Contents

Contributions (Part 2: Group Project)

- **Shinjan Roy (002310501083):**
  - Procedural Terrain Generation
  - Car physics
  - Spring model
  - Collision Detection
  - Stages and unlocking
- **Souradeep De (002310501084):**
  - Wheel physics
  - Rotational Dynamics
- **Arko Dasgupta (002310501085):**
  - Game Introduction Screen.
  - Game Over Screen
  - Pausing, Resuming and Restarting Logics
  - Persistent Coin System and Points scoring
- **Arjeesh Palai (002310501086):**
  - Camera Stabilization.
  - Coin, Fuel, Nitro Feature Integration.
  - Flip Tracker.
  - Key Stroke Logging and HUD.
  - Assets & Media Integration.
  - Scoreboard UI/UX Design and Management.

Part 2: Braking-Bad

1. Abstract & Introduction

Abstract

This project implements a physics-based driving simulation game inspired by "Hill Climb Racing." Utilizing the Qt framework and C++, the application demonstrates fundamental computer graphics concepts including rasterization, 2D geometric transformations, and viewport clipping. The primary objective is to render a dynamic terrain and a controllable vehicle on a custom raster grid, handling physics interactions in real-time.

Introduction

Project Overview: The application features a 2D side-scrolling environment where the player controls a vehicle traversing procedurally generated terrain.

- **Objectives:** To apply affine transformations (translation, rotation) for vehicle movement and use line-drawing algorithms (Bresenham/DDA) for terrain rendering.
- **Scope:** The system handles user input for acceleration/braking, calculates gravity and suspension physics, and performs collision detection between the vehicle polygon and the terrain polyline.

2. Theoretical Background

Coordinate Systems & Transformations

The game world uses a logical coordinate system which is mapped to the device viewport. The vehicle is defined by a set of local vertices V_{local} which are transformed to world coordinates V_{world} using an affine transformation matrix M.

$$M = T(dx, dy) \cdot R(\theta) \cdot S(sx, sy)$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & dx \\ \sin\theta & \cos\theta & dy \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Where (dx, dy) represents the vehicle's position and θ represents its tilt relative to the terrain slope.

Bresenham Line Drawing Algorithm

Bresenham's algorithm is used to rasterize a straight line segment between two points (x_1, y_1) and (x_2, y_2) using only integer arithmetic. For a line with $0 \leq m \leq 1$, we define

$$\Delta x = x_2 - x_1, \qquad \Delta y = y_2 - y_1, \qquad m = \frac{\Delta y}{\Delta x}.$$

Starting from $(x_0, y_0) = (x_1, y_1)$, the decision parameter is initialized as

$$p_0 = 2\Delta y - \Delta x.$$

At step k, we always increment x by one:

$$x_{k+1} = x_k + 1,$$

and choose between staying on the same scanline or moving up one pixel:

$$\text{if } p_k < 0: \quad y_{k+1} = y_k, \quad p_{k+1} = p_k + 2\Delta y,$$

$$\text{else}: \quad y_{k+1} = y_k + 1, \quad p_{k+1} = p_k + 2\Delta y - 2\Delta x.$$

Each (x_{k+1}, y_{k+1}) is plotted, producing an efficient, integer-based approximation of the ideal line.

Midpoint Circle Algorithm

The wheels and circular HUD icons (coins, fuel etc.) are drawn using an integer midpoint circle algorithm. A circle of radius r centred at the origin is described by

$$f(x, y) = x^2 + y^2 - r^2.$$

Starting from $(x_0, y_0) = (0, r)$ in the first octant, a decision parameter

$$p_0 = 1 - r$$

is used to choose between the next pixel to the East $(x_k + 1, y_k)$ or South-East $(x_k + 1, y_k - 1)$. At step k:

$$\text{if } p_k < 0 \Rightarrow (x_{k+1}, y_{k+1}) = (x_k + 1, y_k), \quad p_{k+1} = p_k + 2x_{k+1} + 1,$$

$$\text{else } (x_{k+1}, y_{k+1}) = (x_k + 1, y_k - 1), \quad p_{k+1} = p_k + 2x_{k+1} - 2y_{k+1} + 1.$$

Using eight-way symmetry, all octants are generated from this octant. In the project, the boundary points from the midpoint algorithm are extended into short horizontal spans on each scan-line, effectively producing filled discs rather than just circle outlines.

Scan-Line Polygon Filling

The car body and several HUD elements are rendered as filled polygons using a scan-line fill algorithm. For a simple polygon with vertices (x_i, y_i), we consider each integer scan-line y in the range $[y_{\min}, y_{\max}]$. For every edge $E = (x_1, y_1) \to (x_2, y_2)$ that intersects this y (i.e. $y \in [\min(y_1, y_2), \max(y_1, y_2))$ and $y_1 \neq y_2$), the x-intersection is

$$x(y) = x_1 + (y - y_1)\,\frac{x_2 - x_1}{y_2 - y_1}.$$

The implementation builds an edge table storing for each edge its upper $y_{\max}$, starting x at $y_{\min}$ and inverse slope $1/m = \Delta x/\Delta y$. During filling, an active edge list is maintained per scan-line, sorted by current x_{curr}. Intersections are then taken in pairs (x_1, x_2) and all integer pixels between them are filled using a simple loop. After each scan-line, x_{curr} is incremented by $1/m$ for every active edge. This is the standard edge-table/active-edge realisation of scan-line polygon filling used in class.

3. Implementation Details

Feature: Procedural Terrain Generation and Rasterization

Relevant classes and functions: `MainWindow::generateInitialTerrain()`, `MainWindow::ensureAheadTerrain()`, `MainWindow::rasterizeSegmentToHeightMapWorld()`, `MainWindow::drawFilledTerrain()`.

I designed the terrain system to function as an infinite, procedurally generated 2D height map. Rather than storing a massive array of world coordinates, the terrain is generated just-in-time as the player progresses. The core of my algorithm relies on a discrete stochastic process that updates the slope of the ground segments, ensuring a balance between randomness (to keep the track interesting) and stability (to keep the track drivable and to prevent the height from blowing up in either the positive or negative directions).

Stochastic Slope Evolution. I model the terrain as a sequence of connected line segments $L_0, L_1, \ldots, L_k$. Let the endpoint of the k-th segment be denoted by the tuple (x_k, y_k) in world space. The horizontal step size is fixed at a constant $\Delta x = \texttt{STEP}$. The vertical position y_{k+1} is determined by a state variable m_k representing the current slope intensity.

To calculate the next slope m_{k+1}, I utilize a recurrence relation that combines a random perturbation with a restoring force. Let $\xi \sim \mathcal{U}(0,1)$ be a uniform random variable derived from the Mersenne Twister engine ('std::mt19937'). Let D be the difficulty coefficient and H be the height of the viewport. The update rule is as follows:

$$m_{\text{temp}} = m_k + D \cdot \left(\xi - \left(1 - \frac{\tau}{100} \right) \frac{y_k}{H} \right),$$

where τ is a terrain height parameter. The term $-\frac{y_k}{H}$ acts as a negative feedback loop (a "spring" force) that pushes the terrain downward if it gets too high and upward if it gets too low, preventing the track from drifting off-screen.

I then clamp the slope to prevent vertical walls that would stop the vehicle:

$$m_{k+1} = \text{clamp}(m_{\text{temp}}, -m_{\max}, m_{\max}).$$

Non-Linear Vertical Integration. To determine the vertical change Δy, I do not map the slope linearly. Instead, a non-linearity factor γ (variable `m_irregularity`) is applied to exaggerate steep climbs and flatten gentle slopes. The height of the next vertex is computed as:

$$y_{k+1} = y_k + \lfloor m_{k+1} \cdot |m_{k+1}|^{\gamma} \cdot \Delta x \rceil .$$

This approach creates a "bouncy" terrain feel that becomes progressively more chaotic as the level difficulty (D and γ) increases over time.

Discrete Height Map Rasterization. Once a segment from (x_k, y_k) to (x_{k+1}, y_{k+1}) is generated, I rasterize it into a hash-based sparse height map. Since the game world is a pixel grid, I map the continuous line segment to discrete grid coordinates (g_x, g_y). For a segment defined by the horizontal range $[x_1, x_2]$, and for every grid column g_x such that $x_1 \leq g_x \cdot s_{\text{px}} \leq x_2$ (where s_{px} is the size of the pixels), I compute

the interpolated height w_y:

$$t = \frac{(g_x \cdot s_{\text{px}}) - x_1}{x_2 - x_1}, \quad t \in [0, 1]$$

$$w_y = y_1 + t(y_2 - y_1).$$

The integer grid height is then stored in a hash map:

$$H[g_x] = \left\lfloor \frac{w_y}{s_{\text{px}}} + 0.5 \right\rfloor.$$

This structure 'QHash<int, int>' allows $O(1)$ lookups for collision detection and rendering while handling an effectively infinite X-axis. I also implemented a pruning mechanism ('pruneHeightMap') to remove keys $g_x < x_{\text{camera}}$ to manage memory usage.

Procedural Rendering and Texturing. To render the terrain, I iterate through the visible grid columns on the screen. For a column g_x, I retrieve the ground height $g_y = H[g_x]$. If the ground is visible, I draw a vertical span of pixels from g_y down to the bottom of the screen.

To avoid the repetitive look of standard tiling textures, I implemented a procedural shading function based on spatial hashing. For any block in the world grid (B_x, B_y), I compute a deterministic hash:

$$h(B_x, B_y) = ((k_1 \oplus B_x) \cdot k_2) \oplus B_y \dots$$

This hash is used to index into a color palette (e.g., dirt or grass shades). This ensures that a specific block at world coordinate $(1000, 50)$ always renders with the same color pattern, even if the camera moves away and returns, without needing to store texture data in memory.

Core code snippets. The following excerpts from `mainwindow.cpp` demonstrate the generation loop, the slope mathematics, and the rasterization logic.

```
// mainwindow.cpp - Terrain Generation Logic

void MainWindow::generateInitialTerrain() {
    int currentWorldX = m_lastX;
    m_lastY = height() / 2;

    // Iterate in steps to fill the initial screen width
    for (int i = Constants::STEP; i <= width() + Constants::STEP; i += Constants::
        STEP) {

        // Mathematical model for slope evolution with restoring force
        m_slope += (m_dist(m_rng) - (1 - m_terrain_height/100) * static_cast<float
            >(m_lastY) / height()) * m_difficulty;

        // Clamping to safe limits
        m_slope = std::clamp(m_slope, -(float)Constants::MAX_SLOPE[level_index], (
            float)Constants::MAX_SLOPE[level_index]);

        // Non-linear height integration
        const int newY = m_lastY + std::lround(m_slope * std::pow(std::abs(m_slope
            ), m_irregularity) * Constants::STEP);
```

```
18
19          // Create segment and rasterize
20          Line seg(i - Constants::STEP, m_lastY, i, newY);
21          m_lines.append(seg);
22          rasterizeSegmentToHeightMapWorld(seg.getX1(), m_lastY, seg.getX2(), newY);
23
24          // Prop spawning logic (omitted for brevity)...
25
26          m_lastY = newY;
27          m_lastX = i;
28
29          // Dynamic difficulty adjustment
30          m_difficulty += Constants::DIFFICULTY_INCREMENT[level_index];
31          m_irregularity += Constants::IRREGULARITY_INCREMENT[level_index];
32      }
33  }
34
35  void MainWindow::ensureAheadTerrain(int worldX) {
36      // Infinite generation: Keep adding segments if the camera sees the edge
37      while (m_lastX < worldX) {
38          m_slope += (m_dist(m_rng) - (1 - m_terrain_height/100) * static_cast<float
                >(m_lastY) / height()) * m_difficulty;
39          m_slope = std::clamp(m_slope, -1.0f, 1.0f);
40
41          const int newY = m_lastY + std::lround(m_slope * std::pow(std::abs(m_slope
                ), m_irregularity) * Constants::STEP);
42
43          Line seg(m_lastX, m_lastY, m_lastX + Constants::STEP, newY);
44          m_lines.append(seg);
45          rasterizeSegmentToHeightMapWorld(seg.getX1(), m_lastY, seg.getX2(), newY);
46
47          m_lastY = newY;
48          m_lastX += Constants::STEP;
49
50          // Memory management: Remove old segments
51          if (m_lines.size() > (width() / Constants::STEP) * 3) {
52              m_lines.removeFirst();
53              pruneHeightMap();
54          }
55
56          // ... (Difficulty increment logic)
57      }
58  }
59
60  void MainWindow::rasterizeSegmentToHeightMapWorld(int x1, int y1, int x2, int y2)
        {
61      if (x2 < x1) { std::swap(x1,x2); std::swap(y1,y2); }
62
63      const int gx1 = x1 / Constants::PIXEL_SIZE;
64      const int gx2 = x2 / Constants::PIXEL_SIZE;
65
66      const double dx = double(x2 - x1);
```

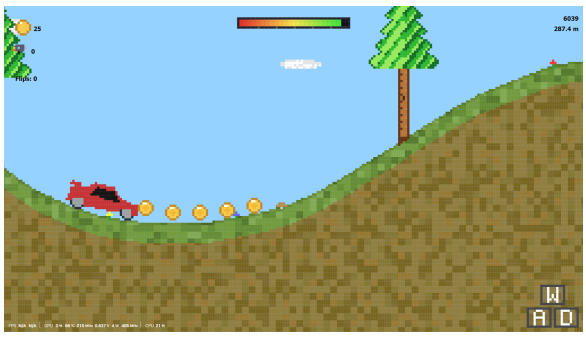
```
67      const double dy = double(y2 - y1);
68
69      // Discrete sampling of the continuous segment
70      for (int gx = gx1; gx <= gx2; ++gx) {
71          const double wx = gx * double(Constants::PIXEL_SIZE);
72          double t = (wx - x1) / dx;
73          t = std::clamp(t, 0.0, 1.0);
74
75          const double wy = y1 + t * dy;
76          const int gy = static_cast<int>(std::floor(wy / double(Constants::
                PIXEL_SIZE) + 0.5));
77
78          // Store in Hash Map
79          m_heightAtGX.insert(gx, gy);
80      }
81  }
82
83  void MainWindow::drawFilledTerrain(QPainter& p) {
84      const int camGX = m_cameraX / Constants::PIXEL_SIZE;
85      const int camGY = m_cameraY / Constants::PIXEL_SIZE;
86
87      for (int sgx = 0; sgx <= gridW(); ++sgx) {
88          const int worldGX = sgx + camGX;
89          auto it = m_heightAtGX.constFind(worldGX);
90          if (it == m_heightAtGX.constEnd()) continue;
91
92          const int groundWorldGY = it.value();
93          // ... (Calculation of screen coordinates)
94
95          for (int sGY = startScreenGY; sGY <= gridH(); ++sGY) {
96              // Procedural texture selection via hashing
97              const QColor shade = grassShadeForBlock(worldGX, worldGY, topZone);
98              plotGridPixel(p, sgx, sGY, shade);
99          }
100     }
101 }
```

Terrain Generation Results

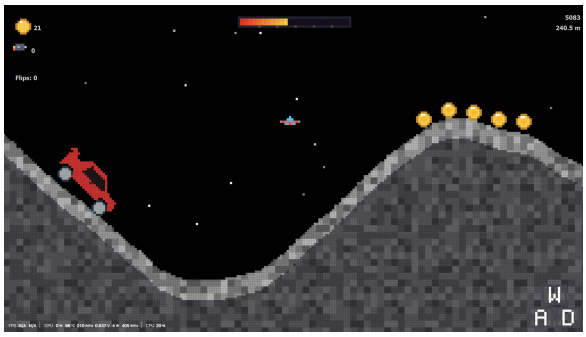
The implementation successfully creates an infinite, seamless terrain that adapts to the game's difficulty progression. The mathematical restoring force ensures that the track never spirals out of the playable area, while the non-linear integration creates "ramps" and "dips" that physically interact with the car's suspension. The ‘QHash‘ rasterization proved highly efficient, maintaining a steady 60 FPS even after the player has traveled tens of thousands of pixels, as old terrain data is aggressively pruned. Visually, the hash-based coloring provides a distinct "pixel-art" texture that is deterministic yet non-repetitive.

All levels have distinct terrain generation parameters which makes the experience unique.

`GAME LOOP :` The `gameLoop()` function advances the entire simulation at a fixed time-step. It updates elapsed time, evaluates wheel positions, accumulates forward distance and recomputes the score from



(a) Smoother curves on Meadows level



(b) Rougher curves on Lunar level

Figure 1: Terrain System: Infinite procedural generation with dynamic irregularity and deterministic texture hashing.

distance, coins and nitro uses. It smoothly follows the car with a damped camera, drives terrain extension and prop/coin/fuel spawning, and updates nitro state and fuel consumption. It also processes coin and fuel pickups, detects crashes or fuel exhaustion, arms game over when needed, and finally triggers a repaint via `update()`.

```
1  void MainWindow::gameLoop() {
2      const qint64 now = m_clock.nsecsElapsed();
3      static qint64 prev = now;
4      const qint64 dtns = now - prev;
5      prev = now;
6      const double dt = std::clamp(dtns / 1e9, 0.001, 0.033);
7
8      m_elapsedSeconds += dt;
9      double fuelBefore = m_fuel;
10     int coinsBefore = m_coinCount;
11     double avgX = 0.0, avgY = 0.0;
12     if (!m_wheels.isEmpty()) {
13         for (const Wheel* w : m_wheels) { avgX += w->x; avgY += w->y; }
14         avgX /= m_wheels.size();
15         avgY /= m_wheels.size();
16     }
17
18     double dx = std::max(0.0, avgX - m_lastScoreX);
19     m_totalDistanceCells += dx / double(Constants::PIXEL_SIZE);
20     m_lastScoreX = avgX;
21
22     m_score = int(std::llround(Constants::SCORE_DIST_PER_CELL *
           m_totalDistanceCells +
23                                Constants::SCORE_PER_COIN * m_coinCount +
24                                Constants::SCORE_PER_NITRO * m_nitroUses));
25
26     double bodyX = (!m_bodies.isEmpty()) ? m_bodies.first()->getX() : avgX;
27     double bodyY = (!m_bodies.isEmpty()) ? m_bodies.first()->getY() : avgY;
28     const double targetX = bodyX - 200.0;
29     const double targetY = -bodyY + height() / 2.0;
30     updateCamera(targetX, targetY, dt);
31     m_cameraX = int(std::lround(m_camX));
```

```
32      m_cameraY = int(std::lround(m_camY));
33      double angleRad = 0.0;
34      if (m_wheels.size() >= 2) {
35          const double dx = (m_wheels[1]->x - m_wheels[0]->x);
36          const double dy = (m_wheels[1]->y - m_wheels[0]->y);
37          angleRad = std::atan2(dy, dx);
38      }
39      double carX = (!m_bodies.isEmpty()) ? m_bodies.first()->getX() : avgX;
40      double carY = (!m_bodies.isEmpty()) ? m_bodies.first()->getY() : avgY;
41
42      m_flip.update(angleRad, carX, carY, m_elapsedSeconds, [this](int bonus){
            m_coinCount += bonus; });
43
44      const int viewRightX = m_cameraX + width();
45      const int marginPx   = Constants::COIN_SPAWN_MARGIN_CELLS * Constants::
            PIXEL_SIZE;
46      const int offRightX  = viewRightX + marginPx;
47      const int maxStreamWidthPx = (Constants::COIN_GROUP_MAX - 1) * Constants::
            COIN_GROUP_STEP_MAX * Constants::PIXEL_SIZE;
48      ensureAheadTerrain(offRightX + maxStreamWidthPx + Constants::PIXEL_SIZE * 20);
49
50      m_coinSys.maybePlaceCoinStreamAtEdge(m_elapsedSeconds, m_cameraX, width(),
            m_heightAtGX, m_lastX, m_rng, m_dist);
51
52      m_nitroSys.update(
53          m_nitroKey, m_fuel, m_elapsedSeconds, avgX,
54          [this](int gx){ return this->groundGyNearestGX(gx); },
55          [this](double wx){ return this->terrainTangentAngleAtX(wx); }
56          );
57
58      if (m_nitroSys.active && !m_prevNitroActive) ++m_nitroUses;
59      m_prevNitroActive = m_nitroSys.active;
60
61      const bool allowInput = (m_fuel > 0.0);
62      bool accelDrive = false, brakeDrive = false, nitroDrive = false;
63
64      if (m_nitroSys.active && allowInput) {
65          accelDrive = false; brakeDrive = false; nitroDrive = true;
66      } else {
67          nitroDrive = false;
68          bool bothKeys = m_accelerating && m_braking;
69          if (allowInput) {
70              if (bothKeys) { accelDrive = true; brakeDrive = true; }
71              else { accelDrive = m_accelerating; brakeDrive = m_braking; }
72          } else { accelDrive = false; brakeDrive = false; }
73      }
74
75      for (Wheel* w : m_wheels) w->simulate(level_index, m_lines, accelDrive,
            brakeDrive, nitroDrive);
76      for (CarBody* b : m_bodies) b->simulate(level_index, m_lines, accelDrive,
            brakeDrive);
77
```

```
78      m_nitroSys.applyThrust(m_wheels);
79
80      if (m_fuel > 0.0) {
81          double baseBurn = Constants::FUEL_BASE_BURN_PER_SEC * dt;
82          double extra = 0.0;
83          if (m_accelerating) extra = std::max(0.0, averageSpeed()) * Constants::
                FUEL_EXTRA_PER_SPEED * dt;
84          double burnMult = m_nitroSys.active ? 3.0 : 1.0;
85          m_fuel = std::max(0.0, m_fuel - burnMult * (baseBurn + extra));
86      }
87
88      const int minX = leftmostTerrainX();
89      for (Wheel* w : m_wheels) {
90          if (w->x < minX) { w->x = minX; w->m_vx = 0; }
91      }
92
93      if (!isFullyUpsideDown()) {
94          m_fuelSys.handlePickups(m_wheels, m_fuel);
95          m_coinSys.handlePickups(m_wheels, m_coinCount);
96      }
97      if (m_coinCount > coinsBefore) m_media->coinPickup();
98      if ((m_fuel - fuelBefore) > 1e-3 && !m_suppressFuelSfx) m_media->fuelPickup();
99      {
100         auto ptSegDist2 = [](double px, double py, const Line& ln)->double {
101             double x1 = ln.getX1(), y1 = ln.getY1();
102             double x2 = ln.getX2(), y2 = ln.getY2();
103             double vx = x2 - x1,    vy = y2 - y1;
104             double wx = px - x1,    wy = py - y1;
105             double len2 = vx*vx + vy*vy;
106             double t = (len2 > 0.0) ? (wx*vx + wy*vy) / len2 : 0.0;
107             if (t < 0.0) t = 0.0; else if (t > 1.0) t = 1.0;
108             double cx = x1 + t*vx, cy = y1 + t*vy;
109             double dx = px - cx,   dy = py - cy;
110             return dx*dx + dy*dy;
111         };
112
113         const double R2 = double(Constants::COIN_PICKUP_RADIUS) * double(Constants
                ::COIN_PICKUP_RADIUS);
114
115         for (auto& coin : m_coinSys.coins) {
116             if (coin.taken) continue;
117             bool hit = false;
118
119             for (CarBody* body : m_bodies) {
120                 const auto edges = body->getLines();
121                 for (const Line& ln : edges) {
122                     if (ptSegDist2(coin.cx, coin.cy, ln) <= R2) {
123                         hit = true;
124                         break;
125                     }
126                 }
127
```

```
128                 if (!hit) {
129                     const auto bodyPoints = body->get(0, 0);
130                     QPolygon polygon;
131                     for (const QPoint& p : bodyPoints)
132                         polygon << QPoint(p.x(), p.y());
133                     if (polygon.containsPoint(QPoint(coin.cx, coin.cy), Qt::
                        OddEvenFill))
134                         hit = true;
135                 }
136
137                 if (hit) break;
138             }
139
140             if (hit) {
141                 coin.taken = true;
142                 ++m_coinCount;
143             }
144         }
145     }
146
147     const bool fuelEmpty = (m_fuel <= 0.0);
148     const bool roofHit   = !m_bodies[0]->isAlive();
149
150     if (roofHit && !m_roofCrashLatched) {
151         m_roofCrashLatched = true;
152         if (!m_bodies.isEmpty() && m_bodies[0]->isAlive()) {
153             m_bodies[0]->kill();
154         }
155     }
156
157     if (fuelEmpty || m_roofCrashLatched) {
158         armGameOver();
159     } else {
160         disarmGameOver();
161     }
162
163     update();
164 }
```

PAINT EVENT : The paintEvent() function renders the entire frame in world and HUD space. After establishing a camera-aligned translation, it optionally draws the grid, starfield, clouds and filled terrain, then renders props, fuel pickups, world coins and nitro flame. Wheels are drawn as filled circles, and the car body plus attachments are filled using a scan-line polygon routine. Finally, it restores the painter and overlays HUD elements: fuel bar, coin counter, nitro HUD, flip HUD, distance, score and key log.

```
1 void MainWindow::paintEvent(QPaintEvent *event) {
2     Q_UNUSED(event);
3     QPainter p(this);
4     p.setRenderHint(QPainter::Antialiasing, true);
5     p.setPen(Qt::NoPen);
6
```

```
7       const int camGX = m_cameraX / Constants::PIXEL_SIZE;
8       const int camGY = m_cameraY / Constants::PIXEL_SIZE;
9       const int offX  = -(m_cameraX - camGX * Constants::PIXEL_SIZE);
10      const int offY  =  (m_cameraY - camGY * Constants::PIXEL_SIZE);
11
12      p.save();
13      p.translate(offX, offY);
14
15      if (m_showGrid) { drawGridOverlay(p); }
16      drawStars(p);
17      drawClouds(p);
18      drawFilledTerrain(p);
19      m_propSys.draw(p, m_cameraX, m_cameraY, width(), height(), m_heightAtGX);
20      m_fuelSys.drawWorldFuel(p, m_cameraX, m_cameraY);
21      m_coinSys.drawWorldCoins(p, m_cameraX, m_cameraY, gridW(), gridH());
22      m_nitroSys.drawFlame(p, m_wheels, m_cameraX, m_cameraY, width(), height());
23
24      for (const Wheel* wheel : m_wheels) {
25          if (auto info = wheel->get(0, 0, width(), height(), -m_cameraX, m_cameraY)
                ) {
26              const int cx = (*info)[0];
27              const int cy = (*info)[1];
28              const int r  = (*info)[2];
29              if(r == 0) continue;
30              const int gcx = cx / Constants::PIXEL_SIZE;
31              const int gcy = cy / Constants::PIXEL_SIZE;
32              const int gr  = r  / Constants::PIXEL_SIZE;
33              drawCircleFilledMidpointGrid(p, gcx, gcy, gr, Constants::
                    WHEEL_COLOR_OUTER);
34              const int tyreCells = std::max(1, Constants::TYRE_THICKNESS /
                    Constants::PIXEL_SIZE);
35              const int innerR = std::max(1, gr - tyreCells);
36              drawCircleFilledMidpointGrid(p, gcx, gcy, innerR, Constants::
                    WHEEL_COLOR_INNER);
37          }
38      }
39
40      for(CarBody* body : m_bodies){
41          auto pts = body->get(-m_cameraX, m_cameraY);
42          QVector<QPoint> normalisedPoints;
43          normalisedPoints.reserve(pts.size());
44          for(auto p2 : pts){
45              normalisedPoints.append(QPoint(p2.x() / Constants::PIXEL_SIZE, p2.y()
                    / Constants::PIXEL_SIZE));
46          }
47          fillPolygon(p, normalisedPoints, Constants::CAR_COLOR);
48
49          auto attach = body->getAttachments(-m_cameraX, m_cameraY);
50          for (const auto& ap : attach) {
51              QVector<QPoint> norm;
52              norm.reserve(ap.first.size());
53              for (const QPoint& q : ap.first) {
```

```
54                  norm.append(QPoint(q.x() / Constants::PIXEL_SIZE, q.y() /
                        Constants::PIXEL_SIZE));
55              }
56              fillPolygon(p, norm, ap.second);
57          }
58      }
59      m_flip.drawWorldPopups(p, m_cameraX, m_cameraY, level_index);
60
61      p.restore();
62
63      drawHUDFuel(p);
64      drawHUDCoins(p);
65      m_nitroSys.drawHUD(p, m_elapsedSeconds, level_index);
66      m_flip.drawHUD(p, level_index);
67      drawHUDDistance(p);
68      drawHUDScore(p);
69      m_keylog.draw(p, width(), height(), Constants::PIXEL_SIZE);
70  }
```

Feature: Physics-Based Spring-Damper Suspension

Relevant classes and functions: `Wheel::simulate()`, `CarBody::simulate()`, `CarBody::addWheel()`.

To simulate realistic vehicle suspension and structural integrity , I implemented a dynamic mass-spring-damper model. This system governs the interaction between the main `CarBody` and its attached `Wheel` objects, as well as the constraints between the wheels themselves. Rather than using rigid body kinematics, I treat the connections as dynamic springs that obey Hooke's Law and utilize viscous damping to stabilize the system.

Mathematical Formulation. Consider the chassis center of mass at position $\mathbf{p}_C$ and a wheel at position $\mathbf{p}_W$. The vector connecting them is $\vec{r} = \mathbf{p}_W - \mathbf{p}_C$. I calculate the Euclidean distance d and the normalized unit vector $\hat{u}$ as:

$$d = \|\vec{r}\| = \sqrt{(x_W - x_C)^2 + (y_W - y_C)^2}, \quad \hat{u} = \frac{\vec{r}}{d}.$$

Let L_0 be the rest length (stored as `desiredDistance`). The scalar displacement x from equilibrium is:

$$x = d - L_0.$$

According to Hooke's Law, the magnitude of the restoring spring force F_s is proportional to this displacement using a spring constant k:

$$F_s = k \cdot x.$$

This force acts along the spring axis. The force vector acting on the wheel (pulling/pushing it relative to the body) is:

$$\mathbf{F}_{\text{spring}} = -F_s \cdot \hat{u}.$$

Projected Viscous Damping. A pure spring system oscillates indefinitely. To simulate the energy dissipation of a real shock absorber , I implemented a damping force that resists motion *only along the axis*

of the spring. This prevents the "jelly" effect where tangential motion is incorrectly dampened.

First, I calculate the relative velocity vector between the wheel and the body:

$$\mathbf{v}_{\text{rel}} = \mathbf{v}_W - \mathbf{v}_C.$$

I then project this relative velocity onto the spring's unit vector $\hat{u}$ using the dot product to find the closing speed v_{proj}:

$$v_{\text{proj}} = \mathbf{v}_{\text{rel}} \cdot \hat{u} = (v_{x,\text{rel}} \cdot u_x + v_{y,\text{rel}} \cdot u_y).$$

The damping force magnitude is proportional to this projected velocity by a damping coefficient c (`Constants::DAMPING`). The resulting damping force vector is:

$$\mathbf{F}_{\text{damping}} = c \cdot v_{\text{proj}} \cdot \hat{u}.$$

Integration and Action-Reaction. The total force applied to the components is the sum of the spring and damping forces. By Newton's Third Law, the forces applied to the wheel and the car body are equal and opposite:

$$\mathbf{F}_{\text{wheel}} = \mathbf{F}_{\text{spring}} - \mathbf{F}_{\text{damping}},$$
$$\mathbf{F}_{\text{body}} = -\mathbf{F}_{\text{wheel}}.$$

I integrate these forces into the velocity of the objects using a semi-implicit Euler approach. For the car body, the velocity update is:

$$\mathbf{v}_C^{t+1} = \mathbf{v}_C^t + \mathbf{F}_{\text{body}}.$$

Core code snippets. The following excerpts demonstrate the spring logic. Note specifically the dot-product projection used in 'CarBody' for accurate suspension damping.

carBody.cpp (Suspension Logic)

```
// Inside CarBody::simulate(...)
for (int i = 0; m_isAlive && i < m_wheels.size(); i++) {
    Wheel* wheel = m_wheels.at(i);
    double desiredDistance = m_attachDistances.at(i);

    // 1. Geometry (Vector and Unit Vector)
    double deltaX = wheel->getX() - m_cx;
    double deltaY = wheel->getY() - m_cy;
    double actualDistance = std::sqrt(deltaX * deltaX + deltaY * deltaY);

    if (actualDistance == 0) continue;
    double unitX = -deltaX / actualDistance;
    double unitY =  deltaY / actualDistance;

    // 2. Hooke's Law (Spring Force)
    double displacement = actualDistance - desiredDistance;
    double springForceMagnitude = displacement * Constants::SPRING_CONSTANT;
    double forceX = unitX * springForceMagnitude;
    double forceY = unitY * springForceMagnitude;
```

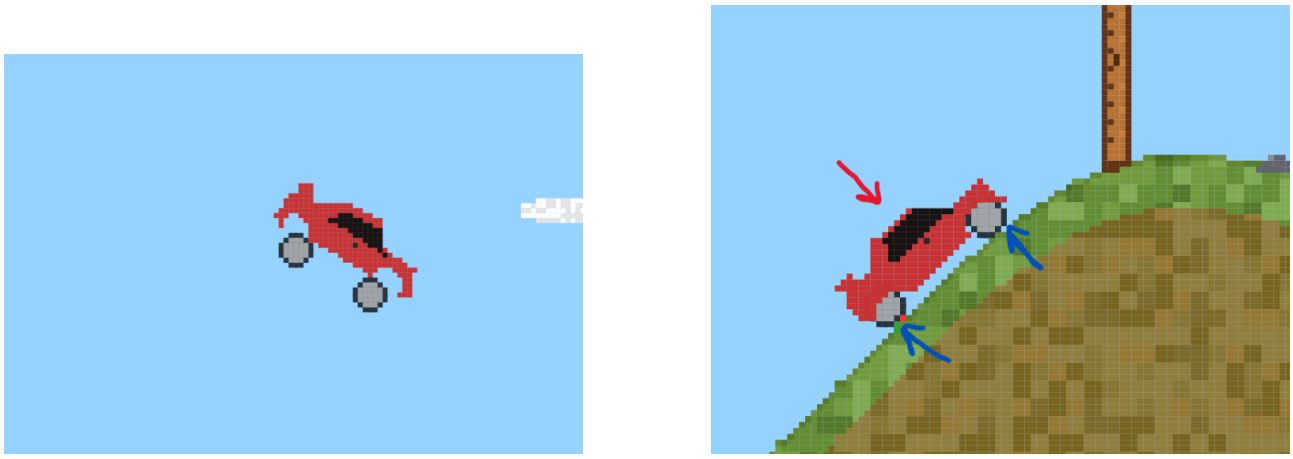
```
20
21      // 3. Projected Damping
22      // Calculate relative velocity
23      double relativeVx = wheel->getVx() - m_vx;
24      double relativeVy = wheel->getVy() - m_vy;
25
26      // Project relative velocity onto the spring axis (Dot Product)
27      // This isolates the component of velocity that is stretching/compressing the
            spring
28      double velocityAlongSpring = relativeVx * unitX + relativeVy * unitY;
29
30      // Calculate damping force components based ONLY on that projection
31      double dampingForceX = velocityAlongSpring * unitX * Constants::DAMPING;
32      double dampingForceY = velocityAlongSpring * unitY * Constants::DAMPING;
33
34      // 4. Integration (Action-Reaction)
35      m_vx -= (forceX - dampingForceX);
36      m_vy -= (forceY - dampingForceY);
37      wheel->updateV(forceX - dampingForceX, forceY - dampingForceY);
38  }
```

wheel.cpp (Inter-wheel constraints)

```
1   // Inside Wheel::simulate(...)
2   for (int i = 0; i < m_others.size(); ++i) {
3       Wheel* other = m_others.at(i);
4       double desiredDistance = m_distances.at(i);
5
6       // Calculate displacement and unit vectors
7       double deltaX = other->x - x;
8       double deltaY = other->y - y;
9       double actualDistance = std::sqrt(deltaX * deltaX + deltaY * deltaY);
10      double unitX = -deltaX / actualDistance;
11      double unitY =  deltaY / actualDistance;
12
13      // Hooke's Law
14      double displacement = actualDistance - desiredDistance;
15      double springForceMagnitude = displacement * Constants::SPRING_CONSTANT;
16      double forceX = unitX * springForceMagnitude;
17      double forceY = unitY * springForceMagnitude;
18
19      // Simplified Damping for wheel-to-wheel struts
20      double relativeVx = other->m_vx - m_vx;
21      double relativeVy = other->m_vy - m_vy;
22      double dampingForceX = relativeVx * Constants::DAMPING;
23      double dampingForceY = relativeVy * Constants::DAMPING;
24
25      // Apply forces
26      m_vx -= (forceX - dampingForceX);
27      m_vy -= (forceY - dampingForceY);
28      other->m_vx += (forceX - dampingForceX);
29      other->m_vy += (forceY - dampingForceY);
30  }
```

Spring Model Results

The implementation resulted in a "soft-body" feel for the vehicle. The projection-based damping in the ‘CarBody‘ was particularly effective; it allowed the suspension to absorb vertical impacts from jumps (where $\vec{v}_{\text{rel}}$ aligns with $\hat{u}$) without artificially slowing down the car's rotational or horizontal motion, which would occur with naive global damping. The car settles back to its resting height naturally after hitting a bump, simulating a working suspension system.



(a) Wheels are suspended freely when in air

(b) Springs absorb impact on terrain collision

Figure 2: Spring model: Simulation of a working suspension system using dampened distance-locked springs.

Feature: Terrain Collision and Impulse Resolution

Relevant classes and functions: `Wheel::simulate()`, `CarBody::simulate()`, `Line::getSlope()`.

To ensure the vehicle interacts physically with the procedural terrain, I implemented a discrete collision detection system based on analytical geometry. Rather than using axis-aligned bounding boxes (AABB), which are unsuitable for the steep, irregular slopes of my terrain, I perform precise point-to-line distance checks for every wheel and every defining vertex of the car's hitbox.

[Image of vector projection physics diagram]

Distance and Intersection Testing. The terrain is stored as a list of line segments. For a given segment defined by slope m and intercept b (equation $y = mx + b$), and a collision point (p_x, p_y), I formulate the line equation as $mx - y + b = 0$. The perpendicular distance d from the point to the infinite line is derived as:

$$d = \frac{|m \cdot p_x - p_y + b|}{\sqrt{m^2 + 1}}.$$

To confirm the collision is valid, I calculate the projected x-coordinate of the intersection, x_{int}, to ensure it lies within the segment's bounds $[x_1, x_2]$:

$$x_{\text{int}} = \frac{m(p_y - b) + p_x}{m^2 + 1}.$$

A collision is detected if $x_1 \leq x_{\text{int}} \leq x_2$ and $d < r$, where r is the collision radius (the wheel radius or the hitbox tolerance).

