DMS Enterprise Upgrade — Walkthrough

Summary

Upgraded the DMS from a prototype to a high-fidelity system across **7 files** in **3 workstreams**.

Changes Made

1. Simulation Logic & Graphics

road_renderer.py

```
# ==============================================================================
# simulation/road_renderer.py
#
# RoadRenderer — draws a scrolling top-down road environment.
#
# Features:
#   • Infinite scrolling asphalt road with lane markings
#   • Animated dashed centre line (scrolls with road speed)
#   • Solid edge lines
#   • Roadside scenery (trees, grass strips)
#   • Road shoulder (for pull-over animation)
#   • All dimensions relative to surface size for easy resizing
# ==============================================================================

import pygame
import random
from config import WINDOW_WIDTH, WINDOW_HEIGHT

_PANEL_H = WINDOW_HEIGHT // 2

# ── Palette ──────────────────────────────────────────────────────────────────
C_SKY        = (20,  24,  36)     # Background / sky
C_GRASS      = (28,  60,  28)     # Roadside grass
C_ASPHALT    = (50,  52,  58)     # Road surface
C_SHOULDER   = (80,  76,  68)     # Road shoulder / gravel
C_LINE_WHITE = (220, 220, 210)    # Edge lines
C_LINE_DASH  = (200, 180,  20)    # Dashed centre line
C_TREE_TRUNK = (80,  55,  30)
C_TREE_TOP   = (30,  90,  30)
C_TREE_TOP2  = (20, 110,  40)


class Tree:
    """A simple roadside tree with randomized position and size."""
    def __init__(self, x: float, y: float, scale: float = 1.0):
        self.x = x
        self.y = y
        self.trunk_w = int(6  * scale)
```

```
        self.trunk_h = int(16 * scale)
        self.crown_r = int(14 * scale)
        self.color   = C_TREE_TOP if random.random() > 0.4 else C_TREE_TOP2

    def draw(self, surf: pygame.Surface, scroll_y: float):
        draw_y = int(self.y - scroll_y) % (_PANEL_H + 60) - 30
        # Trunk
        pygame.draw.rect(surf, C_TREE_TRUNK,
                         (int(self.x) - self.trunk_w // 2,
                          draw_y,
                          self.trunk_w, self.trunk_h))
        # Crown
        pygame.draw.circle(surf, self.color,
                           (int(self.x), draw_y - self.crown_r + 4),
                           self.crown_r)


class RoadRenderer:
    """
    Draws a top-down scrolling road onto a pygame.Surface.

    Usage:
        rr = RoadRenderer(surface_width, surface_height)
        # Each frame:
        rr.scroll(speed_px)          # advance scroll by speed pixels
        surf = rr.render(car_offset) # car_offset: how far right car has drifted
    """

    def __init__(self, width: int = WINDOW_WIDTH, height: int = _PANEL_H):
        self.w = width
        self.h = height

        # Road geometry (all relative to width)
        self.road_w     = int(width * 0.38)          # total road width
        self.road_x     = (width - self.road_w) // 2 # left edge of road
        self.shoulder_w = int(width * 0.055)         # gravel shoulder each side
        self.lane_w     = self.road_w // 2            # each lane width
        self.centre_x   = width // 2

        # Scroll state
        self._scroll_y  = 0.0

        # Dash line parameters
        self._dash_h    = 28
        self._dash_gap  = 20
        self._dash_cycle = self._dash_h + self._dash_gap

        # Generate roadside trees
        random.seed(42)
        self._trees_left  = [
            Tree(
                x=random.randint(20, self.road_x - self.shoulder_w - 10),
```

```
                y=random.randint(0, _PANEL_H + 200) * 1.0,
                scale=random.uniform(0.7, 1.3)
            )
            for _ in range(18)
        ]
        self._trees_right = [
            Tree(
                x=random.randint(
                    self.road_x + self.road_w + self.shoulder_w + 10,
                    width - 20
                ),
                y=random.randint(0, _PANEL_H + 200) * 1.0,
                scale=random.uniform(0.7, 1.3)
            )
            for _ in range(18)
        ]

        # Pre-create surface
        self._surf = pygame.Surface((width, height))

    # ── Public API ─────────────────────────────────────────────────────

    def scroll(self, speed_px: float) -> None:
        """Advance the road scroll by speed_px pixels."""
        self._scroll_y = (self._scroll_y + speed_px) % (self.h + 200)

    def render(self, car_x_offset: float = 0.0) -> pygame.Surface:
        """
        Draw the road and return the surface.

