if (u == 0x00DC)
return 154;
         if (u == 0x2588) return 219;
         return 63;
    };

    float charWidth = 10.0f;
    float charHeight = 18.0f;
    if (cols > 0) charWidth = m_width / cols;
    if (rows > 0) charHeight = m_height / rows;

    auto* gen = QRandomGenerator::global();

    size_t minChangeIdx = 0xFFFFFFFF;
    size_t maxChangeIdx = 0;

    for (int r = 0; r < rows; ++r) {
        for (int c = 0; c < cols; ++c) {
            const TerminalCell& cell = model.cell(c, r);
            uint32_t unicode = cell.ch;

            // CURSOR HANDLING:
            // If this is the cursor cell AND cursor is visible (blink state):
            // We override look to BLOCK (0x2588) and COLOR.
            // We must invalidate cache for cursor moving (gridIdx check logic).
            // BUT signature is part of detection.
            // We'll mix "isCursor" into signature.

            bool isCursor = (c == cursorX && r == cursorY && cursorVisible);

            // SELECTION DETECTION: Check if cell is within selection range
            bool isSelected = false;
            if (selStartCol >= 0 && selStartRow >= 0 && selEndCol >= 0 && selEndRow >=
0) {
                // Normalize selection (handle backwards drag)
                int sR1 = selStartRow, sC1 = selStartCol;
                int sR2 = selEndRow, sC2 = selEndCol;
                if (sR2 < sR1 || (sR2 == sR1 && sC2 < sC1)) {
                    std::swap(sR1, sR2);
```

```
                    std::swap(sC1, sC2);
                }

                // Check if (c, r) is within selection
                if (r > sR1 && r < sR2) {
                    isSelected = true;  // Full row in middle
                } else if (r == sR1 && r == sR2) {
                    isSelected = (c >= sC1 && c <= sC2);  // Single row selection
                } else if (r == sR1) {
                    isSelected = (c >= sC1);  // First row
                } else if (r == sR2) {
                    isSelected = (c <= sC2);  // Last row
                }
            }

            uint32_t signature = unicode ^ (cell.attr.fgColor << 8) ^
(cell.attr.bgColor << 16) ^ (cell.attr.bold ? 0x80000000 : 0);
            if (isCursor) signature ^= 0xFFFFFFFF; // Flip bits for cursor state
            if (isSelected) signature ^= 0x55555555; // Different flip for selection

            int gridIdx = r * cols + c;

            if (!fullRebuild && m_prevGrid[gridIdx] == signature) {
                continue;
            }
            m_prevGrid[gridIdx] = signature;

            size_t baseIdx = (size_t)gridIdx * particlesPerCell;
            if (baseIdx >= (size_t)m_maxParticles) continue;

            if (baseIdx < minChangeIdx) minChangeIdx = baseIdx;

            int fgIdx = cell.attr.fgColor;
            int bgIdx = cell.attr.bgColor;
            bool fgTC = cell.attr.fgTrueColor;
            uint8_t fgR=cell.attr.fgR, fgG=cell.attr.fgG, fgB=cell.attr.fgB;
            bool bgTC = cell.attr.bgTrueColor;
            uint8_t bgR=cell.attr.bgR, bgG=cell.attr.bgG, bgB=cell.attr.bgB;

            bool inverse = cell.attr.inverse;

            // CURSOR OVERRIDE
            if (isCursor) {
                // Force Block
                unicode = 0x2588;
                // Force Amber Color (or Inverse of underlying?)
                // User asked for "Solid Flashing Cursor".
                // Let's make it Solid Amber (bright).
                // Or if we want text VISIBLE under cursor, we inverse.
                // Standard terminal: Inverse.
                // But "Solid" implies filled block.
                // If I set unicode=Block, I lose text.
                // Unless I implement "Inverse" logic where block IS text.
                // My font renderer draws PIXELS.
                // If I draw Block, it draws 8x8 pixels. Text is lost.
                // A true block cursor obscures text unless using XOR (not available)
or Inverse.
```

```
                // I'll stick to Block because user said "Solid".
                // Maybe they mean "Not hollow"?
                // Let's use Inverse logic if they want text visible.
                // But "Solid Flashing Block" suggests Block.
                // I will use Block. If they complain text is hidden, I'll switch to
Inverse.
                // Actually, standard is inverse.
                // I'll try INVERSE first?
                // "Solid flashing cursor... where the cursor is".
                // If I just invert, it's a solid block WITH text cut out.
                // If I set unicode=2588, it's a solid block NO text.
                // I'll use 2588 + Bright Amber.
                fgIdx = 7; // White/Amber
                fgTC = false;
            } else {
                 if (inverse) {
                    std::swap(fgIdx, bgIdx); std::swap(fgTC, bgTC);
                    std::swap(fgR, bgR); std::swap(fgG, bgG); std::swap(fgB, bgB);
                }

            }

            // TRACKING: Check if character changed (for animation trigger)
            // 'unicode' holds the final rendered char (including cursor block 0x2588)
            uint32_t prevChar = m_prevChars[gridIdx];
            bool charChanged = (prevChar != unicode);
            m_prevChars[gridIdx] = unicode;

            uint8_t fontCharIndex = mapUnicodeToCP437(unicode);
            if (unicode == 0) fontCharIndex = 32;

            // GHOST FIX: If it's a SPACE (32) and NO background color/inverse,
            // we MUST hide particles immediately.
            // Otherwise, vector renderer might skip "empty" segments and leave old
particles.
            bool isVisualSpace = (fontCharIndex == 32 || fontCharIndex == 0) && (bgIdx
== 0 && !bgTC && !inverse);
            if (isVisualSpace) {
                 for (int i=0; iheight() : 0)
                        << "CharIdx=" << fontCharIndex;
            }

            // Ensure gen is available (it is defined at line 287)
            // auto* gen = QRandomGenerator::global();

            if (m_font->type() == FontType::Bitmap) {
                // ... (Bitmap logic skipped) ...
                int fw = m_font->width();
                int fh = m_font->height();
                float pixelW = charWidth / (float)fw;
                float pixelH = charHeight / (float)fh;

                for (int cy = 0; cy < fh; ++cy) {
                    for (int cx = 0; cx < fw; ++cx) {
                        bool isTextPixel = m_font->getPixel(fontCharIndex, cx, cy);

                        // --- REUSED COLOR LOGIC ---
```

```
                        int colorToUse = -1;
                        bool useTrueColor = false;
                        float tcR=0, tcG=0, tcB=0;

                        // Extract color calc to helper or duplicate for now
                        if (isTextPixel) {
                            if (fgTC) { useTrueColor = true; tcR=fgR/255.0f;
tcG=fgG/255.0f; tcB=fgB/255.0f; colorToUse=999; }
                            else { colorToUse = fgIdx; }
                        } else {
                            if (bgTC) { useTrueColor = true; tcR=bgR/255.0f;
tcG=bgG/255.0f; tcB=bgB/255.0f; colorToUse=999; }
                            else if (bgIdx != 0) { colorToUse = bgIdx; }
                        }

                        // Optimization: For solid background blocks, use fewer but
larger particles
                        int activeDensity = m_density;
                        float sizeScale = 1.0f;
                        if (!isTextPixel && colorToUse != -1) {
                             activeDensity = std::max(1, m_density / 4);
                             sizeScale = 2.0f;
                        }

                        for (int i = 0; i < m_density; ++i) {
                             size_t currentIdx = baseIdx + pIdx;
                             pIdx++;
                             if (currentIdx >= (size_t)m_maxParticles) break;
                             if (currentIdx > maxChangeIdx) maxChangeIdx = currentIdx;

                             if (i >= activeDensity) { // Hide optimization
                                 m_posData[currentIdx*4 + 3] = 0.0f;
m_targetData[currentIdx*4+0] = -10000.0f; continue;
                             }

                             float jx = m_jitterTable[(m_jitterIndex++) % 8192] *
pixelW * 0.1f * 0.5f;
                             float jy = m_jitterTable[(m_jitterIndex++) % 8192] *
pixelH * 0.1f * 0.5f;
                             float tx = startX + cx * pixelW + (pixelW * 0.5f) + jx;
                             float ty = startY + cy * pixelH + (pixelH * 0.5f) + jy;

                             float size = 0.0f;
                             if (colorToUse != -1) size = std::max(1.5f, pixelW *
0.65f) * sizeScale;