Coordinate Frame Transformation. Applying friction and restitution directly in global coordinates is difficult on slanted terrain. Therefore, upon collision, I rotate the velocity vector $\mathbf{v} = (v_x, v_y)$ into the surface's local reference frame defined by the tangent angle $\theta = -\arctan(m)$. The components parallel ($v_\parallel$) and perpendicular ($v_\perp$) to the surface are:

$$\begin{bmatrix} v_\parallel \\ v_\perp \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} v_x \\ v_y \end{bmatrix}.$$

Impulse Response and Integration. Once in the local frame, I apply physical forces naturally. 1. **Position Correction:** To prevent tunneling, I project the object out of the terrain along the surface normal by the penetration depth $(r - d)$. 2. **Restitution (Bouncing):** I apply an impulse to the normal velocity. For the car body, I implemented a non-linear restitution function using a sigmoid scaling factor to prevent explosive energy gain at high impact speeds:

$$v'_\perp = v_\perp \cdot \epsilon \cdot \frac{1}{1 + e^{-v_\perp}},$$

where ϵ is the restitution coefficient. 3. **Friction and Traction:** I apply kinetic friction to the tangential velocity. If the object is a driving wheel, I also inject acceleration or braking forces directly into $v_\parallel$:

$$v'_\parallel = v_\parallel \cdot (1 - \mu) + F_{\text{drive}}.$$

Finally, I rotate the modified components $v'_\parallel$ and $v'_\perp$ back to the global Cartesian frame to update the object's state for the next integration step.

Core code snippets. The following excerpts demonstrate the collision mathematics, specifically the projection logic and frame rotation.

wheel.cpp (Collision Logic)

```
// Inside Wheel::simulate(...)
for (const Line& line : lines) {
    double m = line.getSlope();
    double b = line.getIntercept();

    // 1. Analytical Distance Check
    double dist = std::abs(m * x - y + b) / std::sqrt(m * m + 1);
    double intersection_x = (m * (y - b) + x) / (m * m + 1);

    int minX = std::min(line.getX1(), line.getX2());
    int maxX = std::max(line.getX1(), line.getX2());

    if (dist < std::max(1, m_radius) && intersection_x >= minX && intersection_x
        <= maxX) {
        double overlap = m_radius - dist;
        double normal_angle = std::atan2(-m, 1.0);

        // 2. Position Correction (Anti-Tunneling)
        y -= overlap * std::cos(normal_angle);
        x -= overlap * std::sin(normal_angle);

```

```
        // 3. Frame Rotation (World -> Slope)
        double theta = -std::atan(m);
        double vAlongLine    = m_vx * std::cos(theta) + m_vy * std::sin(theta);
        double vNormalToLine = m_vy * std::cos(theta) - m_vx * std::sin(theta);

        // 4. Physical Response
        // Bounce
        vNormalToLine *= (vNormalToLine < 0.2) ? Constants::RESTITUTION[
            level_index] : 1;

        // Friction
        if ((vAlongLine > Constants::MAX_VELOCITY/1000) || (vAlongLine < -
            Constants::MAX_VELOCITY/1000)) {
            vAlongLine *= 1 - Constants::FRICTION[level_index];
        }

        // Traction (Acceleration/Braking applied to tangent)
        if (accelerating && isAlive && vAlongLine < Constants::MAX_VELOCITY) {
            vAlongLine += (Constants::ACCELERATION * ... * cos(theta));
        }

        // 5. Frame Rotation (Slope -> World)
        m_vx = vAlongLine * std::cos(theta) - vNormalToLine * std::sin(theta);
        m_vy = vAlongLine * std::sin(theta) + vNormalToLine * std::cos(theta);
    }
}
```

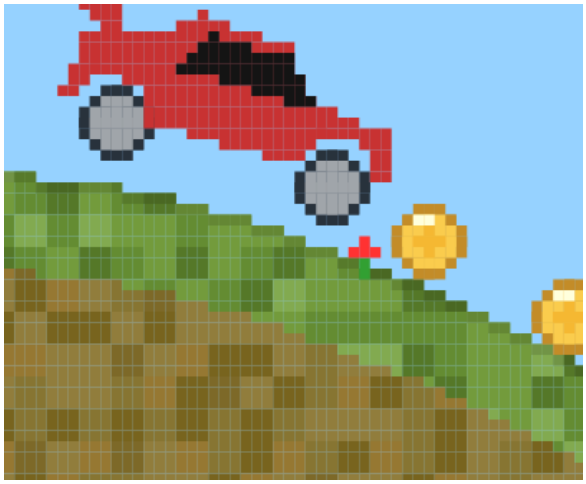
carBody.cpp (Hitbox Collision)

```
// Inside CarBody::simulate(...)
for (const Point& point : hitbox) {
    // ... (Distance calculation similar to Wheel) ...

    if (dist <= 4 && intersectionx >= line.getX1() - 1 && intersectionx <= line.
        getX2() + 1) {
        double theta = -std::atan(m);

        // Position Correction Loop
        while (dist < 4) {
            m_cy -= std::cos(theta);
            m_cx -= std::sin(theta);
            // ... recalculate dist ...
        }

        // Frame Rotation
        double vAlongLine = m_vx * std::cos(theta) + m_vy * std::sin(theta);
        double vNormalToLine = m_vy * std::cos(theta) - m_vx * std::sin(theta);

        // Sigmoid-damped Restitution
        vNormalToLine = vNormalToLine * Constants::RESTITUTION[level_index] / (1 +
            std::exp(-vNormalToLine));
        vAlongLine *= 1 - Constants::FRICTION[level_index];

```

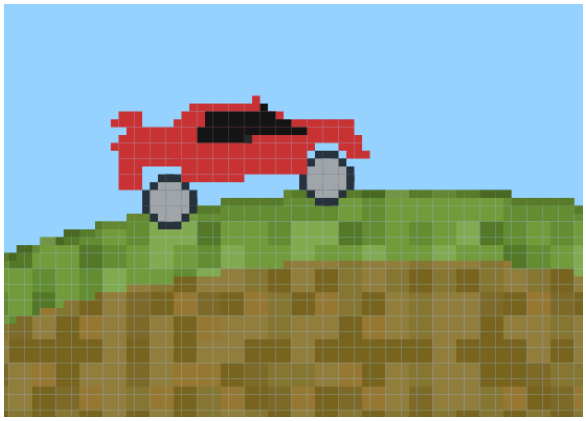
```
23          // Rotate back
24          m_vx = vAlongLine * std::cos(theta) - vNormalToLine * std::sin(theta);
25          m_vy = vAlongLine * std::sin(theta) + vNormalToLine * std::cos(theta);
26      }
27  }
```

Collision Model Results

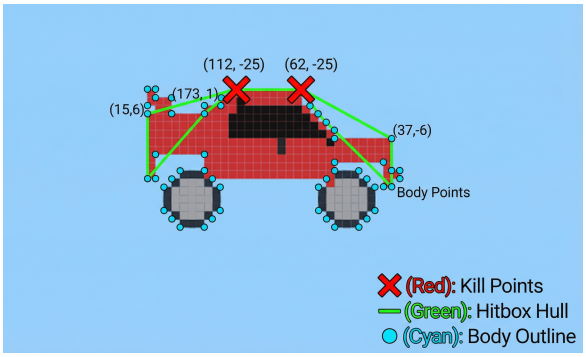
The analytical collision model proved robust against high-speed impacts and irregular terrain geometry. By rotating velocity vectors into the slope's reference frame, the simulation correctly distinguishes between normal forces (bouncing) and tangential forces (friction/traction). This allows the car to climb steep hills where a simple global-axis friction model would fail. The non-linear restitution in the car body successfully prevents numerical instability when the chassis drags along the ground, ensuring that the vehicle slides rather than vibrating uncontrollably.



(a) No clipping into terrain - no collision



(b) Clipping into terrain - collision code activated



(c) Separation of display hull and collision hitbox

Figure 3: Collision detection system: Check point-line distances of wheel centre and car chassis from terrain to ensure car always stays above terrain and responds to collisions naturally.

Feature: Centralized Physics and Procedural Configuration

Relevant file: `constants.h`

To ensure the game engine remains data-driven and easily tunable, I consolidated all physical coefficients, procedural generation parameters, and game balance variables into a single static structure, `Constants`. This design pattern allows for the definition of distinct "biomes" (Levels) by simply altering vectors of values, without modifying the core simulation logic.

Level-Dependent Physics Vectors. The game features multiple environments (e.g., Meadow, Lunar, Martian). To model these distinct physical realities, I utilized `QVector<double>` containers where the index i corresponds to the active level.

- **Gravity (g):** Defined in `GRAVITY`. On the "Lunar" level (index 3), gravity is reduced to 0.04 (half the standard 0.08), directly affecting the vertical velocity integration:

$$v_{y,t+1} = v_{y,t} - \mathbf{g}[i].$$

- **Atmospheric Drag (C_d):** Defined in `AIR_RESISTANCE`. This simulates the density of the atmosphere. For space levels, this approaches zero (10^{-5}), minimizing velocity decay:

$$\mathbf{v}_{t+1} = \mathbf{v}_t \cdot (1 - C_d[i]).$$

- **Surface Interaction (μ, ϵ):** Ground interaction is governed by `FRICTION` (μ) and `RESTITUTION` (ϵ). These determine the loss of tangential velocity and the elasticity of normal collisions respectively.

Procedural Terrain Parametrization. The infinite terrain generation relies on stochastic recurrence relations. I defined initial values and incremental scalers to make the terrain progressively more difficult as the player travels further.

- **Slope Volatility (D):** Controlled by `DIFFICULTY_INCREMENT`. As the game progresses, the range of the random variable ξ used to perturb the slope expands:

$$D_{t+1} = D_t + \texttt{DIFFICULTY\_INCREMENT}[i].$$

- **Non-Linearity (γ):** Controlled by `IRREGULARITY`. This constant powers the slope function, creating sharper peaks and valleys. The height step Δy is calculated as:

$$\Delta y \propto m \cdot |m|^{\gamma}, \quad \gamma = \texttt{m\_irregularity}.$$

Vehicle Geometry and Suspension Constants. The vehicle chassis is not a sprite but a polygon defined physically by vertices relative to a center of mass.

- **Suspension Dynamics:** The mass-spring-damper system is tuned via `SPRING_CONSTANT` ($k = 0.6$) and `DAMPING` ($c = 0.06$). These global constants ensure consistent suspension behavior across all levels, isolating the handling changes to external factors (gravity/traction).

- **Chassis Definition:** I defined the car body using `CAR_BODY_POINTS` for rendering and `CAR_HITBOX_POINTS` for collision. This separation allows for a detailed visual mesh while maintaining a slightly smaller, optimized convex hull for physics calculations to prevent snagging on terrain micro-geometry.

Core code snippets. The following excerpts from `constants.h` illustrate the vector-based level configuration and the hard-coded physics parameters.

```
struct Constants {
    static constexpr int PIXEL_SIZE = 6;

    // LEVEL MECHANICS (Index-based configuration)
    // Index: 0=Meadow, 1=Desert, 2=Tundra, 3=Lunar, 4=Martian, 5=Nightlife
    inline static QVector<double> GRAVITY           = {0.08, 0.08, 0.08, 0.04, 0.06, 0.08};
    inline static QVector<double> AIR_RESISTANCE    = {0.0005, 0.0003, 0.0007, 0.00001, 0.00005, 0.0007};
    inline static QVector<double> RESTITUTION       = {0.8, 0.5, 0.8, 0.7, 0.07, 0.5}; // Bounciness
    inline static QVector<double> FRICTION          = {0.003, 0.03, 0.0001, 0.001, 0.03, 0.03};
    inline static QVector<double> TRACTION          = {1, 1.25, 0.5, 0.5, 0.75, 1.5}; // Grip multiplier

    // TERRAIN GENERATION CONSTANTS
    static constexpr int STEP = 20; // Horizontal distance between vertices
    // Maximum slope clamping per level
    inline static QVector<double> MAX_SLOPE = {1.0, 1.5, 0.8, 2, 1.5, 0.5};

    // Difficulty Evolution parameters
    inline static QVector<double> INITIAL_DIFFICULTY = {0.005, 0.005, 0.008, 0.01, 0.008, 0.001};
    inline static QVector<double> DIFFICULTY_INCREMENT = {0.0001, 0.0002, 0.0001, 0.0003, 0.0002, 0.0001};

    // Irregularity (Power law exponent) parameters
    inline static QVector<double> INITIAL_IRREGULARITY = {0.01, 0.001, 0.01, 0.05, 0.02, 0.05};
    inline static QVector<double> IRREGULARITY_INCREMENT = {0.00001, 0.000001, 0.00003, 0.0001, 0.00005, 0.0001};

    // CAR MECHANICS (Physics Model)
    static constexpr double MAX_VELOCITY    = 30.0;
    static constexpr double ACCELERATION    = 0.8;
    static constexpr double DECELERATION    = 0.8;

    // Suspension (Hooke's Law & Damping)
    static constexpr double SPRING_CONSTANT = 0.6;
    static constexpr double DAMPING         = 0.06;

    // Angular Dynamics (Rotation control)
    static constexpr double ANGULAR_ACCELERATION = 0.0010;
    static constexpr double ANGULAR_DECELERATION = 0.0010;
```

```
37      static constexpr double ANGULAR_DAMPING      = 0.05;
38
39      // CHASSIS GEOMETRY
40      // Points relative to Center of Mass (0,0)
41      inline static const QVector<QPoint> CAR_BODY_POINTS = {
42          QPoint(0,0), QPoint(0,31), QPoint(9,37), QPoint(15,19), /* ... */
43      };
44
45      inline static const QVector<QPoint> CAR_HITBOX_POINTS = {
46          QPoint(15,6), QPoint(173, 1),  QPoint(137,0), QPoint(112,-25), /* ... */
47      };
48  };
```

Configuration Results

By separating these constants, I achieved a highly flexible simulation. For instance, creating the "Lunar" level required no changes to the physics engine code; I simply adjusted the `GRAVITY` vector at index 3 to 0.04 and `AIR_RESISTANCE` to near-zero. Similarly, the `DIFFICULTY_INCREMENT` ensures that the procedural terrain remains drivable at the start but inevitably becomes impassable, creating a natural difficulty curve for an endless-runner style game.

Car Creation

```
void MainWindow::createCar() {
    Wheel* w1 = new Wheel(Constants::WHEEL_REAR_X,  Constants::WHEEL_REAR_Y,
        Constants::WHEEL_REAR_R);
    Wheel* w2 = new Wheel(Constants::WHEEL_FRONT_X, Constants::WHEEL_FRONT_Y,
        Constants::WHEEL_FRONT_R);
    Wheel* w3 = new Wheel(Constants::WHEEL_MID_X,   Constants::WHEEL_MID_Y,
        Constants::WHEEL_MID_R);

    w1->attach(w2); w1->attach(w3); w2->attach(w3);
    m_wheels.append(w1); m_wheels.append(w2); m_wheels.append(w3);

    CarBody* body = new CarBody();
    body->addPoints(Constants::CAR_BODY_POINTS);
    body->addHitbox(Constants::CAR_HITBOX_POINTS);
    body->addKillSwitches(Constants::CAR_KILL_POINTS);

    body->addWheel(w1); body->addWheel(w2); body->addWheel(w3);

    body->addAttachment(Constants::CAR_GLASS_POINTS, Constants::CAR_GLASS_COLOR);
    body->addAttachment(Constants::CAR_HANDLE_POINTS, Constants::CAR_HANDLE_COLOR)
        ;

    body->finish();
    m_bodies.append(body);
}
```

Wheel Physics and Rotational Dynamics

Relevant classes and functions: `Wheel`, `Wheel::simulate()`, `Wheel::attach()`, `Wheel::updateV()`.

The vehicle dynamics in this simulation are built upon a composite rigid-body system. While the suspension constraints are handled via a spring-damper model (detailed in the Spring Physics section), kinematics and rotation are managed by the `Wheel` class. The physics engine integrates linear and angular velocities, resolves constraints against terrain segments, and applies user control inputs (acceleration, braking, and nitro).

Integration and Environmental Forces

Relevant code: `Wheel::simulate()`

The physical state of each wheel is advanced using a semi-implicit Euler integration scheme. This method was chosen for its computational efficiency and stability in games where frame-to-frame velocity changes can be discontinuous (e.g., instant collisions).

Position Integration. For a wheel with position $\mathbf{p} = (x, y)$ and velocity $\mathbf{v} = (v_x, v_y)$, the position update per frame t occurs after all forces and impulses have modified the velocity:

$$x_{t+1} = x_t + v_{x,t}$$
$$y_{t+1} = y_t - v_{y,t} \quad \text{(Note: } y \text{ decreases as } v_y \text{ increases due to screen-space coordinates)}$$

Gravity and Active Downforce. Prior to integration, the vertical velocity v_y is updated by the gravitational constant g. In a standard state, this is a simple subtraction:

$$v_{y,t} \leftarrow v_{y,t} - g \cdot \Delta t$$

However, the simulation includes an "Active Aerodynamics" mechanic. When the player presses both **Accelerate** and **Brake** simultaneously while airborne, the car engages a virtual spoiler. This action applies a down-thrust effectively increasing the gravitational pull by 50%.

$$v_{y,t} \leftarrow v_{y,t} - (g \cdot 0.5) \cdot \Delta t \quad \text{if (Accel AND Brake)}$$

This mechanic serves a tactical purpose: it allows players to force the car down from high jumps quickly ("fast falling"), ensuring they can land on specific terrain features or return to the ground to regain traction and speed sooner. This couples with the angular stabilization logic (described in the Angular Dynamics section) to provide a "stabilized landing" maneuver.

Air Resistance (Drag). To prevent infinite acceleration and simulate atmospheric density, a linear drag factor k_{drag} is applied to both velocity components. This is implemented as a multiplicative damping factor, which provides a smooth terminal velocity curve:

$$\mathbf{v} \leftarrow \mathbf{v} \cdot (1 - k_{\text{drag}}).$$

The following code snippet from `Wheel::simulate` illustrates the implementation of these environmental forces:

```
// 1. Integration of position (Euler)
x += m_vx;
y -= m_vy; // Screen space: Up is negative Y

// 2. Gravity Application
m_vy -= Constants::GRAVITY[level_index];

// Active Downforce / "Spoiler" Mechanic
// If both inputs are active, apply extra down-thrust for fast landing
if(accelerating && braking){
    m_vy -= Constants::GRAVITY[level_index] * 0.5;
}

// 3. Air Drag Application
// Multiplicative damping simulates air resistance
m_vx *= 1 - Constants::AIR_RESISTANCE[level_index];
m_vy *= 1 - Constants::AIR_RESISTANCE[level_index];
```

Angular Dynamics and the `isRoot` Hierarchy Relevant code: `Wheel::attach()`, `Wheel::simulate()`.

A fundamental challenge in simulating a 2D vehicle composed of discrete point-mass wheels is managing the rigid-body rotation of the chassis. Since the chassis itself is not physically simulated as a collision body (only the wheels interact with the terrain), the rotation must be derived from the relationship between the two wheels. To prevent numerical divergence where wheels might simulate conflicting rotations, I implemented a hierarchical ownership model using the `m_isRoot` flag.

The `m_isRoot` Hierarchy. When the vehicle is assembled via the `attach()` method, one wheel is designated as the "Root" (master) and the other as the slave.

$$\text{Let } W_{root} \text{ and } W_{slave} \text{ be the wheel pair.}$$

The `m_isRoot` boolean ensures that angular physics calculations occur exactly once per vehicle per frame. The Root wheel is responsible for:

1. Storing the vehicle's angular velocity ω (`m_omega`).
2. Interpreting control inputs (throttle/brake) into torque.
3. Applying the geometric rotation to both itself and the attached slave wheel.

This centralization is crucial. Without it, floating-point errors would accumulate differently on each wheel, causing the fixed distance constraint to fight against the rotation, leading to jitter or explosion of the constraints.

Angular Velocity as Incremental Rotation ($d\theta$). In this simulation, the variable `m_omega` represents the angular velocity scaled by the timestep Δt.

$$\texttt{m\_omega} \approx \omega \cdot \Delta t = d\theta$$

Consequently, in the integration step, `m_omega` is used directly as the angle of rotation for the current frame. This simplification avoids explicit storage of the vehicle's absolute angle for physics purposes, relying instead on incremental updates to the position vector $\mathbf{r}$.

Control Inputs and Torque Application. The simulation applies angular impulses based on the player's input state. The system models torque τ as a direct modification of ω.

- **Acceleration (Nose-Up / Wheelie):** When the player accelerates, a positive torque is applied. Physically, this reaction torque opposes the wheels' forward spin. In the code, this increments ω:

  $$\omega_{t+1} = \omega_t + \alpha_{\text{accel}} \quad \text{(Counter-Clockwise rotation)}$$

  This causes the vehicle to tilt backwards (wheelie) when facing right, allowing the player to perform backflips.

- **Braking (Nose-Down):** Braking applies negative torque, mimicking the inertia of the chassis continuing forward while wheels slow down:

$$\omega_{t+1} = \omega_t - \alpha_{\text{brake}} \quad \text{(Clockwise rotation)}$$

  This is used for front-flips or correcting a wheelie.

- **Nitro Stabilization:** When nitro is engaged, the gameplay requirement is high-speed stability. We apply a heavy damping factor $k_{\text{nitro}} \in (0, 1)$:

$$\omega_{t+1} = \omega_t \cdot (1 - k_{\text{nitro}}).$$

  This rapidly decays any rotational velocity, ensuring the car "darts" straight forward rather than spinning out of control.

Mid-Air Stabilization. A special case arises when both **Accelerate** and **Brake** are pressed simultaneously while airborne. This triggers an automatic stabilization feature intended to help the player land safely. The system calculates the current pitch angle θ_{curr} relative to the horizon using `atan2`:

$$\theta_{curr} = \text{atan2}(y_{slave} - y_{root}, x_{slave} - x_{root}).$$

The goal is to drive $\theta_{curr} \to 0$. The code implements a Proportional controller (P-controller) logic:

$$\Delta\omega = \begin{cases} +\alpha & \text{if } \theta_{curr} > \epsilon \text{ (Nose too low, rotate up)} \\ -\alpha & \text{if } \theta_{curr} < -\epsilon \text{ (Nose too high, rotate down)} \end{cases}$$

This active correction, combined with aggressive damping, aligns the chassis horizontally for a perfect landing.

Geometric Rotation Implementation. Once the frame's rotation angle $d\theta$ (stored as `m_omega`) is determined, it is applied to the positions of both wheels. First, the centroid (Center of Mass, C) of the system is computed:

$$C_x = \frac{x_{root} + x_{slave}}{2}, \quad C_y = \frac{y_{root} + y_{slave}}{2}.$$

The position vector $\mathbf{r}$ of each wheel relative to C is rotated using the 2D rotation matrix $R(d\theta)$. The code implementation for the updated relative coordinates (nx, ny) is:

```
// Code snippet representing the matrix multiplication:
// | nx |   |  cos(w)  sin(w) | | rx |
// | ny | = | -sin(w)  cos(w) | | ry |
double nx = rx * cosA + ry * sinA;
double ny = -rx * sinA + ry * cosA;
```

Note: The negative sign appears in the y calculation because of the screen-space coordinate system (y increases downwards).

The following code snippet demonstrates the core rotational logic inside `Wheel::simulate()`:

```
// shared body tilt / rotation between two wheels (nitro-aware)
```

```
if (m_isRoot && m_others.size() > 0) {
    Wheel* other = m_others.first();

    // Angular Control Logic
    if (nitro) {
        // 1. Nitro: Heavy damping for straight flight
        m_omega *= 1 - Constants::ANGULAR_DAMPING;
        if (std::abs(m_omega) < 1e-4) m_omega = 0.0;
    }
    else if (accelerating && braking) {
        // 2. Stabilization: Revert to horizontal angle
        m_angle = std::atan2(other->getY() - this->getY(), other->getX() - this->
            getX());

        // Apply corrective torque based on current angle
        if (std::abs(m_angle) > 1e-2) {
            if (m_angle > 0) m_omega += Constants::ANGULAR_ACCELERATION;
            else m_omega -= Constants::ANGULAR_DECELERATION;
        }
        // Apply damping to prevent oscillation
        m_omega *= 1 - Constants::ANGULAR_DAMPING;
        if (std::abs(m_omega) < 1e-4) m_omega = 0.0;
    }
    else if (accelerating) {
        // 3. Accel: Tilt Backward (Counter-Clockwise)
        m_omega += Constants::ANGULAR_ACCELERATION;
        if (m_omega > Constants::MAX_ANGULAR_VELOCITY)
            m_omega = Constants::MAX_ANGULAR_VELOCITY;
    }
    else if (braking) {
        // 4. Brake: Tilt Forward (Clockwise)
        m_omega -= Constants::ANGULAR_DECELERATION;
        if (m_omega < -Constants::MAX_ANGULAR_VELOCITY)
            m_omega = -Constants::MAX_ANGULAR_VELOCITY;
    }
    else {
        // 5. Passive Damping (Air resistance equivalent)
        m_omega *= 1 - Constants::ANGULAR_DAMPING;
        if (std::abs(m_omega) < 1e-4) m_omega = 0.0;
    }

    // Apply incremental rotation of the wheel pair about COM
    if (std::abs(m_omega) > 1e-6) {
        double cx = (x + other->x) / 2.0;
        double cy = (y + other->y) / 2.0;

        double rx1 = x - cx;      double ry1 = y - cy;
        double rx2 = other->x - cx; double ry2 = other->y - cy;

        double sinA = std::sin(m_omega);
        double cosA = std::cos(m_omega);

```

```
53          // Update Root Wheel
54          double nx1 = rx1 * cosA + ry1 * sinA;
55          double ny1 = -rx1 * sinA + ry1 * cosA; // Note Y-down adjustment
56          x = cx + nx1;
57          y = cy + ny1;
58
59          // Update Slave Wheel
60          double nx2 = rx2 * cosA + ry2 * sinA;
61          double ny2 = -rx2 * sinA + ry2 * cosA;
62          other->x = cx + nx2;
63          other->y = cy + ny2;
64      }
65  }
```

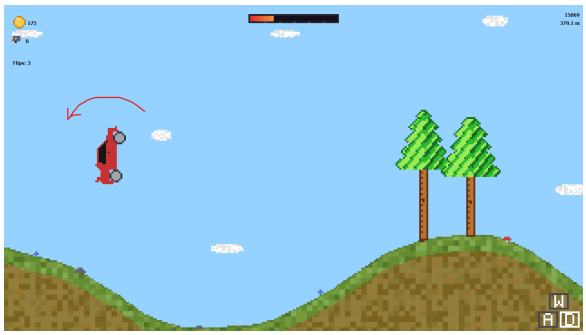
Results and Discussion

Responsiveness and Game Feel: The direct mapping of acceleration to counter-clockwise torque and braking to clockwise torque provided an intuitive "weight" to the car. Players could naturally perform backflips off ramps by holding acceleration, or front-flips by tapping the brake mid-air. This mechanic transformed the vehicle from a simple sliding box into a dynamic, controllable object.

Spoiler Activation: The case of both keys pressed gives a "spoiler" effect that increases down thrust, which helps in "snappier" landings whenever required. This goes together with the angular stabilization dynamic to provide a clean down thrust experience.

Stability vs. Agility: The `isRoot` hierarchy successfully solved the problem of wheel desynchronization. Without this flag, wheels would rotate around multiple pivots, breaking the illusion of a solid chassis, and adding double counting problems. The addition of the “Both Pressed” stabilization logic provided a necessary skill floor, allowing players to recover from bad jumps by instinctively pressing both buttons to level the car before impact.

Nitro Integration: The dynamic damping during nitro usage created a distinct “mode” of travel. The car feels significantly tighter and more aerodynamic when boosting, which contrasts satisfyingly with the floaty, rotational freedom of standard flight.



(a) On Accelerating, produce **anticlockwise** spin

(b) On Braking, produce **clockwise** spin

(c) On both acceleration and braking, produce **downthrust** and angular **stabilization** to **horizontal**

Figure 4: Different cases of generating **rotation**, **downthrust** and **stabilization**.

intro.h

```
// intro.h
#ifndef INTRO_H
#define INTRO_H

#include <QWidget>
#include <QTimer>
#include <QHash>
#include <QColor>
#include <QVector>
#include <QList>
#include "line.h"
#include "constants.h"
#include <QSettings>
#include <random>

class QPainter;
class QMouseEvent;
class QResizeEvent;

struct Cloud {
    int wx;
    int wyCells;
    int wCells;
    int hCells;
    quint32 seed;
};

class IntroScreen : public QWidget {
    Q_OBJECT
    public:
    explicit IntroScreen(QWidget* parent = nullptr, int levelIndex = 0);
    void setGrandCoins(int v);

    signals:
    void startRequested(int levelIndex);
    void exitRequested();

    protected:
    void paintEvent(QPaintEvent*) override;
    void mousePressEvent(QMouseEvent*) override;
    void resizeEvent(QResizeEvent*) override;

    private:
    void drawStars(QPainter& p);
    void drawClouds(QPainter& p);
    void maybeSpawnCloud();
    void drawBackground(QPainter& p);
    void drawFilledTerrain(QPainter& p);
    void plotGridPixel(QPainter& p, int gx, int gy, const QColor& c);
    void rasterizeSegmentToHeightMapWorld(int x1, int y1, int x2, int y2);
```

```
51              void pruneHeightMap();
52              void ensureAheadTerrain(int worldX);
53              QColor grassShadeForBlock(int worldGX, int worldGY, bool greenify)
                    const;
54
55              void drawPixelText(QPainter& p, const QString& s, int gx, int gy, int
                    scale, const QColor& c, bool bold);
56              int  textWidthCells(const QString& s, int scale) const;
57              int  fitTextScaleToRect(int wCells, int hCells, const QString& s)
                    const;
58
59              // Buttons / rects
60              QRect buttonRectStart() const;
61              QRect buttonRectExit() const;
62              QRect buttonRectUnlock() const;
63              QRect buttonRectLevelPrev() const;
64              QRect buttonRectLevelNext() const;
65
66              void saveUnlocks() const;
67              void loadUnlocks();
68              void saveGrandCoins() const;
69              void loadGrandCoins();
70
71              int stageLabelBottomPx() const;
72
73              // Grid helpers
74              inline int gridW() const { return width()  / PIXEL_SIZE; }
75              inline int gridH() const { return height() / PIXEL_SIZE; }
76              void drawCircleFilledMidpointGrid(QPainter& p, int gcx, int gcy, int
                    gr, const QColor& c);
77
78              QTimer m_timer;
79              double m_scrollX = 0.0;
80
81              QList<Line> m_lines;
82              QHash<int,int> m_heightAtGX;
83              QVector<Cloud> m_clouds;
84
85              int m_lastCloudSpawnX = 0;
86              int m_lastX = 0;
87              int m_lastY = 0;
88              float m_slope = 0.0f;
89              float m_difficulty = 0.005f;
90
91              static constexpr int   PIXEL_SIZE = 6;
92              static constexpr int   SHADING_BLOCK = 3;
93              static constexpr int   STEP = 20;
94              static constexpr float DIFF_INC = 0.0001f;
95              static constexpr int CLOUD_SPACING_PX = 700;
96              static constexpr int CLOUD_MIN_W_CELLS = 8;
97              static constexpr int CLOUD_MAX_W_CELLS = 18;
98              static constexpr int CLOUD_MIN_H_CELLS = 3;
```

```
        static constexpr int CLOUD_MAX_H_CELLS = 7;
        static constexpr int CLOUD_SKY_OFFSET_CELLS = 40;

        std::mt19937 m_rng;

        static constexpr int CHAR_ADV = 7;

        int m_camX = 0;
        int m_camY = 200;
        int m_camXFarthest = 0;

        qreal m_blurScale = 0.6;

        quint64 m_grandTotalCoins = 0;

        // Title helpers (existing)
        int titleScale() const;
        int titleYCells() const;
        int startTopCells(int titleScale) const;
        int exitTopCells(int btnHCells) const;

        int level_index;

        QVector<bool> levels_unlocked = {true, false, false, false, false};
    };

    #endif // INTRO_H
```