        Args:
            car_x_offset: How far (px) the road has shifted relative to car
                          (used for pull-over animation — road shifts right
                          as car moves to the shoulder)
        Returns:
            pygame.Surface ready to blit
        """
        surf = self._surf
        cx_off = int(car_x_offset)

        # ── Background (sky / off-road area) ──────────────────────────
        surf.fill(C_SKY)

        # ── Grass strips ──────────────────────────────────────────────
        left_grass_w  = self.road_x + cx_off
        right_grass_x = self.road_x + self.road_w + cx_off
        pygame.draw.rect(surf, C_GRASS, (0, 0, left_grass_w, self.h))
        pygame.draw.rect(surf, C_GRASS, (right_grass_x, 0, self.w - right_grass_x,
self.h))

        # ── Shoulders ─────────────────────────────────────────────────
        pygame.draw.rect(surf, C_SHOULDER,
```

```
                        (self.road_x - self.shoulder_w + cx_off, 0,
                         self.shoulder_w, self.h))
        pygame.draw.rect(surf, C_SHOULDER,
                        (self.road_x + self.road_w + cx_off, 0,
                         self.shoulder_w, self.h))

        # ── Asphalt ──────────────────────────────────────────────────────
        pygame.draw.rect(surf, C_ASPHALT,
                        (self.road_x + cx_off, 0, self.road_w, self.h))

        # ── Edge lines (solid white) ─────────────────────────────────────
        lw = 4
        pygame.draw.rect(surf, C_LINE_WHITE,
                        (self.road_x + cx_off, 0, lw, self.h))
        pygame.draw.rect(surf, C_LINE_WHITE,
                        (self.road_x + self.road_w - lw + cx_off, 0, lw, self.h))

        # ── Dashed centre line ───────────────────────────────────────────
        dash_x = self.centre_x - 3 + cx_off
        offset = int(self._scroll_y) % self._dash_cycle
        y = -offset
        while y < self.h:
            pygame.draw.rect(surf, C_LINE_DASH,
                             (dash_x, y, 6, self._dash_h))
            y += self._dash_cycle

        # ── Trees ────────────────────────────────────────────────────────
        for tree in self._trees_left + self._trees_right:
            tree.draw(surf, self._scroll_y)

        return surf
===
# ============================================================================
# simulation/road_renderer.py
#
# RoadRenderer — draws a scrolling top-down road environment.
#
# Features:
#   • Infinite scrolling asphalt road with lane markings
#   • Animated dashed centre line (scrolls with road speed)
#   • Solid edge lines
#   • Roadside scenery (trees, grass strips) — properly offset during pull-over
#   • Road shoulder with rumble-strip texture
#   • Parallax depth effect on trees
#   • All dimensions relative to surface size for easy resizing
# ============================================================================

import pygame
import random
from config import WINDOW_WIDTH, WINDOW_HEIGHT

_PANEL_H = WINDOW_HEIGHT // 2
```

```
# ── Palette ──────────────────────────────────────────────────────────────
C_SKY        = (20,  24,  36)     # Background / sky
C_GRASS      = (28,  60,  28)     # Roadside grass
C_ASPHALT    = (50,  52,  58)     # Road surface
C_SHOULDER   = (80,  76,  68)     # Road shoulder / gravel
C_LINE_WHITE = (220, 220, 210)    # Edge lines
C_LINE_DASH  = (200, 180,  20)    # Dashed centre line
C_TREE_TRUNK = (80,  55,  30)
C_TREE_TOP   = (30,  90,  30)
C_TREE_TOP2  = (20, 110,  40)
C_RUMBLE     = (100, 95,  80)     # Rumble strip highlight


class Tree:
    """A simple roadside tree with randomized position and size."""
    def __init__(self, x: float, y: float, scale: float = 1.0, side: str = "left"):
        self.base_x = x       # original X position relative to road
        self.y = y
        self.trunk_w = int(6  * scale)
        self.trunk_h = int(16 * scale)
        self.crown_r = int(14 * scale)
        self.color   = C_TREE_TOP if random.random() > 0.4 else C_TREE_TOP2
        self.side    = side    # "left" or "right"
        # Parallax factor: trees farther from road move slightly slower
        self.parallax = 0.92 if random.random() > 0.5 else 1.0

    def draw(self, surf: pygame.Surface, scroll_y: float, car_x_offset: float =
0.0):
        """Draw tree with proper X offset during pull-over."""