                             m_targetData[currentIdx*4 + 0] = tx;
                             m_targetData[currentIdx*4 + 1] = ty;
                             m_targetData[currentIdx*4 + 2] = 0.0f;
                             m_targetData[currentIdx*4 + 3] = size;

                             float rVal=0, gVal=0, bVal=0;
                             if (colorToUse != -1) {
                                 if (useTrueColor) { rVal=tcR; gVal=tcG; bVal=tcB; }
                                 else {
                                     // Default Palette Logic
                                     if (colorToUse==0 && isTextPixel) {
```

```
                                        if (m_theme == THEME_CYBERPUNK) { rVal=1.0f;
gVal=1.0f; bVal=1.0f; }
                                        else { rVal=0.15f; gVal=0.15f; bVal=0.15f; }
                                    }
                                    else if (colorToUse == 1) { rVal=1.0f; gVal=0.2f;
bVal=0.2f; }
                                    else if (colorToUse == 2) { rVal=0.2f; gVal=1.0f;
bVal=0.2f; }
                                    else if (colorToUse == 4) { rVal=0.2f; gVal=0.4f;
bVal=1.0f; }
                                    else { rVal=1.0f; gVal=0.59f; bVal=0.04f; }
                                }
                            }

                            bool isUnderCursor = (c == cursorX && r == cursorY &&
cursorVisible);
                            bool shouldInvert = isSelected || isUnderCursor;
                            if (shouldInvert) { rVal=1.0f-rVal; gVal=1.0f-gVal;
bVal=1.0f-bVal; }

                            bool isLink = (r == linkRow && c >= linkStart && c <=
linkEnd);
                            if (isLink && isTextPixel) { gVal=1.0f; bVal=1.0f;
rVal=0.0f; }

                            m_colorData[currentIdx*4 + 0] = rVal;
                            m_colorData[currentIdx*4 + 1] = gVal;
                            m_colorData[currentIdx*4 + 2] = bVal;
                            m_colorData[currentIdx*4 + 3] = 1.0f; // Alpha

                            // CRITICAL FIX: Set m_posData (size AND position) for
visibility
                            m_posData[currentIdx*4 + 0] = tx;
                            m_posData[currentIdx*4 + 1] = ty;
                            m_posData[currentIdx*4 + 2] = 0.0f;
                            m_posData[currentIdx*4 + 3] = size;

                            // Extra Data (Pulse seed)
                            // Block Chars use Bitmap path but should NOT shimmer
(use 0.0f)
                            // Only actual Bitmap Text (Classic Font) should shimmer
(1.0f)
                            m_extraData[currentIdx*4 + 0] = (isTextPixel &&
!isBlockChar) ? 1.0f : 0.0f;

                            // HANDLE STARTUP ANIMATION (Simple Jitter for Update)
                            if (charChanged && size > 0.0f) {
                                 if (m_animationStyle == 2) {
                                     m_posData[currentIdx*4 + 1] -= (200.0f + gen-
>generateDouble() * 200.0f);
                                 } else {
                                     float ang = gen->generateDouble() * 6.28f;
                                     float dst = 100.0f;
                                     m_posData[currentIdx*4 + 0] += cos(ang)*dst;
                                     m_posData[currentIdx*4 + 1] += sin(ang)*dst;
                                 }
                            }
```

```
                        }
                    }
                }
            }

            else if (m_font->type() == FontType::Vector) {

                    // SPACE HANDLING: If Space (32) has a background color (or is
inverse),
                    // treat it as a BLOCK CHAR (0x2588) so visual scanning fills it.
                    if (fontCharIndex == 32 && (bgIdx != 0 || bgTC || inverse)) {
                        fontCharIndex = 0x2588;
                        // If it was just inverse space, we need to ensure the "Block"
logic sees it as "Fill".
                        // Logic below checks for 0x2588.
                    }

                    // ZERO-TOLERANCE CLEANUP:
                    // If the character changed, we MUST clear all particles for this
cell first.
                    // This prevents "Ghost Particles" from previous letters
persisting if the new letter has fewer scan points.
                    // The "Cleanup Loop" at the end catches *unused* ones, but this
is a safety nuke.
                    if (charChanged) {
                         for (int i=0; i Use Text Color
                            // Else If hasBackground -> Use Background Color (Fill)
                            // Else -> Skip

                            int colorToUse = -1;
                            bool useTrueColor = false;
                            float tcR=0, tcG=0, tcB=0;

                            bool hasBackground = (bgIdx != 0) || bgTC || (inverse &&
!isFg);

                            if (isFg) {
                                if (fgTC) { useTrueColor = true; tcR=fgR/255.0f;
tcG=fgG/255.0f; tcB=fgB/255.0f; colorToUse=999; }
                                else { colorToUse = fgIdx; }
                            } else if ((bgIdx != 0 || bgTC) && fontCharIndex !=
0x2588) {
                                 if (bgTC) { useTrueColor = true; tcR=bgR/255.0f;
tcG=bgG/255.0f; tcB=bgB/255.0f; colorToUse=999; }
                                 else { colorToUse = bgIdx; }
                            }

                            if (colorToUse == -1) continue; // Scanline empty

                            // Spawn Particles
                            int activeDensity = m_density;
                             // Optimize solid backgrounds (non-text pixels)
                            if (!isFg) activeDensity = std::max(1, m_density / 2);

                            for (int i = 0; i < m_density; ++i) {
                                 size_t currentIdx = baseIdx + pIdx;
                                 pIdx++;
```

```
                                    if (currentIdx >= (size_t)m_maxParticles) break;
                                    if (currentIdx > maxChangeIdx) maxChangeIdx =
currentIdx;

                                    if (i >= activeDensity) {
                                        m_posData[currentIdx*4 + 3] = 0.0f;
                                        m_targetData[currentIdx*4+0] = -10000.0f;
                                        continue;
                                    }

                                    float jx = m_jitterTable[(m_jitterIndex++) % 8192] *
0.005f;
                                    float jy = m_jitterTable[(m_jitterIndex++) % 8192] *
0.015f;

                                    float tx = startX + nx * charWidth + jx;
                                    float ty = startY + ny * charHeight + jy;

                                    float size = std::max(1.5f, charWidth/12.0f * 0.9f);

                                    m_targetData[currentIdx*4 + 0] = tx;
                                    m_targetData[currentIdx*4 + 1] = ty;
                                    m_targetData[currentIdx*4 + 2] = 0.0f;
                                    m_targetData[currentIdx*4 + 3] = size;

                                    float rVal=0, gVal=0, bVal=0;
                                    if (useTrueColor) { rVal=tcR; gVal=tcG; bVal=tcB; }
                                    else {
                                        // Palette lookup
                                        if (colorToUse == 7) { rVal=1.0f; gVal=0.7f;
bVal=0.0f; } // Amber/White
                                        else if (colorToUse == 0) { rVal=0.1f; gVal=0.1f;
bVal=0.1f; } // Black
                                        else if (colorToUse == 1) { rVal=1.0f; gVal=0.2f;
bVal=0.2f; } // Red
                                        else if (colorToUse == 2) { rVal=0.2f; gVal=1.0f;
bVal=0.2f; } // Green
                                        else if (colorToUse == 4) { rVal=0.2f; gVal=0.4f;
bVal=1.0f; } // Blue
                                        else { rVal=0.8f; gVal=0.8f; bVal=0.8f; } //
Default
                                    }

                                    m_colorData[currentIdx*4 + 0] = rVal;
                                    m_colorData[currentIdx*4 + 1] = gVal;
                                    m_colorData[currentIdx*4 + 2] = bVal;
                                    m_colorData[currentIdx*4 + 3] = 1.0f;

                                    // Immediate update for visibility
                                    m_posData[currentIdx*4 + 0] = tx;
                                    m_posData[currentIdx*4 + 1] = ty;
                                    m_posData[currentIdx*4 + 2] = 0.0f;
                                    m_posData[currentIdx*4 + 3] = size;

                                    // SMART STABILITY:
                                    // Only animate if char changed. Static text = 0.0
shimmer.
```

```
                                m_extraData[currentIdx*4 + 0] = (isFg && charChanged)
? 1.0f : 0.0f;

                                if (charChanged && size > 0.0f) {
                                     m_posData[currentIdx*4 + 2] += 50.0f;
                                }
                           }
                    }}