intro.cpp

```
    // intro.cpp
    #include "intro.h"
    #include <QPainter>
    #include <QMouseEvent>
    #include <QImage>
    #include <array>
    #include <cmath>
    #include <algorithm>

    constexpr int TITLE_STAGE_GAP_PX = 30;

    IntroScreen::IntroScreen(QWidget* parent, int levelIndex) : QWidget(parent
        ) {
        setAttribute(Qt::WA_OpaquePaintEvent);

        level_index = levelIndex;

        loadGrandCoins();
        loadUnlocks();

        connect(&m_timer, &QTimer::timeout, this, [this]{
```

```
21                  m_scrollX += 2.0;
22                  m_camX = int(m_scrollX);
23
24                  while ((m_camX + width()) > m_camXFarthest) {
25                      m_camXFarthest += STEP;
26
27                      m_slope += (0.5f - float(m_lastY) / std::max(1,height())) *
                            m_difficulty;
28                      m_slope = std::clamp(m_slope, -1.0f, 1.0f);
29
30                      const int newY = m_lastY + std::lround(m_slope * std::pow(std
                            ::abs(m_slope), 0.02f) * STEP);
31
32                      Line seg(m_lastX, m_lastY, m_lastX + STEP, newY);
33                      m_lines.append(seg);
34
35                      rasterizeSegmentToHeightMapWorld(seg.getX1(), m_lastY, seg.
                            getX2(), newY);
36
37                      m_lastY = newY;
38                      m_lastX += STEP;
39
40                      if (m_lines.size() > (width() / STEP) * 3) {
41                          m_lines.removeFirst();
42                          pruneHeightMap();
43                      }
44
45                      m_difficulty += DIFF_INC;
46                      maybeSpawnCloud();
47                  }
48
49                  update();
50              });
51
52              m_lastY = height() / 2;
53              for (int i = STEP; i <= width() + STEP; i += STEP) {
54                  m_slope += (0.5f - float(m_lastY) / std::max(1,height())) *
                        m_difficulty;
55                  m_slope = std::clamp(m_slope, -1.0f, 1.0f);
56
57                  const int newY = m_lastY + std::lround(m_slope * std::pow(std::abs
                        (m_slope), 0.02f) * STEP);
58
59                  Line seg(i - STEP, m_lastY, i, newY);
60                  m_lines.append(seg);
61
62                  rasterizeSegmentToHeightMapWorld(seg.getX1(), m_lastY, seg.getX2()
                        , newY);
63
64                  m_lastY = newY;
65                  m_difficulty += DIFF_INC;
66              }
```

```
67              m_lastX = STEP * m_lines.size();
68
69              m_timer.start(16);
70          }
71
72          void IntroScreen::setGrandCoins(int v){
73              m_grandTotalCoins = v;
74              update();
75          }
76
77          int IntroScreen::titleScale() const {
78              return std::clamp(gridH() / (7 * 12), 2, 4);
79          }
80
81          int IntroScreen::titleYCells() const {
82              return std::max(2, gridH() / 4);
83          }
84
85          int IntroScreen::startTopCells(int) const {
86              int btnH = std::max(10, gridH()/18);
87              int gap  = std::max(10,  gridH()/40);
88              int bottom = std::max(30, gridH()/24);
89              return gridH() - (2*btnH + gap + bottom);
90          }
91
92          int IntroScreen::exitTopCells(int btnHCells) const {
93              int bottom = std::max(30, gridH()/24);
94              return gridH() - (btnHCells + bottom);
95          }
96
97          void IntroScreen::maybeSpawnCloud() {
98              if (m_lastX - m_lastCloudSpawnX < CLOUD_SPACING_PX) return;
99
100             std::uniform_real_distribution<double> dist(0.0, 1.0);
101             if (dist(m_rng) > Constants::CLOUD_PROBABILITY[level_index]) return;
102
103             int gx = m_lastX / PIXEL_SIZE;
104             auto it = m_heightAtGX.constFind(gx);
105             if (it == m_heightAtGX.constEnd()) return;
106
107             int gyGround = it.value();
108
109             auto mix = [](quint32 v, int shift, int span, int base){
110                 return base + int(((v >> shift) % quint32(span)));
111             };
112             quint32 h = 120003212u ^ quint32(m_lastX * 2654435761u);
113
114             int wCells = mix(h, 0,  CLOUD_MAX_W_CELLS - CLOUD_MIN_W_CELLS + 1,
                CLOUD_MIN_W_CELLS);
115             int hCells = mix(h, 8,  CLOUD_MAX_H_CELLS - CLOUD_MIN_H_CELLS + 1,
                CLOUD_MIN_H_CELLS);
116             int skyLift = CLOUD_SKY_OFFSET_CELLS + mix(h, 16, 11, 0);
```

```
117
118             int cloudTopCells = gyGround - skyLift;
119             if (cloudTopCells < 0) cloudTopCells = 0;
120
121             Cloud cl;
122             cl.wx = m_lastX;
123             cl.wyCells = cloudTopCells;
124             cl.wCells = wCells;
125             cl.hCells = hCells;
126             cl.seed = h;
127             m_clouds.append(cl);
128
129             m_lastCloudSpawnX = m_lastX;
130
131             int leftLimit = (m_lines.isEmpty() ? 0 : m_lines.first().getX1()) -
                    width()*2;
132             for (int i = 0; i < m_clouds.size(); ) {
133                 if (m_clouds[i].wx < leftLimit) m_clouds.removeAt(i);
134                 else ++i;
135             }
136         }
137
138         void IntroScreen::drawClouds(QPainter& p) {
139             if (Constants::CLOUD_PROBABILITY[level_index] <= 0.001) return;
140
141             int camGX = m_camX / PIXEL_SIZE;
142             int camGY = m_camY / PIXEL_SIZE;
143
144             auto hash2D = [](int x, int y)->quint32{
145                 quint32 h = 120003212u;
146                 h ^= quint32(x); h *= 16777619u;
147                 h ^= quint32(y); h *= 16777619u;
148                 return h;
149             };
150
151             for (const Cloud& cl : m_clouds) {
152                 int baseGX = (cl.wx / PIXEL_SIZE) - camGX;
153                 int baseGY = cl.wyCells + camGY;
154
155                 for (int yy = 0; yy < cl.hCells; ++yy) {
156                     for (int xx = 0; xx < cl.wCells; ++xx) {
157                         double nx = ((xx + 0.5) - cl.wCells  / 2.0) / (cl.wCells
                                / 2.0);
158                         double ny = ((yy + 0.5) - cl.hCells / 2.0) / (cl.hCells /
                                2.0);
159                         double r2 = nx*nx + ny*ny;
160
161                         quint32 h = hash2D(int(cl.seed) + xx, yy);
162                         double fuzz = (h % 100) / 400.0;
163
164                         if (r2 <= 1.0 + fuzz) {
165                             QColor cMain = Constants::CLOUD_COLOR[level_index];
```

```
                            QColor cSoft(cMain.red() * 0.9, cMain.green() * 0.9,
                            cMain.blue() * 0.9);
                            QColor pix = ((h >> 3) & 1) ? cMain : cSoft;
                            plotGridPixel(p, baseGX + xx, baseGY + yy, pix);
                        }
                    }
                }
            }
        }

        void IntroScreen::drawStars(QPainter& p) {
            if (Constants::STAR_PROBABILITY[level_index] <= 0.001) return;

            const int BLOCK = 20;
            const int camGX = m_camX / PIXEL_SIZE;
            const int camGY = m_camY / PIXEL_SIZE;

            const int startBX = (camGX) / BLOCK - 1;
            const int endBX   = (camGX + gridW()) / BLOCK + 1;
            const int startBY = (-camGY) / BLOCK - 1;
            const int endBY   = (-camGY + gridH()) / BLOCK + 1;

            for (int bx = startBX; bx <= endBX; ++bx) {
                for (int by = startBY; by <= endBY; ++by) {
                    quint32 h = 120003212u;
                    h ^= quint32(bx); h *= 16777619u;
                    h ^= quint32(by); h *= 16777619u;
                    h = (h ^ bx) / (h ^ by) + (bx * by) - (3 * bx*bx + 4 * by*by);

                    std::mt19937 rng(h);
                    std::uniform_real_distribution<float> fdist(0.0f, 1.0f);

                    if (fdist(rng) < Constants::STAR_PROBABILITY[level_index] *
                        0.4) {
                        std::uniform_int_distribution<int> idist(0, BLOCK - 1);
                        int wgx = bx * BLOCK + idist(rng);
                        int wgy = by * BLOCK + idist(rng);

                        int groundGy = 0;
                        auto it = m_heightAtGX.constFind(wgx);
                        if (it != m_heightAtGX.constEnd()) groundGy = it.value();
                        else groundGy = 10000;

                        if (wgy < groundGy - 8) {
                            int sgx = wgx - camGX;
                            int sgy = wgy + camGY;
                            int alpha = std::uniform_int_distribution<int>(100,
                                255)(rng);
                            plotGridPixel(p, sgx, sgy, QColor(255, 255, 255, alpha
                                ));
                        }
                    }
```

```
                }
            }
        }

        void IntroScreen::resizeEvent(QResizeEvent*) {
            m_camXFarthest = m_camX;
        }

        int IntroScreen::textWidthCells(const QString& s, int scale) const {
            if (s.isEmpty()) return 0;
            return (int(s.size()) - 1) * CHAR_ADV * scale + 5 * scale;
        }

        int IntroScreen::fitTextScaleToRect(int wCells, int hCells, const QString&
             s) const {
            int padX = 4;
            int padY = 2;
            int maxScale = 10;

            int wcap = std::max(1, (wCells - padX) / std::max(1, textWidthCells(s,
                 1)));
            int hcap = std::max(1, (hCells - padY) / 7);

            int sc   = std::min(wcap, hcap);
            sc = std::clamp(sc, 1, maxScale);
            return sc;
        }

        void IntroScreen::drawCircleFilledMidpointGrid(QPainter& p, int gcx, int
            gcy, int gr, const QColor& c) {
            int x = 0;
            int y = gr;
            int d = 1 - gr;
            auto span = [&](int cy, int xl, int xr) {
                for (int xg = xl; xg <= xr; ++xg) plotGridPixel(p, xg, cy, c);
            };
            while (y >= x) {
                span(gcy + y, gcx - x, gcx + x);
                span(gcy - y, gcx - x, gcx + x);
                span(gcy + x, gcx - y, gcx + y);
                span(gcy - x, gcx - y, gcx + y);
                ++x;
                if (d < 0) d += 2 * x + 1;
                else { --y; d += 2 * (x - y) + 1; }
            }
        }

        void IntroScreen::paintEvent(QPaintEvent*) {
            QImage bg(size(), QImage::Format_ARGB32_Premultiplied);
            bg.fill(Constants::SKY_COLOR[level_index]);
            {
                QPainter pb(&bg);
```

```
264                 drawBackground(pb);
265             }
266 
267             int sw = std::max(1, int(width() * m_blurScale));
268             int sh = std::max(1, int(height() * m_blurScale));
269             QImage small = bg.scaled(sw, sh, Qt::IgnoreAspectRatio, Qt::
                    SmoothTransformation);
270 
271             QPainter p(this);
272             p.drawImage(rect(), small, small.rect());
273 
274             const int r = 3;
275             int iconGX = 2 + r;
276             int iconGY = 2 + r;
277 
278             drawCircleFilledMidpointGrid(p, iconGX, iconGY, r, QColor(195,140,40))
                    ;
279             drawCircleFilledMidpointGrid(p, iconGX, iconGY, std::max(1, r-1),
                    QColor(250,204,77));
280             plotGridPixel(p, iconGX-1, iconGY-r+1, QColor(255,255,220));
281 
282             double scale = 1;
283             QString total = QString("%1").arg(m_grandTotalCoins);
284             int labelGX = iconGX + r*2 + 2;
285             int labelGY = iconGY - (7*scale)/3;
286             drawPixelText(p, total, labelGX, labelGY, (double)scale, Constants::
                    TEXT_COLOR[level_index], false);
287 
288             const QString title = "Braking Bad";
289             int ts = titleScale();
290             int titleWCells = textWidthCells(title, ts);
291             int tgx = (gridW() - titleWCells) / 2;
292             int tgy = titleYCells();
293             drawPixelText(p, title, tgx, tgy, ts, Constants::TEXT_COLOR[
                    level_index], true);
294 
295             // --- Level selector ---
296             QRect rLevelPrev = buttonRectLevelPrev();
297             QRect rLevelNext = buttonRectLevelNext();
298 
299             p.setPen(Qt::NoPen);
300             p.setBrush(QColor(0,0,0,160));
301             p.drawRect(rLevelPrev.translated(3*PIXEL_SIZE,3*PIXEL_SIZE));
302             p.drawRect(rLevelNext.translated(3*PIXEL_SIZE,3*PIXEL_SIZE));
303 
304             p.setBrush(QColor(150, 150, 160));
305             p.drawRect(rLevelPrev);
306             p.drawRect(rLevelNext);
307 
308             QString sPrev = "<";
309             QString sNext = ">";
```

```
310             int prevScale = fitTextScaleToRect(rLevelPrev.width()/PIXEL_SIZE,
                    rLevelPrev.height()/PIXEL_SIZE, sPrev);
311             int nextScale = fitTextScaleToRect(rLevelNext.width()/PIXEL_SIZE,
                    rLevelNext.height()/PIXEL_SIZE, sNext);
312
313             drawPixelText(p, sPrev,
314             rLevelPrev.left()/PIXEL_SIZE + (rLevelPrev.width()/PIXEL_SIZE -
                    textWidthCells(sPrev, prevScale))/2,
315             rLevelPrev.top()/PIXEL_SIZE  + (rLevelPrev.height()/PIXEL_SIZE - 7*
                    prevScale)/2,
316             prevScale, QColor(25,20,24), false);
317
318             drawPixelText(p, sNext,
319             rLevelNext.left()/PIXEL_SIZE + (rLevelNext.width()/PIXEL_SIZE -
                    textWidthCells(sNext, nextScale))/2,
320             rLevelNext.top()/PIXEL_SIZE  + (rLevelNext.height()/PIXEL_SIZE - 7*
                    nextScale)/2,
321             nextScale, QColor(25,20,24), false);
322
323             QString levelName = m_levelNames[level_index];
324             int levelScale = 2;
325             int levelWCells = textWidthCells(levelName, levelScale);
326             int levelGX = (gridW() - levelWCells) / 2;
327             const int stageGapCells = (TITLE_STAGE_GAP_PX + PIXEL_SIZE - 1) /
                    PIXEL_SIZE;
328
329             int levelGY = rLevelPrev.top()/PIXEL_SIZE
330             + (rLevelPrev.height()/PIXEL_SIZE - 7*levelScale)/2
331             + stageGapCells;
332
333             drawPixelText(p, levelName, levelGX, levelGY, levelScale, Constants::
                    TEXT_COLOR[level_index], true);
334
335             QRect rStart;
336             QRect rExit  = buttonRectExit();
337
338             p.setPen(Qt::NoPen);
339
340             if (levels_unlocked[level_index])
341             {
342                 rStart = buttonRectStart();
343                 p.setBrush(QColor(0,0,0,160));
344                 p.drawRect(rStart.translated(3*PIXEL_SIZE,3*PIXEL_SIZE));
345                 p.setBrush(QColor(240,190,60));
346                 p.drawRect(rStart);
347
348                 QString sStart = "PLAY";
349                 int rStartWc = rStart.width()  / PIXEL_SIZE;
350                 int rStartHc = rStart.height() / PIXEL_SIZE;
351                 int bsStart = fitTextScaleToRect(rStartWc, rStartHc, sStart);
352                 int sWCells = textWidthCells(sStart, bsStart);
353                 int sGX = rStart.left()/PIXEL_SIZE + (rStartWc - sWCells)/2;
```

```
354                 int sGY = rStart.top()/PIXEL_SIZE  + (rStartHc - 7*bsStart)/2;
355                 drawPixelText(p, sStart, sGX, sGY, bsStart,  QColor(20,20,20),
                        false);
356             }
357             else
358             {
359                 rStart = buttonRectUnlock();
360
361                 int cost = m_levelCosts.value(level_index, 999);
362                 bool canAfford = (m_grandTotalCoins >= (quint64)cost);
363
364                 // Draw shadow
365                 p.setBrush(QColor(0,0,0,160));
366                 p.drawRect(rStart.translated(3*PIXEL_SIZE,3*PIXEL_SIZE));
367
368                 // Draw button (gray)
369                 p.setBrush(QColor(100, 100, 110));
370                 p.drawRect(rStart);
371
372                 // Draw text
373                 QString sStart = QString("UNLOCK: %1").arg(cost);
374                 int rStartWc = rStart.width()  / PIXEL_SIZE;
375                 int rStartHc = rStart.height() / PIXEL_SIZE;
376
377                 int bsStart = fitTextScaleToRect(rStartWc, rStartHc, sStart);
378
379                 int sWCells = textWidthCells(sStart, bsStart);
380                 int sGX = rStart.left()/PIXEL_SIZE + (rStartWc - sWCells)/2;
381                 int sGY = rStart.top()/PIXEL_SIZE  + (rStartHc - 7*bsStart)/2;
382                 QColor textColor = canAfford ? QColor(20, 20, 20) : QColor(255,
                        80, 80);
383                 drawPixelText(p, sStart, sGX, sGY, bsStart, textColor, false);
384             }
385
386             p.setBrush(QColor(0,0,0,160));
387             p.drawRect(rExit.translated(3*PIXEL_SIZE,3*PIXEL_SIZE));
388             p.setBrush(QColor(200,80,90));
389             p.drawRect(rExit);
390
391             QString sExit  = "EXIT";
392
393             int rExitWc  = rExit.width()   / PIXEL_SIZE;
394             int rExitHc  = rExit.height()  / PIXEL_SIZE;
395
396             int bsExit  = fitTextScaleToRect(rExitWc,  rExitHc,  sExit);
397
398             int eWCells = textWidthCells(sExit,  bsExit);
399
400             int eGX = rExit.left()/PIXEL_SIZE  + (rExitWc  - eWCells)/2;
401             int eGY = rExit.top()/PIXEL_SIZE   + (rExitHc  - 7*bsExit)/2;
402
403             drawPixelText(p, sExit,  eGX, eGY,  bsExit,  QColor(20,20,20), false);
```

```
404         }
405
406         void IntroScreen::mousePressEvent(QMouseEvent* e) {
407             if (buttonRectLevelPrev().contains(e->pos())) {
408                 level_index--;
409                 if (level_index < 0) level_index = m_levelNames.size() - 1;
410                 update();
411                 return;
412             }
413
414             if (buttonRectLevelNext().contains(e->pos())) {
415                 level_index++;
416                 if (level_index >= m_levelNames.size()) level_index = 0;
417                 update();
418                 return;
419             }
420
421             if (buttonRectStart().contains(e->pos()) && levels_unlocked[
                    level_index]) {
422                 emit startRequested(level_index);
423                 return;
424             }
425
426             if (buttonRectUnlock().contains(e->pos())){
427                 int cost = m_levelCosts.value(level_index, 0);
428                 bool canAfford = (m_grandTotalCoins >= (quint64)cost);
429                 if(canAfford){
430                     m_grandTotalCoins -= cost;
431                     levels_unlocked[level_index] = true;
432                     saveGrandCoins();
433                     saveUnlocks();
434                     update();
435                 }
436                 return;
437             }
438             if (buttonRectExit().contains(e->pos()))  { emit exitRequested();
                            return; }
439         }
440
441         QRect IntroScreen::buttonRectLevelPrev() const {
442             int hCells = std::max(10, gridH()/18);
443             int wCells = hCells;
444             const int stageGapCells = (TITLE_STAGE_GAP_PX + PIXEL_SIZE - 1) /
                    PIXEL_SIZE;
445
446             int yCells = startTopCells(0) - hCells - std::max(10, gridH()/40) +
                    stageGapCells;
447
448             const int fixedOffset = 80;
449             int gx = (gridW() / 2) - fixedOffset - wCells;
450
```

```
451             return QRect(gx*PIXEL_SIZE, yCells*PIXEL_SIZE, wCells*PIXEL_SIZE,
                    hCells*PIXEL_SIZE);
452         }
453
454         QRect IntroScreen::buttonRectLevelNext() const {
455             int hCells = std::max(10, gridH()/18);
456             int wCells = hCells;
457             const int stageGapCells = (TITLE_STAGE_GAP_PX + PIXEL_SIZE - 1) /
                    PIXEL_SIZE;
458
459             int yCells = startTopCells(0) - hCells - std::max(10, gridH()/40) +
                    stageGapCells;
460
461
462             const int fixedOffset = 80;
463             int gx = (gridW() / 2) + fixedOffset;
464
465             return QRect(gx*PIXEL_SIZE, yCells*PIXEL_SIZE, wCells*PIXEL_SIZE,
                    hCells*PIXEL_SIZE);
466         }
467
468         int IntroScreen::stageLabelBottomPx() const {
469             QRect rLevelPrev = buttonRectLevelPrev();
470             const int levelScale = 2;
471             int levelTopCells = rLevelPrev.top()/PIXEL_SIZE
472             + (rLevelPrev.height()/PIXEL_SIZE - 7*levelScale)/2
473             + std::max(3, gridH()/40);
474             int bottomCells = levelTopCells + 7*levelScale;
475             return bottomCells * PIXEL_SIZE;
476         }
477
478         QRect IntroScreen::buttonRectStart() const {
479             int wCells = std::min(std::max(gridW()/6, 30), 50);
480             int hCells = std::max(10, gridH()/18);
481             int gx     = (gridW() - wCells) / 2;
482
483             int topPx  = stageLabelBottomPx() + 100;
484             int maxTop = gridH()*PIXEL_SIZE - hCells*PIXEL_SIZE - 1;
485             if (topPx > maxTop) topPx = maxTop;
486
487             return QRect(gx*PIXEL_SIZE, topPx, wCells*PIXEL_SIZE, hCells*
                    PIXEL_SIZE);
488         }
489
490         QRect IntroScreen::buttonRectExit() const {
491             int wCells = std::min(std::max(gridW()/6, 30), 50);
492             int hCells = std::max(10, gridH()/18);
493             int gx     = (gridW() - wCells) / 2;
494
495             int gapCells = std::max(10, gridH()/40);
496             int topPx    = stageLabelBottomPx() + 50 + hCells*PIXEL_SIZE +
                    gapCells*PIXEL_SIZE;
```

```
497             int maxTop   = gridH()*PIXEL_SIZE - hCells*PIXEL_SIZE - 1;
498             if (topPx > maxTop) topPx = maxTop;
499 
500             return QRect(gx*PIXEL_SIZE, topPx, wCells*PIXEL_SIZE, hCells*
                    PIXEL_SIZE);
501         }
502 
503         QRect IntroScreen::buttonRectUnlock() const{
504             int wCells = std::min(std::max(gridW()/2, 80), 120);
505             int hCells = std::max(10, gridH()/18);
506             int gx     = (gridW() - wCells) / 2;
507 
508             int topPx  = stageLabelBottomPx() + 100;
509             int maxTop = gridH()*PIXEL_SIZE - hCells*PIXEL_SIZE - 1;
510             if (topPx > maxTop) topPx = maxTop;
511 
512             return QRect(gx*PIXEL_SIZE, topPx, wCells*PIXEL_SIZE, hCells*
                    PIXEL_SIZE);
513         }
514 
515         void IntroScreen::drawBackground(QPainter& p) {
516             p.fillRect(rect(), Constants::SKY_COLOR[level_index]);
517             drawStars(p);
518             drawClouds(p);
519             drawFilledTerrain(p);
520         }
521 
522         void IntroScreen::drawFilledTerrain(QPainter& p) {
523             const int camGX = m_camX / PIXEL_SIZE;
524             const int camGY = m_camY / PIXEL_SIZE;
525 
526             for (int sgx = 0; sgx <= gridW(); ++sgx) {
527                 const int worldGX = sgx + camGX;
528                 auto it = m_heightAtGX.constFind(worldGX);
529                 if (it == m_heightAtGX.constEnd()) continue;
530 
531                 const int groundWorldGY = it.value();
532                 int startScreenGY = groundWorldGY + camGY;
533                 if (startScreenGY < 0) startScreenGY = 0;
534                 if (startScreenGY >= gridH()) continue;
535 
536                 for (int sGY = startScreenGY; sGY <= gridH(); ++sGY) {
537                     const int worldGY = sGY - camGY;
538 
539                     int depth = sGY - startScreenGY;
540                     if (level_index == 5) {
541                         QColor c;
542                         if (depth < 14) {
543                             if (depth == 0) c = QColor(80, 80, 85);
544                             else if (depth >= 6 && depth <= 7 && (worldGX % 20 <
                                    10)) {
545                                 c = QColor(240, 190, 40);
```

```
546                             }
547                             else c = QColor(50, 50, 55);
548 
549                             plotGridPixel(p, sgx, sGY, c);
550                             continue;
551                         }
552                     }
553 
554                     bool topZone = (sGY < groundWorldGY + camGY + 3*SHADING_BLOCK)
                            ;
555                     const QColor shade = grassShadeForBlock(worldGX, worldGY,
                        topZone);
556                     plotGridPixel(p, sgx, sGY, shade);
557 
558                     if (level_index != 5) {
559                         const QColor edge = grassShadeForBlock(worldGX,
                            groundWorldGY, true).darker(115);
560                         plotGridPixel(p, sgx, groundWorldGY + camGY, edge);
561                     }
562                 }
563 
564                 const QColor edge = grassShadeForBlock(worldGX, groundWorldGY,
                    true).darker(115);
565                 plotGridPixel(p, sgx, groundWorldGY + camGY, edge);
566             }
567         }
568 
569         QColor IntroScreen::grassShadeForBlock(int worldGX, int worldGY, bool
                greenify) const {
570             auto hash2D = [](int x, int y)->quint32{
571                 quint32 h = 120003212u;
572                 h ^= quint32(x); h *= 16777619u;
573                 h ^= quint32(y); h *= 16777619u;
574                 return (h ^ x) / (h ^ y) + (x * y) - (3 * x*x + 4 * y*y);
575             };
576 
577             const int bx = worldGX / SHADING_BLOCK;
578             const int by = worldGY / SHADING_BLOCK;
579             const quint32 h = hash2D(bx, by);
580 
581             if (greenify) {
582                 const int idxG = int(h % m_grassPalette.size());
583                 return m_grassPalette[level_index][idxG];
584             } else {
585                 const int idxD = int(h % m_dirtPalette.size());
586                 return m_dirtPalette[level_index][idxD];
587             }
588         }
589 
590         void IntroScreen::plotGridPixel(QPainter& p, int gx, int gy, const QColor&
                c) {
591             if (gx < 0 || gy < 0 || gx >= gridW()+1 || gy >= gridH()+1) return;
```

```
592             p.fillRect(gx * PIXEL_SIZE, gy * PIXEL_SIZE, PIXEL_SIZE, PIXEL_SIZE, c
                    );
593         }
594
595         void IntroScreen::rasterizeSegmentToHeightMapWorld(int x1,int y1,int x2,
                int y2){
596             if (x2 < x1) { std::swap(x1,x2); std::swap(y1,y2); }
597
598             const int gx1 = x1 / PIXEL_SIZE;
599             const int gx2 = x2 / PIXEL_SIZE;
600
601             if (x2 == x1) {
602                 const int gy = int(std::floor(y1 / double(PIXEL_SIZE) + 0.5));
603                 m_heightAtGX.insert(gx1, gy);
604                 return;
605             }
606
607             const double dx = double(x2 - x1);
608             double dy = double(y2 - y1);
609
610             for (int gx = gx1; gx <= gx2; ++gx) {
611                 const double wx = gx * double(PIXEL_SIZE);
612                 double t = (wx - x1) / dx;
613                 t = std::clamp(t, 0.0, 1.0);
614                 const double wy = y1 + t * dy;
615                 const int gy = int(std::floor(wy / double(PIXEL_SIZE) + 0.5));
616                 m_heightAtGX.insert(gx, gy);
617             }
618         }
619
620         void IntroScreen::pruneHeightMap() {
621             if (m_lines.isEmpty()) return;
622             const int keepFromGX = (m_lines.first().getX1() / PIXEL_SIZE) - 4;
623
624             QList<int> toRemove;
625             toRemove.reserve(m_heightAtGX.size());
626
627             for (auto it = m_heightAtGX.constBegin(); it != m_heightAtGX.constEnd
                    (); ++it) {
628                 if (it.key() < keepFromGX) toRemove.append(it.key());
629             }
630             for (int k : toRemove) m_heightAtGX.remove(k);
631         }
632
633         void IntroScreen::ensureAheadTerrain(int worldX) {
634             while (m_lastX < worldX) {
635                 m_slope += (0.5f - float(m_lastY) / std::max(1,height())) *
                            m_difficulty;
636                 m_slope = std::clamp(m_slope, -1.0f, 1.0f);
637
638                 const int newY = m_lastY + std::lround(m_slope * std::pow(std::abs
                        (m_slope), 0.02f) * STEP);
```

```
639
640                 Line seg(m_lastX, m_lastY, m_lastX + STEP, newY);
641                 m_lines.append(seg);
642
643                 rasterizeSegmentToHeightMapWorld(seg.getX1(), m_lastY, seg.getX2()
                        , newY);
644
645                 m_lastY = newY;
646                 m_lastX += STEP;
647
648                 if (m_lines.size() > (width() / STEP) * 3) {
649                     m_lines.removeFirst();
650                     pruneHeightMap();
651                 }
652
653                 m_difficulty += DIFF_INC;
654             }
655         }
656
657         void IntroScreen::drawPixelText(QPainter& p, const QString& s, int gx, int
                gy, int scale, const QColor& c, bool bold)
658         {
659             auto plot = [&](int x,int y,const QColor& col) {
660                 plotGridPixel(p, gx + x, gy + y, col);
661             };
662
663             for (int i = 0; i < s.size(); ++i) {
664                 const QChar ch = s.at(i).toUpper();
665                 const auto rows = font_map.value(font_map.contains(ch) ? ch :
                        QChar(' '));
666                 int baseOff = i * CHAR_ADV * scale;
667                 for (int ry = 0; ry < 7; ++ry) {
668                     uint8_t row = rows[ry];
669                     for (int rx = 0; rx < 5; ++rx) {
670                         if (row & (1 << (4 - rx))) {
671                             for (int sy = 0; sy < scale; ++sy) {
672                                 for (int sx = 0; sx < scale; ++sx) {
673                                     if (bold) {
674                                         plot(baseOff + rx*scale + sx - 1, ry*scale
                                                + sy, c);
675                                         plot(baseOff + rx*scale + sx + 1, ry*scale
                                                + sy, c);
676                                         plot(baseOff + rx*scale + sx, ry*scale +
                                            sy - 1, c);
677                                         plot(baseOff + rx*scale + sx, ry*scale +
                                            sy + 1, c);
678                                     }
679                                     plot(baseOff + rx*scale + sx, ry*scale + sy, c
                                        );
680                                 }
681                             }
682                         }
```

```
                    }
                }
            }
        }


        void IntroScreen::saveGrandCoins() const {
            QSettings s("JU","F1PixelGrid");
            s.setValue("grandCoins", m_grandTotalCoins);
            s.sync();
        }

        void IntroScreen::loadGrandCoins() {
            QSettings s("JU","F1PixelGrid");
            m_grandTotalCoins = s.value("grandCoins", 0).toInt();
        }

        void IntroScreen::saveUnlocks() const {
            QSettings s("JU","F1PixelGrid");

            QVariantList unlockList;

            for(bool unlocked : levels_unlocked) {
                unlockList.append(unlocked);
            }

            s.setValue("unlocks", unlockList);
            s.sync();
        }

        void IntroScreen::loadUnlocks() {
            QSettings s("JU","F1PixelGrid");
            QVariant unlockData = s.value("unlocks");

            if (unlockData.isValid()) {

                QVariantList savedList = unlockData.toList();

                levels_unlocked.clear();

                for(const QVariant& item : savedList) {
                    levels_unlocked.append(item.toBool());
                }

                while(levels_unlocked.size() != m_levelCosts.size()) {
                    levels_unlocked.append(false);
                }

                saveUnlocks();

                if (levels_unlocked.size() > 5) {
                    levels_unlocked[5] = false;
```

```
735                 }
736             }
737             else
738             {
739                 levels_unlocked = {true, false, false, false, false, false};
740             }
741         }
```

Intro Screen Feature

The intro screen is implemented as a dedicated `IntroScreen` widget, which `MainWindow` uses as the initial scene before gameplay. Its primary purpose is to provide a visually rich landing page that already reflects the in–game environment. Internally, it constructs a scrolling terrain by generating line segments and rasterising them into a height map. A `QTimer` advances a virtual camera and triggers terrain extension and pruning, so the background continuously moves even while the player is idle on the menu.

Rendering is handled in `paintEvent()`, which first draws into an off–screen `QImage`. Helper functions such as `drawBackground()`, `drawStars()`, `drawClouds()` and `drawFilledTerrain()` are used to compose the sky, star field, cloud layers and ground, all in a pixel–art style controlled by a logical grid. The result is then scaled and blitted to the widget to produce a slightly softened backdrop. On top of this, the intro screen overlays the game title (and any summary information) using a custom 5x7 pixel font. Mouse events are interpreted to detect clicks on the "PLAY" and "EXIT" areas, and the widget emits the high–level signals `startRequested()` and `exitRequested()`, which `MainWindow` uses to transition into the game or close the application.

(a) Intro Screen: Stage – Meadow

(b) Intro Screen: Stage – Nightlife

Figure 5: Intro screen for two different stages.

outro.h

```
1   // outro.h
2   #pragma once
3   #include <QWidget>
4   #include <QRect>
5
6   class QPushButton;
7   class QPaintEvent;
8   class QResizeEvent;
```

```
class QMouseEvent;

class OutroScreen : public QWidget {
    Q_OBJECT
    public:
    explicit OutroScreen(QWidget* parent = nullptr);
    void setStats(int coinCount, int nitroCount, int score, double
        distanceMeters);
    void setFlips(int flips);

    signals:
    void exitRequested();
    void restartRequested();

    protected:
    void paintEvent(QPaintEvent* e) override;
    void resizeEvent(QResizeEvent* e) override;
    void mousePressEvent(QMouseEvent* e) override;

    private:
    void centerInParent();
    void drawPixelCoin(QPainter& p, int gx, int gy, int rCells);
    void drawPixelFlame(QPainter& p, int gx, int gy, int lenCells);

    int m_flips = 0;

    private:
    QPushButton* m_exitBtn = nullptr;
    int  m_cell   = 6;

    int m_coins = 0;
    int m_nitros = 0;
    int m_score = 0;
    double m_distanceM = 0.0;

    QRect m_btnExitRect;
    QRect m_btnRestartRect;
};
```

outro.cpp

```
// outro.cpp
#include "outro.h"
#include "intro.h"
#include "constants.h"
#include <QPainter>
#include <QPaintEvent>
#include <QPushButton>
#include <QMouseEvent>
#include <QMap>
#include <array>
```

```
#include <initializer_list>
#include <algorithm>

namespace {

    constexpr int CHAR_ADV = 7;

    inline int textHeightCells(int scale){ return 7*scale; }

    inline void plotGridPixel(QPainter& p, int cell, int gx, int gy, const QColor& c) {
        p.fillRect(gx*cell, gy*cell, cell, cell, c);
    }

    inline int textWidthCells(const QString& s, int scale){
        return s.isEmpty()?0:((int)s.size()-1)*CHAR_ADV*scale + 5*scale;
    }

    inline int fitTextScale(int wCells, int hCells, const QString& s){
        int wcap = std::max(1, (wCells-4)/std::max(1,textWidthCells(s,1)));
        int hcap = std::max(1, (hCells-2)/7);
        return std::clamp(std::min(wcap,hcap),1,18);
    }

    void drawPixelText(QPainter& p, const QString& s, int cell, int gx, int gy, int scale, const QColor& c, bool bold){
        auto plot=[&](int x,int y,const QColor& col){ plotGridPixel(p,cell,gx+x,gy+y,col); };
        for(int i=0;i<s.size();++i){
            const QChar ch = s.at(i).toUpper();
            const auto rows = ::font_map.value(::font_map.contains(ch) ? ch : QChar(' '));
            int base=i*CHAR_ADV*scale;
            for(int ry=0;ry<7;++ry){
                uint8_t row=rows[ry];
                for(int rx=0;rx<5;++rx){
                    if(row&(1<<(4-rx))){
                        for(int sy=0;sy<scale;++sy){
                            for(int sx=0;sx<scale;++sx){
                                if(bold){
                                    plot(base+rx*scale+sx-1,ry*scale+sy,QColor(20,20,22));
                                    plot(base+rx*scale+sx+1,ry*scale+sy,QColor(20,20,22));
                                    plot(base+rx*scale+sx,ry*scale+sy-1,QColor(20,20,22));
                                    plot(base+rx*scale+sx,ry*scale+sy+1,QColor(20,20,22));
                                }
                                plot(base+rx*scale+sx,ry*scale+sy,c);
                            }
```

```
                                }
                            }
                        }
                    }
                }
            }
        }

        OutroScreen::OutroScreen(QWidget* parent)
        : QWidget(parent)
        {
            setAttribute(Qt::WA_StyledBackground, true);
            setAutoFillBackground(true);
            QPalette pal=palette();
            pal.setColor(QPalette::Window, QColor(15,12,20,230));
            setPalette(pal);

            m_exitBtn=new QPushButton("Exit",this);
            m_exitBtn->setCursor(Qt::PointingHandCursor);
            connect(m_exitBtn,&QPushButton::clicked,this,[this]{ emit
                exitRequested(); });
            m_exitBtn->hide();

            // Larger panel
            if(parent){
                int w = int(parent->width()  * 0.60);
                int h = int(parent->height() * 0.50);
                resize(w, h);
                centerInParent();
            }
            m_cell=std::max(4,std::min(width(),height())/90);
            setFocusPolicy(Qt::StrongFocus);
            setVisible(true);
            raise();
        }

        void OutroScreen::centerInParent(){
            if(!parentWidget()) return;
            move(parentWidget()->width()/2 - width()/2, parentWidget()->height()/2
                - height()/2);
        }

        void OutroScreen::setStats(int coinCount, int nitroCount, int score,
            double distanceMeters){
            m_coins  = coinCount;
            m_nitros = nitroCount;
            m_score  = score;
            m_distanceM = distanceMeters;
            update();
        }

        void OutroScreen::paintEvent(QPaintEvent*){
```

```
103             QPainter p(this);
104             p.setRenderHint(QPainter::Antialiasing,false);
105             p.setRenderHint(QPainter::TextAntialiasing,false);
106 
107             const int cell = m_cell;
108             const int pad = cell;
109 
110             const int gw = (width()-2*pad)/cell;
111             const int gh = (height()-2*pad)/cell;
112             const int panelX = pad;
113             const int panelY = pad;
114             const QRect panel(panelX,panelY,gw*cell,gh*cell);
115 
116             p.fillRect(panel,QColor(28,24,36));
117             p.setPen(QColor(65,60,80));
118             p.drawRect(panel.adjusted(0,0,-1,-1));
119 
120             const int pgx=panelX/cell;
121             const int pgy=panelY/cell;
122             auto GX=[&](int c){ return pgx+c; };
123             auto GY=[&](int c){ return pgy+c; };
124 
125             const QString title="GAME OVER";
126             const int titleScale = fitTextScale(gw-10, 9, title);
127             const int titleW = textWidthCells(title,titleScale);
128             const int tGX = GX((gw-titleW)/2);
129             const int tGY = GY(std::max(2, gh/10));
130             drawPixelText(p,title,cell,tGX,tGY,titleScale,QColor(230,230,240),
                false);
131 
132             const int titleH = textHeightCells(titleScale);
133             const int topGap = std::max(12, gh/18);
134             const int rowGap = std::max(10, gh/24);
135 
136             const int contentTop = tGY + titleH + topGap;
137 
138             // Left column (coins / nitro)
139             int iconR = std::clamp(gw/50, 2, 5);
140             int gapL  = std::max(8, gw/40);
141             int leftGX = GX(std::max(6, gw/12));
142 
143             // Right column strings
144             QString scoreStr = QString("SCORE: %1").arg(m_score);
145             QString distStr  = QString("DIST: %1 m").arg(QString::number(
                m_distanceM,'f',1));
146             QString flipsStr = QString("FLIPS: x%1").arg(m_flips);
147 
148             int numScale    = std::max(1, titleScale/2);
149             int rightScale  = std::max(1, titleScale/2);
150 
151             auto leftEndGX = [&](const QString& s, int baseGX){
152                 return baseGX + gapL + textWidthCells(s, numScale);
```

```
153             };
154             auto rightStartGX = [&](const QString& s){
155                 int rightPadCells = std::clamp(gw/10, 12, 30);
156                 int w = textWidthCells(s, rightScale);
157                 return GX(gw - rightPadCells - w);
158             };
159 
160             // Equal vertical spacing for the three right rows (score, dist, flips)
161             const int rightRowH = textHeightCells(rightScale);
162             const int row1Y = contentTop;
163             const int row2Y = row1Y + rightRowH + rowGap;
164             const int row3Y = row2Y + rightRowH + rowGap;
165 
166             // Ensure left rows don't collide with right rows; shrink scales if necessary
167             int iter=12;
168             while (iter--){
169                 int sGX = rightStartGX(scoreStr);
170                 int dGX = rightStartGX(distStr);
171                 int fGX = rightStartGX(flipsStr);
172                 int l1End = leftEndGX(QString("x%1").arg(m_coins), leftGX + iconR);
173                 int l2End = leftEndGX(QString("x%1").arg(m_nitros), leftGX + iconR);
174                 bool overlap = (l1End + 2 >= sGX) || (l2End + 2 >= dGX) || (l2End + 2 >= fGX);
175                 if (!overlap) break;
176                 if (rightScale > 1) --rightScale;
177                 if (numScale   > 1) --numScale;
178             }
179 
180             // Left column draw
181             drawPixelCoin(p,leftGX,row1Y+3,iconR);
182             drawPixelText(p,QString("x%1").arg(m_coins),cell,leftGX+iconR+std::max(8, gw/40),row1Y,numScale,QColor(230,230,240),false);
183 
184             drawPixelFlame(p,leftGX,row2Y+3,iconR*2);
185             drawPixelText(p,QString("x%1").arg(m_nitros),cell,leftGX+iconR+std::max(8, gw/40),row2Y,numScale,QColor(230,230,240),false);
186 
187             drawPixelText(p, scoreStr, cell, rightStartGX(scoreStr), row1Y, rightScale, QColor(230,230,240), false);
188             drawPixelText(p, distStr,  cell, rightStartGX(distStr),  row2Y, rightScale, QColor(230,230,240), false);
189             drawPixelText(p, flipsStr, cell, rightStartGX(flipsStr), row3Y, rightScale, QColor(230,230,240), false);
190 
191             // Buttons: RESTART (left, dark green) and EXIT (right, red)
192             const int gwBtn = gw;
193             int btnWCells = std::min(std::max(int(gwBtn/4), 24), gwBtn - 12);
194             int btnHCells = std::max(9, gh/10);
```

```
195             int gapCells  = std::max(12, gw/12);
196
197             int totalW = btnWCells*2 + gapCells;
198             int bGX0 = GX((gwBtn - totalW)/2);
199             int bGY  = GY(gh - btnHCells - std::max(3, gh/40));
200
201             QRect rRestart(bGX0*cell, bGY*cell, btnWCells*cell, btnHCells*cell);
202             QRect rExit   ((bGX0 + btnWCells + gapCells)*cell, bGY*cell, btnWCells
                    *cell, btnHCells*cell);
203
204             m_btnRestartRect = rRestart;
205             m_btnExitRect = rExit;
206
207             // RESTART button (dark green bg, white text)
208             p.setPen(Qt::NoPen);
209             p.setBrush(QColor(20,100,40));
210             p.drawRect(rRestart);
211
212             const QString sRestart = "RESTART";
213             int innerW = btnWCells-6;
214             int innerH = btnHCells-4;
215             int rsW = std::max(1, innerW/std::max(1,textWidthCells(sRestart,1)));
216             int rsH = std::max(1, innerH/7);
217             int rs  = std::clamp(std::min(rsW,rsH),1,12);
218             int rLabelW = textWidthCells(sRestart,rs);
219             int rGX = rRestart.left()/cell + (btnWCells - rLabelW)/2;
220             int rGY = rRestart.top() /cell + (btnHCells - 7*rs)/2;
221             drawPixelText(p, sRestart, cell, rGX, rGY, rs, QColor(255,255,255),
                    false);
222
223             // EXIT button (red bg, white text)
224             p.setBrush(QColor(255,0,0));
225             p.drawRect(rExit);
226
227             const QString sExit = "EXIT";
228             int bsW = std::max(1, innerW/std::max(1,textWidthCells(sExit,1)));
229             int bsH = std::max(1, innerH/7);
230             int bs  = std::clamp(std::min(bsW,bsH),1,12);
231             int labelW = textWidthCells(sExit,bs);
232             int txGX = rExit.left()/cell + (btnWCells - labelW)/2;
233             int tyGY = rExit.top() /cell + (btnHCells - 7*bs)/2;
234             drawPixelText(p, sExit, cell, txGX, tyGY, bs, QColor(255,255,255),
                    false);
235
236         }
237
238         void OutroScreen::resizeEvent(QResizeEvent*){
239             m_cell=std::max(4,std::min(width(),height())/90);
240             centerInParent();
241             update();
242         }
243
```

```
244         void OutroScreen::mousePressEvent(QMouseEvent* e){
245             if (m_btnRestartRect.contains(e->pos())) { emit restartRequested();
                    return; }
246             if(m_btnExitRect.contains(e->pos())){ emit exitRequested(); return; }
247             QWidget::mousePressEvent(e);
248         }
249
250         void OutroScreen::drawPixelCoin(QPainter& p, int gx, int gy, int rCells){
251             auto plot=[&](int x,int y,const QColor& c){
252                 p.fillRect(x*m_cell,y*m_cell,m_cell,m_cell,c);
253             };
254
255             int x0=gx,y0=gy,r=rCells,x=0,y=r,d=1-r;
256
257             auto span=[&](int cy,int xl,int xr,const QColor& c){
258                 for(int xx=xl;xx<=xr;++xx)
259                 plot(xx,cy,c);
260             };
261
262             QColor c1(195,140,40),c2(250,204,77),hl(255,255,220);
263             while(y>=x){
264                 span(y0+y,x0-x,x0+x,c1);
265                 span(y0-y,x0-x,x0+x,c1);
266                 span(y0+x,x0-y,x0+y,c1);
267                 span(y0-x,x0-y,x0+y,c1);
268                 ++x;
269                 if(d<0) d+=2*x+1;
270                 else { --y; d+=2*(x-y)+1; }
271             }
272
273             for(int yy=-r+1; yy<=r-1; ++yy){
274                 int rad=int(std::floor(std::sqrt(double(r*r-yy*yy))));
275                 span(y0+yy,x0-rad+1,x0+rad-1,c2);
276             }
277
278             plot(x0-1,y0-r+1,hl);
279         }
280
281         void OutroScreen::drawPixelFlame(QPainter& p, int gx, int gy, int lenCells
                ){
282             auto plot=[&](int x,int y,const QColor& c){
283                 p.fillRect(x*m_cell,y*m_cell,m_cell,m_cell,c);
284             };
285             QColor cOuter(255,100,35),cMid(255,160,45),cCore(255,240,120);
286             int x0=gx,y0=gy;
287
288             for(int i=0;i<lenCells;++i){
289                 int cx=x0-i, cy=y0;
290
291                 int w=std::max(0,2-i/2);
292                 for(int j=-w;j<=w;++j) plot(cx,cy+j,cOuter);
293
```

```
                int w2=std::max(0,w-1);
                for(int j=-w2;j<=w2;++j) plot(cx,cy+j,cMid);

                if(w2>=0) plot(cx,cy,cCore);
            }
        }

        void OutroScreen::setFlips(int flips) {
            m_flips = std::max(0, flips);
            update();
        }
```

Game Over / Outro Screen Feature

The game over interface is implemented as a dedicated `OutroScreen` widget, created as a child overlay of `MainWindow`. When a run ends, `MainWindow` calls `setStats()` and `setFlips()` to pass the final coins collected, nitro usage, score, distance travelled (in metres), and number of flips to this widget. The constructor configures a semi–transparent dark background using a styled palette and sizes the panel to a fixed fraction of the parent window, with `centerInParent()` keeping it visually centred and `resizeEvent()` recalculating the logical cell size for different resolutions.