        # Apply car_x_offset to tree X position so trees move with the road
        draw_x = int(self.base_x + car_x_offset * self.parallax)
        draw_y = int(self.y - scroll_y) % (_PANEL_H + 60) - 30
        # Trunk
        pygame.draw.rect(surf, C_TREE_TRUNK,
                         (draw_x - self.trunk_w // 2,
                          draw_y,
                          self.trunk_w, self.trunk_h))
        # Crown
        pygame.draw.circle(surf, self.color,
                           (draw_x, draw_y - self.crown_r + 4),
                           self.crown_r)


class RoadRenderer:
    """
    Draws a top-down scrolling road onto a pygame.Surface.

    Usage:
        rr = RoadRenderer(surface_width, surface_height)
        # Each frame:
        rr.scroll(speed_px)          # advance scroll by speed pixels
```

```
    surf = rr.render(car_offset) # car_offset: how far right car has drifted
"""

def __init__(self, width: int = WINDOW_WIDTH, height: int = _PANEL_H):
    self.w = width
    self.h = height

    # Road geometry (all relative to width)
    self.road_w     = int(width * 0.38)          # total road width
    self.road_x     = (width - self.road_w) // 2 # left edge of road
    self.shoulder_w = int(width * 0.055)         # gravel shoulder each side
    self.lane_w     = self.road_w // 2            # each lane width
    self.centre_x   = width // 2

    # Scroll state
    self._scroll_y  = 0.0

    # Dash line parameters
    self._dash_h    = 28
    self._dash_gap  = 20
    self._dash_cycle = self._dash_h + self._dash_gap

    # Rumble strip parameters
    self._rumble_h   = 8
    self._rumble_gap = 12
    self._rumble_cycle = self._rumble_h + self._rumble_gap

    # Generate roadside trees
    random.seed(42)
    self._trees_left  = [
        Tree(
            x=random.randint(20, self.road_x - self.shoulder_w - 10),
            y=random.randint(0, _PANEL_H + 200) * 1.0,
            scale=random.uniform(0.7, 1.3),
            side="left"
        )
        for _ in range(18)
    ]
    self._trees_right = [
        Tree(
            x=random.randint(
                self.road_x + self.road_w + self.shoulder_w + 10,
                width - 20
            ),
            y=random.randint(0, _PANEL_H + 200) * 1.0,
            scale=random.uniform(0.7, 1.3),
            side="right"
        )
        for _ in range(18)
    ]

    # Pre-create surface
```

```
        self._surf = pygame.Surface((width, height))

    # ── Public API ─────────────────────────────────────────────────────────

    def scroll(self, speed_px: float) -> None:
        """Advance the road scroll by speed_px pixels."""
        self._scroll_y = (self._scroll_y + speed_px) % (self.h + 200)

    def render(self, car_x_offset: float = 0.0) -> pygame.Surface:
        """
        Draw the road and return the surface.