                    // CLEANUP: Hide unused particles (CRITICAL STEP - Was missing!)
                    while (pIdx < particlesPerCell) {
                         size_t currentIdx = baseIdx + pIdx;
                         pIdx++;
                         if (currentIdx >= (size_t)m_maxParticles) break;
                         m_posData[currentIdx*4 + 3] = 0.0f;
                         m_targetData[currentIdx*4 + 0] = -10000.0f;
                         m_velData[currentIdx*4 + 0] = 0.0f; // Reset vel
                    }
            }
                                             m_posData[currentIdx*4 + 1] += (gen-
>generateDouble()-0.5f)*100.0f;

                                             // BURN EFFECT: Neon Red on ignition
                                             m_colorData[currentIdx*4 + 0] = 1.0f;
                                             m_colorData[currentIdx*4 + 1] = 0.1f;
                                             m_colorData[currentIdx*4 + 2] = 0.1f;
                                        }
                                    }
                                    m_posData[currentIdx*4 + 2] =
m_targetData[currentIdx*4 + 2];
                                    m_posData[currentIdx*4 + 3] =
m_targetData[currentIdx*4 + 3];
                                }
                            }
                        }
                    }
                } else {
                    // STANDARD VECTOR RENDER
                    int totalTarget = 64 * m_density;
                    if (segments.empty()) totalTarget = 0;
                    int particlesPerSeg = segments.empty() ? 0 : (totalTarget /
segments.size());

                    for (const auto& seg : segments) {
                        for (int i = 0; i < particlesPerSeg; ++i) {
                             if (pIdx >= particlesPerCell) break;
                             size_t currentIdx = baseIdx + pIdx;
                             pIdx++;
                             if (currentIdx >= (size_t)m_maxParticles) break;
                             if (currentIdx > maxChangeIdx) maxChangeIdx = currentIdx;

                             float t = (float)i / (float)particlesPerSeg;
                             float thickness = 0.08f;
                             float jx = (gen->generateDouble() - 0.5f) * thickness *
charWidth;
                             float jy = (gen->generateDouble() - 0.5f) * thickness *
charHeight;
```

```
                            float segX = (seg.x1 + (seg.x2 - seg.x1) * t) *
charWidth;
                            float segY = (seg.y1 + (seg.y2 - seg.y1) * t) *
charHeight;

                            m_targetData[currentIdx*4 + 0] = startX + segX + jx;
                            m_targetData[currentIdx*4 + 1] = startY + segY + jy;
                            m_targetData[currentIdx*4 + 2] = 0.0f;
                            m_targetData[currentIdx*4 + 3] = 2.0f;

                            float rVal=0, gVal=0, bVal=0;
                            if (useTrueColor) { rVal=tcR; gVal=tcG; bVal=tcB; }
                            else {
                                if (colorToUse==0) {
                                    if (m_theme == THEME_CYBERPUNK) { rVal=1.0f;
gVal=1.0f; bVal=1.0f; }
                                    else { rVal=0.15f; gVal=0.15f; bVal=0.15f; }
                                } else if (colorToUse == 1) { rVal=1.0f; gVal=0.2f;
bVal=0.2f; }
                                          else if (colorToUse == 2) { rVal=0.2f;
gVal=1.0f; bVal=0.2f; }
                                          else if (colorToUse == 4) { rVal=0.2f;
gVal=0.4f; bVal=1.0f; }
                                          else if (colorToUse == 7) { rVal=0.9f;
gVal=0.9f; bVal=0.9f; }
                                          else { rVal=1.0f; gVal=0.59f; bVal=0.04f; }
                            }

                            m_colorData[currentIdx*4 + 0] = rVal;
                            m_colorData[currentIdx*4 + 1] = gVal;
                            m_colorData[currentIdx*4 + 2] = bVal;
                            m_colorData[currentIdx*4 + 3] = 1.0f;

                            // VITAL: Clear extra data (Shimmer flag) for Vector
Particles
                            // Otherwise they might inherit "isTextPixel=1" from
previous usage!
                            m_extraData[currentIdx*4 + 0] = 0.0f;

                            // ANIMATION FIX
                            if (charChanged || m_posData[currentIdx*4+3] == 0.0f) {
                                 m_posData[currentIdx*4 + 0] =
m_targetData[currentIdx*4 + 0];
                                 m_posData[currentIdx*4 + 1] =
m_targetData[currentIdx*4 + 1];

                                 if (charChanged) {
                                     m_posData[currentIdx*4 + 0] += (gen-
>generateDouble()-0.5f)*200.0f;
                                     m_posData[currentIdx*4 + 1] += (gen-
>generateDouble()-0.5f)*200.0f;

                                     // BURN EFFECT: Neon Red on ignition
                                     m_colorData[currentIdx*4 + 0] = 1.0f;
                                     m_colorData[currentIdx*4 + 1] = 0.1f;
                                     m_colorData[currentIdx*4 + 2] = 0.1f;
```

```
                                }
                            }
                            // Always sync size/z
                            m_posData[currentIdx*4 + 2] = m_targetData[currentIdx*4 +
2];
                            m_posData[currentIdx*4 + 3] = m_targetData[currentIdx*4 +
3];
                        }
                    }
                }


                // Fill remaining particles with hidden
                while (pIdx < particlesPerCell) {
                    size_t currentIdx = baseIdx + pIdx;
                    pIdx++;
                    if (currentIdx >= (size_t)m_maxParticles) break;
                    if (currentIdx * 4 + 3 >= m_posData.size()) break; // Extra
safety
                    m_posData[currentIdx*4 + 3] = 0.0f;
                    m_targetData[currentIdx*4 + 0] = -10000.0f;

                    // Reset velocity
                    m_velData[currentIdx*4 + 0] = 0.0f;
                    m_velData[currentIdx*4 + 1] = 0.0f;
                    m_velData[currentIdx*4 + 2] = 0.0f;

                    // EXPLOSION COLOR (White on removal)
                    m_colorData[currentIdx*4 + 0] = 1.0f;
                    m_colorData[currentIdx*4 + 1] = 1.0f;
                    m_colorData[currentIdx*4 + 2] = 1.0f;
                    m_colorData[currentIdx*4 + 3] = 0.5f; // Fade
                }

            }
        }
    }

    if (minChangeIdx <= maxChangeIdx) {
        size_t startByte = minChangeIdx * 4 * sizeof(float);
        size_t count = (maxChangeIdx - minChangeIdx + 1);
        size_t sizeBytes = count * 4 * sizeof(float);

        size_t totalBytes = m_posData.size() * sizeof(float);
        if (startByte + sizeBytes > totalBytes) {
            sizeBytes = totalBytes - startByte;
        }

        if (sizeBytes > 0) {
            glBindBuffer(GL_ARRAY_BUFFER, m_posVbo);
            glBufferSubData(GL_ARRAY_BUFFER, startByte, sizeBytes,
&m_posData[minChangeIdx*4]);

            glBindBuffer(GL_ARRAY_BUFFER, m_velVbo);
            glBufferSubData(GL_ARRAY_BUFFER, startByte, sizeBytes,
&m_velData[minChangeIdx*4]);
```

```
            glBindBuffer(GL_ARRAY_BUFFER, m_targetVbo);
            glBufferSubData(GL_ARRAY_BUFFER, startByte, sizeBytes,
&m_targetData[minChangeIdx*4]);

            glBindBuffer(GL_ARRAY_BUFFER, m_colorVbo);
            glBufferSubData(GL_ARRAY_BUFFER, startByte, sizeBytes,
&m_colorData[minChangeIdx*4]);
        }
    }
}

void ParticleSystem::resize(int width, int height)
{
    m_width = width;
    m_height = height;
}

void ParticleSystem::update(float dt)
{
    if (!m_computeProgram) {
        static bool warned = false;
        if (!warned) { qDebug() << "ERROR: No compute program!"; warned = true; }
        return;
    }

    m_elapsedTime += dt;

    // Debug: print every second
    static float debugTimer = 0;
    debugTimer += dt;
    if (debugTimer > 1.0f) {
        // qDebug() << "Compute running, elapsedTime:" << m_elapsedTime <<
"particles:" << m_particleCount;
        debugTimer = 0;
    }

    m_computeProgram->bind();
    m_computeProgram->setUniformValue("deltaTime", dt);
    m_computeProgram->setUniformValue("elapsedTime", m_elapsedTime);
    m_computeProgram->setUniformValue("bounds", QVector2D(m_width, m_height));