The core visual layout is handled in `paintEvent()`, which renders into a coarse pixel grid. A titled panel ("GAME OVER") is drawn with the shared 5x7 bitmap font via `drawPixelText()`, using helper functions `textWidthCells()` and `fitTextScale()` to choose an appropriate font scale. The left column shows a coin icon and nitro flame, rendered with `drawPixelCoin()` and `drawPixelFlame()`, together with their counts. The right column displays the score, distance, and flips in evenly spaced rows, with dynamic scaling to prevent overlap. At the bottom, two large buttons ("RESTART" and "EXIT") are drawn as rectangular regions with pixel text; `mousePressEvent()` tests clicks against these rectangles and emits `restartRequested()` or `exitRequested()`, which `MainWindow` uses to either restart the level or return to the intro screen.

Figure 6: Outro Screen: Game Over

pause.h

```
        #pragma once
```

```
#include <QWidget>
#include <QTimer>
#include <QRect>
#include <QString>
#include "constants.h"

class PauseOverlay : public QWidget {
    Q_OBJECT
    public:
    explicit PauseOverlay(QWidget* parent=nullptr);
    void setLevelIndex(int idx);
    void showPaused();
    signals:
    void resumeRequested();
    protected:
    void paintEvent(QPaintEvent*) override;
    void mousePressEvent(QMouseEvent*) override;
    void resizeEvent(QResizeEvent*) override;
    private:
    int m_levelIndex = 0;
    enum State { Paused, CountingDown } m_state = Paused;
    int m_count = 3;
    QTimer m_timer;
    QRect m_resumeRectPx;
    static constexpr int CHAR_ADV = 7;
    inline int gridW() const { return width()  / Constants::PIXEL_SIZE; }
    inline int gridH() const { return height() / Constants::PIXEL_SIZE; }
    int textWidthCells(const QString& s, int scale) const;
    void drawPixelText(QPainter& p, const QString& s, int gx, int gy, int
        scale, const QColor& c, bool bold);
    void plotGridPixel(QPainter& p, int gx, int gy, const QColor& c);
    QRect resumeRectPx() const;
};
```

pause.cpp

```
#include "pause.h"
#include <QPainter>
#include <QMouseEvent>
#include <algorithm>

PauseOverlay::PauseOverlay(QWidget* parent) : QWidget(parent) {
    setAttribute(Qt::WA_TranslucentBackground);
    setAttribute(Qt::WA_NoSystemBackground, false);
    setFocusPolicy(Qt::NoFocus);
    connect(&m_timer, &QTimer::timeout, this, [this]{
        if (--m_count <= 0) { m_timer.stop(); emit resumeRequested(); }
        update();
    });
}

```

```
16          void PauseOverlay::setLevelIndex(int idx) {
17              m_levelIndex = idx;
18          }
19
20          void PauseOverlay::showPaused() {
21              m_state = Paused;
22              m_count = 3;
23              m_timer.stop();
24              show();
25              raise();
26              update();
27          }
28
29          void PauseOverlay::paintEvent(QPaintEvent*) {
30              QPainter p(this);
31              p.fillRect(rect(), QColor(0,0,0,150));
32
33              const int gw = gridW();
34              const int gh = gridH();
35              const QString title = "GAME PAUSED";
36
37              int ts = std::clamp(gh / (7 * 8), 2, 6);
38              int tw = textWidthCells(title, ts);
39              int tgx = (gw - tw) / 2;
40              int tgy = gh / 3;
41              drawPixelText(p, title, tgx, tgy, ts, Constants::TEXT_COLOR[
                    m_levelIndex], true);
42
43              if (m_state == Paused) {
44                  QRect r = resumeRectPx();
45                  p.setPen(Qt::NoPen);
46                  p.setBrush(QColor(0,0,0,160));
47                  p.drawRect(r.translated(3*Constants::PIXEL_SIZE, 3*Constants::
                        PIXEL_SIZE));
48                  p.setBrush(QColor(180,180,190));
49                  p.drawRect(r);
50                  const QString lab = "RESUME";
51                  int rc = r.width()  / Constants::PIXEL_SIZE;
52                  int rr = r.height() / Constants::PIXEL_SIZE;
53                  int s  = std::clamp(std::min(rc / (int(lab.size())*CHAR_ADV - (
                        CHAR_ADV-5)), rr / 2), 1, 6);
54                  int gx = r.left()/Constants::PIXEL_SIZE + (rc - textWidthCells(lab
                        , s))/2;
55                  int gy = r.top() /Constants::PIXEL_SIZE + (rr - 7*s)/2;
56                  drawPixelText(p, lab, gx, gy, s, QColor(25,25,28), false);
57                  m_resumeRectPx = r;
58              } else {
59                  const QString num = QString::number(m_count);
60                  int ns = std::clamp(gh / (7 * 4), 3, 10);
61                  int nw = textWidthCells(num, ns);
62                  int ngx = (gw - nw) / 2;
63                  int ngy = tgy + 7*ts + 8;
```

```
64                  drawPixelText(p, num, ngx, ngy, ns, Constants::TEXT_COLOR[
                        m_levelIndex], true);
65              }
66          }
67
68          void PauseOverlay::mousePressEvent(QMouseEvent* e) {
69              if (m_state != Paused) return;
70              if (resumeRectPx().contains(e->pos())) {
71                  m_state = CountingDown;
72                  m_count = 3;
73                  m_timer.start(1000);
74                  update();
75              }
76          }
77
78          void PauseOverlay::resizeEvent(QResizeEvent*) {
79              update();
80          }
81
82          int PauseOverlay::textWidthCells(const QString& s, int scale) const {
83              if (s.isEmpty()) return 0;
84              return (int(s.size()) - 1) * CHAR_ADV * scale + 5 * scale;
85          }
86
87          void PauseOverlay::drawPixelText(QPainter& p, const QString& s, int gx,
                int gy, int scale, const QColor& c, bool bold) {
88              auto plot = [&](int x,int y,const QColor& col){ plotGridPixel(p, gx +
                    x, gy + y, col); };
89              for (int i = 0; i < s.size(); ++i) {
90                  const QChar ch = s.at(i).toUpper();
91                  const auto rows = font_map.value(font_map.contains(ch) ? ch :
                        QChar(' '));
92                  int baseOff = i * CHAR_ADV * scale;
93                  for (int ry=0; ry<7; ++ry) {
94                      uint8_t row = rows[ry];
95                      for (int rx=0; rx<5; ++rx) {
96                          if (row & (1<<(4-rx))) {
97                              for (int sy=0; sy<scale; ++sy) {
98                                  for (int sx=0; sx<scale; ++sx) {
99                                      if (bold) {
100                                         plot(baseOff + rx*scale+sx-1, ry*scale+sy,
                                                c);
101                                         plot(baseOff + rx*scale+sx+1, ry*scale+sy,
                                                c);
102                                         plot(baseOff + rx*scale+sx, ry*scale+sy-1,
                                                c);
103                                         plot(baseOff + rx*scale+sx, ry*scale+sy+1,
                                                c);
104                                     }
105                                     plot(baseOff + rx*scale+sx, ry*scale+sy, c);
106                                 }
107                             }
```

```
                        }
                    }
                }
            }
        }

        void PauseOverlay::plotGridPixel(QPainter& p, int gx, int gy, const QColor
            & c) {
            if (gx < 0 || gy < 0 || gx >= gridW()+1 || gy >= gridH()+1) return;
            p.fillRect(gx * Constants::PIXEL_SIZE, gy * Constants::PIXEL_SIZE,
                Constants::PIXEL_SIZE, Constants::PIXEL_SIZE, c);
        }

        QRect PauseOverlay::resumeRectPx() const {
            int rc = std::min(std::max(gridW()/6, 30), 60);
            int rr = std::max(10, gridH()/18);
            int gx = (gridW() - rc) / 2;
            int gy = gridH()/2;
            return QRect(gx*Constants::PIXEL_SIZE, gy*Constants::PIXEL_SIZE, rc*
                Constants::PIXEL_SIZE, rr*Constants::PIXEL_SIZE);
        }
```

Pause Overlay Feature

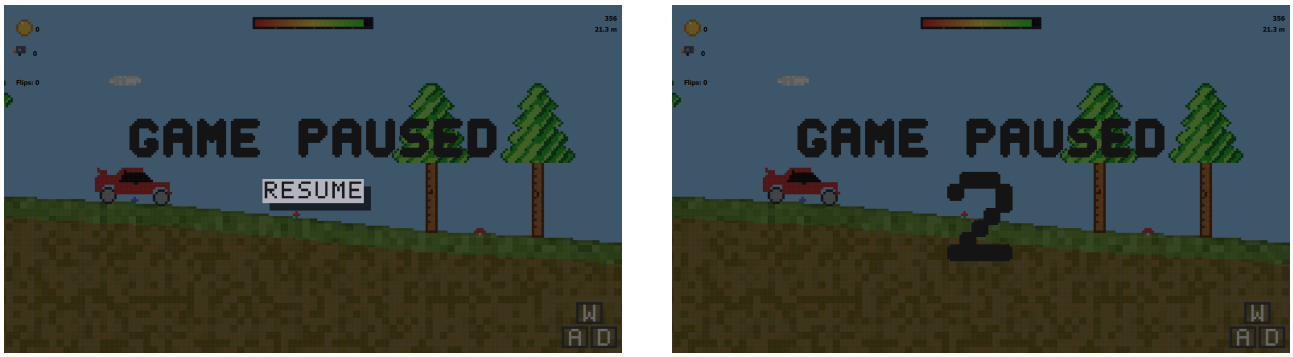
The pause functionality is implemented as a dedicated `PauseOverlay` widget that is owned and controlled by `MainWindow`. When the player presses the pause key, `MainWindow` invokes `showPaused()`, which switches the internal state to `Paused`, resets a 3 second counter, and shows a semi transparent dark overlay above the game. The overlay uses the global pixel size (`Constants::PIXEL_SIZE`) and logical grid dimensions `gridW()` and `gridH()` to render a centered panel independent of window resolution.

The core rendering is implemented in `paintEvent()`. It draws a translucent black full screen rectangle and a large pixel font title "GAME PAUSED" using `drawPixelText()`, with text color selected from `Constants::TEXT_COLOR` via `setLevelIndex()`. In the paused state, a rectangular "RESUME" button is drawn using grid based metrics from `resumeRectPx()`, and the text scale is computed by `textWidthCells()` to fit inside the button. When the button is clicked (`mousePressEvent()`), the overlay switches to the `CountingDown` state, starts a 1 Hz `QTimer`, and displays a large countdown (3, 2, 1) in the same pixel font. Each timer tick decrements `m_count`, and when it reaches zero, the timer stops and the signal `resumeRequested()` is emitted, allowing `MainWindow` to resume the game loop cleanly.

Feature: Coins persistence across runs

Relevant functions: `loadGrandCoins()`, `saveGrandCoins()`, `showGameOver()`, `returnToIntro()`, `closeEvent()`.

This feature uses a separate aggregate counter `m_grandTotalCoins` to preserve the total coins earned across all runs, independent of the per-run `m_coinCount`. On startup and whenever a new round is prepared, `loadGrandCoins()` reads the last saved value from `QSettings` (organisation `"JU"`, application `"F1PixelGrid"`) and initialises `m_grandTotalCoins`. At the end of a run, both the restart path and the exit path in `showGameOver()` and `returnToIntro()` add the current `m_coinCount` into `m_grandTotalCoins`



(a) Game Paused (b) Game resuming with countdown

Figure 7: Pause overlay and countdown sequence.

before resetting the round. `saveGrandCoins()` then writes this updated total back to `QSettings` and is also called from `closeEvent()` to guard against abrupt application exits. As a result, the overall coin total survives across sessions and never resets to zero when starting a new game.

Relevant excerpts from mainwindow.cpp (for reference)

```
1   void MainWindow::loadGrandCoins() {
2       QSettings s("JU","F1PixelGrid");
3       m_grandTotalCoins = s.value("grandCoins", 0).toInt();
4   }
5
6   void MainWindow::saveGrandCoins() const {
7       QSettings s("JU","F1PixelGrid");
8       s.setValue("grandCoins", m_grandTotalCoins);
9       s.sync();
10  }
11
12  void MainWindow::showGameOver() {
13      if (m_media) m_media->playGameOverOnce();
14      if (m_leaderboardMgr) {
15          QString stageName = QStringLiteral("UNKNOWN");
16          if (level_index >= 0 && level_index < m_levelNames.size()) {
17              stageName = m_levelNames[level_index];
18          }
19          m_leaderboardMgr->submitScore(stageName, m_score);
20      }
21      if (m_outro) return;
22      if (m_timer) m_timer->stop();
23
24      m_outro = new OutroScreen(this);
25      m_outro->setStats(m_coinCount, m_nitroUses, m_score, (
            m_totalDistanceCells * Constants::PIXEL_SIZE) / 100.0);
26      m_outro->setFlips(m_flip.total());
27      m_outro->show();
28      m_outro->raise();
29
30      connect(m_outro, &OutroScreen::restartRequested, this, [this]{
31          if (m_outro) {
```

```
32                      m_outro->hide();
33                      m_outro->deleteLater();
34                      m_outro = nullptr;
35                  }
36                  m_grandTotalCoins += m_coinCount;
37                  saveGrandCoins();
38
39                  resetGameRound();
40                  m_pause->hide();
41                  m_roofCrashLatched = false;
42                  m_gameOverArmed = false;
43                  if (m_timer) {
44                      m_clock.restart();
45                      m_timer->start(10);
46                  }
47                  setFocus();
48              });
49
50              connect(m_outro, &OutroScreen::exitRequested, this, [this]{
                    returnToIntro(); });
51          }
```

Feature: Exit flow and return to intro screen

Relevant functions: `showGameOver()`, `returnToIntro()`, lambda connected to `IntroScreen::startRequested` inside `returnToIntro()`.

This behaviour is mainly handled by `showGameOver()` and `returnToIntro()`, together with the `IntroScreen` connections created inside `returnToIntro()`. When the player clicks the `EXIT` button on the game-over panel, `OutroScreen` emits `exitRequested()`, which `MainWindow::showGameOver()` connects directly to `returnToIntro()`. In `returnToIntro()`, the current run is gracefully closed: the timer is stopped, the session id is advanced, game-over flags are cleared, and any remaining run coins are folded into `m_grandTotalCoins` and persisted via `saveGrandCoins()`. The existing `OutroScreen` is deleted and all input state is reset. A fresh `IntroScreen` instance is then constructed, initialised with the last played `level_index` and current `m_grandTotalCoins`, and shown full-screen. New signal connections from this intro instance back to `MainWindow` allow the next stage selection to start a completely reset game round.

Relevant excerpts from mainwindow.cpp (for reference)

```
1   void MainWindow::returnToIntro() {
2       loadGrandCoins();
3       ++m_sessionId;
4       m_gameOverArmed = false;
5       m_roofCrashLatched = false;
6
7       if (m_timer) m_timer->stop();
8
9       m_grandTotalCoins += m_coinCount;
10      saveGrandCoins();
11
```

```
12          if (m_outro) {
13              m_outro->hide();
14              m_outro->deleteLater();
15              m_outro = nullptr;
16          }
17
18          m_accelerating = m_braking = m_nitroKey = false;
19          m_prevNitroActive = false;
20
21          m_intro = new IntroScreen(this, level_index);
22          m_intro->setGeometry(rect());
23          m_intro->setGrandCoins(m_grandTotalCoins);
24          m_intro->show();
25
26          connect(m_intro, &IntroScreen::exitRequested, this, &QWidget::close);
27          connect(m_intro, &IntroScreen::startRequested, this, [this](int levelIndex
                ){
28              if (m_intro) {
29                  m_intro->hide();
30                  m_intro->deleteLater();
31                  m_intro = nullptr;
32              }
33
34              level_index = levelIndex;
35
36
37              if (m_media) {
38                  m_media->setStageBgm(level_index);
39              }
40
41              QPalette pal = palette();
42              pal.setColor(QPalette::Window, Constants::SKY_COLOR[level_index]);
43              setAutoFillBackground(true);
44              setPalette(pal);
45
46              resetGameRound();
47              setFocus();
48              if (m_timer) m_timer->start();
49          });
50      }
51
52      void MainWindow::closeEvent(QCloseEvent* e) {
53          saveGrandCoins();
54          QWidget::closeEvent(e);
55      }
```

Feature: Restarting the same stage after Game Over

Relevant functions: `showGameOver()`, `resetGameRound()`, `saveGrandCoins()`.

Restarting the same stage after game over is implemented inside `showGameOver()` through the connection to `OutroScreen::restartRequested` and the shared reset routine `resetGameRound()`. When

the game-over overlay is created, `MainWindow` connects the restart signal to a lambda that first hides and deletes the existing `OutroScreen`. Before reinitialising gameplay, the lambda adds the coins earned in the finished run (`m_coinCount`) into `m_grandTotalCoins` and calls `saveGrandCoins()`, ensuring progression is preserved. It then invokes `resetGameRound()`, which regenerates terrain, re-creates the car, resets fuel, score, nitro state, flip tracker and camera variables, and restarts the internal clock. Game-over latches and pause state are cleared, and the main timer is restarted at a fixed timestep. Because `level_index` is left unchanged, the player automatically restarts on the same stage configuration.

Relevant excerpts from mainwindow.cpp (for reference)

```
void MainWindow::disarmGameOver() { m_gameOverArmed = false; }

void MainWindow::resetGameRound() {
    loadGrandCoins();
    QPalette pal = palette();
    pal.setColor(QPalette::Window, Constants::SKY_COLOR[level_index]);
    setAutoFillBackground(true);
    setPalette(pal);

    m_gameOverArmed = false;
    m_roofCrashLatched  = false;
    ++m_sessionId;

    m_nitroSys = NitroSystem();
    m_fuelSys  = FuelSystem();
    m_coinSys  = CoinSystem();

    m_fuel = Constants::FUEL_MAX;
    m_coinCount = 0;
    m_nitroUses = 0;
    m_elapsedSeconds= 0.0;

    m_camX = m_camY = m_camVX = m_camVY = 0.0;
    m_cameraX = 0; m_cameraY = 200; m_cameraXFarthest = 0;

    m_lines.clear();
    m_heightAtGX.clear();
    m_lastX = 0;
    m_lastY = 0;
    m_slope = 0;
    m_difficulty = Constants::INITIAL_DIFFICULTY[level_index];
    m_irregularity = Constants::INITIAL_IRREGULARITY[level_index];
    m_terrain_height = Constants::INITIAL_TERRAIN_HEIGHT[level_index];
    generateInitialTerrain();

    m_clouds.clear();
    m_lastCloudSpawnX = 0;
    m_propSys.clear();

    qDeleteAll(m_wheels); m_wheels.clear();
    qDeleteAll(m_bodies); m_bodies.clear();

```

```
43              createCar();
44
45              m_totalDistanceCells = 0.0;
46              m_score = 0;
47
48              {
49                  double ax = 0.0;
50                  for (const Wheel* w : m_wheels) ax += w->x;
51                  m_lastScoreX = m_wheels.isEmpty() ? 0.0 : ax / m_wheels.size();
52              }
53
54              m_accelerating = m_braking = m_nitroKey = false;
55              m_prevNitroActive = false;
56              m_flip.reset();
57              m_clock.restart();
58          }
```

Feature: On–the–fly HUD score and distance rendering

Relevant functions: `MainWindow::drawHUDDistance()`, `MainWindow::drawHUDScore()`.

The HUD display of score and distance is layered on top of the world rendering inside `paintEvent()`. After drawing terrain, props, car, wheels, fuel, coins, nitro flame and flip popups, `paintEvent()` calls `drawHUDDistance(p)` and `drawHUDScore(p)` every frame, ensuring that both values update continuously during gameplay. These helpers use a bold monospace `QFont`, the per–stage text colour from `Constants::TEXT_COLOR[level_index]`, and `QFontMetrics` to right–align text against the window edge. `drawHUDDistance()` converts the accumulated distance in grid cells (`m_totalDistanceCells`) into metres using `PIXEL_SIZE` and formats it with one decimal place, while `drawHUDScore()` renders `m_score` as an integer directly above the distance label. Together, these functions provide a clean, always–visible summary of run progress.

Feature: Continuous score and distance calculation

Relevant functions: `MainWindow::gameLoop()`, `MainWindow::resetGameRound()`.

Score and travel distance are computed incrementally in `gameLoop()`, with initialisation performed in `resetGameRound()`. At the start of each round,

$$D(0) = 0, \quad S(0) = 0, \quad x_{\text{last}} = \overline{x}(0),$$

where D is the total distance in cells, S is the score and $\overline{x}$ is the average x–position of all wheels. At simulation step k,

$$\overline{x}(k) = \frac{1}{N}\sum_{i=1}^{N} x_i(k), \qquad \Delta x(k) = \max\bigl(0,\ \overline{x}(k) - x_{\text{last}}(k-1)\bigr),$$

so reversing never decreases distance. The increment in grid cells is

$$\Delta D(k) = \frac{\Delta x(k)}{\text{PIXEL\_SIZE}}, \qquad D(k) = D(k-1) + \Delta D(k),$$

and $x_{\text{last}}(k)$ is updated to $\overline{x}(k)$. The final scalar score is then computed as

$$S = \left\lfloor \, \text{round}\big(\alpha\, D + \beta\, C + \gamma\, N\big) \, \right\rfloor,$$

where C is the coin count, N is the number of nitro uses, and

$\alpha =$ `Constants::SCORE_DIST_PER_CELL`, $\beta =$ `Constants::SCORE_PER_COIN`, $\gamma =$ `Constants::SCORE_PER_NI`

Relevant excerpts from mainwindow.cpp (for reference)

```
1    void MainWindow::drawHUDDistance(QPainter& p) {
2        double meters = (m_totalDistanceCells * Constants::PIXEL_SIZE) / 100.0;
3        QString s = QString::number(meters, 'f', 1) + " m";
4        QFont f; f.setFamily("Monospace"); f.setBold(true); f.setPointSize(12);
5        p.setFont(f);
6        p.setPen(Constants::TEXT_COLOR[level_index]);
7        QFontMetrics fm(f);
8        int px = width() - fm.horizontalAdvance(s) - 12;
9        int py = (Constants::HUD_TOP_MARGIN + Constants::COIN_RADIUS_CELLS + 2) *
             Constants::PIXEL_SIZE;
10       p.drawText(px, py, s);
11   }
12
13   void MainWindow::drawHUDScore(QPainter& p) {
14       const QString s = QString::number(m_score);
15       QFont f; f.setFamily("Monospace"); f.setBold(true); f.setPointSize(12);
16       p.setFont(f);
17       p.setPen(Constants::TEXT_COLOR[level_index]);
18       QFontMetrics fm(f);
19       const int rightPadPx = 12;
20       const int px = width() - fm.horizontalAdvance(s) - rightPadPx;
21       const int distancePy = (Constants::HUD_TOP_MARGIN + Constants::
             COIN_RADIUS_CELLS + 2) * Constants::PIXEL_SIZE;
22       const int gapPx = 8;
23       const int py = distancePy - fm.height() - gapPx;
24       p.drawText(px, py, s);
25   }
```

(a) The grand total coins is maintained across all runs. (Top left of the image). It is refreshed every time the game is opened and every time the game gets over.

(b) On the fly score and distance calculation. (Top right of the image). The score calculations have also been described.

Figure 8: grand total coins and on the fly score & dist metrics

Feature: Coin System and HUD Coins Indicator

Relevant classes and functions: `CoinSystem`, `CoinSystem::maybePlaceCoinStreamAtEdge()`, `CoinSystem::draw` `CoinSystem::handlePickups()`, `MainWindow::gameLoop()`, `MainWindow::drawHUDCoins()`.

In my implementation, coins are modelled as light-weight world objects stored in a dedicated `CoinSystem`. Each coin is represented by integer world coordinates (c_x, c_y) in pixel space and a Boolean flag:

$$C_i = (c_{x,i}, c_{y,i}, \texttt{taken}_i), \quad \texttt{taken}_i \in \{0, 1\}.$$

Coins are spawned just outside the current view frustum in short "streams" that follow the terrain profile; at run-time they are rendered in world space using a midpoint circle rasteriser and collected via an Euclidean distance test around the car wheels and body.

Temporal and spatial spawning conditions. Coin streams are not spawned continuously, but are throttled by a simple time condition. Let t be the current elapsed time in seconds and t_{last} the time at which the previous stream was spawned. A new stream is considered only if

$$\Delta t_{\text{spawn}} = t - t_{\text{last}} \;\geq\; 5.0 \text{ s}.$$

The stream is also constrained to lie on terrain that already exists ahead of the current camera. If x_{cam} is the camera world x coordinate, W_{view} is the view width in pixels, and $x_{\text{terr}}^{\max}$ denotes the last generated terrain x (in pixels), then I compute:

$$\begin{aligned} x_{\text{right}}^{\text{view}} &= x_{\text{cam}} + W_{\text{view}}, \\ x_{\text{off}} &= x_{\text{right}}^{\text{view}} + M_{\text{coin}}, \\ x_{\text{terr}}^{\max} &= x_{\text{lastTerrain}} - 10 \cdot s_{\text{px}}, \end{aligned}$$

where $M_{\text{coin}} = \texttt{COIN\_SPAWN\_MARGIN\_CELLS} \cdot s_{\text{px}}$ and $s_{\text{px}} = \texttt{PIXEL\_SIZE}$ is the world-to-grid scale. A coin stream is only spawned if

$$x_{\text{terr}}^{\max} > x_{\text{off}},$$

ensuring there is sufficient terrain to place the entire arc of coins ahead of the player.

Randomised coin streams and sinusoidal vertical offset. Every stream draws a random integer length N and horizontal spacing (in grid cells) from uniform distributions:

$$N \sim \mathcal{U}(\texttt{COIN\_GROUP\_MIN}, \texttt{COIN\_GROUP\_MAX}), \qquad s_{\text{step}} \sim \mathcal{U}(\texttt{COIN\_GROUP\_STEP\_MIN}, \texttt{COIN\_GROUP\_STEP\_MAX}).$$

The total horizontal width of the stream in pixels is

$$W_{\text{stream}} = (N - 1)\, s_{\text{step}}\, s_{\text{px}}.$$

I initially place the leftmost coin at

$$x_{\text{start}} = x_{\text{off}} + 2\, s_{\text{px}},$$

and clamp $(x_{\text{start}}, x_{\text{end}})$ to ensure the group fits before the current terrain limit:

$$x_{\text{end}} = x_{\text{start}} + W_{\text{stream}}, \qquad x_{\text{end}} \le x_{\text{terr}}^{\max}.$$

If the naive placement would cross the terrain limit, I simply shift the whole group left so that $x_{\text{end}} = x_{\text{terr}}^{\max}$.

For each coin index $i \in \{0, \dots, N-1\}$, the world x coordinate is

$$w_x(i) = x_{\text{start}} + i \cdot s_{\text{step}} \cdot s_{\text{px}}.$$

I convert this to a terrain grid column

$$g_x(i) = \left\lfloor \frac{w_x(i)}{s_{\text{px}}} \right\rfloor,$$

look up the corresponding ground height $g_y^{\text{ground}}(i)$ from the integer height map `heightAtGX`, and then place the coin a fixed number of cells above the floor plus a sinusoidal arc. With amplitude $A =$ `COIN_STREAM_AMP_CELLS` and a random phase $\varphi \in [0, 2\pi)$, the vertical offset in cells is

$$\Delta g_y(i) = \lfloor \text{round}\big(\sin(\varphi + 0.55\, i)\, A\big) \rfloor,$$

so the final grid y coordinate for coin i becomes

$$g_y(i) = g_y^{\text{ground}}(i) - \texttt{COIN\_FLOOR\_OFFSET\_CELLS} - \Delta g_y(i),$$

and the world pixel coordinates stored in the `Coin` struct are

$$c_{x,i} = w_x(i), \qquad c_{y,i} = g_y(i)\, s_{\text{px}}.$$

World-space rasterisation of coin sprites. To render coins in the world, I reuse the integer midpoint circle algorithm on the logical grid, similar to the wheel and HUD icons. For each coin that has not yet been taken, I convert its world position (c_x, c_y) into screen grid coordinates relative to the camera

$$g_x^{\text{screen}} = \frac{c_x}{s_{\text{px}}} - g_x^{\text{cam}}, \qquad g_y^{\text{screen}} = \frac{c_y}{s_{\text{px}}} + g_y^{\text{cam}},$$

where $g_x^{\text{cam}} = \lfloor x_{\text{cam}}/s_{\text{px}} \rfloor$ and $g_y^{\text{cam}} = \lfloor y_{\text{cam}}/s_{\text{px}} \rfloor$ are the camera grid offsets. I then draw three concentric filled discs with radii

$$r, \quad r-1, \quad r-2, \qquad r = \texttt{COIN\_RADIUS\_CELLS},$$

using different colours for rim and fill, plus a small highlight pixel in the top-left of the sprite to suggest specular shine.

Pickup detection with Euclidean distance. Coin collection is implemented in two stages. First, I do a radial pickup test against all wheel centres. For each coin C_i that is still active and every wheel centre $(w_{x,j}, w_{y,j})$, I compute the squared Euclidean distance

$$d_{ij}^2 = (w_{x,j} - c_{x,i})^2 + (w_{y,j} - c_{y,i})^2,$$

and keep the minimum

$$d_i^2 = \min_j d_{ij}^2.$$

With a global pickup radius $R =$ `COIN_PICKUP_RADIUS`, the coin is collected if

$$d_i^2 \le R^2.$$

When this condition holds, I mark `taken` for that coin and increment the run-specific counter `m_coinCount`. This `m_coinCount` is later folded into the total score and also added to a persistent `m_grandTotalCoins` at the end of each run (both on restart and when exiting to the intro screen).

In addition to wheel-based pickups, I perform a more precise overlap check between each remaining coin and the car polygon to catch edge cases where the body intersects a coin without a wheel touching it. For each polygon edge segment $\overline{AB}$, I compute the orthogonal projection of the coin centre P onto the segment:

$$t = \mathrm{clamp}_{[0,1]}\left(\frac{(P-A)\cdot(B-A)}{\|B-A\|^2}\right), \quad C = A + t\,(B-A),$$

and test the squared distance $\|P-C\|^2$ against R^2. If no edge collision is found, I also check whether P lies inside the body polygon via an odd-even scan, analogous to a standard point-in-polygon test. Any hit again marks the coin as taken and increments `m_coinCount`.

HUD coins indicator. On the HUD, I show a compact representation of the current coin count in the top-left corner using the same pixel-grid renderer as the rest of the UI. The HUD circle icon is drawn in grid space at

$$g_x^{\mathrm{HUD}} = L_{\mathrm{HUD}} + r + 1, \qquad g_y^{\mathrm{HUD}} = T_{\mathrm{HUD}} + r,$$

where $L_{\mathrm{HUD}} =$ `HUD_LEFT_MARGIN`, $T_{\mathrm{HUD}} =$ `HUD_TOP_MARGIN` and $r =$ `COIN_RADIUS_CELLS`. These grid coordinates are converted to pixels by multiplying with s_{px}. The icon is rendered as two concentric filled circles plus a highlight, and the numeric coin count is then drawn to the right using a bold monospace `QFont`. The layout ensures that increments in `m_coinCount` are immediately visible to the player after each pickup.

Core code snippets. The following excerpts show the core of the coin system and HUD implementation used in the project.

coin.h

```
#ifndef COIN_H
#define COIN_H

#include <QVector>
#include <QColor>
#include <QPainter>
#include <QHash>
#include <random>
#include "constants.h"
#include "wheel.h"

struct Coin {
    int cx;
```

```
    int cy;
    bool taken = false;
};

class CoinSystem {
public:
    QVector<Coin> coins;
    int    lastPlacedCoinX = 0;
    double lastSpawnTimeSec = 0.0;

    void maybePlaceCoinStreamAtEdge(
        double elapsedSeconds,
        int cameraX,
        int viewWidth,
        const QHash<int,int>& heightAtGX,
        int lastTerrainX,
        std::mt19937& rng,
        std::uniform_real_distribution<float>& dist
        );

    void drawWorldCoins(QPainter& p, int cameraX, int cameraY, int gridW, int
        gridH) const;

    void handlePickups(const QList<Wheel*>& wheels, int& coinCount);
};

#endif // COIN_H
```

coin.cpp

```
#include "coin.h"
#include <cmath>
#include <algorithm>

void CoinSystem::maybePlaceCoinStreamAtEdge(
    double elapsedSeconds,
    int cameraX,
    int viewWidth,
    const QHash<int,int>& heightAtGX,
    int lastTerrainX,
    std::mt19937& rng,
    std::uniform_real_distribution<float>& dist
    ) {
    if ((elapsedSeconds - lastSpawnTimeSec) < 5.0) {
        return;
    }

    const int viewRightX = cameraX + viewWidth;
    const int marginPx   = Constants::COIN_SPAWN_MARGIN_CELLS * Constants::
        PIXEL_SIZE;
    const int offRightX  = viewRightX + marginPx;

    const int terrainLimitX = lastTerrainX - Constants::PIXEL_SIZE * 10;
```

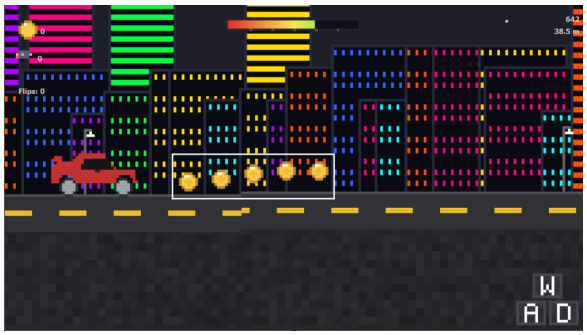
```
23      if (terrainLimitX <= offRightX) {
24          return;
25      }
26
27      std::uniform_int_distribution<int> coinLenDist(Constants::COIN_GROUP_MIN,
            Constants::COIN_GROUP_MAX);
28      const int groupN = coinLenDist(rng);
29
30      std::uniform_int_distribution<int> stepDist(Constants::COIN_GROUP_STEP_MIN,
            Constants::COIN_GROUP_STEP_MAX);
31      const int stepCells = stepDist(rng);
32
33      const int streamWidthPx = (groupN - 1) * stepCells * Constants::PIXEL_SIZE;
34
35      int startX = offRightX + Constants::PIXEL_SIZE * 2;
36      int endX   = startX + streamWidthPx;
37
38      if (endX > terrainLimitX) {
39          startX = terrainLimitX - streamWidthPx;
40          endX   = terrainLimitX;
41      }
42
43      if (startX <= offRightX) {
44          return;
45      }
46
47      const int ampCells = Constants::COIN_STREAM_AMP_CELLS;
48      const double phase = dist(rng) * 6.2831853;
49
50      for (int i = 0; i < groupN; ++i) {
51          const int wx = startX + i * (stepCells * Constants::PIXEL_SIZE);
52          const int gx = wx / Constants::PIXEL_SIZE;
53
54          auto it = heightAtGX.constFind(gx);
55          if (it == heightAtGX.constEnd()) {
56              continue;
57          }
58
59          const int gyGround = it.value();
60          const int arcOffsetCells =
61              int(std::lround(std::sin(phase + i * 0.55) * ampCells));
62          const int gy = gyGround - Constants::COIN_FLOOR_OFFSET_CELLS -
                arcOffsetCells;
63
64          Coin c;
65          c.cx = wx;
66          c.cy = gy * Constants::PIXEL_SIZE;
67          c.taken = false;
68          coins.append(c);
69      }
70
71      lastPlacedCoinX   = endX;
```

```
72      lastSpawnTimeSec  = elapsedSeconds;
73  }
74
75  void CoinSystem::drawWorldCoins(QPainter& p, int cameraX, int cameraY, int /*gridW
        */, int /*gridH*/) const {
76      const int camGX = cameraX / Constants::PIXEL_SIZE;
77      const int camGY = cameraY / Constants::PIXEL_SIZE;
78
79      QColor rim  (195,140,40);
80      QColor fill (250,204,77);
81      QColor fill2(245,184,50);
82      QColor shine(255,255,220);
83
84      auto plotGridPixel = [&](int gx, int gy, const QColor& c) {
85          p.fillRect(gx * Constants::PIXEL_SIZE,
86                     gy * Constants::PIXEL_SIZE,
87                     Constants::PIXEL_SIZE,
88                     Constants::PIXEL_SIZE, c);
89      };
90
91      auto drawCircleFilledMidpointGrid = [&](int gcx, int gcy, int gr, const QColor
            & color) {
92          auto span = [&](int cy, int xl, int xr) {
93              for (int xg = xl; xg <= xr; ++xg) {
94                  plotGridPixel(xg, cy, color);
95              }
96          };
97          int x = 0;
98          int y = gr;
99          int d = 1 - gr;
100         while (y >= x) {
101             span(gcy + y, gcx - x, gcx + x);
102             span(gcy - y, gcx - x, gcx + x);
103             span(gcy + x, gcx - y, gcx + y);
104             span(gcy - x, gcx - y, gcx + y);
105             ++x;
106             if (d < 0) {
107                 d += 2 * x + 1;
108             } else {
109                 --y;
110                 d += 2 * (x - y) + 1;
111             }
112         }
113     };
114
115     for (const Coin& c : coins) {
116         if (c.taken) continue;
117
118         int scx = (c.cx / Constants::PIXEL_SIZE) - camGX;
119         int scy = (c.cy / Constants::PIXEL_SIZE) + camGY;
120         int r   = Constants::COIN_RADIUS_CELLS;
121
```

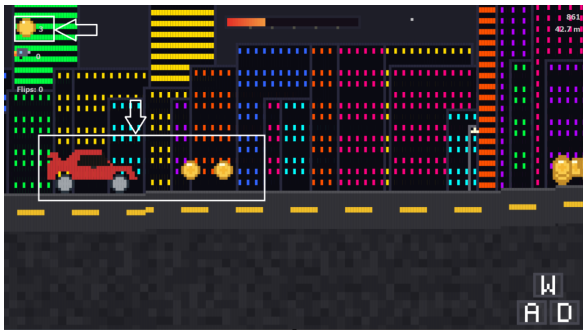
```
        drawCircleFilledMidpointGrid(scx, scy, r,   rim);
        if (r-1 > 0) {
            drawCircleFilledMidpointGrid(scx, scy, r-1, fill);
        }
        if (r-2 > 0) {
            drawCircleFilledMidpointGrid(scx, scy, r-2, fill2);
        }

        plotGridPixel(scx-1, scy-r+1, shine);
        plotGridPixel(scx,   scy-r+1, shine);
    }
}

void CoinSystem::handlePickups(const QList<Wheel*>& wheels, int& coinCount) {
    if (wheels.isEmpty()) return;

    for (Coin& c : coins) {
        if (c.taken) continue;
        double minD2 = 1e18;
        for (const Wheel* w : wheels) {
            const double dx = w->x - c.cx;
            const double dy = w->y - c.cy;
            const double d2 = dx*dx + dy*dy;
            if (d2 < minD2) minD2 = d2;
        }
        const double R = Constants::COIN_PICKUP_RADIUS;
        if (minD2 <= R*R) { c.taken = true; ++coinCount; }
    }
}
```

mainwindow.cpp (HUD coins excerpt)

```
void MainWindow::drawHUDCoins(QPainter& p) {
    int iconGX = Constants::HUD_LEFT_MARGIN + Constants::COIN_RADIUS_CELLS + 1;
    int iconGY = Constants::HUD_TOP_MARGIN  + Constants::COIN_RADIUS_CELLS;
    drawCircleFilledMidpointGrid(p, iconGX, iconGY, Constants::COIN_RADIUS_CELLS,
        QColor(195,140,40));
    drawCircleFilledMidpointGrid(p, iconGX, iconGY, std::max(1, Constants::
        COIN_RADIUS_CELLS-1), QColor(250,204,77));
    plotGridPixel(p, iconGX-1, iconGY-Constants::COIN_RADIUS_CELLS+1, QColor
        (255,255,220));
    QFont f; f.setFamily("Monospace"); f.setBold(true); f.setPointSize(12);
    p.setFont(f);
    p.setPen(Constants::TEXT_COLOR[level_index]);
    int px = (Constants::HUD_LEFT_MARGIN + Constants::COIN_RADIUS_CELLS*2 + 3) *
        Constants::PIXEL_SIZE;
    int py = (Constants::HUD_TOP_MARGIN  + Constants::COIN_RADIUS_CELLS + 2) *
        Constants::PIXEL_SIZE;
    p.drawText(px, py, QString::number(m_coinCount));
}
```



(a) Procedural coin stream spawning ahead of the vehicle



(b) Coin collection and HUD counter update

Figure 9: Coin system: terrain-following stream placement and real-time collection with HUD feedback.