        Args:
            car_x_offset: How far (px) the road has shifted relative to car
                          (used for pull-over animation — road shifts right
                          as car moves to the shoulder)
        Returns:
            pygame.Surface ready to blit
        """
        surf = self._surf
        cx_off = int(car_x_offset)

        # ── Background (sky / off-road area) ──────────────────────────────
        surf.fill(C_SKY)

        # ── Grass strips ──────────────────────────────────────────────────
        left_grass_w  = max(0, self.road_x - self.shoulder_w + cx_off)
        right_grass_x = self.road_x + self.road_w + self.shoulder_w + cx_off
        pygame.draw.rect(surf, C_GRASS, (0, 0, left_grass_w, self.h))
        right_grass_w = max(0, self.w - right_grass_x)
        if right_grass_w > 0:
            pygame.draw.rect(surf, C_GRASS, (right_grass_x, 0, right_grass_w,
self.h))

        # ── Shoulders ─────────────────────────────────────────────────────
        left_shoulder_x = self.road_x - self.shoulder_w + cx_off
        right_shoulder_x = self.road_x + self.road_w + cx_off
        pygame.draw.rect(surf, C_SHOULDER,
                         (left_shoulder_x, 0, self.shoulder_w, self.h))
        pygame.draw.rect(surf, C_SHOULDER,
                         (right_shoulder_x, 0, self.shoulder_w, self.h))

        # ── Rumble strips on shoulders ────────────────────────────────────
        rumble_offset = int(self._scroll_y) % self._rumble_cycle
        y = -rumble_offset
        while y < self.h:
            # Left shoulder rumble
            pygame.draw.rect(surf, C_RUMBLE,
                             (left_shoulder_x + 2, y,
                              self.shoulder_w - 4, self._rumble_h))
            # Right shoulder rumble
            pygame.draw.rect(surf, C_RUMBLE,
```

```
                        (right_shoulder_x + 2, y,
                         self.shoulder_w - 4, self._rumble_h))
            y += self._rumble_cycle

        # ── Asphalt ──────────────────────────────────────────────────────
        pygame.draw.rect(surf, C_ASPHALT,
                         (self.road_x + cx_off, 0, self.road_w, self.h))

        # ── Edge lines (solid white) ─────────────────────────────────────
        lw = 4
        pygame.draw.rect(surf, C_LINE_WHITE,
                         (self.road_x + cx_off, 0, lw, self.h))
        pygame.draw.rect(surf, C_LINE_WHITE,
                         (self.road_x + self.road_w - lw + cx_off, 0, lw, self.h))

        # ── Dashed centre line ───────────────────────────────────────────
        dash_x = self.centre_x - 3 + cx_off
        offset = int(self._scroll_y) % self._dash_cycle
        y = -offset
        while y < self.h:
            pygame.draw.rect(surf, C_LINE_DASH,
                             (dash_x, y, 6, self._dash_h))
            y += self._dash_cycle

        # ── Trees (now properly offset with car_x_offset) ────────────────
        for tree in self._trees_left + self._trees_right:
            tree.draw(surf, self._scroll_y, car_x_offset=car_x_offset)

        return surf
        return surf
```

Key fixes:

- **Tree offset bug fixed** — Trees now receive `car_x_offset` and shift with the road during pull-over
- **Rumble strip texture** on shoulders (scrolling with road)
- **Parallax depth** — some trees scroll at 0.92× speed for depth illusion
- **Clamped grass rendering** — prevents negative-width rectangles at extreme offsets

[car_simulation.py]

```
# ============================================================================
# simulation/car_simulation.py
#
# CarSimulation — a top-down car whose behavior is driven by AnalyticsState.
#
# Car states and transitions:
#
#   DRIVING  ──(DROWSY)──→  SLOWING  ──(SLEEPING)──→  PULLING_OVER
#            ←─(ALERT)──                                  │
#                                                         ▼
#                                                     STOPPED
#                          ←─(ALERT reset)──────────────┘
```

```
#
# Physics:
#   • Speed interpolates smoothly toward target speed
#   • Pull-over: car steers right toward shoulder over ~2 seconds
#   • Hazard lights: alternating left/right blink at 1Hz when stopped
#
# Rendering:
#   • Top-down car sprite drawn with pygame primitives
#   • Speed indicator
#   • State label
#   • Hazard light flash overlay
# ============================================================================

import math
import time
import pygame
from config import (
    WINDOW_WIDTH, WINDOW_HEIGHT,
    CAR_NORMAL_SPEED, CAR_DROWSY_SPEED, CAR_PULLOVER_SPEED,
)

_PANEL_H = WINDOW_HEIGHT // 2

# ── Palette ──────────────────────────────────────────────────────────────────
C_CAR_BODY    = (220, 60,  60)    # Red car body
C_CAR_ROOF    = (180, 40,  40)
C_WINDOW      = (140, 200, 230)
C_TYRE        = (30,  30,  30)
C_HEADLIGHT   = (255, 255, 180)
C_TAILLIGHT   = (255, 60,  60)
C_HAZARD_ON   = (255, 180,  0)
C_HAZARD_OFF  = (120, 80,   0)
C_WHITE       = (255, 255, 255)
C_YELLOW      = (255, 210,  0)
C_DARK        = (20,  20,  30)

# Car geometry (pixels)
CAR_W = 36
CAR_H = 68


class Car:
    """
    Top-down car sprite with state-driven movement.