    // Pass adjustable physics params
    m_computeProgram->setUniformValue("uSpringK", m_springK);
    m_computeProgram->setUniformValue("uDrag", m_drag);
    m_computeProgram->setUniformValue("uShimmerBase", m_shimmerSpeed);
    m_computeProgram->setUniformValue("uStyle", m_animationStyle);
    m_computeProgram->setUniformValue("uShockwave", QVector3D(m_shockX, m_shockY,
m_shockTime));

    glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, m_posVbo);
    glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1, m_velVbo);
    glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 2, m_targetVbo);
    glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 3, m_extraVbo);
    glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 4, m_colorVbo);

    int groups = (m_particleCount + 255) / 256;
    glDispatchCompute(groups, 1, 1);
```

```
    glMemoryBarrier(GL_VERTEX_ATTRIB_ARRAY_BARRIER_BIT);
}

void ParticleSystem::setZoomLevel(float zoom)
{
    m_zoomLevel = zoom;

    // Increased Density for Brightness (was 8/6/4/3/2)
    if (zoom > 1.8) m_density = 16;
    else if (zoom > 1.3) m_density = 12;
    else if (zoom > 0.8) m_density = 8;   // Standard (was 4)
    else if (zoom > 0.5) m_density = 6;
    else m_density = 4;
}

void ParticleSystem::adjustQuality(float currentFps, float targetFps)
{
    if (currentFps < targetFps * 0.9f) {
        m_glowIntensity *= 0.9f;
        if (m_glowIntensity < 0.5f) m_glowIntensity = 0.5f;
    } else if (currentFps > targetFps * 1.1f) {
        m_glowIntensity = std::min(2.0f, m_glowIntensity * 1.05f);
    }
}

void ParticleSystem::triggerShockwave(float x, float y) {
    m_shockX = x;
    m_shockY = y;
    m_shockTime = m_elapsedTime;
}

void ParticleSystem::setTheme(int theme) {
    m_theme = theme;

    // Apply Presets
    if (theme == THEME_CYBERPUNK) {
        m_scanlineIntensity = 0.05f;
        m_colorTint = QVector3D(1.5f, 0.45f, 0.05f); // DEEP Orange
    }
    else if (theme == THEME_RETRO) {
        m_scanlineIntensity = 0.50f; // Strong scanlines
        m_colorTint = QVector3D(0.8f, 1.0f, 0.8f); // Greenish tint
    }
    else if (theme == THEME_SYNTHWAVE) {
        m_scanlineIntensity = 0.15f;
        m_colorTint = QVector3D(1.0f, 1.0f, 1.0f); // Gradient handled in shader
    }
}

void ParticleSystem::render(const QMatrix4x4& projection)
{
    if (!m_renderProgram) return;

    m_renderProgram->bind();

    QMatrix4x4 finalProj = projection;
```

```
    finalProj.translate(m_width/2, m_height/2);
    finalProj.scale(m_zoomLevel);
    finalProj.translate(-m_width/2, -m_height/2);

    m_renderProgram->setUniformValue("projection", finalProj);
    m_renderProgram->setUniformValue("glowIntensity", m_glowIntensity);
    m_renderProgram->setUniformValue("uBrightness", m_brightness);
    m_renderProgram->setUniformValue("uVibrance", m_vibrance); // FIXED
    m_renderProgram->setUniformValue("uShimmerSpeed", m_shimmerSpeed);
    m_renderProgram->setUniformValue("elapsedTime", m_elapsedTime);

    // Theme Uniforms
    m_renderProgram->setUniformValue("uScanlineIntensity", m_scanlineIntensity);
    m_renderProgram->setUniformValue("uColorTint", m_colorTint); // vec3
    m_renderProgram->setUniformValue("uTheme", m_theme);
    m_renderProgram->setUniformValue("uResolution", QVector2D(m_width, m_height));

    glBindVertexArray(m_vao);
    glDrawArraysInstanced(GL_TRIANGLE_FAN, 0, 4, m_particleCount);
    glBindVertexArray(0);
}
```

File: src/particles/ParticleSystem.h

```
#pragma once

#include
#include
#include
#include
#include "../fonts/FontAsset.h"
#include "../fonts/ClassicFont.h"
#include "../fonts/HighResFont.h"
#include "../fonts/TechVectorFont.h"
#include "../fonts/ModernTermFont.h"
#include "../fonts/CodeProFont.h"
#include "../fonts/CrtRetroFont.h"
#include "../fonts/TechVectorFont.h" // NEW

// SoA Layout for strict cache coherency on CPU (if needed) and direct mapping to GPU
buffers
class ParticleSystem : protected QOpenGLFunctions_4_5_Core
{
public:
    ParticleSystem();
    ~ParticleSystem();

    enum AnimationStyle {
        STYLE_NORMAL = 0,
        STYLE_TWIST = 1,
        STYLE_RAIN = 2,
        STYLE_QUANTUM = 3,
        STYLE_SONIC = 4,
        STYLE_MAGNETIC = 5
    };

    enum Theme {
        THEME_CYBERPUNK = 0,
        THEME_RETRO = 1,
        THEME_SYNTHWAVE = 2,
        THEME_CUSTOM = 99
    };

    void init();
    void resize(int width, int height);
    void update(float dt);
    void render(const QMatrix4x4& viewProjection);

    // Optimization Systems
    void setZoomLevel(float zoom);
    void adjustQuality(float currentFps, float targetFps);

    // Initial setup
    void seedParticles(int count);
```

```
    // Text Rendering
    void updateParticlesFromTerminal(const class TerminalModel& model, int cursorX =
-1, int cursorY = -1, bool cursorVisible = false,
                                     int selStartCol = -1, int selStartRow = -1, int
selEndCol = -1, int selEndRow = -1,
                                     int linkRow = -1, int linkStart = -1, int linkEnd
= -1);

    void triggerShockwave(float x, float y);

    void setAudioLevel(float level) { m_audioLevel = level; }
    float getAudioLevel() const { return m_audioLevel; }

    void setTheme(int theme); // 0=Cyberpunk, 1=Retro, 2=Synthwave
    void setScanlineIntensity(float val) { m_scanlineIntensity = val; }
    void setColorTint(QVector3D tint) { m_colorTint = tint; }

    void setGlowIntensity(float val) { m_glowIntensity = val; }
    void setBrightness(float val) { m_brightness = val; }
    void setVibrance(float val) { m_vibrance = val; } // NEW
    void setSpringK(float val) { m_springK = val; }
    void setDrag(float val) { m_drag = val; }
    void setShimmerSpeed(float val) { m_shimmerSpeed = val; }
    void setDensity(int val) {
        if (m_density != val) {
            m_density = val;
            m_prevGrid.clear(); // Force rebuild
            m_prevGrid.clear(); // Force rebuild
        }
    }

    void setFont(FontAsset* font);
    void setFontById(int id);
    int getFontId() const { return m_fontId; }

    // Getters for UI init
    float getGlowIntensity() const { return m_glowIntensity; }
    float getBrightness() const { return m_brightness; }
    float getVibrance() const { return m_vibrance; } // NEW
    float getSpringK() const { return m_springK; }
    float getDrag() const { return m_drag; }
    float getShimmerSpeed() const { return m_shimmerSpeed; }
    int getDensity() const { return m_density; }
    int getAnimationStyle() const { return m_animationStyle; }
    float getZoomLevel() const { return m_zoomLevel; }

    int getTheme() const { return m_theme; }
    float getScanlineIntensity() const { return m_scanlineIntensity; }
    QVector3D getColorTint() const { return m_colorTint; }

    void setAnimationStyle(int style) { m_animationStyle = style; }

private:
    void initBuffers();
    void initShaders();

    // GPU Buffers
```

```
    GLuint m_vao;
    GLuint m_posVbo;      // Current Position (vec4: x, y, z, w)
    GLuint m_velVbo;      // Velocity (vec4: vx, vy, vz, vw)
    GLuint m_targetVbo;   // Target Position (vec4: tx, ty, tz, padding)
    GLuint m_extraVbo;    // Extra: x=pulse, y=flicker, z=radius, w=unused - NEW
    GLuint m_colorVbo;    // Color (vec4: r, g, b, a)
    GLuint m_baseQuadVbo; // The single quad geometry

    // Shader Programs
    std::unique_ptr m_renderProgram;
    std::unique_ptr m_computeProgram;

    // Data
    int m_particleCount;
    int m_maxParticles;

    // Member vectors to avoid re-allocation
    std::vector m_posData;
    std::vector m_velData;
    std::vector m_targetData;
    std::vector m_extraData;
    std::vector m_colorData;

    // Bounds
    float m_width;
    float m_height;