Coin System Results

In practice, the sinusoidal coin streams remained tightly aligned with the generated terrain and always spawned just outside the current view, avoiding unfair or unreachable placements. During gameplay, every successful pickup immediately updated the on-screen coin counter and the persistent grand-total, confirming the correctness of both the world-space collision logic and the HUD integration.

Feature: Fuel System, Consumption Model and HUD Fuel Bar

Relevant classes and functions: `FuelSystem`, `FuelSystem::currentFuelSpacing()`, `FuelSystem::maybePlaceFuelAtEdge()`, `FuelSystem::drawWorldFuel()`, `FuelSystem::handlePickups()`, `MainWindow::gameLoop()`, `MainWindow::drawHUDFuel()`.

In my implementation, fuel is treated as a continuous scalar resource

$$f(t) \in [0, F_{\max}], \qquad F_{\max} = \texttt{Constants::FUEL\_MAX},$$

normalized to the range $[0, 1]$ for HUD display. At the beginning of a run I set

$$f(0) = F_{\max},$$

and the rest of the system either dissipates or refills this quantity as the car moves, accelerates and collects fuel cans.

Fuel consumption per frame. Each simulation step has duration $\Delta t = \texttt{dt}$ (in seconds). If there is still fuel available, I subtract a base burn rate and an additional speed-dependent term. Let

$$k_{\text{base}} = \texttt{FUEL\_BASE\_BURN\_PER\_SEC}, \qquad k_{\text{speed}} = \texttt{FUEL\_EXTRA\_PER\_SPEED},$$

and let $\bar{v}(t)$ denote the average horizontal speed of the car at time t. The instantaneous burn components are:

$$b_{\text{base}}(t) = k_{\text{base}}\,\Delta t,$$

$$b_{\text{speed}}(t) = \begin{cases} 0, & \text{if the accelerator is not pressed,} \\ \max(0, \bar{v}(t))\,k_{\text{speed}}\,\Delta t, & \text{if accelerating.} \end{cases}$$

Nitro increases the fuel drain multiplicatively. I define

$$\alpha(t) = \begin{cases} 3, & \text{if nitro is active at time } t, \\ 1, & \text{otherwise,} \end{cases}$$

and update the continuous fuel value whenever $f(t) > 0$ as

$$f(t + \Delta t) \;=\; \max\!\big(0,\; f(t) - \alpha(t)\big(b_{\text{base}}(t) + b_{\text{speed}}(t)\big)\big).$$

In other words, nitro simply triples the total burn. Once f reaches zero, inputs are suppressed and the run will eventually end when either the car flips or forward progress stops.

Spatially adaptive fuel spawning. Fuel pickups are modelled as light-weight world-space objects:

$$F_i = (w_{x,i}, w_{y,i}, \texttt{taken}_i), \quad \texttt{taken}_i \in \{0, 1\},$$

where $(w_{x,i}, w_{y,i})$ are pixel coordinates. To avoid clustering and to make later stages more demanding, I make the spacing between successive cans an explicit function of both difficulty and elapsed run time.

Let d denote the current difficulty (a small positive number) and t the elapsed time in seconds. I first compute an unscaled spacing in pixels:

$$S_{\text{raw}}(d, t) = 700 + 500\left(\frac{d}{0.005}\right) + 35\,t.$$

This is then globally scaled by the factor $E = \texttt{FUEL\_SPAWN\_EASE}$ and clamped to a practical range:

$$S(d, t) = \text{clip}\Big(E\,S_{\text{raw}}(d, t),\; S_{\min},\; S_{\max}\Big),$$
$$S_{\min} = 10000\,E, \qquad S_{\max} = 200000\,E.$$

Here $\text{clip}(x, a, b) = \min(\max(x, a), b)$. Let x_{lastTerr} be the world x of the last generated terrain vertex and x_{lastFuel} the x coordinate at which the previous fuel can was placed. A new can is spawned only if

$$x_{\text{lastTerr}} - x_{\text{lastFuel}} \;\geq\; S(d, t),$$

so that increasing difficulty and elapsed time push fuel cans further apart.

When the spacing condition is satisfied, I convert the current terrain tip from pixels to grid space

$$g_x = \left\lfloor \frac{x_{\text{lastTerr}}}{s_{\text{px}}} \right\rfloor, \qquad s_{\text{px}} = \texttt{PIXEL\_SIZE},$$

look up the ground height $g_y^{\text{ground}}(g_x)$ from the height map, and place the new can a fixed offset above the floor:

$$\begin{aligned} w_{x,\text{new}} &= x_{\text{lastTerr}}, \\ w_{y,\text{new}} &= \left(g_y^{\text{ground}}(g_x) - \texttt{FUEL\_FLOOR\_OFFSET\_CELLS}\right) s_{\text{px}}, \\ \texttt{taken}_{\text{new}} &= 0. \end{aligned}$$

This ensures that each fuel can sits at a visually consistent height above the terrain, right at the edge of the currently generated world.

World-space rendering of fuel cans. Rendering uses the same logical grid as the rest of the pixel-art world. For each fuel can that has not been collected, I transform its world coordinates into camera-relative grid coordinates:

$$s_x = \frac{w_x}{s_{\text{px}}} - g_x^{\text{cam}}, \qquad s_y = \frac{w_y}{s_{\text{px}}} + g_y^{\text{cam}},$$

with $(g_x^{\text{cam}}, g_y^{\text{cam}})$ the camera offset in grid cells. Around this origin I draw a small jerry-can sprite using a few hard-coded pixels:

- a grey cap on top,
- a red rectangular body,
- a yellow "label" band across the centre,
- a semi-transparent shadow offset to one side.

Everything is filled using `plotGridPixel` calls on the grid, which results in a crisp, stylised jerry-can icon that is consistent with the overall pixel-art aesthetic.

Pickup detection and full refuel. For collision, I approximate the visual centre of each can as

$$P_i = (f_{x,i}, f_{y,i}) = (w_{x,i} + 2s_{\text{px}},\ w_{y,i} + 3s_{\text{px}}),$$

which roughly corresponds to the middle of the painted body rectangle. For each can and every wheel centre $W_j = (w_{x,j}, w_{y,j})$, I compute the squared Euclidean distance

$$d_{ij}^2 = \|P_i - W_j\|^2 = (w_{x,j} - f_{x,i})^2 + (w_{y,j} - f_{y,i})^2,$$

and keep the minimum over wheels

$$d_i^2 = \min_j d_{ij}^2.$$

Using an effective pickup radius

$$R_{\text{fuel}} = \texttt{FUEL\_PICKUP\_RADIUS} + 20,$$

the can is collected if

$$d_i^2 \le R_{\text{fuel}}^2.$$

On a successful pickup I mark

$$\texttt{taken}_i \leftarrow 1, \qquad f(t^+) \leftarrow F_{\max},$$

i.e., a single can currently refills the tank completely. I also play a short fuel-pickup sound and reset the spawn spacing logic accordingly. Although there is a configuration constant `FUEL_PICKUP_AMOUNT`, in the current build I chose the simpler "full refill" semantics for clarity of gameplay.

HUD fuel bar and low-fuel warning. On the HUD, I present the fuel level as a horizontal colour-gradient bar centred at the top of the screen. Let the bar width in grid cells be

$$W_{\text{bar}} = \text{clip}(\lfloor \tfrac{1}{4}\,\text{gridW} \rfloor, 24, 48),$$

and height $H_{\text{bar}} = 3$ cells. The current fuel fraction

$$\varphi(t) = \text{clamp}\left(\frac{f(t)}{F_{\max}}, 0, 1\right)$$

is mapped to a number of filled cells

$$N_{\text{filled}}(t) = \lfloor W_{\text{bar}}\,\varphi(t) \rfloor\,.$$

For $x = 0, \ldots, W_{\text{bar}} - 1$ I define a normalised horizontal parameter

$$\tau(x) = \frac{x}{\max(W_{\text{bar}} - 1, 1)},$$

and assign each filled column a colour obtained by piecewise-linear interpolation between three key colours:

$$C_{\text{bar}}(x) = \begin{cases} \text{lerp}(C_{\text{red}}, C_{\text{yellow}}, 2\tau(x)), & \tau(x) < 0.5, \\ \text{lerp}(C_{\text{yellow}}, C_{\text{green}}, 2(\tau(x) - 0.5)), & \tau(x) \ge 0.5, \end{cases}$$

where C_{red}, C_{yellow} and C_{green} are fixed RGB values. The background and border of the bar are drawn in darker tones, and I overlay small tick marks every few cells to give a coarse sense of consumption rate.

To emphasise imminent depletion, I introduce a low-fuel threshold

$$f_{\text{low}} = 0.25\,F_{\max}.$$

Whenever $f(t) \leq f_{\text{low}}$, I trigger a blinking warning symbol below the bar. Blinking is implemented via

$$\texttt{flashOn}(t) = \begin{cases} \text{true}, & \text{frac}(t) < 0.5, \\ \text{false}, & \text{otherwise}, \end{cases}$$

where $\text{frac}(t)$ is the fractional part of the global elapsed time in seconds. If both the low-fuel and flashing conditions hold, I draw a red triangular warning icon followed by the pixel text "LOW", centred under the bar, using the same grid-based rasteriser.

Core code snippets. The following excerpts show the complete fuel system implementation and the HUD fuel bar rendering as used in the project.

fuel.h

```
#ifndef FUEL_H
#define FUEL_H

#include <QVector>
#include <QColor>
#include <QPainter>
#include <QHash>
#include <random>
#include "constants.h"
#include "wheel.h"

struct FuelCan {
    int wx;
    int wy;
    bool taken = false;
};

class FuelSystem {
public:
    QVector<FuelCan> cans;
    int lastPlacedFuelX = 0;

    int currentFuelSpacing(double difficulty, double elapsedSeconds) const;

    void maybePlaceFuelAtEdge(int lastTerrainX,
                              const QHash<int,int>& heightAtGX,
                              double difficulty,
                              double elapsedSeconds);

    void drawWorldFuel(QPainter& p, int cameraX, int cameraY) const;
    void handlePickups(const QList<Wheel*>& wheels, double& fuel);
};

#endif // FUEL_H
```

fuel.cpp

```
#include "fuel.h"
#include <cmath>
```

```
#include <algorithm>

int FuelSystem::currentFuelSpacing(double difficulty,
                                   double elapsedSeconds) const {
    int base    = 700;
    int byDiff  = int(500 * (difficulty / 0.005));
    int byTime  = int(35 * elapsedSeconds);
    int spacing = base + byDiff + byTime;
    spacing     = int(std::round(spacing * Constants::FUEL_SPAWN_EASE));

    int minSpacing = int(std::round(10000 * Constants::FUEL_SPAWN_EASE));
    int maxSpacing = int(std::round(200000 * Constants::FUEL_SPAWN_EASE));
    if (spacing < minSpacing) spacing = minSpacing;
    if (spacing > maxSpacing) spacing = maxSpacing;

    return spacing;
}

void FuelSystem::maybePlaceFuelAtEdge(
    int lastTerrainX,
    const QHash<int,int>& heightAtGX,
    double difficulty,
    double elapsedSeconds
    ) {
    if (lastTerrainX - lastPlacedFuelX <
        currentFuelSpacing(difficulty, elapsedSeconds)) return;

    int gx = lastTerrainX / Constants::PIXEL_SIZE;
    auto it = heightAtGX.constFind(gx);
    if (it == heightAtGX.constEnd()) return;

    const int gyGround = it.value();

    FuelCan f;
    f.wx = lastTerrainX;
    f.wy = (gyGround - Constants::FUEL_FLOOR_OFFSET_CELLS)
            * Constants::PIXEL_SIZE;
    f.taken = false;
    cans.append(f);

    lastPlacedFuelX = lastTerrainX;
}

void FuelSystem::drawWorldFuel(QPainter& p,
                               int cameraX,
                               int cameraY) const {
    const int camGX = cameraX / Constants::PIXEL_SIZE;
    const int camGY = cameraY / Constants::PIXEL_SIZE;

    QColor body(230, 60, 60);
    QColor cap(230,230,230);
    QColor label(255,200,50);
```

```
55      QColor shadow(0,0,0,90);
56
57      auto plotGridPixel = [&](int gx, int gy, const QColor& c) {
58          p.fillRect(gx * Constants::PIXEL_SIZE,
59                     gy * Constants::PIXEL_SIZE,
60                     Constants::PIXEL_SIZE,
61                     Constants::PIXEL_SIZE, c);
62      };
63
64      for (const FuelCan& f : cans) {
65          if (f.taken) continue;
66
67          int sx = (f.wx / Constants::PIXEL_SIZE) - camGX;
68          int sy = (f.wy / Constants::PIXEL_SIZE) + camGY;
69
70          auto px = [&](int gx, int gy, const QColor& c){
71              plotGridPixel(sx+gx, sy+gy, c);
72          };
73
74          // shadow
75          px(2,1,shadow);
76          px(5,2,shadow);
77
78          // cap
79          px(1,0,cap);
80          px(2,0,cap);
81
82          // body
83          for (int y=1; y<=5; ++y) {
84              for (int x=0; x<=4; ++x) {
85                  px(x,y,body);
86              }
87          }
88
89          // label stripe
90          for (int x=1; x<=3; ++x) {
91              px(x,3,label);
92          }
93      }
94  }
95
96  void FuelSystem::handlePickups(const QList<Wheel*>& wheels,
97                                 double& fuel) {
98      if (wheels.isEmpty()) return;
99
100     for (FuelCan& f : cans) {
101         if (f.taken) continue;
102         const double fx = f.wx + 2 * Constants::PIXEL_SIZE;
103         const double fy = f.wy + 3 * Constants::PIXEL_SIZE;
104         double minD2 = 1e18;
105         for (const Wheel* w : wheels) {
106             const double dx = w->x - fx;
```

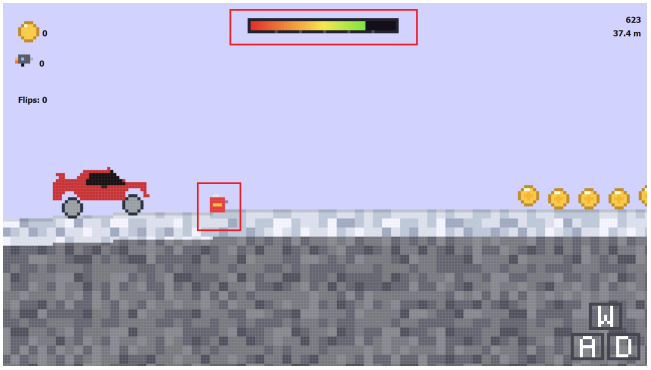
```
            const double dy = w->y - fy;
            const double d2 = dx*dx + dy*dy;
            if (d2 < minD2) minD2 = d2;
        }
        const double R = Constants::FUEL_PICKUP_RADIUS + 20;
        if (minD2 <= R*R) {
            f.taken = true;
            fuel = Constants::FUEL_MAX;
        }
    }
}
```

mainwindow.cpp (HUD fuel excerpt)

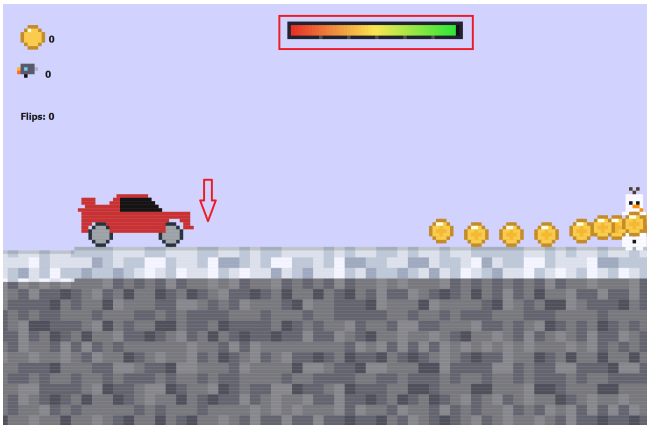
```
void MainWindow::drawHUDFuel(QPainter& p) {
    int gy = Constants::HUD_TOP_MARGIN;
    int wcells = std::min(std::max(gridW()/4, 24), 48);
    int gx = (gridW() - wcells)/2;
    int barH = 3;

    double frac = std::clamp(m_fuel / Constants::FUEL_MAX, 0.0, 1.0);
    int filled = int(std::floor(wcells * frac));

    auto lerp = [](const QColor& c1, const QColor& c2, double t)->QColor {
        int r = int((1-t)*c1.red()   + t*c2.red());
        int g = int((1-t)*c1.green() + t*c2.green());
        int b = int((1-t)*c1.blue()  + t*c2.blue());
        return QColor(r,g,b);
    };

    QColor startC(230,50,40), midC(250,230,80), endC(40,230,55);

    // background
    for (int x=0; x<wcells; ++x)
        for (int y=0; y<barH; ++y)
            plotGridPixel(p, gx+x, gy+y, QColor(20,14,24));

    // filled gradient
    for (int x=0; x<filled; ++x) {
        double t = double(x)/std::max(wcells-1,1);
        QColor c = (t<0.5)
            ? lerp(startC, midC, t*2)
            : lerp(midC, endC, (t-0.5)*2);
        for (int y=0; y<barH; ++y)
            plotGridPixel(p, gx+x, gy+y, c);
    }

    // vertical borders
    for (int y=0; y<barH; ++y) {
        plotGridPixel(p,gx-1,gy+y,QColor(35,35,48));
        plotGridPixel(p,gx+wcells,gy+y,QColor(35,35,48));
    }

```

```
40      // horizontal borders
41      for (int x=-1; x<=wcells; ++x) {
42          plotGridPixel(p,gx+x,gy-1,QColor(35,35,48));
43          plotGridPixel(p,gx+x,gy+barH,QColor(35,35,48));
44      }
45
46      // tick marks
47      int tickEvery = std::max(6, wcells/6);
48      for (int x=tickEvery; x<wcells; x+=tickEvery)
49          plotGridPixel(p, gx+x, gy+barH, QColor(80,80,70));
50
51      const double lowFuelThreshold = Constants::FUEL_MAX * 0.25;
52      const bool isLow = (m_fuel <= lowFuelThreshold);
53      const bool isFlashingOn =
54          (std::fmod(m_elapsedSeconds, 1.0) < 0.5);
55
56      if (isLow && isFlashingOn) {
57          const QColor red(230, 50, 40);
58          const QColor white(255, 255, 255);
59          const int warningGap = 3;
60          const int warningTopGY = gy + barH + 1 + warningGap;
61
62          const int totalWidth = 5 + 2 + 3 + 1 + 3 + 1 + 5;
63          int currentGX = (gridW() - totalWidth) / 2;
64
65          const int triTopGY = warningTopGY;
66
67          // triangle with exclamation
68          plotGridPixel(p, currentGX + 2, triTopGY,     red);
69          plotGridPixel(p, currentGX + 1, triTopGY + 1, red);
70          plotGridPixel(p, currentGX + 2, triTopGY + 1, white);
71          plotGridPixel(p, currentGX + 3, triTopGY + 1, red);
72          plotGridPixel(p, currentGX,     triTopGY + 2, red);
73          plotGridPixel(p, currentGX + 1, triTopGY + 2, red);
74          plotGridPixel(p, currentGX + 2, triTopGY + 2, white);
75          plotGridPixel(p, currentGX + 3, triTopGY + 2, red);
76          plotGridPixel(p, currentGX + 4, triTopGY + 2, red);
77          plotGridPixel(p, currentGX + 2, triTopGY + 4, white);
78
79          currentGX += 5 + 2;
80
81          const int textTopGY = warningTopGY;
82
83          // 'L'
84          for(int y=0; y<5; ++y)
85              plotGridPixel(p, currentGX, textTopGY+y, red);
86          plotGridPixel(p, currentGX+1, textTopGY+4, red);
87          plotGridPixel(p, currentGX+2, textTopGY+4, red);
88          currentGX += 3 + 1;
89
90          // 'O'
91          for(int y=0; y<5; ++y) {
```

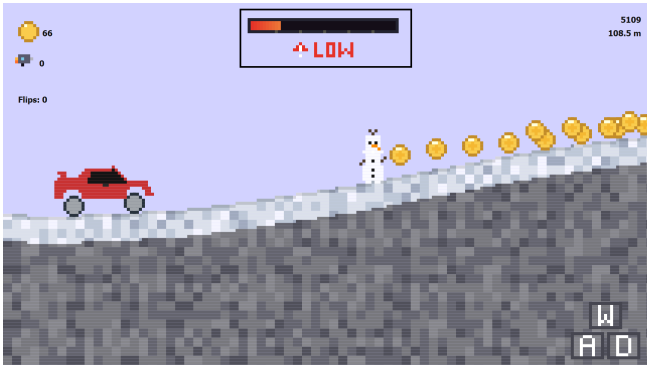
```
92              plotGridPixel(p, currentGX,   textTopGY+y, red);
93              plotGridPixel(p, currentGX+2, textTopGY+y, red);
94          }
95          plotGridPixel(p, currentGX+1, textTopGY,   red);
96          plotGridPixel(p, currentGX+1, textTopGY+4, red);
97          currentGX += 3 + 1;
98
99          // 'W'
100         for(int y=0; y<5; ++y) {
101             plotGridPixel(p, currentGX,   textTopGY+y, red);
102             plotGridPixel(p, currentGX+4, textTopGY+y, red);
103         }
104         plotGridPixel(p, currentGX+1, textTopGY+3, red);
105         plotGridPixel(p, currentGX+2, textTopGY+2, red);
106         plotGridPixel(p, currentGX+3, textTopGY+3, red);
107     }
108 }
```



(a) Procedural fuel can spawning at the terrain edge



(b) Fuel pickup and HUD bar refill



(c) Blinking low fuel warning indicator on the HUD

Figure 10: Fuel system: adaptive fuel can placement, world-space collection, and HUD feedback including low fuel indication.

Fuel System Results

In practice, the adaptive spacing function produced fuel cans that became noticeably rarer as difficulty and run time increased, forcing me to drive more efficiently while still avoiding unwinnable "fuel starvation" scenarios. The continuous consumption model, coupled with nitro's triple burn, gave a clear risk–reward trade-off, and every successful can pickup immediately refilled the HUD bar and cancelled the low-fuel warning, confirming the correctness of the world-space collision logic and the HUD fuel indicator.

Feature: Nitro Boost System

Relevant classes and functions: `NitroSystem`, `NitroSystem::update()`, `NitroSystem::applyThrust()`, `NitroSystem::drawHUD()`, `NitroSystem::drawFlame()`, `MainWindow::gameLoop()`, `Wheel::simulate()`.

In my implementation, the nitro boost is modelled as a short-lived state machine that directly injects additional thrust into the wheels while a dedicated key is held, with activation gated by the current fuel level. At any physical time t (measured in seconds of game time), the nitro subsystem is described by the state

$$S_{\text{nitro}}(t) = \big(\texttt{active}(t),\, t_{\text{end}},\, t_{\text{cool}},\, y_{\text{ceil}}\big),$$

where $\texttt{active}(t) \in \{0, 1\}$ encodes whether nitro is currently firing, t_{end} is the scheduled shutoff time for the current burst, t_{cool} is the earliest time at which a new burst may be triggered (cooldown), and y_{ceil} is a vertical altitude ceiling for the wheels expressed in world pixels. This state lives inside a dedicated `NitroSystem` object and is updated once per frame from `MainWindow::gameLoop()` using the current input, fuel value and local terrain geometry.

The global fuel level is maintained as a scalar $f(t)$ in `m_fuel`, with invariant

$$0 \;\le\; f(t) \;\le\; F_{\max} = \texttt{Constants::FUEL\_MAX}.$$

Nitro can only be activated while $f(t) > 0$, and all bursts are forcibly terminated once fuel reaches zero. Thus nitro behaves as a higher-intensity consumer of the same fuel resource rather than a separate energy pool.

Activation and cooldown logic. Let $k(t) \in \{0, 1\}$ denote the nitro key state passed into `NitroSystem::update()` as `nitroKey`, and let $f(t)$ be the current fuel level. Conceptually, I implement the following piecewise rule for the `active` flag:

$$\texttt{active}(t^{+}) = \begin{cases} 1, & \text{if } \texttt{active}(t^{-}) = 0,\ k(t) = 1,\ f(t) > 0,\ t \ge t_{\text{cool}}, \\ 0, & \text{if } \texttt{active}(t^{-}) = 1 \text{ and } \big(k(t) = 0 \ \vee\ f(t) \le 0 \ \vee\ t \ge t_{\text{end}}\big), \\ \texttt{active}(t^{-}), & \text{otherwise.} \end{cases}$$

When a new nitro burst is started at time t, I schedule its end time and leave cooldown unset for the moment:

$$t_{\text{end}} = t + T_{\text{burst}}, \qquad T_{\text{burst}} = \texttt{Constants::NITRO\_DURATION\_SECOND}.$$

Once the burst ends (because the key is released, fuel hits zero, or the timer expires), nitro is switched off and a cooldown window is scheduled:

$$t_{\text{cool}} = t + T_{\text{cool}}, \qquad T_{\text{cool}} \approx 2.0 \text{ s}.$$

During $[t, t_{\text{cool}})$, even if the player presses the nitro key and still has fuel, the `update()` function will refuse reactivation.

Altitude ceiling from terrain height. At the start of a burst, I also compute a maximum "hop" height relative to the ground under the car's average x coordinate. Let x_{avg} be the average of the two wheel x positions and $s_{\text{px}} = \texttt{PIXEL\_SIZE}$ be the world-to-grid scale. I convert to a terrain grid column

$$g_x = \left\lfloor \frac{x_{\text{avg}}}{s_{\text{px}}} \right\rfloor,$$

and query the corresponding ground row via a function object `groundGyNearestGX` supplied by the main window:

$$g_y^{\text{ground}} = \texttt{groundGyNearestGX}(g_x).$$

With a maximum allowed vertical offset $H_{\max} = \texttt{Constants::NITRO\_MAX\_ALT\_CELLS}$ (in grid cells), I define the ceiling grid row

$$g_y^{\text{ceil}} = g_y^{\text{ground}} - H_{\max}, \qquad y_{\text{ceil}} = g_y^{\text{ceil}}\, s_{\text{px}}.$$

In screen coordinates (where smaller y means visually "higher"), the constraint $y \geq y_{\text{ceil}}$ means that neither wheel is allowed to move above this ceiling; any such motion is immediately clamped in `applyThrust()`, preventing nitro from flinging the car indefinitely upwards.

Boost direction from terrain tangent. The direction of the nitro thrust is derived from the local terrain slope. From `MainWindow::gameLoop()` I pass a function `terrainTangentAngleAtX(wx)` which returns the tangent angle of the procedural terrain at a world x position w_x. At activation time, I sample it at the average x:

$$\theta_{\text{terr}} = \texttt{terrainTangentAngleAtX}(x_{\text{avg}}),$$

and then add a fixed launch angle offset

$$\theta_{\text{boost}} = \theta_{\text{terr}} + \texttt{Constants::NITRO\_LAUNCH\_ANGLE\_RADIANS}.$$

The unit direction vector used to accelerate the wheels is

$$\mathbf{u} = (u_x, u_y) = (\cos\theta_{\text{boost}},\ -\sin\theta_{\text{boost}}),$$

where the minus sign on the sine term compensates for the inverted y axis in screen coordinates (larger y is visually "downwards").

Thrust application and altitude clamping on wheels. Actual velocity changes are applied to the physical wheels in `NitroSystem::applyThrust()`. Let the back and front wheels have positions (x_b, y_b)

and (x_f, y_f) and velocities $(v_{x,b}, v_{y,b})$ and $(v_{x,f}, v_{y,f})$. First, I compute the chassis direction based on the two wheels:

$$\mathbf{d} = (d_x, d_y) = (x_f - x_b,\ y_b - y_f), \qquad \mathbf{u}_{\text{chassis}} = \frac{\mathbf{d}}{\|\mathbf{d}\|},$$

again using $y_b - y_f$ to compensate for screen coordinates. In this implementation, the instantaneous boost is applied along $\mathbf{u}_{\text{chassis}}$ with a fixed magnitude $K_{\text{nitro}} = \texttt{Constants::NITRO\_THRUST}$:

$$\mathbf{v}_w(t^+) = \mathbf{v}_w(t^-) + K_{\text{nitro}}\,\mathbf{u}_{\text{chassis}}$$

for each wheel $w \in \{\text{back}, \text{front}\}$. After this update, I enforce the altitude ceiling by clamping

$$y_w(t^+) \leftarrow \max(y_w(t^+),\, y_{\text{ceil}}),$$

and if a wheel has been pushed above the ceiling ($y_w < y_{\text{ceil}}$) I reset its vertical velocity component to zero to avoid numerical tunnelling through the ceiling.

Exhaust flame rendering behind the rear wheel. The visual nitro flame is rendered in `NitroSystem::drawFlame()` as a small pixel-art plume attached to the rear wheel. I first project the back wheel centre and radius into screen grid coordinates using the existing helper:

$$(c_x, c_y, r_{\text{px}}) = \texttt{back->get}(\ldots), \qquad g_{cx} = \frac{c_x}{s_{\text{px}}}, \quad g_{cy} = \frac{c_y}{s_{\text{px}}},$$

and derive a nozzle position by stepping a few cells backwards along the chassis direction. With $\mathbf{u}_{\text{chassis}}$ as above and an offset in grid cells

$$N_{\text{off}} = \left\lfloor \frac{r_{\text{px}}}{s_{\text{px}}} \right\rfloor + 2,$$

the nozzle grid coordinates are

$$g_x^{\text{nozzle}} = g_{cx} - \lfloor u_x\, N_{\text{off}} \rfloor, \qquad g_y^{\text{nozzle}} = g_{cy} - \lfloor u_y\, N_{\text{off}} \rfloor.$$

Around this nozzle point I plot a small cross-shaped cluster of pixels using three colours (outer orange, mid yellow, inner white) via a lambda

$$\texttt{plot}(g_x, g_y) \rightarrow \texttt{fillRect}(g_x s_{\text{px}}, g_y s_{\text{px}}, s_{\text{px}}, s_{\text{px}}),$$

which yields a stylised exhaust flame that rotates with the car as it climbs and descends hills.

HUD nitro countdown and fuel gating. The nitro HUD is drawn by `NitroSystem::drawHUD()`, which renders a small rocket icon plus a numeric countdown in the same grid-based style as the rest of the UI. The icon is anchored at a base grid coordinate

$$g_x^{\text{base}} = \texttt{HUD\_LEFT\_MARGIN}, \qquad g_y^{\text{base}} = \texttt{HUD\_TOP\_MARGIN} + 2\,\texttt{COIN\_RADIUS\_CELLS} + 4,$$

and implemented as a fixed 5×5 pattern of coloured cells (hull, window, tip, flame and shadow).

The text next to the rocket shows a remaining time value $\tau(t)$ computed from the nitro state:

$$\tau(t) = \begin{cases} \max(0,\, t_{\text{end}} - t), & \text{if } \texttt{active}(t) = 1, \\ \max(0,\, t_{\text{cool}} - t), & \text{if } \texttt{active}(t) = 0 \text{ and } t < t_{\text{cool}}, \\ 0, & \text{otherwise.} \end{cases}$$

I convert this to an integer via $\lceil \tau(t) \rceil$ and draw it as a bold monospace number to the right of the rocket sprite. Since activation is gated by $f(t) > 0$, the combination of the on-screen countdown and the fuel check makes nitro availability transparent to the player.