    The car is always drawn at a fixed screen position (centre lane).
    The ROAD scrolls past it to simulate forward motion.
    For the pull-over, the road shifts and the car moves right on screen.
    """

    def __init__(self, start_x: float, start_y: float):
        # Screen position (car centre)
```

```
        self.x = float(start_x)
        self.y = float(start_y)

        # Target X for pull-over (set when SLEEPING detected)
        self._target_x = start_x
        self._home_x   = start_x

        # Speed (pixels / frame of road scroll)
        self.speed        = CAR_NORMAL_SPEED
        self._target_speed = CAR_NORMAL_SPEED

        # State
        self.state = "DRIVING"   # DRIVING | SLOWING | PULLING_OVER | STOPPED

        # Hazard light state
        self._hazard_tick  = 0
        self._hazard_left  = False   # True when left hazard is lit this tick
        self._hazard_period = 18     # frames per half-cycle (~0.5s at 30fps)

        # Steering angle for visual tilt during pull-over (degrees)
        self.steer_angle = 0.0

        # Pull-over progress (0→1)
        self._pullover_progress = 0.0

        # Timestamp for pull-over initiation
        self._pullover_start_t = None

    # ── Public API ──────────────────────────────────────────────────────

    def update(self, driver_state: str, road_right_edge: float) -> float:
        """
        Update car physics for one frame.

        Args:
            driver_state:    "ALERT" | "DROWSY" | "SLEEPING"
            road_right_edge: X pixel of the road's right edge (for pull-over target)

        Returns:
            Current road scroll speed (pass to RoadRenderer.scroll())
        """
        self._update_state_machine(driver_state, road_right_edge)
        self._update_speed()
        self._update_position()
        self._update_hazards()

        return self.speed

    def draw(self, surf: pygame.Surface) -> None:
        """Draw the car at its current position onto surf."""
        cx, cy = int(self.x), int(self.y)
        angle  = self.steer_angle
```

```
        # Build car on a temp surface, then rotate and blit
        car_surf = self._build_car_surface()
        if abs(angle) > 0.5:
            car_surf = pygame.transform.rotate(car_surf, -angle)

        rect = car_surf.get_rect(center=(cx, cy))
        surf.blit(car_surf, rect)

        # Hazard flash overlay (full-screen tint when hazards active)
        if self.state == "STOPPED" and self._hazard_left:
            overlay = pygame.Surface((surf.get_width(), surf.get_height()),
pygame.SRCALPHA)
            overlay.fill((255, 160, 0, 18))
            surf.blit(overlay, (0, 0))

    # ── State Machine ──────────────────────────────────────────────────

    def _update_state_machine(self, driver_state: str, road_right_edge: float):
        if driver_state == "SLEEPING":
            if self.state not in ("PULLING_OVER", "STOPPED"):
                self.state = "PULLING_OVER"
                self._pullover_start_t = time.time()
                # Target: shoulder position (right edge of road + 40px)
                self._target_x = road_right_edge - CAR_W // 2 - 10
                self._target_speed = CAR_PULLOVER_SPEED

        elif driver_state == "DROWSY":
            if self.state == "DRIVING":
                self.state = "SLOWING"
                self._target_speed = CAR_DROWSY_SPEED

        elif driver_state == "ALERT":
            if self.state in ("SLOWING",):
                self.state = "DRIVING"
                self._target_speed = CAR_NORMAL_SPEED
                self._target_x     = self._home_x
            # Note: once STOPPED, car stays stopped until manual reset

        # Transition PULLING_OVER → STOPPED once speed is near zero
        if self.state == "PULLING_OVER":
            if abs(self.x - self._target_x) < 4 and self.speed < 0.3:
                self.state = "STOPPED"
                self.speed = 0.0
                self._target_speed = 0.0
                self.steer_angle = 0.0

    def _update_speed(self):
        """Smoothly interpolate speed toward target."""