    // Optimization Settings
    float m_zoomLevel = 1.0f;
    int m_density = 8; // Increased default density for brightness
    int m_gridCols = 0;
    int m_gridRows = 0;
    int m_gridDensity = 0; // Track density used for allocation
    std::vector m_prevGrid; // Store char codes to detect changes
    std::vector m_prevChars; // Store actual character codes for animation triggers
    std::vector m_jitterTable; // Optimization: Pre-computed random noise
    int m_jitterIndex = 0;

    // Visual Parameters
    float m_glowIntensity = 1.0f;
    float m_brightness = 1.0f; // New global multiplier
    float m_vibrance = 1.0f; // New saturation
    float m_springK = 80.0f; // Much faster (was 15.0)
    float m_drag = 0.90f;   // Higher damping (was 0.85)
    float m_shimmerSpeed = 4.0f; // Base speed

    float m_elapsedTime = 0.0f; // For wave animation timing
    int m_animationStyle = 0; // 0 = Normal

    // Shockwave Data
    float m_shockX = -1000.0f;
    float m_shockY = -1000.0f;
    float m_shockTime = -10.0f;

    // Theme Data
    int m_theme = 0;
    float m_scanlineIntensity = 0.05f; // Default subtle
```

```
    QVector3D m_colorTint = QVector3D(1.0f, 1.0f, 1.0f);

    // Audio
    float m_audioLevel = 0.0f;

    // Font
    FontAsset* m_font = nullptr;
    int m_fontId = 0;
};
```

File: shaders/particle.vert

```
#version 450 core

layout(location = 0) in vec2 inPos;       // Quad vertex position (0..1)
layout(location = 1) in vec3 inInstancePos; // Per-instance position
layout(location = 2) in float inSize;       // Per-instance size
layout(location = 3) in vec4 inColor;       // Per-instance color
layout(location = 4) in vec4 inExtra;       // Pulse, Flicker, Radius, Unused

out vec4 vColor;
out vec2 vTexCoord;

uniform mat4 projection;
uniform float elapsedTime; // For animation
uniform float uShimmerSpeed;

// Pseudo-random noise
float rand(vec2 co){
    return fract(sin(dot(co.xy ,vec2(12.9898,78.233))) * 43758.5453);
}

void main() {
    vTexCoord = inPos;

    float pulse = inExtra.x;
    float flicker = inExtra.y; // Seed from C++

    // Time wrapper to preserve precision
    float time = mod(elapsedTime, 100.0);

    // NEW SHIMMER LOGIC
    // Base glow pulsing
    float speed = max(0.5, uShimmerSpeed); // Ensure non-zero
    float wave = sin(time * speed + pulse * 10.0); // Offset by particle ID

    // Organic Pulse: Normalize sin to 0.7 - 1.3
    float brightness = 1.0 + wave * 0.3;

    // Occasional Sparkle (High frequency noise)
    if (uShimmerSpeed > 2.0) {
        float spark = rand(vec2(time * speed * 5.0, flicker));
        if (spark > 0.95) brightness += 0.5; // Sparkle
    }

    float finalBrightness = brightness;

    // Boost Color
    vColor = inColor * finalBrightness * 1.3; // 130% brightness boost

    // Tiny Jitter (Arcing Movement) - Sub-pixel only
    // "Staying within confines" -> very small amplitude
    vec2 jitter = vec2((rand(vec2(time * 5.0, flicker)) - 0.5),
```

```
                        (rand(vec2(time * 5.0, pulse)) - 0.5)) * inSize * 0.1;

    vec2 pos = inPos * inSize + jitter;
    vec3 finalPos = inInstancePos + vec3(pos, 0.0);

    gl_Position = projection * vec4(finalPos, 1.0);
}
```

File: shaders/particle.frag

```
#version 450 core

in vec4 vColor;
in vec2 vTexCoord;

out vec4 fragColor;

uniform float glowIntensity;
uniform float uBrightness;
uniform float uVibrance; // NEW
uniform float uScanlineIntensity;
uniform vec3 uColorTint;
uniform int uTheme;
uniform vec2 uResolution;

void main() {
    vec2 coord = vTexCoord - vec2(0.5);
    float dist = length(coord);

    if (dist > 0.5) discard;

    // === CRT DOT MATRIX ===
    // Sharp bright center dot + subtle phosphor glow

    // Sharp core (the actual dot)
    float core = smoothstep(0.25, 0.1, dist);

    // Soft phosphor glow around it
    float glow = exp(-dist * 8.0) * 0.5;

    float intensity = (core + glow) * glowIntensity;

    fragColor = vColor * intensity;

    // === THEMES ===
    // 1. Color Tint (Retro/Global)
    fragColor.rgb *= uColorTint;

    // Saturation / Vibrance
    float gray = dot(fragColor.rgb, vec3(0.299, 0.587, 0.114));
    fragColor.rgb = mix(vec3(gray), fragColor.rgb, uVibrance);

    fragColor.rgb *= uBrightness;

    // 2. Scanlines
    if (uScanlineIntensity > 0.001) {
        // High frequency sine wave (every 2-3 pixels)
        float scanline = 0.5 + 0.5 * sin(gl_FragCoord.y * 1.5);
        fragColor.rgb *= mix(1.0, scanline, uScanlineIntensity);
    }
```

```
    // 3. Synthwave Gradient (Theme 2)
    if (uTheme == 2) {
       // Purple top (t=1), Orange bottom (t=0)
       // Add slight curvature?
       float t = gl_FragCoord.y / uResolution.y;
       vec3 topColor = vec3(0.7, 0.0, 1.0); // Purple
       vec3 botColor = vec3(1.0, 0.2, 0.0); // Orange

       // Use lighter mix for text legibility
       vec3 grad = mix(botColor, topColor, t * 1.2 - 0.1);
       fragColor.rgb *= grad * 2.0; // Boost brightness standard
    }
}
```

File: src/renderer/TerminalWidget.cpp

```
#include "TerminalWidget.h"
#include "../particles/ParticleSystem.h"
#include
#include
#include
#include
#include
#include
#include
#include
#include

TerminalWidget::TerminalWidget(QWidget* parent)
    : QOpenGLWidget(parent)
    , m_frameTimer(new QTimer(this))
    , m_frameCount(0)
    , m_particleSystem(new ParticleSystem())
    , m_sshClient(new SshClient(this))
    , m_terminalModel(new TerminalModel(80, 25, this))
    , m_parent(parent)
{
    // Connect SSH -> Terminal (Incoming Data)
    connect(m_sshClient, &SshClient::dataReceived, m_terminalModel,
&TerminalModel::processInput);
    connect(m_sshClient, &SshClient::errorOccurred, m_terminalModel,
&TerminalModel::showMessage);
    connect(m_sshClient, &SshClient::debugMessage, m_terminalModel,
&TerminalModel::showMessage);

    // Connect Terminal -> SSH (Outgoing Data)
    connect(m_terminalModel, &TerminalModel::dataOutput, m_sshClient,
&SshClient::sendData);

    // Connect Terminal -> Particles (Render Update)
    connect(m_terminalModel, &TerminalModel::screenChanged, this, [this]() {
        m_screenDirty = true;
    });

    // Connect poll timer
    connect(m_frameTimer, &QTimer::timeout, m_sshClient, &SshClient::poll);

    // 120 FPS target -> ~8.33ms
    m_frameTimer->setInterval(8);
    connect(m_frameTimer, &QTimer::timeout, this, [this]() {
        update(); // Schedules paintGL
    });
    m_frameTimer->start();
    m_elapsedTimer.start();

    setFocusPolicy(Qt::StrongFocus);
    setMouseTracking(true);
```

```
    // Blink Timer
    m_blinkTimer = new QTimer(this);
    connect(m_blinkTimer, &QTimer::timeout, this, [this](){
        m_cursorBlinkState = !m_cursorBlinkState;

        // ASYMMETRIC BLINK:
        // If ON: Stay ON for 1000ms (Solid)
        // If OFF: Stay OFF for 200ms (Explosion duration)
        if (m_cursorBlinkState) {
            m_blinkTimer->setInterval(1000);
        } else {
            m_blinkTimer->setInterval(200);
        }

        m_screenDirty = true;
        update();
    });
    // Start in ON state, so ensure first timeout happens after 1000ms
    m_cursorBlinkState = true;
    m_blinkTimer->start(1000);
}