Core code snippets. The following excerpts show the core of the nitro system and its integration with the main update and rendering loop.

nitro.h

```
#ifndef NITRO_H
#define NITRO_H

#include <QPainter>
#include <QList>
#include <QHash>
#include <functional>
#include <cmath>

#include "constants.h"
#include "wheel.h"

class NitroSystem {
public:
    bool   active        = false;
    double endTime       = 0.0;
    double cooldownUntil = 0.0;
    double dirX          = 0.0;
    double dirY          = -1.0;
    int    ceilY         = -1000000000;

    void reset() {
        active        = false;
        endTime       = 0.0;
        cooldownUntil = 0.0;
        dirX          = 0.0;
        dirY          = -1.0;
        ceilY         = -1000000000;
    }

    void update(bool nitroKey,
                double fuel,
                double elapsedSeconds,
                double avgX,
                const std::function<int(int)>& groundGyNearestGX,
                const std::function<double(double)>& terrainTangentAngleAtX);
```

```
37
38      void applyThrust(QList<Wheel*>& wheels) const;
39
40      void drawHUD(QPainter& p,
41                   double elapsedSeconds,
42                   int levelIndex) const;
43
44      void drawFlame(QPainter& p,
45                     const QList<Wheel*>& wheels,
46                     int cameraX,
47                     int cameraY,
48                     int viewW,
49                     int viewH) const;
50  };
51
52  #endif // NITRO_H
```

nitro.cpp

```
1   #include "nitro.h"
2   #include <QtMath>
3   #include <algorithm>
4
5   void NitroSystem::update(bool nitroKey,
6                            double fuel,
7                            double elapsedSeconds,
8                            double avgX,
9                            const std::function<int(int)>& groundGyNearestGX,
10                           const std::function<double(double)>&
                                 terrainTangentAngleAtX)
11  {
12      const bool wantNitro = nitroKey;
13
14      if (!active) {
15          if (wantNitro && fuel > 0.0 && elapsedSeconds >= cooldownUntil) {
16              active  = true;
17              endTime = elapsedSeconds + Constants::NITRO_DURATION_SECOND;
18
19              const double launchAngle =
20                  terrainTangentAngleAtX(avgX)
21                  + Constants::NITRO_LAUNCH_ANGLE_RADIANS;
22              dirX = std::cos(launchAngle);
23              dirY = -std::sin(launchAngle);
24
25              const int gx = int(std::round(avgX / Constants::PIXEL_SIZE));
26
27              int gyGround = 0;
28              if (groundGyNearestGX) {
29                  gyGround = groundGyNearestGX(gx);
30              }
31
32              ceilY = (gyGround - Constants::NITRO_MAX_ALT_CELLS)
33                      * Constants::PIXEL_SIZE;
```

```
        }
    } else {
        if (!wantNitro || fuel <= 0.0 || elapsedSeconds >= endTime) {
            active        = false;
            cooldownUntil = elapsedSeconds + 2.0;
        }
    }
}

void NitroSystem::applyThrust(QList<Wheel*>& wheels) const
{
    if (!active) return;
    if (wheels.size() < 2) return;

    Wheel* back  = wheels[0];
    Wheel* front = wheels[1];

    const double dx  = front->x - back->x;
    const double dy  = back->y  - front->y;
    const double len = std::sqrt(dx*dx + dy*dy);
    if (len < 1e-6) return;

    const double ux = dx / len;
    const double uy = dy / len;

    const double thrust = Constants::NITRO_THRUST;

    for (Wheel* w : wheels) {
        w->m_vx += thrust * ux;
        w->m_vy += thrust * uy;

        if (w->y < ceilY) {
            w->y = ceilY;
            if (w->m_vy > 0.0) {
                w->m_vy = 0.0;
            }
        }
    }
}

void NitroSystem::drawHUD(QPainter& p,
                          double elapsedSeconds,
                          int levelIndex) const
{
    const int baseGX = Constants::HUD_LEFT_MARGIN;
    const int baseGY = Constants::HUD_TOP_MARGIN
                       + Constants::COIN_RADIUS_CELLS * 2 + 4;

    QColor hull   (180, 180, 190);
    QColor tip    (230, 230, 240);
    QColor windowC(120, 170, 220);
    QColor flame1 (250, 160,  60);
```

```
    QColor flame2 (255, 210,  90);
    QColor shadow ( 10,  10,  20);

    auto plot = [&](int gxCell, int gyCell, const QColor& c) {
        p.fillRect((baseGX + gxCell) * Constants::PIXEL_SIZE,
                   (baseGY + gyCell) * Constants::PIXEL_SIZE,
                   Constants::PIXEL_SIZE,
                   Constants::PIXEL_SIZE,
                   c);
    };

    // Simple 5x5 rocket sprite
    plot(1,1,hull);    plot(2,1,hull);    plot(3,1,hull);    plot(4,1,hull);
    plot(1,2,hull);    plot(2,2,windowC); plot(3,2,hull);    plot(4,2,hull);
    plot(5,2,tip);
    plot(1,3,hull);    plot(2,3,hull);    plot(3,3,hull);    plot(4,3,hull);
    plot(0,2,flame1);  plot(0,3,flame2);
    plot(2,4,shadow);

    QFont f;
    f.setFamily("Monospace");
    f.setBold(true);
    f.setPointSize(12);
    p.setFont(f);
    p.setPen(Constants::TEXT_COLOR[levelIndex]);

    double tleft = 0.0;
    if (active) {
        tleft = std::max(0.0, endTime - elapsedSeconds);
    } else if (elapsedSeconds < cooldownUntil) {
        tleft = std::max(0.0, cooldownUntil - elapsedSeconds);
    }

    const int pxText = (baseGX + 8) * Constants::PIXEL_SIZE;
    const int pyText = (baseGY + 5) * Constants::PIXEL_SIZE;
    p.drawText(pxText, pyText,
               QString::number(int(std::ceil(tleft))));
}

void NitroSystem::drawFlame(QPainter& p,
                            const QList<Wheel*>& wheels,
                            int cameraX,
                            int cameraY,
                            int viewW,
                            int viewH) const
{
    if (!active) return;
    if (wheels.size() < 2) return;

    const Wheel* back  = wheels[0];
    const Wheel* front = wheels[1];

```

```
    auto screenInfo = back->get(cameraX, cameraY,
                                viewW, viewH,
                                -cameraX, cameraY);
    if (!screenInfo.has_value()) return;

    const int cx = (*screenInfo)[0];
    const int cy = (*screenInfo)[1];
    const int r  = (*screenInfo)[2];
    if (r <= 0) return;

    const int gcx = cx / Constants::PIXEL_SIZE;
    const int gcy = cy / Constants::PIXEL_SIZE;

    const double dx  = front->x - back->x;
    const double dy  = back->y  - front->y;
    const double len = std::sqrt(dx*dx + dy*dy);
    if (len < 1e-6) return;

    const double ux = dx / len;
    const double uy = dy / len;

    const int offsetCells = r / Constants::PIXEL_SIZE + 2;
    const int nozzleGX =
        gcx - int(std::round(ux * offsetCells));
    const int nozzleGY =
        gcy - int(std::round(uy * offsetCells));

    QColor cOuter(255,160,60);
    QColor cMid  (255,210,90);
    QColor cCore (255,255,255);

    auto plot = [&](int gx, int gy, const QColor& c) {
        p.fillRect(gx * Constants::PIXEL_SIZE,
                   gy * Constants::PIXEL_SIZE,
                   Constants::PIXEL_SIZE,
                   Constants::PIXEL_SIZE,
                   c);
    };

    plot(nozzleGX - 1, nozzleGY,     cOuter);
    plot(nozzleGX - 2, nozzleGY,     cOuter);
    plot(nozzleGX - 2, nozzleGY + 1, cMid);
    plot(nozzleGX - 3, nozzleGY,     cCore);
}
```

wheel.h (excerpt: nitro-aware simulation signature)

```
#ifndef WHEEL_H
#define WHEEL_H

#include <QPainter>
#include <QVector>
#include <optional>
```

```
#include <array>
#include "constants.h"

class Wheel {
public:
    double x = 0.0;
    double y = 0.0;
    double m_vx = 0.0;
    double m_vy = 0.0;
    double radius = 0.0;
    bool   onGround = false;

    Wheel(double cx, double cy, double r);

    void simulate(int level_index,
                  const QVector<QLine>& groundLines,
                  bool accelerating,
                  bool braking,
                  bool nitro);

    void draw(QPainter& p,
              int cameraX,
              int cameraY,
              int gridW,
              int gridH) const;

    std::optional<std::array<int,3>> get(int cameraX,
                                         int cameraY,
                                         int viewW,
                                         int viewH,
                                         int offsetX,
                                         int offsetY) const;
};

#endif // WHEEL_H
```

mainwindow.cpp (nitro integration excerpts)

```
void MainWindow::gameLoop() {
    ...
    // Average wheel X used as sample point for nitro terrain queries
    double avgX = 0.0;
    if (m_wheels.size() >= 2) {
        avgX = 0.5 * (m_wheels[0]->x + m_wheels[1]->x);
    }

    m_nitroSys.update(
        m_nitroKey,
        m_fuel,
        m_elapsedSeconds,
        avgX,
        [this](int gx) {
            return this->groundGyNearestGX(gx);
```

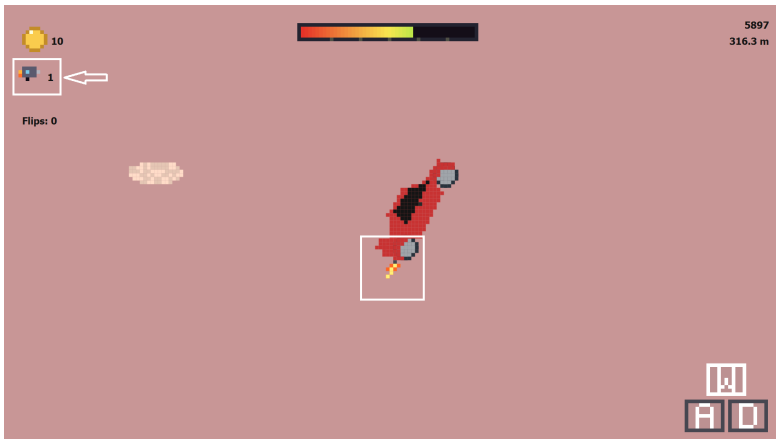
```
        },
        [this](double wx) {
            return this->terrainTangentAngleAtX(wx);
        }
    );

    for (Wheel* w : m_wheels) {
        w->simulate(level_index,
                    m_lines,
                    accelDrive,
                    brakeDrive,
                    m_nitroKey);
    }

    // Inject nitro thrust on top of normal wheel dynamics
    m_nitroSys.applyThrust(m_wheels);
    ...
}

void MainWindow::paintEvent(QPaintEvent* /*ev*/) {
    QPainter p(this);
    ...
    // World-space rendering (terrain, car, wheels, etc.)
    ...
    // Nitro exhaust and HUD overlay
    m_nitroSys.drawFlame(p,
                         m_wheels,
                         m_cameraX,
                         m_cameraY,
                         width(),
                         height());

    m_nitroSys.drawHUD(p,
                       m_elapsedSeconds,
                       level_index);
}
```

Nitro System Results

In practice, the nitro subsystem produced short, controlled bursts of acceleration that let the car clear steep hills and gaps without destabilising the lightweight physics. The altitude ceiling and the fuel-gated activation meant players could not "fly" indefinitely; instead, they had to choose their boost windows carefully based on remaining fuel. The combination of the animated exhaust plume and the rocket-style countdown HUD made nitro availability and remaining burst time immediately readable during gameplay, so the system felt responsive and intentional rather than arbitrary.

Feature: Camera Stabilisation and Smooth Follow

Relevant classes and functions: `MainWindow`, `MainWindow::gameLoop()`, `MainWindow::updateCamera()`, `CarBody`, `Wheel`.



(a) Nitro exhaust flame and HUD countdown indicator

Figure 11: Nitro boost system: world-space flame rendering attached to the rear wheel and rocket-style HUD countdown for burst and cooldown timing.

In my implementation, the world camera is modelled as a damped second–order system that tracks a moving target attached to the car body. Instead of snapping the viewport directly to the car position, I maintain a continuous camera state

$$C_{\text{cam}}(t) = (x_{\text{cam}}(t),\, y_{\text{cam}}(t),\, v_x(t),\, v_y(t)),$$

where $(x_{\text{cam}}, y_{\text{cam}})$ is the camera centre in world pixels and (v_x, v_y) is the camera velocity. This state is updated once per frame to follow a target point $(x_{\text{tgt}}, y_{\text{tgt}})$ derived from the car position, with overshoot and oscillations controlled by a natural frequency ω_n and damping ratio ζ.

Target point in world and screen space. Each frame, I first compute an average wheel position

$$x_{\text{avg}} = \frac{1}{N}\sum_{i=1}^{N} x_i, \qquad y_{\text{avg}} = \frac{1}{N}\sum_{i=1}^{N} y_i,$$

over all wheels (with $N = \texttt{m\_wheels.size()}$). For the actual camera anchor, I prefer the car body centre when available:

$$x_{\text{body}} = \begin{cases} \texttt{m\_bodies.first()->getX()}, & \text{if body exists,} \\ x_{\text{avg}}, & \text{otherwise,} \end{cases} \qquad y_{\text{body}} = \begin{cases} \texttt{m\_bodies.first()->getY()}, & \text{if body exist} \\ y_{\text{avg}}, & \text{otherwise.} \end{cases}$$

To give the player a better look ahead, I offset the camera target a fixed horizontal amount behind the car and vertically centre the car on the screen. With window height $H = \texttt{height()}$ I define

$$x_{\text{tgt}} = x_{\text{body}} - \Delta_x, \qquad y_{\text{tgt}} = -y_{\text{body}} + \frac{H}{2},$$

where $\Delta_x = 200$ pixels in my current tuning, and the minus sign on y_{body} compensates for the inverted screen y axis (larger y is "downwards" on screen).

Second-order smoothing model. To avoid jitter when the car bounces over rough terrain, I do not set $(x_{\text{cam}}, y_{\text{cam}})$ directly to $(x_{\text{tgt}}, y_{\text{tgt}})$. Instead, I approximate a critically damped mass–spring system per axis. For one dimension (say x), the continuous-time model is

$$\ddot{x}_{\text{cam}}(t) + 2\zeta\omega_n \dot{x}_{\text{cam}}(t) + \omega_n^2(x_{\text{cam}}(t) - x_{\text{tgt}}(t)) = 0,$$

with natural frequency $\omega_n = \texttt{m\_camWN}$ and damping ratio $\zeta = \texttt{m\_camZeta}$. In my implementation I use

$$\omega_n = 20.0, \qquad \zeta = 0.98,$$

which yields a fast, slightly over-damped response: the camera catches up quickly but without visible oscillations.

Given the current state (x_{cam}, v_x) and target x_{tgt}, I compute the instantaneous acceleration as

$$a_x = \omega_n^2\,(x_{\text{tgt}} - x_{\text{cam}}) - 2\zeta\omega_n\, v_x,$$

and analogously for y:

$$a_y = \omega_n^2\,(y_{\text{tgt}} - y_{\text{cam}}) - 2\zeta\omega_n\, v_y.$$

Time-stepping and numerical stability. The game loop passes a raw frame time step Δt_{raw} into `updateCamera()`, which I clamp to a safe range to avoid numerical instability:

$$\Delta t = \text{clamp}(\Delta t_{\text{raw}},\, \Delta t_{\min},\, \Delta t_{\max}), \qquad \Delta t_{\min} = 10^{-4}, \quad \Delta t_{\max} = 0.05.$$

I then integrate the second-order system using semi-implicit Euler:

$$v_x(t + \Delta t) = v_x(t) + a_x(t)\,\Delta t,$$
$$x_{\text{cam}}(t + \Delta t) = x_{\text{cam}}(t) + v_x(t + \Delta t)\,\Delta t,$$

and similarly for y. This scheme is inexpensive and, combined with the fixed (ω_n, ζ) and Δt clamp, keeps the camera motion smooth even under fluctuating frame rates.

Once the continuous camera centre $(x_{\text{cam}}, y_{\text{cam}})$ has been updated, I quantise it to integer pixels for use as the draw offset:

$$\texttt{m\_cameraX} = \lfloor x_{\text{cam}} + 0.5 \rfloor, \qquad \texttt{m\_cameraY} = \lfloor y_{\text{cam}} + 0.5 \rfloor.$$

All world elements (terrain, car body, wheels, coins, fuel, etc.) are then rendered relative to $(\texttt{m\_cameraX}, \texttt{m\_cameraY})$, so the entire scene inherits this stabilised camera motion.

Core code snippets. The following excerpts show the camera state, its integration in the main loop, and the stabilisation update step.

mainwindow.h (camera state)

```
class MainWindow : public QWidget
{
    Q_OBJECT
```

```
    ...
    // Smoothed camera in world space
    double m_camX   = 0.0;
    double m_camY   = 0.0;
    double m_camVX  = 0.0;
    double m_camVY  = 0.0;
    double m_camWN  = 20.0;   // Natural frequency
    double m_camZeta= 0.98;   // Damping ratio

    // Integer camera used for rendering offsets
    int m_cameraX   = 0;
    int m_cameraY   = 0;
    ...
    void updateCamera(double targetX, double targetY, double rawDt);
    ...
};
```

mainwindow.cpp (camera target in gameLoop() and quantisation)

```
void MainWindow::gameLoop() {
    ...
    double avgX = 0.0, avgY = 0.0;
    if (!m_wheels.isEmpty()) {
        for (Wheel* w : m_wheels) {
            avgX += w->x;
            avgY += w->y;
        }
        avgX /= m_wheels.size();
        avgY /= m_wheels.size();
    }

    ...
    double bodyX = (!m_bodies.isEmpty()) ? m_bodies.first()->getX() : avgX;
    double bodyY = (!m_bodies.isEmpty()) ? m_bodies.first()->getY() : avgY;

    const double targetX = bodyX - 200.0;
    const double targetY = -bodyY + height() / 2.0;

    updateCamera(targetX, targetY, dt);

    m_cameraX = int(std::lround(m_camX));
    m_cameraY = int(std::lround(m_camY));
    ...
}
```

mainwindow.cpp (camera stabilisation update)

```
void MainWindow::updateCamera(double targetX, double targetY, double rawDt) {
    const double dt = std::clamp(rawDt, 1e-4, 0.05);

    const double wn2       = m_camWN * m_camWN;
    const double twoZetaWN = 2.0 * m_camZeta * m_camWN;

    const double ax = wn2 * (targetX - m_camX) - twoZetaWN * m_camVX;
```

```
    const double ay = wn2 * (targetY - m_camY) - twoZetaWN * m_camVY;

    m_camVX += ax * dt;
    m_camVY += ay * dt;

    m_camX  += m_camVX * dt;
    m_camY  += m_camVY * dt;
}
```

Camera Stabilisation Results

In practice, this damped second–order camera produced a smooth, "weighty" follow behaviour that filters out high-frequency jitter from the wheels while still keeping the car centred and readable. The chosen (ω_n, ζ) parameters let the view catch up quickly after jumps and crashes without overshooting or oscillating, and the horizontal look-ahead offset gave players extra reaction time when approaching steep slopes and gaps. Overall, the stabilised camera significantly improved comfort and perceived polish compared to a naïve, instant-follow implementation.

Feature: Aerial Flip Detection and Reward System

Relevant classes and functions: `FlipTracker`, `MainWindow::gameLoop()`, `MainWindow::paintEvent()`.

In my implementation, I developed a dedicated system to track the car's rotational history to detect and reward 360-degree aerial flips. Unlike simple state checks (e.g., checking if the car is upside down), detecting a flip requires integrating the angular velocity over time to distinguish between a full rotation and a simple oscillation. This logic is encapsulated in the `FlipTracker` class.

Angular Integration and Normalization. The core challenge in tracking rotation is the discontinuity of angles at $\pm\pi$. The physics engine provides the car's current angle $\theta_t \in (-\pi, \pi]$. Simply subtracting the current angle from the previous angle θ_{t-1} can yield incorrect large deltas when the angle crosses the cut line (e.g., moving from $\pi - \epsilon$ to $-\pi + \epsilon$).

To solve this, I calculate the raw difference $\delta_{\text{raw}} = \theta_t - \theta_{t-1}$ and normalize it into the principal interval $(-\pi, \pi]$:

$$\Delta\theta = \begin{cases} \delta_{\text{raw}} - 2\pi, & \text{if } \delta_{\text{raw}} > \pi \\ \delta_{\text{raw}} + 2\pi, & \text{if } \delta_{\text{raw}} \leq -\pi \\ \delta_{\text{raw}}, & \text{otherwise} \end{cases}$$

This normalized $\Delta\theta$ represents the true physical rotation magnitude and direction that occurred during the frame Δt. I maintain a continuous accumulator Φ_{accum}:

$$\Phi_{\text{accum}}(t) = \Phi_{\text{accum}}(t-1) + \Delta\theta$$

Flip Detection Logic. A complete flip is defined as a net rotation of 2π radians. The system checks Φ_{accum} against this threshold every frame:

- **Counter-Clockwise (Backflip):** If $\Phi_{\text{accum}} \geq 2\pi$, a flip is recorded, the flip count N_{ccw} is incremented, and 2π is subtracted from the accumulator to allow detection of subsequent flips (double flips).

- **Clockwise (Frontflip):** If $\Phi_{\text{accum}} \leq -2\pi$, a flip is recorded, N_{cw} is incremented, and 2π is added to the accumulator.

Upon detecting a flip, the system triggers a callback to the `MainWindow` to award coins and spawns a floating "Flip!" popup text at the car's world coordinates $(x_{\text{car}}, y_{\text{car}})$.

Visual Feedback. To provide immediate feedback, I implemented a lightweight particle system within `FlipTracker`. When a flip occurs, a `Popup` struct is created containing the world location and an expiration timestamp:

$$T_{\text{expire}} = t_{\text{now}} + 1.5\text{s}$$

During the render pass, these popups are drawn by manually plotting pixels to form the word "Flip!", creating a retro aesthetic consistent with the grid-based terrain.

Source Code. Below is the complete implementation of the flip tracking module and the relevant integration points in the main game loop.

flip.h

```
#pragma once
#include <QPainter>
#include <QList>
#include <QString>
#include <functional>
#include "constants.h"

class FlipTracker {
public:
    void reset() {
        m_init = false;
        m_lastAng = 0.0;
        m_accum = 0.0;
        m_cw = m_ccw = 0;
        m_popups.clear();
    }

    void update(double angleRad, double carX, double carY, double nowSec, const
        std::function<void(int)>& onAward);
    void drawHUD(QPainter& p, int levelIndex) const;
    void drawWorldPopups(QPainter& p, int cameraX, int cameraY, int level_index)
        const;

    int total() const { return m_cw + m_ccw; }
    int cw()    const { return m_cw; }
    int ccw()   const { return m_ccw; }

private:
    static constexpr double TWO_PI = 6.283185307179586;
    static constexpr int    COINS_PER_FLIP     = 50;
    static constexpr double POPUP_LIFETIME     = 1.5; // seconds
    static constexpr int    POPUP_OFFSET_CELLS = 10;

```

```
32      struct Popup {
33          int wx, wy;
34          double until;
35      };
36
37      static void drawPixelWordFlip(QPainter& p, int gx, int gy, int cell, const
            QColor& c);
38
39      bool   m_init = false;
40      double m_lastAng = 0.0;
41      double m_accum   = 0.0;
42      int    m_cw = 0, m_ccw = 0;
43      QList<Popup> m_popups;
44  };
```

flip.cpp

```
1   #include "flip.h"
2   #include <QtMath>
3   #include <algorithm>
4
5   static inline int gx_from_px(int px){
6       return px / Constants::PIXEL_SIZE;
7   }
8
9   static inline int gy_from_px(int py){
10      return py / Constants::PIXEL_SIZE;
11  }
12
13  void FlipTracker::update(double angleRad, double carX, double carY, double nowSec,
        const std::function<void(int)>& onAward)
14  {
15      if (!m_init) {
16          m_init = true;
17          m_lastAng = angleRad;
18          m_accum = 0.0;
19          return;
20      }
21
22      // Normalize angle difference to (-PI, PI]
23      double d = angleRad - m_lastAng;
24      while (d >  M_PI) d -= TWO_PI;
25      while (d <= -M_PI) d += TWO_PI;
26
27      m_accum += d;
28      m_lastAng = angleRad;
29
30      // Check for CCW Flip (Backflip)
31      while (m_accum >= TWO_PI) {
32          ++m_ccw; m_accum -= TWO_PI;
33          if (onAward) onAward(COINS_PER_FLIP);
34          m_popups.append({int(carX), int(carY - POPUP_OFFSET_CELLS * Constants::
                PIXEL_SIZE),
```

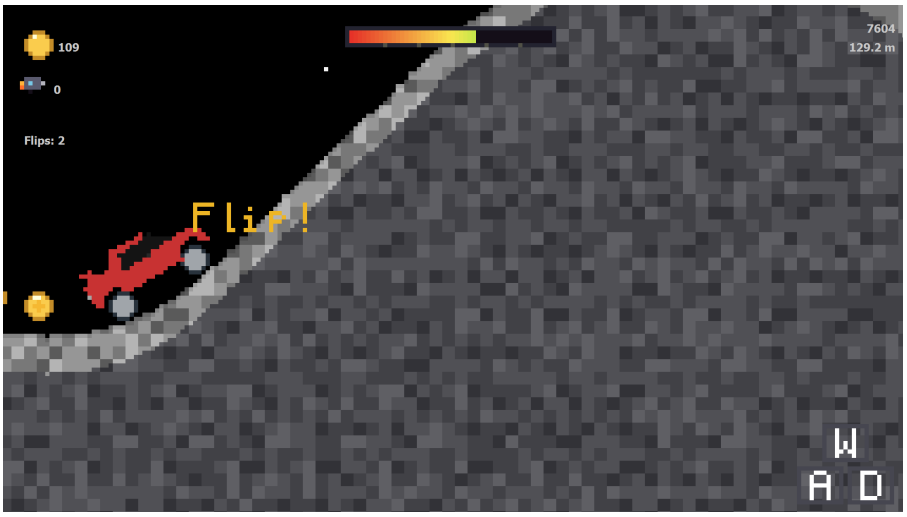
```
35                          nowSec + POPUP_LIFETIME});
36      }
37      // Check for CW Flip (Frontflip)
38      while (m_accum <= -TWO_PI) {
39          ++m_cw; m_accum += TWO_PI;
40          if (onAward) onAward(COINS_PER_FLIP);
41          m_popups.append({int(carX), int(carY - POPUP_OFFSET_CELLS * Constants::
                PIXEL_SIZE),
42                          nowSec + POPUP_LIFETIME});
43      }
44
45      // Remove expired popups
46      for (int i = 0; i < m_popups.size();) {
47          if (m_popups[i].until <= nowSec) m_popups.removeAt(i);
48          else ++i;
49      }
50  }
51
52  void FlipTracker::drawHUD(QPainter& p, int levelIndex) const
53  {
54      const int textGX = Constants::HUD_LEFT_MARGIN + 1;
55      const int nitroBaselineGY = Constants::HUD_TOP_MARGIN + Constants::
            COIN_RADIUS_CELLS*2 + 4;
56      const int extraGapCells = 10;
57      const int textGY = nitroBaselineGY + 7 + extraGapCells;
58      QFont f; f.setFamily("Monospace"); f.setBold(true); f.setPointSize(12);
59      p.setFont(f);
60      p.setPen(Constants::TEXT_COLOR[levelIndex]);
61      p.drawText(textGX * Constants::PIXEL_SIZE,
62                 textGY  * Constants::PIXEL_SIZE,
63                 QString("Flips: %1").arg(total()));
64  }
65
66  void FlipTracker::drawPixelWordFlip(QPainter& p, int gx, int gy, int cell, const
        QColor& c)
67  {
68      auto plot=[&](int x,int y){ p.fillRect((gx+x)*cell,(gy+y)*cell,cell,cell,c);
            };
69
70      // Bitmaps for letters F, l, i, p, !
71      static const uint8_t F[7]={0x1F,0x10,0x1E,0x10,0x10,0x10,0x10};
72      static const uint8_t l[7]={0x04,0x04,0x04,0x04,0x04,0x04,0x06};
73      static const uint8_t i[7]={0x00,0x08,0x00,0x18,0x08,0x08,0x1C};
74      static const uint8_t glyphP[7]={0x00,0x00,0x1C,0x12,0x1C,0x10,0x10};
75      static const uint8_t ex[7]={0x04,0x04,0x04,0x04,0x04,0x00,0x04};
76
77      auto drawChar=[&](const uint8_t rows[7], int ox){
78          for(int ry=0; ry<7; ++ry){
79              uint8_t row=rows[ry];
80              for(int rx=0; rx<5; ++rx)
81                  if (row & (1<<(4-rx))) plot(ox+rx, ry);
82          }
```

```
    };

    int adv=6;
    drawChar(F, 0);
    drawChar(l, adv*1);
    drawChar(i, adv*2);
    drawChar(glyphP, adv*3);
    drawChar(ex,adv*4);
}

void FlipTracker::drawWorldPopups(QPainter& p, int cameraX, int cameraY, int
    level_index) const
{
    const int cell = Constants::PIXEL_SIZE;
    const int screenPadCells = 10;

    for (const auto& pop : m_popups) {
        const int gx = gx_from_px(pop.wx - cameraX) + screenPadCells;
        const int gy = gy_from_px(pop.wy + cameraY);
        drawPixelWordFlip(p, gx, gy, cell, Constants::FLIP_COLOR[level_index]);
    }
}
```

mainwindow.cpp (Integration Excerpt)

```
void MainWindow::gameLoop() {
    ...
    double angleRad = 0.0;
    if (m_wheels.size() >= 2) {
        const double dx = (m_wheels[1]->x - m_wheels[0]->x);
        const double dy = (m_wheels[1]->y - m_wheels[0]->y);
        angleRad = std::atan2(dy, dx);
    }
    double carX = (!m_bodies.isEmpty()) ? m_bodies.first()->getX() : avgX;
    double carY = (!m_bodies.isEmpty()) ? m_bodies.first()->getY() : avgY;

    // Update tracker with current car angle and position
    m_flip.update(angleRad, carX, carY, m_elapsedSeconds, [this](int bonus){
        m_coinCount += bonus;
    });
    ...
}

void MainWindow::paintEvent(QPaintEvent *event) {
    ...
    p.save();
    p.translate(offX, offY);

    // Draw world objects
    ...
    m_flip.drawWorldPopups(p, m_cameraX, m_cameraY, level_index);

    p.restore();
```

```
29
30      // Draw HUD elements
31      ...
32      m_flip.drawHUD(p, level_index);
33      ...
34  }
```



(a) World-space text popup triggered by a $360°$ rotation

Figure 12: Aerial Flip Detection: Visual feedback showing the generated "Flip!" pixel-art text spawned relative to the car's position and the corresponding increment on the HUD counter.

Results

The `FlipTracker` successfully rewards skilled play by detecting loops in the car's trajectory. The normalization logic ensures that crossing the $180°$ boundary does not reset or glitch the flip counter, and the pixel-art "Flip!" popup provides satisfying visual confirmation of the stunt without requiring complex texture assets.

Feature: Real-time Input Visualization (KeyLog) and HUD

Relevant classes and functions: `KeyLog`, `MainWindow::paintEvent()`, `MainWindow::keyPressEvent()`, `MainWindow::keyReleaseEvent()`.

To provide immediate visual feedback on the player's control inputs—a feature commonly seen in speed-running tools and replays—I implemented a modular `KeyLog` class. This system overlays a pixel-art representation of the directional keys (W, A, D) onto the screen, lighting them up dynamically as the user presses the corresponding buttons.

Grid-based Layout and Positioning. Since the game renders in a low-resolution pixel grid scaled up by a factor P_{size} (typically 4 or 5 screen pixels per game pixel), the input overlay must align perfectly with this grid. I calculate the grid dimensions as:

$$W_{\text{grid}} = \lfloor W_{\text{screen}} / P_{\text{size}} \rfloor, \qquad H_{\text{grid}} = \lfloor H_{\text{screen}} / P_{\text{size}} \rfloor.$$

The keys are anchored to the bottom-right corner of the screen to avoid obstructing the gameplay view. If a key cluster has dimensions $w_{\text{pack}} \times h_{\text{pack}}$ (in grid cells) and a margin m, the top-left origin (x_0, y_0) of the cluster is calculated as:

$$x_0 = W_{\text{grid}} - m - w_{\text{pack}}, \qquad y_0 = H_{\text{grid}} - m - h_{\text{pack}}.$$

From this anchor, the specific keys ('A', 'W', 'D') are offset relatively. For example, the 'W' key is centered above the 'A' and 'D' keys:

$$x_W = x_0 + \frac{w_{\text{pack}} - w_{\text{key}}}{2}, \qquad y_W = y_0.$$

Bitwise Glyph Rendering. To maintain the retro aesthetic without importing external font textures, I implemented a lightweight bitmapped font renderer within `KeyLog::drawGlyph`. Each character is represented by an array of bytes, where each byte corresponds to a row of pixels.

For a given row r containing the byte value B_r, and a glyph width w_g, the visibility of a pixel at column c is determined by a bitwise check:

$$\text{pixel}(c, r) = \begin{cases} \text{on}, & \text{if } (B_r \gg (w_g - 1 - c))\&1 \\ \text{off}, & \text{otherwise} \end{cases}$$

This allows me to draw crisp, scalable alphanumeric characters directly onto the `QPainter` surface using the game's unified `plot` function.

State Management and Rendering. The visual state (pressed vs. unpressed) is managed via boolean flags updated in the Qt event loop. During the render pass, I select the border color C_{border} based on this state:

$$C_{\text{border}} = \begin{cases} C_{\text{hot}} = (255, 255, 255), & \text{if pressed} \\ C_{\text{dim}} = (70, 70, 80), & \text{otherwise} \end{cases}$$

This provides a high-contrast response that is instantly readable even during fast-paced gameplay.

Core Code Snippets. Below is the complete implementation of the input visualization module and its integration into the main window's event handling.

keylog.h

```
#pragma once
#include <QPainter>
#include <Qt>

class KeyLog {
public:
    void setPressed(int key, bool down) {
        if (key == Qt::Key_W) m_w = down;
        else if (key == Qt::Key_A) m_a = down;
        else if (key == Qt::Key_D) m_d = down;
    }
    void draw(QPainter& p, int screenW, int screenH, int pixelSize);

```

```
14 private:
15     bool m_w = false, m_a = false, m_d = false;
16 
17     static inline void plot(QPainter& p, int gx, int gy, int ps, const QColor& c)
           {
18         p.fillRect(gx*ps, gy*ps, ps, ps, c);
19     }
20     void drawKey(QPainter& p, int gx, int gy, int w, int h, int ps, QChar label,
           bool pressed);
21     void drawGlyph(QPainter& p, int gx, int gy, int w, int h, int ps, QChar ch,
           const QColor& color);
22 };
```

keylog.cpp

```
1  #include "keylog.h"
2  #include "constants.h"
3  #include <QRect>
4  #include <algorithm>
5  
6  void KeyLog::draw(QPainter& p, int screenW, int screenH, int ps) {
7      const int gridW = screenW / ps;
8      const int gridH = screenH / ps;
9  
10     const int margin = 2;
11     const int gap    = 1;
12     const int keyW   = 11;
13     const int keyH   = 9;
14 
15     const int packW  = 2*keyW + gap;
16     const int packH  = 2*keyH + gap;
17 
18     // Calculate anchor position (Bottom-Right)
19     const int gx0 = gridW - margin - packW;
20     const int gy0 = gridH - margin - packH;
21 
22     // Relative positions for W, A, D
23     const int gxW = gx0 + (packW - keyW)/2;
24     const int gyW = gy0;
25 
26     const int gxA = gx0;
27     const int gyA = gy0 + keyH + gap;
28 
29     const int gxD = gx0 + keyW + gap;
30     const int gyD = gyA;
31 
32     p.save();
33     drawKey(p, gxW, gyW, keyW, keyH, ps, QChar('W'), m_w);
34     drawKey(p, gxA, gyA, keyW, keyH, ps, QChar('A'), m_a);
35     drawKey(p, gxD, gyD, keyW, keyH, ps, QChar('D'), m_d);
36     p.restore();
37 }
38 
```

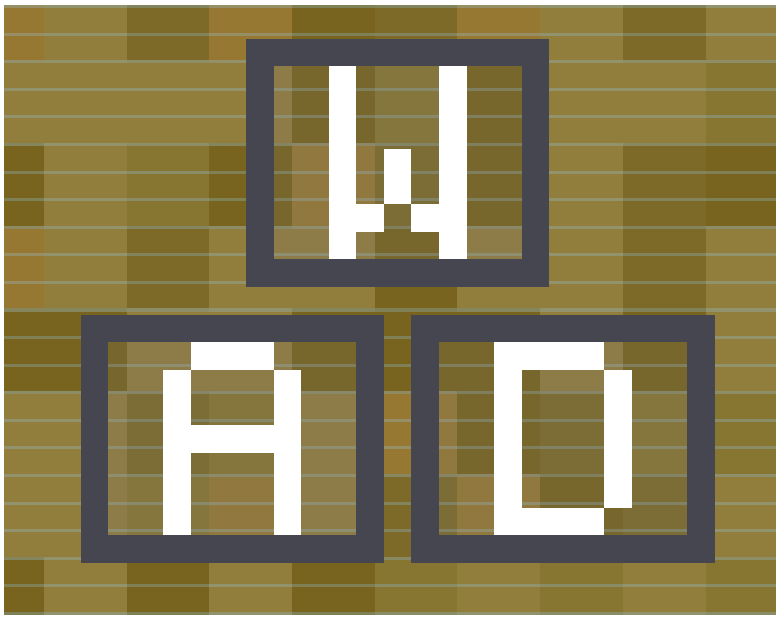
```
39 void KeyLog::drawKey(QPainter& p, int gx, int gy, int w, int h, int ps, QChar
       label, bool pressed) {
40     const QColor fill(120,120,130,40);
41     const QColor borderDim(70,70,80);
42     const QColor borderHot(255,255,255);
43 
44     // Fill background
45     for (int y=0; y<h; ++y)
46         for (int x=0; x<w; ++x)
47             plot(p, gx+x, gy+y, ps, fill);
48 
49     // Draw Border based on state
50     const QColor bc = pressed ? borderHot : borderDim;
51     for (int x=0; x<w; ++x) { plot(p, gx+x, gy+0, ps, bc); plot(p, gx+x, gy+h-1,
           ps, bc); }
52     for (int y=0; y<h; ++y) { plot(p, gx+0, gy+y, ps, bc); plot(p, gx+w-1, gy+y,
           ps, bc); }
53 
54     drawGlyph(p, gx, gy, w, h, ps, label, QColor(255,255,255));
55 }
56 
57 void KeyLog::drawGlyph(QPainter& p, int gx, int gy, int w, int h, int ps, QChar ch
       , const QColor& color) {
58     auto it = font_map.find(ch.toUpper()); // font_map defined in constants.h
59     if (it == font_map.end()) return;
60 
61     const int gw = 5, gh = 7;
62     const int pad = 2;
63     const int innerW = std::max(0, w - 2*pad);
64     const int innerH = std::max(0, h - 2*pad);
65     const int cell = std::max(1, std::min(innerW / gw, innerH / gh));
66     const int ox = gx + pad + (innerW - gw*cell)/2;
67     const int oy = gy + pad + (innerH - gh*cell)/2;
68 
69     const auto& rows = it.value();
70     for (int r=0; r<gh; ++r) {
71         uint8_t row = rows[r];
72         for (int c=0; c<gw; ++c) {
73             // Bitwise check for pixel presence
74             if (row & (1 << (gw-1-c))) {
75                 for (int yy=0; yy<cell; ++yy)
76                     for (int xx=0; xx<cell; ++xx)
77                         plot(p, ox + c*cell + xx, oy + r*cell + yy, ps, color);
78             }
79         }
80     }
81 }
```

mainwindow.cpp (Event Handling and HUD Integration)

```
1 void MainWindow::keyPressEvent(QKeyEvent *event) {
2     if (event->isAutoRepeat()) return;
3 
```

```
4       switch (event->key()) {
5           case Qt::Key_D:
6           case Qt::Key_Right:
7               // ... physics logic ...
8               m_keylog.setPressed(Qt::Key_D, true);
9               break;
10
11          case Qt::Key_A:
12          case Qt::Key_Left:
13              // ... physics logic ...
14              m_keylog.setPressed(Qt::Key_A, true);
15              break;
16
17          case Qt::Key_W:
18          case Qt::Key_Up:
19              // ... physics logic ...
20              m_keylog.setPressed(Qt::Key_W, true);
21              break;
22          // ...
23      }
24  }
25
26  void MainWindow::keyReleaseEvent(QKeyEvent *event) {
27      if (event->isAutoRepeat()) return;
28
29      switch (event->key()) {
30      case Qt::Key_D:
31      case Qt::Key_Right:
32          // ...
33          m_keylog.setPressed(Qt::Key_D, false);
34          break;
35      case Qt::Key_A:
36      case Qt::Key_Left:
37          // ...
38          m_keylog.setPressed(Qt::Key_A, false);
39          break;
40      case Qt::Key_W:
41      case Qt::Key_Up:
42          // ...
43          m_keylog.setPressed(Qt::Key_W, false);
44          break;
45      // ...
46      }
47  }
48
49  void MainWindow::paintEvent(QPaintEvent *event) {
50      // ... (Scene rendering) ...
51
52      // Draw HUD overlay elements
53      drawHUDFuel(p);
54      drawHUDCoins(p);
55      m_nitroSys.drawHUD(p, m_elapsedSeconds, level_index);
```

```
56      drawHUDDistance(p);
57      drawHUDScore(p);
58
59      // Draw Input Visualizer on top
60      m_keylog.draw(p, width(), height(), Constants::PIXEL_SIZE);
61  }
```



(a) Real-time input visualization overlay

Figure 13: KeyLog HUD: Pixel-perfect rendering of the directional keys (W, A, D) anchored to the bottom-right corner, providing immediate visual feedback of the player's control inputs during gameplay.

Results

The `KeyLog` system was successfully integrated, providing a clean, non-intrusive visual indicator of input states. The mathematical scaling ensures it remains pixel-perfect regardless of the window size or screen resolution, maintaining the game's coherent 8-bit aesthetic.

Feature: Procedural Environmental Props and Audio Management

Relevant classes and functions: `PropSystem`, `Media`, `assets.qrc`.

To enhance the visual diversity of the infinite terrain and provide auditory feedback, I implemented two distinct subsystems: a procedural prop generator and a low-latency audio manager.

Programmatic Prop Generation. Rather than relying on external sprite textures, which can cause scaling artifacts in a pixel-grid renderer, I implemented the `PropSystem` to draw environmental elements programmatically using geometric primitives. This ensures that trees, buildings, and rocks align perfectly with the global pixel grid G_{size}.

The spawning logic in `maybeSpawnProp` utilizes a stochastic process constrained by terrain topology.

For a given world coordinate x_w and a generated random variable $r \in [0, 1]$, a prop of type T is spawned if:

$$r < P(\text{spawn}|\text{level}) \quad \text{and} \quad |\text{slope}(x_w)| < \theta_{\text{max}},$$

where θ_{max} is a slope threshold (e.g., 0.15 for camels/igloos) ensuring that structure-like props only appear on relatively flat ground.

For the "Nightlife" level, I implemented a specific procedural generation algorithm for buildings. The system generates skyscrapers with varying widths, heights, and window patterns. To simulate neon lighting without lighting engines, I use modulus arithmetic on the coordinate space to determine window states:

$$\text{Window}(x, y) = \begin{cases} \text{On}, & \text{if } (x \mod S_x = 0) \wedge ((x + y) \mod S_y \neq 0) \\ \text{Off}, & \text{otherwise} \end{cases}$$

This creates complex, scrolling interference patterns that resemble lit windows as the camera moves.