TerminalWidget::~TerminalWidget()
{
    makeCurrent();
    delete m_particleSystem;
    doneCurrent();
}

void TerminalWidget::showConnectionDialog()
{
    ConnectionDialog* dialog = new ConnectionDialog(this);
    connect(dialog, &ConnectionDialog::connectClicked, this, [this](QString h, int p,
QString u, QString pwd, QString key){
        m_sshClient->connectToHost(h, p, u, pwd, key);
        int cols = m_terminalModel->cols();
        int rows = m_terminalModel->rows();
        m_sshClient->setPtySize(cols, rows);
    });
    dialog->show();
}

void TerminalWidget::connectToHost(const QString& h, int p, const QString& u, const
QString& pwd, const QString& key, const QList& rules)
{
     m_sshClient->connectToHost(h, p, u, pwd, key);
     int cols = m_terminalModel->cols();
     int rows = m_terminalModel->rows();
     m_sshClient->setPtySize(cols, rows);

     for (const auto& rule : rules) {
         if (rule.type == SshClient::PortForwardRule::Local) {
             m_sshClient->addLocalForward(rule.bindPort, rule.targetHost,
rule.targetPort);
         }
     }
```

```
}

bool TerminalWidget::isConnected() const
{
    return m_sshClient && m_sshClient->isConnected();
}

void TerminalWidget::setRenderEnabled(bool enabled)
{
    if (enabled) {
        if (!m_frameTimer->isActive()) m_frameTimer->start();
    } else {
        if (m_frameTimer->isActive()) m_frameTimer->stop();
    }
}

void TerminalWidget::initializeGL()
{
    if (!initializeOpenGLFunctions()) {
        qFatal("OpenGL 4.5 functions initialization failed");
    }

    glClearColor(0.0f, 0.0f, 0.0f, m_opacity);
    glEnable(GL_BLEND);
    glBlendFuncSeparate(GL_ONE, GL_ONE_MINUS_SRC_ALPHA, GL_ONE, GL_ONE);

    m_particleSystem->init();
    m_particleSystem->resize(width(), height());
}

void TerminalWidget::resizeGL(int w, int h)
{
    int charWidth = 16;
    int charHeight = 29;
    int cols = std::max(80, w / charWidth);
    int rows = std::max(24, h / charHeight);

    if (m_terminalModel && (m_terminalModel->cols() != cols || m_terminalModel->rows()
!= rows)) {
        m_terminalModel->resize(cols, rows);
        if (m_sshClient) {
            m_sshClient->setPtySize(cols, rows);
        }
    }

    if (m_particleSystem) {
        m_particleSystem->resize(w, h);
        if (m_terminalModel) {
            m_particleSystem->updateParticlesFromTerminal(*m_terminalModel);
        }
    }
}

void TerminalWidget::paintGL()
{
    float current = m_elapsedTimer.nsecsElapsed() / 1000000000.0f;
    static float last = 0;
```

```
    m_deltaTime = current - last;
    last = current;

    if (m_screenDirty) {
        if (m_particleSystem && m_terminalModel) {
             m_particleSystem->updateParticlesFromTerminal(*m_terminalModel,
                m_terminalModel->cursorX(),
                m_terminalModel->cursorY(),
                m_cursorBlinkState,
                m_selStart.x(), m_selStart.y(),
                m_selEnd.x(), m_selEnd.y(),
                m_hoveredLink.row, m_hoveredLink.startCol, m_hoveredLink.endCol);
        }
        m_screenDirty = false;
    }

    m_frameCount++;
    static float timeAccum = 0;
    timeAccum += m_deltaTime;
    if (timeAccum >= 1.0f) {
        m_particleSystem->adjustQuality(m_frameCount, 120.0f);
        m_frameCount = 0;
        timeAccum = 0;
    }

    // Beat Sim (Removed)

    updatePhysics();
    renderParticles();
}

void TerminalWidget::paintEvent(QPaintEvent *event)
{
    // 1. Render OpenGL (Particles)
    QOpenGLWidget::paintEvent(event);

    // 2. Render Minimap Overlay (QPainter)
    if (!m_terminalModel) return;

    QPainter p(this);
    p.setRenderHint(QPainter::Antialiasing, false);

    int mapWidth = 120;
    int w = width();
    int h = height();
    QRect mapRect(w - mapWidth, 0, mapWidth, h);

    // Background
    p.fillRect(mapRect, QColor(0, 0, 0, 150)); // Semi-transparent black

    // Borders
    p.setPen(QColor(60, 60, 60));
    p.drawLine(w - mapWidth, 0, w - mapWidth, h);

    // Calculate scaling
    int historyLines = m_terminalModel->historySize();
    int termLines = m_terminalModel->rows();
```

```
    int totalLines = historyLines + termLines;
    if (totalLines == 0) return;

    // Map total lines to widget height
    // If fewer lines than pixels, 1:1 or stretched? usually 1:1 or compressed.
    // If more lines than pixels, skip lines.

    double scaleY = (double)h / std::max(h, totalLines);
    if (totalLines > h) scaleY = (double)h / totalLines;

    // Ideally we want pixel-perfect mapping for small buffers
    // For large buffers, we sample.

    p.setPen(Qt::NoPen);

    // Draw History
    // Optimization: Draw 1 rect per line if scaleY > 1, else draw blocks

    int step = 1;
    if (scaleY < 1.0) step = (int)(1.0 / scaleY);
    if (step < 1) step = 1;

    for (int r = 0; r < historyLines; r += step) {
        const auto& line = m_terminalModel->historyLine(r);
        int y = (int)(r * scaleY);

        // Sample line content
        for (int c = 0; c < line.size(); c++) {
            if (line[c].ch > 32) { // visible char
                // Map column to map width
                int x = w - mapWidth + (int)((float)c / m_terminalModel->cols() *
(mapWidth - 4) + 2);

                // Color based on attribute or default amber
                QColor col(255, 176, 0, 120); // Amber semi-trans
                if (line[c].attr.fgColor == 1) col = QColor(255, 80, 80, 150); // Red
                else if (line[c].attr.fgColor == 2) col = QColor(80, 255, 80, 150); //
Green
                else if (line[c].attr.fgColor == 4) col = QColor(80, 120, 255, 150);
// Blue

                p.fillRect(x, y, 2, std::max(1, (int)scaleY), col);
            }
        }
    }

    // Draw Viewport Highlight (Visible Area)
    // viewOffset 0 = bottom (visible grid)
    // viewOffset > 0 = history
    // We need to map current view range [start, end] to Y pixels

    int viewOffset = m_terminalModel->viewOffset(); // 0 is active grid
    // Logic:
    // Total lines = History + Active
    // Active grid is at index [historyLines ... historyLines+termLines-1] if offset=0
    // If offset > 0, we shift up into history.
```

```
    // Let's assume logical index 0 is oldest history.
    // Max index is (totalLines - 1).
    // Visible window starts at: (historyLines - viewOffset)
    // Visible window height: termLines

    int visibleStart = historyLines - viewOffset;
    int visibleEnd = visibleStart + termLines;

    // Clamp
    if (visibleStart < 0) visibleStart = 0;

    int yStart = (int)(visibleStart * scaleY);
    int yEnd = (int)(visibleEnd * scaleY);
    int hRect = std::max(4, yEnd - yStart);

    // Draw highlight rect
    p.fillRect(w - mapWidth, yStart, mapWidth, hRect, QColor(255, 255, 255, 30)); //
Highlight
    p.setPen(QColor(255, 255, 255, 80));
    p.drawRect(w - mapWidth, yStart, mapWidth - 1, hRect - 1); // Border
}

void TerminalWidget::updatePhysics()
{
    m_particleSystem->update(m_deltaTime);
}

void TerminalWidget::renderParticles()
{
    glClearColor(0.0f, 0.0f, 0.0f, m_opacity);
    glClear(GL_COLOR_BUFFER_BIT);
    QMatrix4x4 projection;
    projection.ortho(0, width(), height(), 0, -1, 1);
    m_particleSystem->render(projection);
}


void TerminalWidget::sendData(const QByteArray& data)
{
    if (m_sshClient) {
        m_sshClient->sendData(data);
    }
}

bool TerminalWidget::event(QEvent *event)
{
    if (event->type() == QEvent::KeyPress) {
        QKeyEvent *keyEvent = static_cast(event);
        if (keyEvent->key() == Qt::Key_Tab || keyEvent->key() == Qt::Key_Backtab) {
            keyPressEvent(keyEvent);
            return true;
        }
    }
    return QOpenGLWidget::event(event);
}

void TerminalWidget::keyPressEvent(QKeyEvent *event)
```

```
{
    if (!m_sshClient) return;

    // Clipboard & Split Shortcuts
    if (event->modifiers() == (Qt::ControlModifier | Qt::ShiftModifier)) {
        if (event->key() == Qt::Key_C) { copySelection(); return; }
        if (event->key() == Qt::Key_V) { pasteClipboard(); return; }
        if (event->key() == Qt::Key_E) { emit splitRequest(Qt::Horizontal); return; }
        if (event->key() == Qt::Key_O) { emit splitRequest(Qt::Vertical); return; }
    }

    // Zoom Shortcuts (Ctrl +/-)
    if (event->modifiers() & Qt::ControlModifier) {
         if (event->key() == Qt::Key_Plus || event->key() == Qt::Key_Equal) {
             emit zoomRequest(0.1f);
             return;
         }
         if (event->key() == Qt::Key_Minus) {
             emit zoomRequest(-0.1f);
             return;
         }
    }

    // Navigation (Ctrl+Arrows)
    if (event->modifiers() & Qt::ControlModifier) {
        if (event->key() == Qt::Key_Up) { emit navigateRequest(0, -1); return; }
        if (event->key() == Qt::Key_Down) { emit navigateRequest(0, 1); return; }
        if (event->key() == Qt::Key_Left) { emit navigateRequest(-1, 0); return; }
        if (event->key() == Qt::Key_Right) { emit navigateRequest(1, 0); return; }
    }

    VTermModifier mod = VTERM_MOD_NONE;
    if (event->modifiers() & Qt::ShiftModifier) mod = (VTermModifier)(mod |
VTERM_MOD_SHIFT);
    if (event->modifiers() & Qt::AltModifier)   mod = (VTermModifier)(mod |
VTERM_MOD_ALT);
    if (event->modifiers() & Qt::ControlModifier) mod = (VTermModifier)(mod |
VTERM_MOD_CTRL);

    VTermKey vtKey = VTERM_KEY_NONE;
    switch(event->key()) {
        case Qt::Key_Enter:
        case Qt::Key_Return:
            vtKey = VTERM_KEY_ENTER;
            // SONIC BOOM TRIGGER
            if (m_particleSystem && getAnimationStyle() == 4) { // 4 = Sonic
                // Get cursor position in pixels
                 if (m_terminalModel) {
                     float cw = (float)width() / std::max(1, m_terminalModel->cols());
                     float ch = (float)height() / std::max(1, m_terminalModel-
>rows());
                     float x = m_terminalModel->cursorX() * cw + (cw/2);
                     float y = m_terminalModel->cursorY() * ch + (ch/2);
                     m_particleSystem->triggerShockwave(x, y);
                 }
            }
            break;
```

```
        case Qt::Key_Tab: vtKey = VTERM_KEY_TAB; break;
        case Qt::Key_Backspace: vtKey = VTERM_KEY_BACKSPACE; break;
        case Qt::Key_Escape: vtKey = VTERM_KEY_ESCAPE; break;
        case Qt::Key_Up: vtKey = VTERM_KEY_UP; break;
        case Qt::Key_Down: vtKey = VTERM_KEY_DOWN; break;
        case Qt::Key_Left: vtKey = VTERM_KEY_LEFT; break;
        case Qt::Key_Right: vtKey = VTERM_KEY_RIGHT; break;
        case Qt::Key_Insert: vtKey = VTERM_KEY_INS; break;
        case Qt::Key_Delete: vtKey = VTERM_KEY_DEL; break;
        case Qt::Key_Home: vtKey = VTERM_KEY_HOME; break;
        case Qt::Key_End: vtKey = VTERM_KEY_END; break;
        case Qt::Key_PageUp: vtKey = VTERM_KEY_PAGEUP; break;
        case Qt::Key_PageDown: vtKey = VTERM_KEY_PAGEDOWN; break;
        default:
            if (event->key() >= Qt::Key_F1 && event->key() <= Qt::Key_F12)
                vtKey = (VTermKey)(VTERM_KEY_FUNCTION(event->key() - Qt::Key_F1 + 1));
            break;
    }

    if (vtKey != VTERM_KEY_NONE) {
        m_terminalModel->sendKey(vtKey, mod);
    } else {
        QString text = event->text();
        if (!text.isEmpty()) {
            m_terminalModel->sendText(text);
        }
    }
}

void TerminalWidget::focusInEvent(QFocusEvent *event)
{
    QOpenGLWidget::focusInEvent(event);
    emit focused();
}

void TerminalWidget::mousePressEvent(QMouseEvent *event)
{
    // If app is tracking mouse, forward to terminal
    if (m_terminalModel->mouseTracking()) {
        int button = 1;
        if (event->button() == Qt::MiddleButton) button = 2;
        else if (event->button() == Qt::RightButton) button = 3;

        VTermModifier mod = VTERM_MOD_NONE;
        if (event->modifiers() & Qt::ShiftModifier) mod = (VTermModifier)(mod |
VTERM_MOD_SHIFT);
        if (event->modifiers() & Qt::AltModifier)   mod = (VTermModifier)(mod |
VTERM_MOD_ALT);
        if (event->modifiers() & Qt::ControlModifier) mod = (VTermModifier)(mod |
VTERM_MOD_CTRL);

        m_terminalModel->sendMouse(button, true, mod);
        return;
    }

    // Middle-click: Paste
    if (event->button() == Qt::MiddleButton) {
```

```
        pasteClipboard();
        return;
    }

    // Left-click: Link or Selection
    if (event->button() == Qt::LeftButton) {
        if (m_hoveredLink.isValid()) {
            QDesktopServices::openUrl(QUrl(m_hoveredLink.url));
            return;
        }

        clearSelection();
        m_selecting = true;
        m_selStart = pixelToCell(event->position());
        m_selEnd = m_selStart;
        m_screenDirty = true;
        update();
    }
}

void TerminalWidget::mouseReleaseEvent(QMouseEvent *event)
{
    if (m_terminalModel->mouseTracking()) {
        int button = 1;
        if (event->button() == Qt::MiddleButton) button = 2;
        else if (event->button() == Qt::RightButton) button = 3;

        VTermModifier mod = VTERM_MOD_NONE;
        if (event->modifiers() & Qt::ShiftModifier) mod = (VTermModifier)(mod |
VTERM_MOD_SHIFT);
        if (event->modifiers() & Qt::AltModifier)   mod = (VTermModifier)(mod |
VTERM_MOD_ALT);
        if (event->modifiers() & Qt::ControlModifier) mod = (VTermModifier)(mod |
VTERM_MOD_CTRL);

        m_terminalModel->sendMouse(button, false, mod);
        return;
    }

    // Left-click release: Finalize selection and auto-copy
    if (event->button() == Qt::LeftButton && m_selecting) {
        m_selecting = false;
        m_selEnd = pixelToCell(event->position());

        // Auto-copy on selection (PuTTY behavior)
        if (hasSelection()) {
            copySelection();
        }
        m_screenDirty = true;
        update();
    }
}

void TerminalWidget::mouseMoveEvent(QMouseEvent *event)
{
    if (m_terminalModel->mouseTracking()) {
        float cellW = (float)width() / std::max(1, m_terminalModel->cols());
```

```
        float cellH = (float)height() / std::max(1, m_terminalModel->rows());
        int col = (int)(event->position().x() / cellW);
        int row = (int)(event->position().y() / cellH);

        VTermModifier mod = VTERM_MOD_NONE;
        if (event->modifiers() & Qt::ShiftModifier) mod = (VTermModifier)(mod |
VTERM_MOD_SHIFT);
        if (event->modifiers() & Qt::AltModifier)   mod = (VTermModifier)(mod |
VTERM_MOD_ALT);
        if (event->modifiers() & Qt::ControlModifier) mod = (VTermModifier)(mod |
VTERM_MOD_CTRL);

        m_terminalModel->sendMouseMove(col, row, mod);
        return;
    }

    // Detect Links
    detectLinkAt(event->position().toPoint());

    // Update selection during drag
    if (m_selecting) {
        m_selEnd = pixelToCell(event->position());
        m_screenDirty = true;
        update();
    }
}

void TerminalWidget::wheelEvent(QWheelEvent *event)
{
    if (m_terminalModel->mouseTracking()) {
        // ... (existing mouse code) ...
        // VTerm doesn't have wheel buttons directly in API typically,
        // but often encoded as buttons 4/5
        int button = (event->angleDelta().y() > 0) ? 4 : 5;