Audio Pooling and Dynamic Mixing. The `Media` class manages the game's soundscape. A significant challenge in fast-paced games is "sound cutting," where a new sound effect (like a coin pickup) terminates the previous one if a single audio channel is reused. To solve this, I implemented an audio pool (Round-Robin scheduling) for high-frequency events.

For coin pickups, I maintain a pool of $N = 3$ media players. When a coin is collected at step k, the player index i is selected as:

$$i_k = (i_{k-1} + 1) \mod N$$

This allows up to three coin sounds to overlap naturally.

Additionally, I implemented a dynamic engine sound loop. To avoid abrupt audio cuts when the player releases the accelerator, I use a `QPropertyAnimation` to interpolate the volume V over time t:

$$V(t) = V_{\text{initial}} \cdot (1 - \frac{t}{T_{\text{fade}}})$$

This provides a smooth audio decay that mimics physical engine spin-down.

Source Code. Below are the complete implementations for the prop generation system, the media manager, and the resource configuration.

prop.h

```
#ifndef PROP_H
#define PROP_H

#include <QVector>
#include <QColor>
#include <random>
#include <QPainter>
#include <QHash>
#include "constants.h"

enum class PropType {
    Tree, Rock, Flower, Mushroom,
```

```
    Cactus, Tumbleweed, Camel,
    Igloo, Penguin, Snowman, IceSpike,
    UFO,
    Rover, Alien,
    Building, StreetLamp
};

struct Prop {
    int wx;
    int wy;
    PropType type;
    int variant;
    bool flipped;
};

class PropSystem {
public:
    PropSystem();

    void maybeSpawnProp(int worldX, int groundGy, int levelIndex, float slope, std::mt19937& rng);
    void draw(QPainter& p, int camX, int camY, int screenW, int screenH, const QHash<int,int>& heightMap);
    void prune(int minWorldX);
    void clear();

private:
    QVector<Prop> m_props;

    void plot(QPainter& p, int gx, int gy, const QColor& c);

    void drawTree(QPainter& p, int gx, int gy, int worldGX, int wx, int wy, int variant, const QHash<int,int>& heightMap);
    void drawRock(QPainter& p, int gx, int gy, int variant);
    void drawFlower(QPainter& p, int gx, int gy, int variant);
    void drawMushroom(QPainter& p, int gx, int gy, int variant);
    void drawCactus(QPainter& p, int gx, int gy, int variant);
    void drawTumbleweed(QPainter& p, int gx, int gy, int variant);
    void drawCamel(QPainter& p, int gx, int gy, int variant, bool flipped);
    void drawIgloo(QPainter& p, int gx, int gy, int worldGX, int variant, const QHash<int,int>& heightMap);
    void drawPenguin(QPainter& p, int gx, int gy, int variant, bool flipped);
    void drawSnowman(QPainter& p, int gx, int gy, int variant);
    void drawIceSpike(QPainter& p, int gx, int gy, int variant);
    void drawUFO(QPainter& p, int gx, int gy, int variant);
    void drawRover(QPainter& p, int gx, int gy, int worldGX, int variant, bool flipped, const QHash<int,int>& heightMap);
    void drawAlien(QPainter& p, int gx, int gy, int variant);

    void drawBuilding(QPainter& p, int gx, int gy, int worldGX, int variant, const QHash<int,int>& heightMap);
    void drawStreetLamp(QPainter& p, int gx, int gy, int worldGX, int variant,
```

```
        const QHash<int,int>& heightMap);
};

#endif
```

prop.cpp

```
#include "prop.h"
#include <cmath>
#include <algorithm>
#include <vector>

PropSystem::PropSystem() {}

void PropSystem::clear() {
    m_props.clear();
}

void PropSystem::prune(int minWorldX) {
    auto it = std::remove_if(m_props.begin(), m_props.end(), [minWorldX](const
        Prop& p){
        return p.wx < minWorldX - 500;
    });
    m_props.erase(it, m_props.end());
}

void PropSystem::maybeSpawnProp(int worldX, int groundGy, int levelIndex, float
    slope, std::mt19937& rng) {
    std::uniform_real_distribution<float> dist(0.0f, 1.0f);
    std::uniform_int_distribution<int> varDist(0, 6);
    std::uniform_int_distribution<int> flipDist(0, 1);

    if (!m_props.isEmpty()) {
        int minSpacing = 120;
        if (levelIndex == 5) minSpacing = 30;

        if (std::abs(worldX - m_props.last().wx) < minSpacing) {
            return;
        }
    }

    float chance = dist(rng);
    int wy = groundGy * Constants::PIXEL_SIZE;

    if (levelIndex == 0) {
        if (chance < 0.03f)      m_props.append({worldX, wy, PropType::Tree,
            varDist(rng), (bool)flipDist(rng)});
        else if (chance < 0.06f) m_props.append({worldX, wy, PropType::Rock,
            varDist(rng), (bool)flipDist(rng)});
        else if (chance < 0.12f) m_props.append({worldX, wy, PropType::Flower,
            varDist(rng), false});
        else if (chance < 0.14f) m_props.append({worldX, wy, PropType::Mushroom,
            varDist(rng), false});
```

```
41      }
42      else if (levelIndex == 1) {
43          if (chance < 0.02f) {
44              if (std::abs(slope) < 0.15f) {
45                  bool camelNearby = false;
46                  for(const auto& p : m_props) {
47                      if (p.type == PropType::Camel && std::abs(p.wx - worldX) <
                            3000) {
48                          camelNearby = true;
49                          break;
50                      }
51                  }
52                  if (!camelNearby) {
53                      m_props.append({worldX, wy, PropType::Camel, varDist(rng), (
                            bool)flipDist(rng)});
54                  }
55              }
56          }
57          else if (chance < 0.06f) {
58              m_props.append({worldX, wy, PropType::Cactus, varDist(rng), (bool)
                    flipDist(rng)});
59          }
60          else if (chance > 0.998f) {
61              bool exists = false;
62              for(const auto& p : m_props) {
63                  if (p.type == PropType::Tumbleweed && std::abs(p.wx - worldX) <
                        1000) {
64                      exists = true;
65                      break;
66                  }
67              }
68              if (!exists) {
69                  m_props.append({worldX, wy, PropType::Tumbleweed, varDist(rng), (
                        bool)flipDist(rng)});
70              }
71          }
72      }
73      else if (levelIndex == 2) {
74          if (chance < 0.018f) {
75              m_props.append({worldX, wy, PropType::Penguin, varDist(rng), (bool)
                    flipDist(rng)});
76          }
77          else if (chance < 0.048f) {
78              if (std::abs(slope) < 0.15f) {
79                  bool iglooNearby = false;
80                  for (const auto& p : m_props) {
81                      if (p.type == PropType::Igloo && std::abs(p.wx - worldX) <
                            2500) {
82                          iglooNearby = true;
83                          break;
84                      }
85                  }
```

```
86                  if (!iglooNearby) {
87                      m_props.append({worldX, wy, PropType::Igloo, varDist(rng),
                            false});
88                  }
89              }
90          }
91          else if (chance < 0.06f) {
92              m_props.append({worldX, wy, PropType::Snowman, varDist(rng), (bool)
                    flipDist(rng)});
93          }
94          else if (chance < 0.08f) {
95              m_props.append({worldX, wy, PropType::IceSpike, varDist(rng), (bool)
                    flipDist(rng)});
96          }
97      }
98      else if (levelIndex == 3) {
99          if (chance < 0.009f) {
100             int liftCells = 50 + (varDist(rng) * 2);
101             int skyWy = wy - (liftCells * Constants::PIXEL_SIZE);
102             m_props.append({worldX, skyWy, PropType::UFO, varDist(rng), false});
103         }
104     }
105     else if (levelIndex == 4) {
106         if (chance < 0.015f) {
107             if (std::abs(slope) < 0.25f) {
108                 m_props.append({worldX, wy, PropType::Rover, varDist(rng), (bool)
                        flipDist(rng)});
109             }
110         }
111         else if (chance > 0.998f) {
112             m_props.append({worldX, wy, PropType::Alien, varDist(rng), false});
113         }
114     }
115     else if (levelIndex == 5) {
116         if (chance < 0.5f) {
117             m_props.append({worldX, wy, PropType::Building, varDist(rng), false});
118         }
119
120         float lampChance = dist(rng);
121         if (lampChance < 0.1f) {
122             m_props.append({worldX + 15, wy, PropType::StreetLamp, varDist(rng),
                    false});
123         }
124     }
125 }
126
127 void PropSystem::draw(QPainter& p, int camX, int camY, int screenW, int screenH,
        const QHash<int,int>& heightMap) {
128     int camGX = camX / Constants::PIXEL_SIZE;
129     int camGY = camY / Constants::PIXEL_SIZE;
130
131     auto drawProp = [&](const Prop& prop) {
```

```
132         if (prop.wx < camX - 200 || prop.wx > camX + screenW + 200) return;
133
134         int gx = (prop.wx / Constants::PIXEL_SIZE) - camGX;
135         int gy = (prop.wy / Constants::PIXEL_SIZE) + camGY;
136         int worldGX = prop.wx / Constants::PIXEL_SIZE;
137
138         switch (prop.type) {
139         case PropType::Tree:       drawTree(p, gx, gy, worldGX, prop.wx, prop.wy,
                prop.variant, heightMap); break;
140         case PropType::Rock:       drawRock(p, gx, gy, prop.variant); break;
141         case PropType::Flower:     drawFlower(p, gx, gy, prop.variant); break;
142         case PropType::Mushroom:   drawMushroom(p, gx, gy, prop.variant); break;
143         case PropType::Cactus:     drawCactus(p, gx, gy, prop.variant); break;
144         case PropType::Tumbleweed: drawTumbleweed(p, gx, gy, prop.variant); break;
145         case PropType::Camel:      drawCamel(p, gx, gy, prop.variant, prop.flipped
                ); break;
146         case PropType::Igloo:      drawIgloo(p, gx, gy, worldGX, prop.variant,
                heightMap); break;
147         case PropType::Penguin:    drawPenguin(p, gx, gy, prop.variant, prop.
                flipped); break;
148         case PropType::Snowman:    drawSnowman(p, gx, gy, prop.variant); break;
149         case PropType::IceSpike:   drawIceSpike(p, gx, gy, prop.variant); break;
150         case PropType::UFO:        drawUFO(p, gx, gy, prop.variant); break;
151         case PropType::Rover:      drawRover(p, gx, gy, worldGX, prop.variant,
                prop.flipped, heightMap); break;
152         case PropType::Alien:      drawAlien(p, gx, gy, prop.variant); break;
153         case PropType::Building:   drawBuilding(p, gx, gy, worldGX, prop.variant,
                heightMap); break;
154         case PropType::StreetLamp: drawStreetLamp(p, gx, gy, worldGX, prop.variant
                , heightMap); break;
155         }
156     };
157
158     for (const Prop& prop : m_props) {
159         if (prop.type == PropType::Building) drawProp(prop);
160     }
161
162     for (const Prop& prop : m_props) {
163         if (prop.type != PropType::Building) drawProp(prop);
164     }
165 }
166
167 void PropSystem::plot(QPainter& p, int gx, int gy, const QColor& c) {
168     p.fillRect(gx * Constants::PIXEL_SIZE, gy * Constants::PIXEL_SIZE,
169                Constants::PIXEL_SIZE, Constants::PIXEL_SIZE, c);
170 }
171
172 void PropSystem::drawBuilding(QPainter& p, int gx, int gy, int worldGX, int
        variant, const QHash<int,int>& heightMap) {
173     QColor bDark(10, 10, 18);
174     QColor bFrame(40, 40, 60);
175
```

```
176     std::vector<QColor> neons = {
177         QColor(30, 180, 180), QColor(180, 0, 90), QColor(120, 0, 200),
178         QColor(50, 180, 40),  QColor(200, 180, 40), QColor(200, 80, 40),
179         QColor(80, 100, 180)
180     };
181     QColor neon = neons[(variant + worldGX) % neons.size()];
182 
183     int h = 30 + (variant * 4);
184     int w = 32 + (variant % 3) * 8;
185 
186     if (variant == 5) { w = 25; h = 80; }
187     if (variant == 3) { w = 36; h = 100; }
188 
189     int leftRel = -w/2;
190     int rightRel = w/2;
191     int topScreenY = gy - h;
192 
193     for (int dx = leftRel; dx <= rightRel; dx++) {
194         int currentWX = worldGX + dx;
195         int currentScreenX = gx + dx;
196 
197         int centerGroundY = heightMap.value(worldGX, 0);
198         int groundYAtCol = heightMap.value(currentWX, centerGroundY);
199         int groundScreenY = gy + (groundYAtCol - centerGroundY);
200 
201         int columnHeight = groundScreenY - topScreenY;
202         if (columnHeight > 0) {
203             bool isSideEdge = (dx == leftRel || dx == rightRel);
204             p.fillRect(currentScreenX * Constants::PIXEL_SIZE,
205                        topScreenY * Constants::PIXEL_SIZE,
206                        Constants::PIXEL_SIZE,
207                        columnHeight * Constants::PIXEL_SIZE,
208                        isSideEdge ? bFrame : bDark);
209         }
210 
211         plot(p, currentScreenX, topScreenY, bFrame);
212 
213         if (dx > leftRel && dx < rightRel) {
214             int yStart = topScreenY + 2;
215 
216             for (int y = yStart; y < groundScreenY - 1; y += 4) {
217                 bool windowExists = false;
218                 int relY = y - topScreenY;
219 
220                 if (variant == 5) {
221                     windowExists = true;
222                 } else if (variant == 3) {
223                     if ((dx % 4) == 0) windowExists = true;
224                 } else {
225                     if ((dx % 3 == 0) && ((dx + relY) % 3 != 0)) windowExists =
                            true;
226                 }
```

```
227
228                 if (windowExists) {
229                     p.fillRect(currentScreenX * Constants::PIXEL_SIZE,
230                                y * Constants::PIXEL_SIZE,
231                                Constants::PIXEL_SIZE,
232                                2 * Constants::PIXEL_SIZE,
233                                neon);
234                 }
235             }
236         }
237     }
238
239     if (variant == 3 || variant == 5) {
240         for(int i=1; i<=10; i++){
241             plot(p, gx, topScreenY - i, bFrame);
242         }
243     }
244 }
245
246 void PropSystem::drawStreetLamp(QPainter& p, int gx, int gy, int worldGX, int
        variant, const QHash<int,int>& heightMap) {
247     QColor pole(100, 100, 110);
248     QColor light(255, 255, 220);
249
250     if (variant % 3 == 1) light = QColor(200, 240, 255);
251     else if (variant % 3 == 2) light = QColor(255, 200, 180);
252
253     int h = 24;
254     int groundScreenY = gy;
255
256     p.fillRect(gx * Constants::PIXEL_SIZE, (groundScreenY - h) * Constants::
            PIXEL_SIZE,
257                Constants::PIXEL_SIZE, h * Constants::PIXEL_SIZE, pole);
258
259     plot(p, gx + 1, groundScreenY - h, pole);
260     plot(p, gx + 2, groundScreenY - h, pole);
261     plot(p, gx + 2, groundScreenY - h + 1, light);
262     plot(p, gx + 1, groundScreenY - h + 2, light);
263     plot(p, gx + 3, groundScreenY - h + 2, light);
264     plot(p, gx + 2, groundScreenY - h + 2, light);
265 }
266
267 void PropSystem::drawTree(QPainter& p, int gx, int gy, int worldGX, int wx, int wy
        , int variant, const QHash<int,int>& heightMap) {
268     QColor cTrunk(184, 115, 51); QColor cTrunkDark(100, 50, 20); QColor cHole(80,
            40, 10);
269     QColor cLeafBase(46, 184, 46); QColor cLeafLight(154, 235, 90); QColor
            cLeafDark(20, 110, 35);
270     int trunkW = 6; int trunkH = 30 + (variant * 2);
271     int centerGroundWorldY = heightMap.value(worldGX, 0); int camYOffset = gy -
            centerGroundWorldY;
272     int peakScreenY = 999999; int halfTrunk = trunkW / 2;
```

```
273     for(int dx = -halfTrunk; dx < halfTrunk; dx++) {
274         int wgx = worldGX + dx;
275         if(heightMap.contains(wgx)) {
276             int sGY = heightMap.value(wgx) + camYOffset;
277             if(sGY < peakScreenY) peakScreenY = sGY;
278         }
279     }
280     if(peakScreenY == 999999) peakScreenY = gy;
281     int effectiveBaseY = peakScreenY - 1;
282     for(int dx = -halfTrunk + 1; dx < halfTrunk; dx++) {
283         int wgx = worldGX + dx;
284         int groundScreenY = gy;
285         if(heightMap.contains(wgx)) groundScreenY = heightMap.value(wgx) +
              camYOffset;
286         int trunkTopY = effectiveBaseY - trunkH;
287         for(int y = trunkTopY; y <= groundScreenY; y++) {
288             bool isBorder = (dx == -halfTrunk+1 || dx == halfTrunk-1);
289             bool isShadow = (dx == -1 && (effectiveBaseY - y) > 0 && (
                  effectiveBaseY - y) % 3 == 0);
290             bool isLight  = (dx == 1  && (effectiveBaseY - y) > 0 && (
                  effectiveBaseY - y) % 4 == 0);
291             bool isRootFill = (y > effectiveBaseY);
292             QColor c = cTrunk;
293             if (isRootFill) c = cTrunkDark;
294             else if (isBorder) c = cTrunkDark;
295             else if (isShadow) c = cTrunkDark;
296             else if (isLight)  c = cTrunk.lighter(110);
297             plot(p, gx + dx, y, c);
298         }
299     }
300     plot(p, gx, effectiveBaseY - trunkH/2, cHole); plot(p, gx, effectiveBaseY -
          trunkH/2 - 1, cHole);
301     int folBot = effectiveBaseY - trunkH + 2;
302     std::vector<int> rows = { 26, 28, 30, 30, 28, 26, 22, 20, 22, 24, 24, 22, 20,
          16, 14, 18, 20, 18, 16, 14, 12, 10, 14, 16, 14, 12, 10, 8, 6, 8, 6, 4, 2
          };
303     for(int i=0; i<rows.size(); i++) {
304         int w = rows[i]; if (variant % 2 == 0) w += 2;
305         int py = folBot - i; int startX = gx - w/2; int endX = gx + w/2;
306         for(int px = startX; px <= endX; px++) {
307             int snX = (wx / Constants::PIXEL_SIZE) + (px - gx);
308             int snY = (wy / Constants::PIXEL_SIZE) - i;
309             int pat = (snX * 17 + snY * 13 + variant * 7) % 100;
310             int lightThresh = 50; int shadowThresh = 15;
311             if (px < gx) { lightThresh -= 15; shadowThresh -= 10; }
312             else if (px > gx) { lightThresh += 25; shadowThresh += 20; }
313             QColor c = cLeafBase;
314             if (pat > lightThresh) c = cLeafLight;
315             else if (pat < shadowThresh) c = cLeafDark;
316             if (px == startX || px == endX || i == rows.size()-1) { c = cLeafDark;
                  }
317             plot(p, px, py, c);
```

```
318         }
319     }
320 }
321
322 void PropSystem::drawRock(QPainter& p, int gx, int gy, int variant) {
323     QColor c(100, 100, 110); QColor highlight(140, 140, 150); int r = 2 + (variant
        % 2);
324     for(int dy = -r; dy <= 0; dy++) {
325         for(int dx = -r; dx <= r; dx++) {
326             if (dx*dx + (dy*dy)*1.5 <= r*r) { plot(p, gx+dx, gy+dy, (dx<0 && dy<-r
                /2) ? highlight : c); }
327         }
328     }
329 }
330
331 void PropSystem::drawFlower(QPainter& p, int gx, int gy, int variant) {
332     QColor stem(50, 160, 50);
333     QColor petal = (variant % 3 == 0) ? QColor(255, 50, 50) : ((variant % 3 == 1)
        ? QColor(255, 255, 50) : QColor(100, 100, 255));
334     plot(p, gx, gy, stem); plot(p, gx, gy-1, stem); plot(p, gx-2, gy-1, petal);
        plot(p, gx-1, gy-1, petal);
335     plot(p, gx, gy-1, petal); plot(p, gx+1, gy-1, petal); plot(p, gx+2, gy-1,
        petal);
336     plot(p, gx-1, gy-2, petal); plot(p, gx, gy-2, petal); plot(p, gx+1, gy-2,
        petal); plot(p, gx, gy-3, petal);
337 }
338
339 void PropSystem::drawMushroom(QPainter& p, int gx, int gy, int variant) {
340     QColor stalk(220, 220, 210);
341     QColor cap = (variant % 2 == 0) ? QColor(200, 60, 60) : QColor(180, 140, 80);
342     plot(p, gx, gy, stalk); plot(p, gx, gy-1, stalk); plot(p, gx-2, gy-1, cap);
        plot(p, gx-1, gy-1, cap);
343     plot(p, gx, gy-1, cap); plot(p, gx+1, gy-1, cap); plot(p, gx+2, gy-1, cap);
344     plot(p, gx-1, gy-2, cap); plot(p, gx, gy-2, cap); plot(p, gx+1, gy-2, cap);
345 }
346
347 void PropSystem::drawCactus(QPainter& p, int gx, int gy, int variant) {
348     QColor c(40, 150, 40); int h = 10 + variant * 2;
349     for(int y=0; y<h; y++) { plot(p, gx, gy - y, c); plot(p, gx - 1, gy - y, c);
        plot(p, gx + 1, gy - y, c); }
350     plot(p, gx, gy - h, c);
351     if (variant > 0) {
352         int armY = gy - (h/2); plot(p, gx-2, armY, c); plot(p, gx-3, armY, c);
            plot(p, gx-2, armY+1, c);
353         plot(p, gx-3, armY+1, c); plot(p, gx-3, armY-1, c); plot(p, gx-4, armY-1,
            c); plot(p, gx-3, armY-2, c); plot(p, gx-4, armY-2, c);
354     }
355     if (variant > 2) {
356         int armY2 = gy - (h/2) - 2; plot(p, gx+2, armY2, c); plot(p, gx+3, armY2,
            c); plot(p, gx+2, armY2+1, c);
357         plot(p, gx+3, armY2+1, c); plot(p, gx+3, armY2-1, c); plot(p, gx+4, armY2
            -1, c); plot(p, gx+3, armY2-2, c); plot(p, gx+4, armY2-2, c);
```

```
358     }
359 }
360
361 void PropSystem::drawTumbleweed(QPainter& p, int gx, int gy, int variant) {
362     QColor twigDark(100, 80, 50); QColor twigLight(180, 140, 90); int r = 7 + (
            variant % 3); int cy = gy - r;
363     for(int dy = -r; dy <= r; dy++) {
364         for(int dx = -r; dx <= r; dx++) {
365             double dist = std::sqrt(dx*dx + dy*dy);
366             if (dist <= r) {
367                 int lines1 = (dx * 3 + dy * 3 + variant * 11) % 7; int lines2 = (
                        dx * -3 + dy * 4 + variant * 5) % 6; int lines3 = (dx * 5 + dy
                         + variant * 2) % 9;
368                 bool isBranch = false; QColor c = twigDark;
369                 if (lines1 == 0 || lines2 == 0) isBranch = true;
370                 if (lines3 == 0 && dist < r - 2) isBranch = true;
371                 if (dist > r - 1.5) { isBranch = true; c = twigDark; } else if (
                        isBranch) { c = twigLight; }
372                 int noise = (dx * 97 + dy * 89) % 100;
373                 if (isBranch && lines1 != 0 && lines2 != 0 && noise < 20) {
                        isBranch = false; }
374                 if (isBranch) { plot(p, gx+dx, cy+dy, c); }
375             }
376         }
377     }
378 }
379
380 void PropSystem::drawCamel(QPainter& p, int gx, int gy, int variant, bool flipped)
        {
381     int d = flipped ? -1 : 1; QColor bodyColor(218, 165, 32); QColor legColor(139,
            69, 19);
382     for (int y = 0; y < 8; ++y) plot(p, gx + (4 * d), gy - y, legColor);
383     for (int y = 0; y < 8; ++y) plot(p, gx - (6 * d), gy - y, legColor);
384     for (int y = 1; y < 8; ++y) plot(p, gx + (3 * d), gy - y, bodyColor);
385     for (int y = 1; y < 8; ++y) plot(p, gx - (5 * d), gy - y, bodyColor);
386     for (int x = -7; x <= 5; ++x) { for (int y = 8; y < 14; ++y) { plot(p, gx + (x
            * d), gy - y, bodyColor); } }
387     bool twoHumps = (variant % 2 == 0);
388     if (twoHumps) {
389         plot(p, gx - (4 * d), gy - 14, bodyColor); plot(p, gx - (3 * d), gy - 14,
                bodyColor);
390         plot(p, gx - (4 * d), gy - 15, bodyColor); plot(p, gx - (3 * d), gy - 15,
                bodyColor);
391         plot(p, gx + (1 * d), gy - 14, bodyColor); plot(p, gx + (2 * d), gy - 14,
                bodyColor);
392         plot(p, gx + (1 * d), gy - 15, bodyColor); plot(p, gx + (2 * d), gy - 15,
                bodyColor);
393     } else {
394         for(int x = -2; x <= 1; x++) { plot(p, gx + (x * d), gy - 14, bodyColor);
                plot(p, gx + (x * d), gy - 15, bodyColor); }
395         plot(p, gx - (1 * d), gy - 16, bodyColor); plot(p, gx, gy - 16, bodyColor)
                ;
```

```
396     }
397     for(int y = 12; y < 18; y++) { plot(p, gx + (6 * d), gy - y, bodyColor); plot(
            p, gx + (7 * d), gy - y, bodyColor); }
398     plot(p, gx + (6 * d), gy - 18, bodyColor); plot(p, gx + (7 * d), gy - 18,
            bodyColor);
399     plot(p, gx + (8 * d), gy - 18, bodyColor); plot(p, gx + (6 * d), gy - 19,
            bodyColor);
400     plot(p, gx + (7 * d), gy - 19, bodyColor); plot(p, gx + (5 * d), gy - 19,
            legColor);
401     plot(p, gx + (7 * d), gy - 19, legColor); plot(p, gx - (8 * d), gy - 10,
            legColor); plot(p, gx - (8 * d), gy - 9, bodyColor);
402 }
403
404 void PropSystem::drawIgloo(QPainter& p, int gx, int gy, int worldGX, int variant,
        const QHash<int,int>& heightMap) {
405     QColor ice(220, 230, 255); QColor iceShadow(180, 190, 220); QColor dark(50,
            50, 60);
406     int r = 14 + (variant % 3); int centerGroundWorldY = heightMap.value(worldGX,
            0); int camYOffset = gy - centerGroundWorldY;
407     int peakScreenY = 999999;
408     for(int dx = -r; dx <= r; dx++) {
409         int wgx = worldGX + dx; if(heightMap.contains(wgx)) {
410             int groundScreenY = heightMap.value(wgx) + camYOffset;
411             if(groundScreenY < peakScreenY) { peakScreenY = groundScreenY; }
412         }
413     }
414     if (peakScreenY == 999999) peakScreenY = gy;
415     for(int dx = -r; dx <= r; dx++) {
416         int wgx = worldGX + dx; int groundScreenY = gy;
417         if(heightMap.contains(wgx)) { groundScreenY = heightMap.value(wgx) +
                camYOffset; }
418         int h = std::round(std::sqrt(r*r - dx*dx)); int domeTopY = peakScreenY - h
                ;
419         for (int y = domeTopY; y < groundScreenY; y++) {
420             bool isFoundation = (y >= peakScreenY); bool isShadow = (dx > r/3) ||
                    (y > peakScreenY - r/4 && !isFoundation);
421             QColor c = (isShadow || isFoundation) ? iceShadow : ice; plot(p, gx +
                    dx, y, c);
422         }
423     }
424     int tunW = 6; int tunH = 8; int tunBaseY = peakScreenY;
425     for(int dx = -tunW; dx <= tunW; dx++) {
426         int wgx = worldGX + dx; int groundScreenY = gy;
427         if(heightMap.contains(wgx)) groundScreenY = heightMap.value(wgx) +
                camYOffset;
428         int tunTopY = tunBaseY - tunH;
429         for(int y = tunTopY; y < groundScreenY; y++) { plot(p, gx + dx, y,
                iceShadow); }
430     }
431     for(int dx = -3; dx <= 3; dx++) {
432         int wgx = worldGX + dx; int groundScreenY = gy;
433         if(heightMap.contains(wgx)) groundScreenY = heightMap.value(wgx) +
```

```
            camYOffset;
434         int holeTopY = tunBaseY - (tunH - 2);
435         for(int y = holeTopY; y < groundScreenY; y++) { plot(p, gx + dx, y, dark);
                }
436     }
437 }
438
439 void PropSystem::drawPenguin(QPainter& p, int gx, int gy, int variant, bool
        flipped) {
440     int d = flipped ? -1 : 1; QColor black(30, 30, 40); QColor white(240, 240,
            250); QColor orange(255, 140, 0);
441     plot(p, gx+(1*d), gy, orange); plot(p, gx+(2*d), gy, orange); plot(p, gx-(1*d)
            , gy, orange);
442     for(int y=1; y<9; y++) for(int x=-2; x<=2; x++) plot(p, gx+(x*d), gy-y, black)
            ;
443     for(int y=1; y<8; y++) { plot(p, gx+(1*d), gy-y, white); plot(p, gx+(2*d), gy-
            y, white); }
444     for(int y=9; y<=11; y++) for(int x=-2; x<=2; x++) plot(p, gx+(x*d), gy-y,
            black);
445     plot(p, gx+(1*d), gy-10, white); plot(p, gx+(3*d), gy-10, orange); plot(p, gx
            -(1*d), gy-5, black); plot(p, gx-(2*d), gy-4, black);
446 }
447
448 void PropSystem::drawSnowman(QPainter& p, int gx, int gy, int variant) {
449     QColor snow(250, 250, 255); QColor carrot(255, 140, 0); QColor stick(80, 60,
            40); QColor coal(20, 20, 20); QColor tooth(255, 255, 255);
450     plot(p, gx-2, gy, snow); plot(p, gx-1, gy, snow); plot(p, gx+1, gy, snow);
            plot(p, gx+2, gy, snow);
451     for(int y=1; y<6; y++) { for(int x=-3; x<=3; x++) plot(p, gx+x, gy-y, snow); }
452     plot(p, gx, gy-2, coal); plot(p, gx, gy-4, coal);
453     for(int y=6; y<9; y++) { for(int x=-2; x<=2; x++) plot(p, gx+x, gy-y, snow); }
454     plot(p, gx, gy-7, coal);
455     for(int y=9; y<16; y++) { for(int x=-2; x<=2; x++) plot(p, gx+x, gy-y, snow);
            }
456     plot(p, gx-3, gy-10, snow); plot(p, gx+3, gy-10, snow); plot(p, gx-1, gy-13,
            coal); plot(p, gx+1, gy-13, coal);
457     plot(p, gx, gy-12, carrot); plot(p, gx+1, gy-12, carrot); plot(p, gx+2, gy-11,
             carrot); plot(p, gx, gy-10, tooth);
458     plot(p, gx, gy-16, stick); plot(p, gx-1, gy-17, stick); plot(p, gx+1, gy-17,
            stick);
459     plot(p, gx-3, gy-7, stick); plot(p, gx-4, gy-6, stick); plot(p, gx+3, gy-7,
            stick); plot(p, gx+4, gy-8, stick);
460 }
461
462 void PropSystem::drawIceSpike(QPainter& p, int gx, int gy, int variant) {
463     QColor ice(180, 230, 255); int h = 5 + variant * 2;
464     for(int y=0; y<h; y++) { plot(p, gx, gy-y, ice); if(y < h/2) { plot(p, gx-1,
            gy-y, ice); plot(p, gx+1, gy-y, ice); } }
465 }
466
467 void PropSystem::drawUFO(QPainter& p, int gx, int gy, int variant) {
468     QColor metal(150, 150, 160); QColor glass(100, 200, 255); QColor light = (
```

```
            variant % 2 == 0) ? QColor(255, 50, 50) : QColor(50, 255, 50);
469     plot(p, gx, gy-2, glass); plot(p, gx-1, gy-2, glass); plot(p, gx+1, gy-2,
            glass); plot(p, gx, gy-3, glass);
470     for(int x=-4; x<=4; x++) plot(p, gx+x, gy-1, metal); for(int x=-2; x<=2; x++)
            plot(p, gx+x, gy, metal);
471     plot(p, gx-3, gy-1, light); plot(p, gx+3, gy-1, light); plot(p, gx, gy, light)
            ;
472 }
473
474 void PropSystem::drawRover(QPainter& p, int gx, int gy, int worldGX, int variant,
        bool flipped, const QHash<int,int>& heightMap) {
475     int d = flipped ? -1 : 1;
476     QColor wheelC(30, 30, 35); QColor chassisC(220, 220, 220); QColor detailC(50,
            50, 60); QColor lensC(20, 30, 80); QColor gold(200, 170, 50); QColor
            strutC(40, 40, 50);
477     int centerGroundWorldY = heightMap.value(worldGX, 0); int camYOffset = gy -
            centerGroundWorldY;
478     int peakScreenY = 999999;
479     for(int dx = -6; dx <= 6; dx++) { int wgx = worldGX + dx; if(heightMap.
            contains(wgx)) { int sGY = heightMap.value(wgx) + camYOffset; if(sGY <
            peakScreenY) peakScreenY = sGY; } }
480     if(peakScreenY == 999999) peakScreenY = gy;
481     int chassisBaseY = peakScreenY - 2;
482     auto drawAdaptiveWheel = [&](int offsetX) {
483         int wheelWorldGX = worldGX + offsetX; int wheelScreenX = gx + offsetX; int
                groundY = peakScreenY + 5;
484         if (heightMap.contains(wheelWorldGX)) { groundY = heightMap.value(
                wheelWorldGX) + camYOffset; }
485         int wheelY = groundY;
486         for(int y = chassisBaseY; y < wheelY; y++) { plot(p, wheelScreenX, y,
                strutC); plot(p, wheelScreenX + 1, y, strutC); }
487         plot(p, wheelScreenX, wheelY, wheelC); plot(p, wheelScreenX+1, wheelY,
                wheelC); plot(p, wheelScreenX, wheelY-1, wheelC); plot(p, wheelScreenX
                +1, wheelY-1, wheelC);
488     };
489     drawAdaptiveWheel(-5 * d); drawAdaptiveWheel(-1 * d); drawAdaptiveWheel(5 * d)
            ;
490     int bodyY = chassisBaseY - 1; plot(p, gx-(5*d), bodyY, detailC); plot(p, gx
            -(1*d), bodyY, detailC); plot(p, gx+(5*d), bodyY, detailC);
491     for(int x=-6; x<=6; x++) { plot(p, gx+(x*d), bodyY-1, chassisC); plot(p, gx+(x
            *d), bodyY-2, chassisC); }
492     plot(p, gx-(5*d), bodyY-3, detailC); plot(p, gx-(6*d), bodyY-3, detailC); plot
            (p, gx-(5*d), bodyY-4, detailC);
493     int mastX = gx + (4*d); plot(p, mastX, bodyY-3, detailC); plot(p, mastX, bodyY
            -4, detailC); plot(p, mastX, bodyY-5, detailC);
494     plot(p, mastX+(1*d), bodyY-6, chassisC); plot(p, mastX+(1*d), bodyY-6, lensC);
495     int dishX = gx - (1*d); plot(p, dishX, bodyY-3, detailC); plot(p, dishX-1,
            bodyY-4, gold); plot(p, dishX, bodyY-4, gold); plot(p, dishX+1, bodyY-4,
            gold); plot(p, dishX-2, bodyY-5, gold); plot(p, dishX+2, bodyY-5, gold);
496 }
497
498 void PropSystem::drawAlien(QPainter& p, int gx, int gy, int variant) {
```

```
    QColor skin(50, 220, 80); QColor dark(30, 150, 50); QColor eyeWhite(255, 255, 255); QColor eyeBlack(0, 0, 0);
    for(int y=0; y<6; y++) { plot(p, gx, gy-y, skin); plot(p, gx-1, gy-y, skin); plot(p, gx+1, gy-y, skin); }
    plot(p, gx-2, gy, dark); plot(p, gx+2, gy, dark);
    if (variant % 2 == 0) { plot(p, gx-2, gy-3, skin); plot(p, gx-3, gy-4, skin); plot(p, gx+2, gy-3, skin); }
    else { plot(p, gx+2, gy-3, skin); plot(p, gx+3, gy-4, skin); plot(p, gx-2, gy-3, skin); }
    for(int y=6; y<10; y++) { for(int x=-2; x<=2; x++) plot(p, gx+x, gy-y, skin); }
    plot(p, gx, gy-10, dark); plot(p, gx, gy-11, dark); plot(p, gx, gy-12, skin);
    plot(p, gx-1, gy-7, eyeBlack); plot(p, gx-1, gy-8, eyeBlack); plot(p, gx+1, gy-7, eyeBlack); plot(p, gx+1, gy-8, eyeWhite);
}
```

media.h

```
#pragma once

#include <QObject>
#include <QVector>

class QAudioOutput;
class QMediaPlayer;
class QSoundEffect;
class QPropertyAnimation;

class Media : public QObject {
    Q_OBJECT
public:
    explicit Media(QObject* parent = nullptr);
    ~Media();

    void setupBgm();
    void setBgmVolume(qreal v);
    void playBgm();
    void stopBgm();

    void setStageBgm(int levelIndex);

    void startAccelLoop();
    void stopAccelLoop();

    void playNitroOnce();

    void coinPickup();
    void fuelPickup();

    void playGameOverOnce();

private:
    QAudioOutput* m_bgmOut = nullptr;
```

```
    QMediaPlayer* m_bgm    = nullptr;

    QVector<QSoundEffect*> m_driveSfx;
    QPropertyAnimation* m_accelFade = nullptr;

    QSoundEffect* m_nitroSfx = nullptr;

    QVector<QMediaPlayer*> m_coinPlayers;
    QVector<QAudioOutput*> m_coinOuts;
    int m_nextCoinPl = -1;

    QVector<QMediaPlayer*> m_fuelPlayers;
    QVector<QAudioOutput*> m_fuelOuts;
    int m_nextFuelPl = -1;

    QAudioOutput* m_gameOverOut = nullptr;
    QMediaPlayer* m_gameOver    = nullptr;
};
```

media.cpp

```
#include "media.h"

#include <QAudioOutput>
#include <QMediaPlayer>
#include <QMediaDevices>
#include <QSoundEffect>
#include <QPropertyAnimation>
#include <QCoreApplication>
#include <QFile>
#include <QUrl>

namespace {

QUrl pickBgmUrl(const QString& alias, const QString& fileName)
{
    const QString qrcPath = QString(":/audio/%1").arg(alias);
    if (QFile::exists(qrcPath)) {
        return QUrl(QStringLiteral("qrc:/audio/") + alias);
    }

    const QString fsPath =
        QCoreApplication::applicationDirPath() + "/assets/audio/" + fileName;
    if (QFile::exists(fsPath)) {
        return QUrl::fromLocalFile(fsPath);
    }

    return {};
}

QUrl defaultBgmUrl()
{
    const QString qrcPath = QStringLiteral(":/audio/bgm.mp3");
```

```
33      if (QFile::exists(qrcPath)) {
34          return QUrl(QStringLiteral("qrc:/audio/bgm.mp3"));
35      }
36
37      const QString fsPath =
38          QCoreApplication::applicationDirPath() + "/assets/audio/bgm.mp3";
39      if (QFile::exists(fsPath)) {
40          return QUrl::fromLocalFile(fsPath);
41      }
42
43      return {};
44  }
45
46  }
47
48  Media::Media(QObject* parent)
49      : QObject(parent)
50  {
51      auto* e = new QSoundEffect(this);
52      e->setSource(QUrl(QStringLiteral("qrc:/sfx/accelerate.wav")));
53      e->setLoopCount(1);
54      e->setVolume(0.35);
55      m_driveSfx.push_back(e);
56
57      m_nitroSfx = new QSoundEffect(this);
58      m_nitroSfx->setSource(QUrl(QStringLiteral("qrc:/sfx/nitro.wav")));
59      m_nitroSfx->setLoopCount(1);
60      m_nitroSfx->setVolume(0.35);
61
62      m_gameOverOut = new QAudioOutput(this);
63      m_gameOverOut->setVolume(0.35);
64      m_gameOver = new QMediaPlayer(this);
65      m_gameOver->setAudioOutput(m_gameOverOut);
66      m_gameOver->setSource(QUrl(QStringLiteral("qrc:/sfx/gameOver.mp3")));
67
68      const char* coinPath = "qrc:/sfx/coin.mp3";
69      for (int i = 0; i < 3; ++i) {
70          auto* out = new QAudioOutput(this);
71          out->setVolume(0.35);
72          auto* p = new QMediaPlayer(this);
73          p->setAudioOutput(out);
74          p->setSource(QUrl(coinPath));
75          m_coinOuts.push_back(out);
76          m_coinPlayers.push_back(p);
77      }
78
79      const char* fuelPath = "qrc:/sfx/fuel.mp3";
80      for (int i = 0; i < 3; ++i) {
81          auto* out = new QAudioOutput(this);
82          out->setVolume(0.35);
83          auto* p = new QMediaPlayer(this);
84          p->setAudioOutput(out);
```

```
85          p->setSource(QUrl(fuelPath));
86          m_fuelOuts.push_back(out);
87          m_fuelPlayers.push_back(p);
88      }
89  }
90
91  Media::~Media() = default;
92
93  void Media::setupBgm()
94  {
95      m_bgmOut = new QAudioOutput(this);
96      m_bgmOut->setVolume(1);
97      m_bgm = new QMediaPlayer(this);
98      m_bgm->setAudioOutput(m_bgmOut);
99      m_bgm->setLoops(QMediaPlayer::Infinite);
100
101     const QUrl src = defaultBgmUrl();
102     if (!src.isEmpty()) {
103         m_bgm->setSource(src);
104     }
105 }
106
107 void Media::setBgmVolume(qreal v)
108 {
109     if (m_bgmOut) {
110         m_bgmOut->setVolume(v);
111     }
112 }
113
114 void Media::playBgm()
115 {
116     if (m_bgm) {
117         m_bgm->play();
118     }
119 }
120
121 void Media::stopBgm()
122 {
123     if (m_bgm) {
124         m_bgm->stop();
125     }
126 }
127
128 void Media::setStageBgm(int levelIndex)
129 {
130     if (!m_bgm) return;
131
132     QUrl src;
133     switch (levelIndex) {
134     case 0: src = pickBgmUrl("bgm_meadow.mp3", "bgm_meadow.mp3"); break;
135     case 1: src = pickBgmUrl("bgm_desert.mp3", "bgm_desert.mp3"); break;
136     case 2: src = pickBgmUrl("bgm_tundra.mp3", "bgm_tundra.mp3"); break;
```