        VTermModifier mod = VTERM_MOD_NONE;
        if (event->modifiers() & Qt::ShiftModifier) mod = (VTermModifier)(mod |
VTERM_MOD_SHIFT);
        if (event->modifiers() & Qt::AltModifier)   mod = (VTermModifier)(mod |
VTERM_MOD_ALT);
        if (event->modifiers() & Qt::ControlModifier) mod = (VTermModifier)(mod |
VTERM_MOD_CTRL);

        m_terminalModel->sendMouse(button, true, mod);
        return;
    }

    int steps = event->angleDelta().y() / 40;
    // Invert direction for natural scrolling
    m_terminalModel->scrollView(-steps);
}

// Graphics Settings Delegation
void TerminalWidget::setGlowIntensity(float val) { if (m_particleSystem)
m_particleSystem->setGlowIntensity(val); }
void TerminalWidget::setOpacity(float val) { m_opacity = val; update(); }
void TerminalWidget::setBrightness(float val) { if (m_particleSystem)
```

```
m_particleSystem->setBrightness(val); }
void TerminalWidget::setVibrance(float val) { if (m_particleSystem) m_particleSystem-
>setVibrance(val); } // NEW
void TerminalWidget::setSpringK(float val) { if (m_particleSystem) m_particleSystem-
>setSpringK(val); }
void TerminalWidget::setDrag(float val) { if (m_particleSystem) m_particleSystem-
>setDrag(val); }
void TerminalWidget::setShimmerSpeed(float val) { if (m_particleSystem)
m_particleSystem->setShimmerSpeed(val); }
void TerminalWidget::setFont(int val) { if (m_particleSystem) m_particleSystem-
>setFontById(val); }
void TerminalWidget::setDensity(int val) {
    if (m_particleSystem) {
        m_particleSystem->setDensity(val);
        m_screenDirty = true;
        update();
    }
}
void TerminalWidget::setZoomLevel(float zoom) {
    if (m_particleSystem) {
        m_particleSystem->setZoomLevel(zoom);
        m_screenDirty = true;
        update();
    }
}
void TerminalWidget::setAnimationStyle(int style) { if (m_particleSystem)
m_particleSystem->setAnimationStyle(style); }
void TerminalWidget::setTheme(int theme) { if (m_particleSystem) m_particleSystem-
>setTheme(theme); }

// Getters
float TerminalWidget::getGlowIntensity() const { return m_particleSystem ?
m_particleSystem->getGlowIntensity() : 1.0f; }
float TerminalWidget::getOpacity() const { return m_opacity; }
float TerminalWidget::getBrightness() const { return m_particleSystem ?
m_particleSystem->getBrightness() : 1.0f; }
float TerminalWidget::getVibrance() const { return m_particleSystem ?
m_particleSystem->getVibrance() : 1.0f; } // NEW
float TerminalWidget::getSpringK() const { return m_particleSystem ? m_particleSystem-
>getSpringK() : 15.0f; }
float TerminalWidget::getDrag() const { return m_particleSystem ? m_particleSystem-
>getDrag() : 0.85f; }
float TerminalWidget::getShimmerSpeed() const { return m_particleSystem ?
m_particleSystem->getShimmerSpeed() : 4.0f; }
int TerminalWidget::getDensity() const { return m_particleSystem ? m_particleSystem-
>getDensity() : 8; }
int TerminalWidget::getFont() const { return m_particleSystem ? m_particleSystem-
>getFontId() : 0; }
float TerminalWidget::getZoomLevel() const { return m_particleSystem ?
m_particleSystem->getZoomLevel() : 1.0f; }
int TerminalWidget::getTheme() const { return m_particleSystem ? m_particleSystem-
>getTheme() : 0; }
int TerminalWidget::getAnimationStyle() const { return m_particleSystem ?
m_particleSystem->getAnimationStyle() : 0; }

// ==== Text Selection Methods ====
```

```
QPoint TerminalWidget::pixelToCell(const QPointF& pos) const
{
    if (!m_terminalModel) return QPoint(-1, -1);

    float cellW = (float)width() / std::max(1, m_terminalModel->cols());
    float cellH = (float)height() / std::max(1, m_terminalModel->rows());

    int col = qBound(0, (int)(pos.x() / cellW), m_terminalModel->cols() - 1);
    int row = qBound(0, (int)(pos.y() / cellH), m_terminalModel->rows() - 1);

    return QPoint(col, row);
}

void TerminalWidget::clearSelection()
{
    m_selStart = QPoint(-1, -1);
    m_selEnd = QPoint(-1, -1);
    m_selecting = false;
    m_screenDirty = true;
    update();
}

bool TerminalWidget::hasSelection() const
{
    return m_selStart != QPoint(-1, -1) && m_selEnd != QPoint(-1, -1) && m_selStart !=
m_selEnd;
}

// Smart Link Detection
void TerminalWidget::detectLinkAt(QPoint pos)
{
    if (!m_terminalModel) return;
    QPoint cell = pixelToCell(pos);
    if (cell.x() < 0) {
        if (m_hoveredLink.isValid()) {
             m_hoveredLink.clear();
             m_screenDirty = true;
             setCursor(Qt::ArrowCursor);
             update();
        }
        return;
    }

    // If mouse moved but still traversing same link, verify range
    if (m_hoveredLink.isValid() && m_hoveredLink.row == cell.y() &&
        cell.x() >= m_hoveredLink.startCol && cell.x() <= m_hoveredLink.endCol) {
        return; // Still over same link
    }

    // New potential link
    m_hoveredLink.clear();
    setCursor(Qt::ArrowCursor);

    // Extract line text
    QString lineText;
    int cols = m_terminalModel->cols();
    for (int c = 0; c < cols; ++c) {
```

```
        const TerminalCell& tc = m_terminalModel->cell(c, cell.y());
        lineText += QChar(tc.ch > 0 ? (char16_t)tc.ch : ' ');
    }

    // Regex for URLs
    static QRegularExpression urlRegex(R"((https?://\S+))");
    QRegularExpressionMatchIterator i = urlRegex.globalMatch(lineText);
    while (i.hasNext()) {
        QRegularExpressionMatch match = i.next();
        if (cell.x() >= match.capturedStart() && cell.x() < match.capturedEnd()) {
            m_hoveredLink.url = match.captured();
            m_hoveredLink.row = cell.y();
            m_hoveredLink.startCol = match.capturedStart();
            m_hoveredLink.endCol = match.capturedEnd() - 1;

            setCursor(Qt::PointingHandCursor);
            m_screenDirty = true;
            update();
            return;
        }
    }
}

QString TerminalWidget::getSelectedText() const
{
    if (!hasSelection() || !m_terminalModel) return QString();

    // Normalize start/end (start is top-left, end is bottom-right)
    QPoint start = m_selStart;
    QPoint end = m_selEnd;

    // Swap if end is before start (dragged backwards)
    if (end.y() < start.y() || (end.y() == start.y() && end.x() < start.x())) {
        std::swap(start, end);
    }

    QString result;
    int cols = m_terminalModel->cols();

    for (int row = start.y(); row <= end.y(); ++row) {
        int colStart = (row == start.y()) ? start.x() : 0;
        int colEnd = (row == end.y()) ? end.x() : cols - 1;

        for (int col = colStart; col <= colEnd; ++col) {
            const TerminalCell& cell = m_terminalModel->cell(col, row);
            if (cell.ch != 0) {
                // Handle Unicode chars including those > 0xFFFF (needs surrogate
pairs)
                if (cell.ch <= 0xFFFF) {
                    result += QChar(static_cast(cell.ch));
                } else {
                    // Use fromUcs4 for characters outside BMP
                    char32_t ch32 = cell.ch;
                    result += QString::fromUcs4(&ch32, 1);
                }
            } else {
                result += ' ';
```

```
            }
        }

        // Add newline between rows (but not after last row)
        if (row < end.y()) {
            result += '\n';
        }
    }

    // Trim trailing spaces from each line
    QStringList lines = result.split('\n');
    for (QString& line : lines) {
        while (line.endsWith(' ')) line.chop(1);
    }
    result = lines.join('\n');

    return result;
}

void TerminalWidget::copySelection()
{
    QString text = getSelectedText();
    if (!text.isEmpty()) {
        QClipboard* clipboard = QGuiApplication::clipboard();
        clipboard->setText(text);
        qDebug() << "Copied to clipboard:" << text.length() << "chars";
    }
}

void TerminalWidget::pasteClipboard()
{
    QClipboard* clipboard = QGuiApplication::clipboard();
    QString text = clipboard->text();
    if (!text.isEmpty() && m_terminalModel) {
        m_terminalModel->sendText(text);
    }
}
```