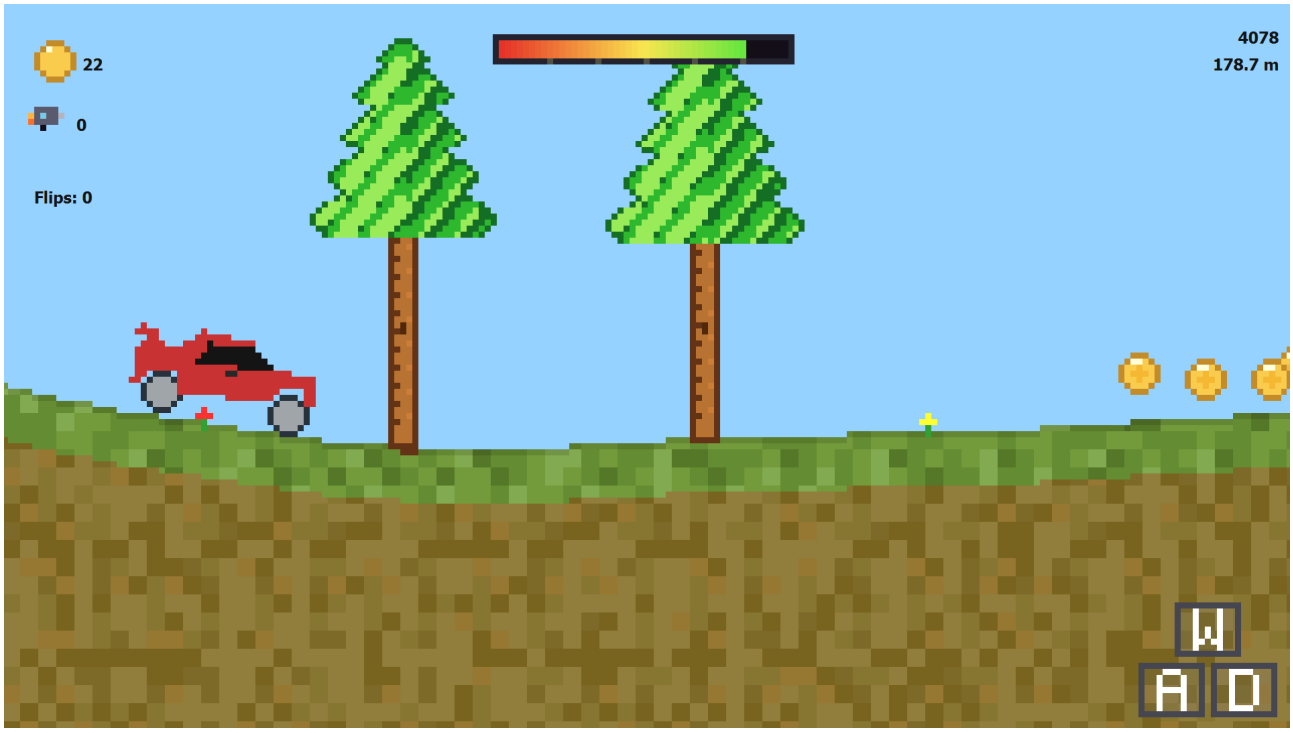
```
137     case 3: src = pickBgmUrl("bgm_lunar.mp3", "bgm_lunar.mp3"); break;
138     case 4: src = pickBgmUrl("bgm_martian.mp3", "bgm_martian.mp3"); break;
139     case 5: src = pickBgmUrl("bgm_nightlife.mp3", "bgm_nightlife.mp3"); break;
140     default: break;
141     }
142 
143     if (src.isEmpty()) src = defaultBgmUrl();
144     if (!src.isEmpty()) m_bgm->setSource(src);
145     m_bgm->play();
146 }
147 
148 void Media::startAccelLoop()
149 {
150     if (m_driveSfx.isEmpty()) return;
151     auto* e = m_driveSfx[0];
152 
153     if (m_accelFade) {
154         m_accelFade->stop();
155         delete m_accelFade;
156         m_accelFade = nullptr;
157     }
158 
159     e->setLoopCount(QSoundEffect::Infinite);
160     e->setVolume(1.0);
161     if (!e->isPlaying()) e->play();
162 }
163 
164 void Media::stopAccelLoop()
165 {
166     if (m_driveSfx.isEmpty()) return;
167     auto* e = m_driveSfx[0];
168 
169     if (m_accelFade) {
170         m_accelFade->stop();
171         delete m_accelFade;
172         m_accelFade = nullptr;
173     }
174 
175     m_accelFade = new QPropertyAnimation(e, "volume", this);
176     m_accelFade->setStartValue(e->volume());
177     m_accelFade->setEndValue(0.0);
178     m_accelFade->setDuration(250);
179 
180     connect(m_accelFade, &QPropertyAnimation::finished, this, [this, e] {
181         e->stop();
182         e->setVolume(0.0);
183         if (m_accelFade) {
184             m_accelFade->deleteLater();
185             m_accelFade = nullptr;
186         }
187     });
188 
```

```
    m_accelFade->start();
}

void Media::playNitroOnce()
{
    if (!m_nitroSfx) return;
    m_nitroSfx->stop();
    m_nitroSfx->setLoopCount(1);
    m_nitroSfx->setVolume(0.35);
    m_nitroSfx->play();
}

void Media::coinPickup()
{
    if (m_coinPlayers.isEmpty()) return;
    m_nextCoinPl = (m_nextCoinPl + 1) % m_coinPlayers.size();
    auto* p = m_coinPlayers[m_nextCoinPl];
    p->setPosition(0);
    p->play();
}

void Media::fuelPickup()
{
    if (m_fuelPlayers.isEmpty()) return;
    m_nextFuelPl = (m_nextFuelPl + 1) % m_fuelPlayers.size();
    auto* p = m_fuelPlayers[m_nextFuelPl];
    p->setPosition(0);
    p->play();
}

void Media::playGameOverOnce()
{
    if (!m_gameOver) return;
    m_gameOver->setPosition(0);
    m_gameOver->play();
}
```

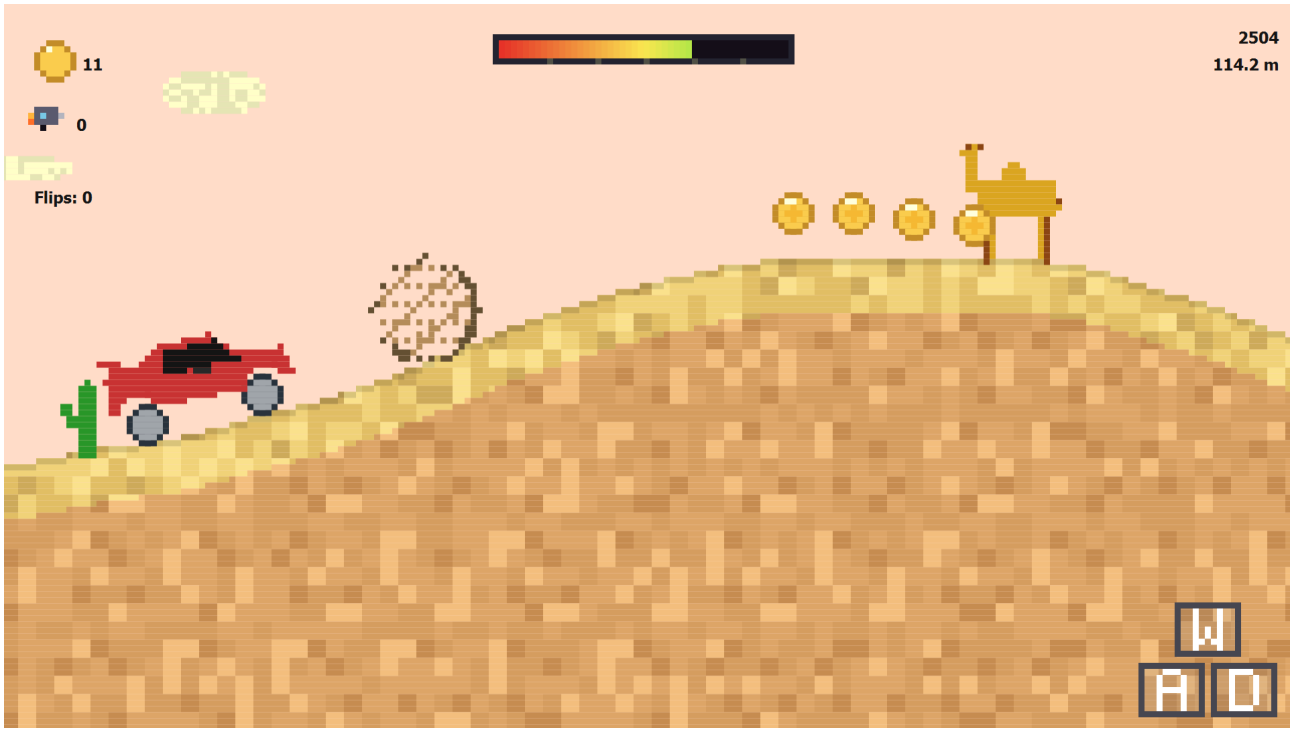
assets.qrc (Resource Mapping)

```
<RCC>
  <qresource prefix="/audio">
    <file alias="bgm.mp3">assets/audio/bgm.mp3</file>
    <file alias="bgm_meadow.mp3">assets/audio/bgm.mp3</file>
    <file alias="bgm_desert.mp3">assets/audio/bgm_desert.mp3</file>
    <file alias="bgm_tundra.mp3">assets/audio/bgm_tundra.mp3</file>
    <file alias="bgm_lunar.mp3">assets/audio/bgm_lunar.mp3</file>
    <file alias="bgm_martian.mp3">assets/audio/bgm_martian.mp3</file>
    <file alias="bgm_nightlife.mp3">assets/audio/bgm_nightlife.mp3</file>
  </qresource>
  <qresource prefix="/sfx">
    <file alias="accelerate.wav">assets/audio/accelerate.wav</file>
    <file alias="nitro.wav">assets/audio/nitro.wav</file>
    <file alias="coin.mp3">assets/audio/coin.mp3</file>
```

```
    <file alias="fuel.mp3">assets/audio/fuel.mp3</file>
    <file alias="gameOver.mp3">assets/audio/gameOver.mp3</file>
  </qresource>
</RCC>
```



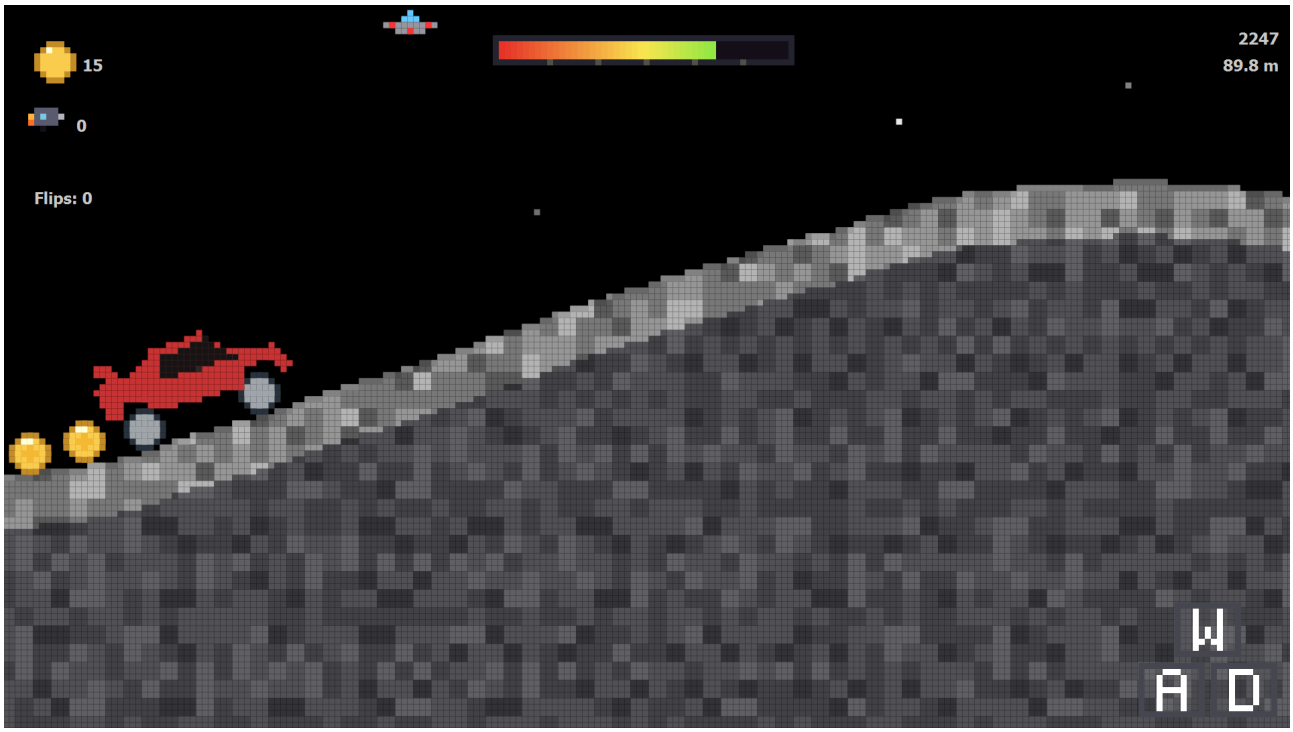
(a) Meadow: The starting biome featuring grassy slopes, procedural trees, and scattered rocks.



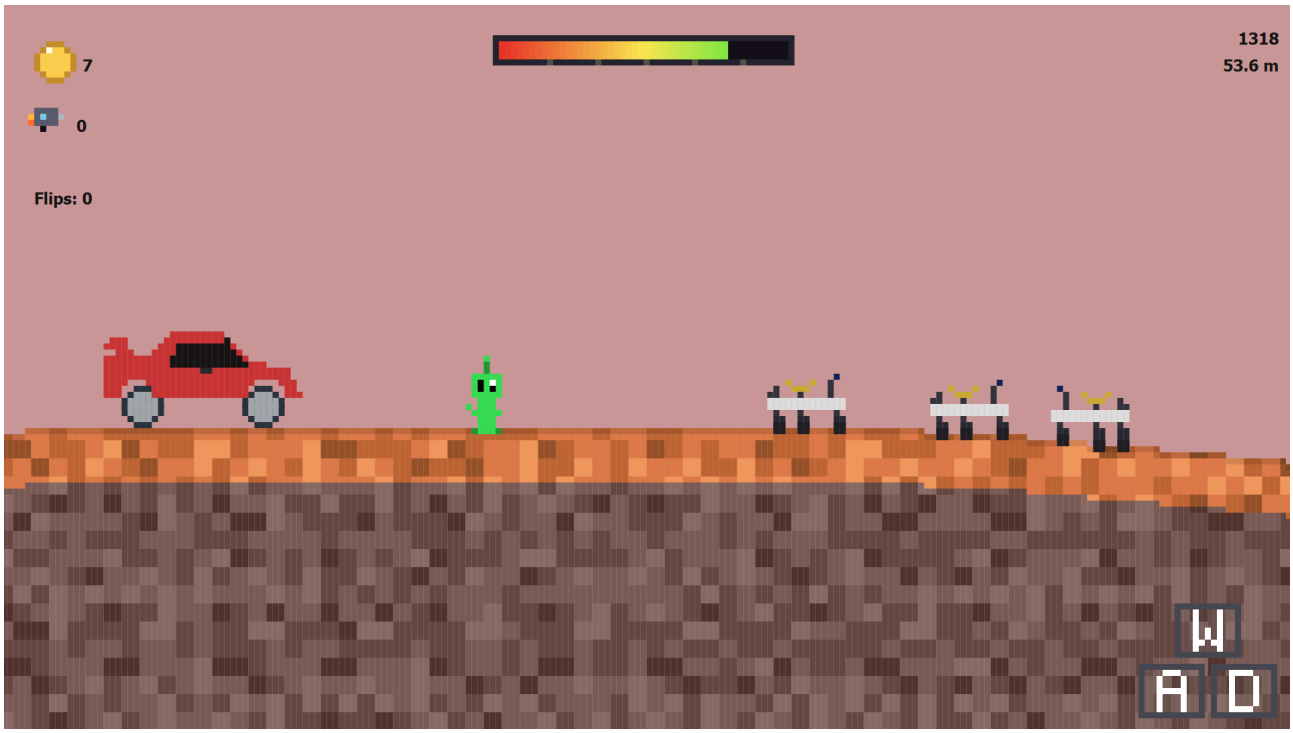
(b) Desert: Arid sand dunes populated by cacti, tumbleweeds, and camels.

(c) Tundra: Snowy peaks and icy terrain with penguins, igloos, and snowmen.

Figure 14: Earth-based biomes showcasing distinct color palettes and procedural environmental props.

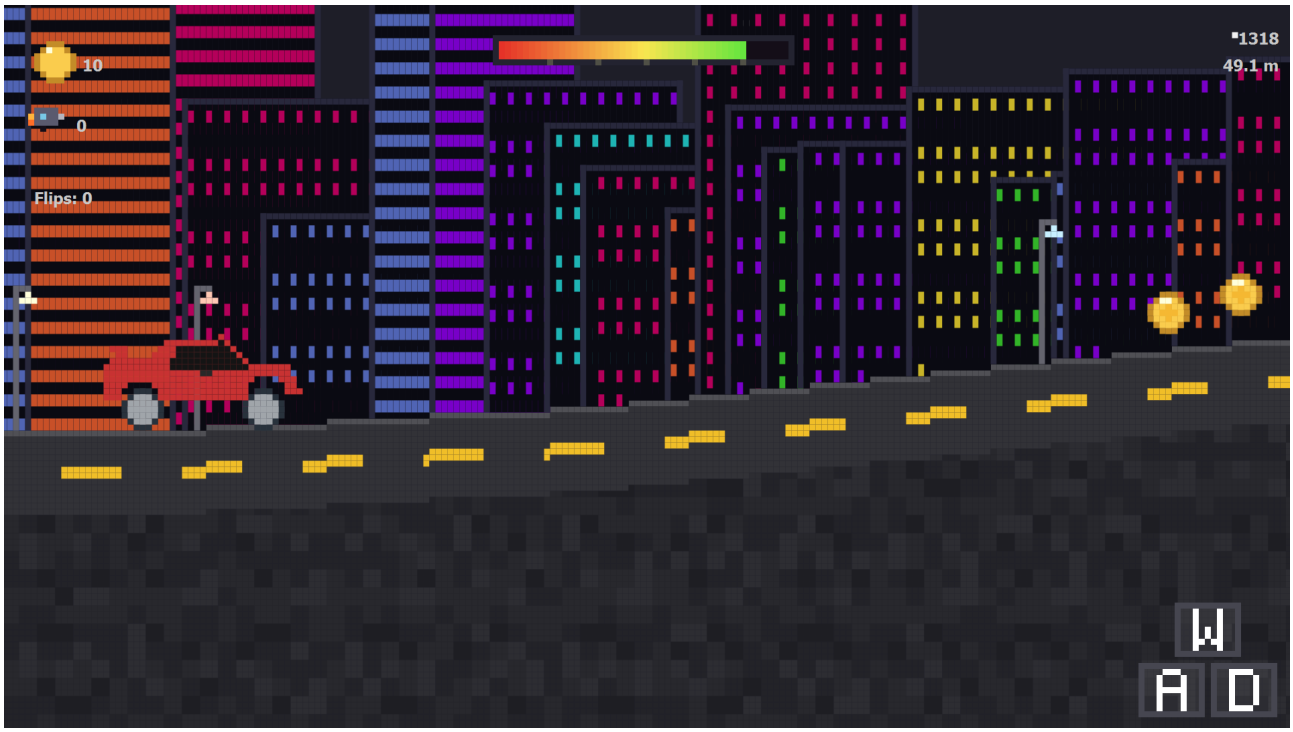


(a) Lunar: A dark, low-gravity moonscape with craters and hovering UFOs.



(b) Martian: Red planet terrain featuring exploration rovers and alien lifeforms.

Figure 15: Exotic and urban biomes introducing unique visual styles and lighting effects.



(c) Nightlife: A neon-lit urban environment with procedurally generated skyscrapers and street lamps.

Figure 15: Exotic and urban biomes.

Feature: Local Leaderboard and Persistence System

Relevant classes and functions: `LeaderboardManager`, `LeaderboardWidget`, `MainWindow::showGameOver()`, `QSettings`.

To provide long-term player engagement, I implemented a persistent local leaderboard system that tracks high scores for each biome. The system is architected using a separation of concerns pattern: the `LeaderboardManager` handles data persistence and sorting logic, while the `LeaderboardWidget` handles the custom pixel-art rendering of the overlay.

Data Persistence and Identity. Instead of using an external database, I utilized `QSettings` to store high scores locally in the system registry or configuration files. To distinguish entries, I generate a unique device identifier using `QSysInfo::machineUniqueId()`. This allows the system to be extended in the future for network synchronization while currently serving as a local user tag.

The data is structured as a list of `LeaderboardEntry` structs, each containing the stage name, user ID, and score. When the application starts, these are deserialized from storage. When a game ends, the new score is compared against existing records. If a personal best for that specific stage is achieved, the record is updated, the list is re-sorted, and the data is flushed back to storage.

Sorting Logic. To ensure a consistent display, I enforce a deterministic sorting order. Entries are primarily grouped by `stageName` and secondarily sorted by `score` in descending order:

$$\text{Order}(A, B) = \begin{cases} \text{Score}_A > \text{Score}_B, & \text{if Stage}_A = \text{Stage}_B \\ \text{Stage}_A < \text{Stage}_B, & \text{otherwise} \end{cases}$$

Custom UI Rendering. The `LeaderboardWidget` is a custom QWidget that overlays the game window. Rather than using standard Qt UI controls, I implemented a custom `paintEvent` to draw the scoreboard using the game's specific color palette and pixel-font aesthetic. The background is rendered as a semi-transparent dark overlay (‘rgba(28, 24, 36, 240)‘) to maintain visual continuity with the game's "Nightlife" and "Space" themes.

Source Code. Below are the complete implementations of the scoreboard logic and the integration points within the main window.

scoreboard.h

```
#pragma once

#include <QObject>
#include <QWidget>
#include <QVector>
#include <QString>

class QPaintEvent;
class QKeyEvent;
class QMouseEvent;

struct LeaderboardEntry {
    QString stageName;
```

```
    QString userName;
    int     score = 0;
};

class LeaderboardWidget : public QWidget {
    Q_OBJECT
public:
    explicit LeaderboardWidget(QWidget* parent = nullptr);
    void setEntries(const QVector<LeaderboardEntry>& entries);

signals:
    void closed();

protected:
    void paintEvent(QPaintEvent* event) override;
    void keyPressEvent(QKeyEvent* event) override;
    void mousePressEvent(QMouseEvent* event) override;

private:
    QVector<LeaderboardEntry> m_entries;
};

class LeaderboardManager : public QObject {
    Q_OBJECT
public:
    explicit LeaderboardManager(QObject* parent = nullptr);
    void submitScore(const QString& stageName, int score);
    void refreshLeaderboard();

signals:
    void leaderboardUpdated(const QVector<LeaderboardEntry>& entries);

private:
    QString deviceId() const;
    void    loadFromSettings();
    void    saveToSettings() const;

    QVector<LeaderboardEntry> m_entries;
};
```

scoreboard.cpp

```
#include "scoreboard.h"
#include <QPainter>
#include <QPaintEvent>
#include <QKeyEvent>
#include <QMouseEvent>
#include <QPalette>
#include <QSysInfo>
#include <QSettings>
#include <algorithm>

LeaderboardWidget::LeaderboardWidget(QWidget* parent) : QWidget(parent)
```

```
{
    setAttribute(Qt::WA_StyledBackground, true);
    setAutoFillBackground(true);

    QPalette pal = palette();
    pal.setColor(QPalette::Window, QColor(0, 0, 0, 210));
    setPalette(pal);

    setFocusPolicy(Qt::StrongFocus);
    hide();
}

void LeaderboardWidget::setEntries(const QVector<LeaderboardEntry>& entries)
{
    m_entries = entries;
    update();
}

void LeaderboardWidget::paintEvent(QPaintEvent* event)
{
    Q_UNUSED(event);
    QPainter p(this);
    p.setRenderHint(QPainter::Antialiasing, false);
    p.setRenderHint(QPainter::TextAntialiasing, true);

    const QRect panel = rect().adjusted(width() * 0.1, height() * 0.1, -width() *
        0.1, -height() * 0.1);

    p.fillRect(panel, QColor(28, 24, 36, 240));
    p.setPen(QColor(80, 70, 100));
    p.drawRect(panel.adjusted(0, 0, -1, -1));

    QFont titleFont;
    titleFont.setFamily("Monospace");
    titleFont.setBold(true);
    titleFont.setPixelSize(24);
    titleFont.setStyleStrategy(QFont::NoAntialias);
    p.setFont(titleFont);
    p.setPen(QColor(230, 230, 240));

    const QString title = QStringLiteral("SCOREBOARD");
    QFontMetrics tfm(titleFont);
    int titleW = tfm.horizontalAdvance(title);
    int titleX = panel.center().x() - titleW / 2;
    int titleY = panel.top() + tfm.ascent() + 16;
    p.drawText(titleX, titleY, title);

    QFont headerFont;
    headerFont.setFamily("Monospace");
    headerFont.setBold(true);
    headerFont.setPixelSize(14);
    headerFont.setStyleStrategy(QFont::NoAntialias);
```

```
63
64      QFont rowFont = headerFont;
65      rowFont.setBold(false);
66
67      const int topMargin = 60;
68      const int rowHeight = 26;
69
70      int colStageX = panel.left() + 40;
71      int colScoreX = panel.right() - 160;
72
73      p.setFont(headerFont);
74      p.setPen(QColor(200, 200, 210));
75      int headerY = panel.top() + topMargin;
76      p.drawText(colStageX, headerY, QStringLiteral("STAGE"));
77      p.drawText(colScoreX, headerY, QStringLiteral("BEST SCORE"));
78
79      p.setPen(QColor(80, 70, 100));
80      p.drawLine(panel.left() + 20, headerY + 6,
81                 panel.right() - 20, headerY + 6);
82
83      p.setFont(rowFont);
84      p.setPen(QColor(230, 230, 240));
85
86      if (m_entries.isEmpty()) {
87          const QString msg = QStringLiteral("No scores yet. Play a game to create a
                record.");
88          QFontMetrics rfm(rowFont);
89          int w = rfm.horizontalAdvance(msg);
90          int x = panel.center().x() - w / 2;
91          int y = headerY + rowHeight * 2;
92          p.drawText(x, y, msg);
93          return;
94      }
95
96      int y = headerY + rowHeight;
97      for (int i = 0; i < m_entries.size(); ++i) {
98          const LeaderboardEntry& e = m_entries[i];
99
100         if (i % 2 == 0) {
101             QRect rowRect(panel.left() + 10, y - rowHeight + 6, panel.width() -
                    20, rowHeight);
102             p.fillRect(rowRect, QColor(40, 34, 50, 160));
103         }
104
105         p.setPen(QColor(220, 220, 230));
106         p.drawText(colStageX, y, e.stageName);
107         p.drawText(colScoreX, y, QString::number(e.score));
108
109         y += rowHeight;
110         if (y > panel.bottom() - 20) break;
111     }
112
```

```
    QFont hintFont;
    hintFont.setFamily("Monospace");
    hintFont.setPixelSize(10);
    hintFont.setStyleStrategy(QFont::NoAntialias);
    p.setFont(hintFont);
    p.setPen(QColor(170, 170, 190));
    const QString hint = QStringLiteral("[S] or [Esc] to close");
    QFontMetrics hfm(hintFont);
    int hw = hfm.horizontalAdvance(hint);
    int hx = panel.center().x() - hw / 2;
    int hy = panel.bottom() - 12;
    p.drawText(hx, hy, hint);
}

void LeaderboardWidget::keyPressEvent(QKeyEvent* event)
{
    if (event->key() == Qt::Key_S || event->key() == Qt::Key_Escape) {
        hide();
        emit closed();
        return;
    }
    QWidget::keyPressEvent(event);
}

void LeaderboardWidget::mousePressEvent(QMouseEvent* event)
{
    Q_UNUSED(event);
    hide();
    emit closed();
}

LeaderboardManager::LeaderboardManager(QObject* parent)
    : QObject(parent)
{
    loadFromSettings();
}

QString LeaderboardManager::deviceId() const
{
    QByteArray id = QSysInfo::machineUniqueId();
    if (!id.isEmpty())
        return QString::fromLatin1(id.toHex());

    QString host = QSysInfo::machineHostName();
    if (!host.isEmpty())
        return host;

    return QStringLiteral("UNKNOWN_DEVICE");
}

void LeaderboardManager::submitScore(const QString& stageName, int score)
{
```

```
    const QString user = deviceId();

    bool found = false;
    for (auto &e : m_entries) {
        if (e.stageName == stageName && e.userName == user) {
            if (score > e.score)
                e.score = score;
            found = true;
            break;
        }
    }

    if (!found) {
        LeaderboardEntry e;
        e.stageName = stageName;
        e.userName  = user;
        e.score     = score;
        m_entries.push_back(e);
    }

    std::sort(m_entries.begin(), m_entries.end(),
              [](const LeaderboardEntry& a, const LeaderboardEntry& b) {
                  if (a.stageName == b.stageName)
                      return a.score > b.score;
                  return a.stageName < b.stageName;
              });

    saveToSettings();
    emit leaderboardUpdated(m_entries);
}

void LeaderboardManager::refreshLeaderboard()
{
    emit leaderboardUpdated(m_entries);
}

void LeaderboardManager::loadFromSettings()
{
    m_entries.clear();

    QSettings s("JU", "F1PixelGrid");
    int n = s.beginReadArray("leaderboard");
    for (int i = 0; i < n; ++i) {
        s.setArrayIndex(i);
        LeaderboardEntry e;
        e.stageName = s.value("stage").toString();
        e.userName  = s.value("user").toString();
        e.score     = s.value("score").toInt();
        if (!e.stageName.isEmpty())
            m_entries.push_back(e);
    }
    s.endArray();
```

```

    std::sort(m_entries.begin(), m_entries.end(),
              [](const LeaderboardEntry& a, const LeaderboardEntry& b) {
                  if (a.stageName == b.stageName)
                      return a.score > b.score;
                  return a.stageName < b.stageName;
              });
}

void LeaderboardManager::saveToSettings() const
{
    QSettings s("JU", "F1PixelGrid");
    s.beginWriteArray("leaderboard");
    for (int i = 0; i < m_entries.size(); ++i) {
        s.setArrayIndex(i);
        s.setValue("stage", m_entries[i].stageName);
        s.setValue("user",  m_entries[i].userName);
        s.setValue("score", m_entries[i].score);
    }
    s.endArray();
    s.sync();
}
```

mainwindow.cpp (Integration Excerpts)

```
// Integration within the Constructor
MainWindow::MainWindow(QWidget *parent)
    : QWidget(parent),
    m_rng(std::random_device{}() ),
    m_dist(0.0f, 1.0f)
{
    // ... initialization ...

    m_leaderboardMgr = new LeaderboardManager(this);
    m_leaderboardWidget = new LeaderboardWidget(this);
    m_leaderboardWidget->setGeometry(rect());
    m_leaderboardWidget->hide();

    connect(m_leaderboardMgr, &LeaderboardManager::leaderboardUpdated,
            m_leaderboardWidget, &LeaderboardWidget::setEntries);

    connect(m_leaderboardWidget, &LeaderboardWidget::closed, this, [this]{
        if (m_timer && !m_intro && !m_outro) {
            m_timer->start();
        }
        setFocus();
    });

    // ... rest of constructor ...
}

// Integration within Game Over Logic
void MainWindow::showGameOver() {
```

```
    if (m_media) m_media->playGameOverOnce();

    if (m_leaderboardMgr) {
        QString stageName = QStringLiteral("UNKNOWN");
        if (level_index >= 0 && level_index < m_levelNames.size()) {
            stageName = m_levelNames[level_index];
        }
        m_leaderboardMgr->submitScore(stageName, m_score);
    }

    // ... existing game over logic ...
}

// Integration within Key Handling
void MainWindow::keyPressEvent(QKeyEvent *event) {
    // ... other keys ...

    switch (event->key()) {
        case Qt::Key_S:
            if (m_leaderboardWidget && m_leaderboardMgr) {
                if (m_timer && m_timer->isActive()) {
                    m_timer->stop();
                }
                m_leaderboardWidget->setGeometry(rect());
                m_leaderboardWidget->show();
                m_leaderboardWidget->raise();
                m_leaderboardWidget->setFocus();
                m_leaderboardMgr->refreshLeaderboard();
            }
            break;

    // ... remaining cases ...
    }
}
```

Figure 16: Local Leaderboard System: The custom overlay widget displaying persistent high scores for each biome, rendered with a semi-transparent background and pixel-perfect typography to match the game's aesthetic.

4. Challenges & Limitation

Challenges

- **Physics tuning:** Designing simple custom physics such that the three–wheel car feels responsive, but still unstable enough to punish careless driving, required repeated tuning of gravity, traction, damping and spring parameters across all stages.
- **Procedural terrain:** Generating terrain that looks organic while avoiding impossible slopes or unfair obstacles was challenging. The random walk–based height model had to be carefully constrained for each biome.
- **Pixel-grid UI:** All HUD elements and overlays are rendered through a manual pixel–grid system. Ensuring text readability, layout consistency and visual clarity across different window sizes and aspect ratios was non-trivial.

Limitations

- **Simplified physics:** The physics model is intentionally lightweight and 2D, without full rigid–body dynamics or continuous collision detection, so extreme configurations can occasionally produce visually unrealistic behaviour.
- **Limited content variety:** Terrain, coins, fuel pickups and props are procedurally generated but follow fixed parameter ranges per stage. Over long play sessions this can lead to repetitive patterns and predictable difficulty.
- **Local-only experience:** The leaderboard is strictly local, with no networking, cloud sync or cross-device progression. Additionally, there is no support for configurable key bindings, mobile/touch input schemes or accessibility-oriented options.

5. Conclusion & Future Scope

Conclusion

- **Integration of CG concepts:** The project combines line and circle rasterization, scan-line polygon filling, affine transformations and basic viewing ideas into a single interactive application, reinforcing core computer graphics topics.
- **Coherent game loop:** A simple physics engine, procedural terrain and HUD rendering are integrated within a Qt-based framework using a custom pixel-grid renderer and a fixed-timestep update loop.
- **Educational value:** The implementation illustrates trade-offs between visual style, responsiveness and algorithmic simplicity, providing a concrete, playable demonstration of techniques taught in the course.

Future Scope

- **Enhanced physics:** Upgrade to a more robust rigid-body model with better suspension, contact handling and continuous collision detection, improving realism without sacrificing responsiveness.
- **Richer content:** Diversify terrain and prop generation, add dynamic events (bridges, hazards, alternative routes) and introduce structured missions, obstacles and additional power-ups to increase replay value.
- **Multiplayer gameplay:** Add local split-screen or online multiplayer modes, enabling races between players on the same terrain seed, shared leaderboards and cooperative or competitive challenges.
- **Platform and UX improvements:** Introduce online leaderboards, configurable controls, support for gamepads and touch input, as well as basic accessibility settings (color/contrast options, UI scaling).