        diff = self._target_speed - self.speed
        self.speed += diff * 0.04   # ease factor
```

```
    def _update_position(self):
        """Move car toward target X (for pull-over) with steering tilt."""
        dx = self._target_x - self.x
        if abs(dx) > 1.0:
            step = dx * 0.025
            self.x += step
            # Steering tilt proportional to lateral movement speed
            self.steer_angle = max(-18.0, min(18.0, step * 2.5))
        else:
            self.steer_angle *= 0.85   # straighten out

    def _update_hazards(self):
        if self.state == "STOPPED":
            self._hazard_tick += 1
            if self._hazard_tick >= self._hazard_period:
                self._hazard_tick = 0
                self._hazard_left = not self._hazard_left
        else:
            self._hazard_left = False
            self._hazard_tick = 0

    # ── Drawing ──────────────────────────────────────────────────────────

    def _build_car_surface(self) -> pygame.Surface:
        """Construct the top-down car sprite onto a transparent surface."""
        surf = pygame.Surface((CAR_W + 10, CAR_H + 10), pygame.SRCALPHA)
        ox, oy = 5, 5   # offset so rotated surface has room

        w, h = CAR_W, CAR_H

        # ── Tyres (drawn first, behind body) ──────────────────────────
        tyre_w, tyre_h = 7, 14
        positions = [
            (ox - 3,         oy + 8),           # front-left
            (ox + w - tyre_w + 3, oy + 8),      # front-right
            (ox - 3,         oy + h - tyre_h - 8),  # rear-left
            (ox + w - tyre_w + 3, oy + h - tyre_h - 8),  # rear-right
        ]
        for tx, ty in positions:
            pygame.draw.rect(surf, C_TYRE, (tx, ty, tyre_w, tyre_h),
border_radius=2)

        # ── Body ──────────────────────────────────────────────────────
        pygame.draw.rect(surf, C_CAR_BODY, (ox, oy, w, h), border_radius=6)

        # ── Roof (darker inner rectangle) ─────────────────────────────
        roof_margin = 6
        pygame.draw.rect(surf, C_CAR_ROOF,
                         (ox + roof_margin, oy + h // 5,
                          w - 2 * roof_margin, h * 3 // 5),
                         border_radius=4)
```

```
        # ── Windows ──────────────────────────────────────────────────────
        win_margin = 9
        pygame.draw.rect(surf, C_WINDOW,
                         (ox + win_margin, oy + h // 5 + 4,
                          w - 2 * win_margin, h * 3 // 10),
                         border_radius=3)

        # ── Headlights (front = top of surface since car faces up) ──────
        hl_y = oy + 4
        pygame.draw.rect(surf, C_HEADLIGHT, (ox + 4,     hl_y, 9,  5),
border_radius=2)
        pygame.draw.rect(surf, C_HEADLIGHT, (ox + w - 13, hl_y, 9,  5),
border_radius=2)

        # ── Taillights ───────────────────────────────────────────────────
        tl_y = oy + h - 8
        pygame.draw.rect(surf, C_TAILLIGHT, (ox + 4,     tl_y, 9,  5),
border_radius=2)
        pygame.draw.rect(surf, C_TAILLIGHT, (ox + w - 13, tl_y, 9,  5),
border_radius=2)

        # ── Hazard lights (replace tail/head lights when active) ────────
        if self.state == "STOPPED":
            h_col = C_HAZARD_ON if self._hazard_left else C_HAZARD_OFF
            pygame.draw.rect(surf, h_col, (ox + 4,     tl_y, 9, 5), border_radius=2)
            pygame.draw.rect(surf, h_col, (ox + 4,     hl_y, 9, 5), border_radius=2)
            h_col2 = C_HAZARD_ON if not self._hazard_left else C_HAZARD_OFF
            pygame.draw.rect(surf, h_col2, (ox + w - 13, tl_y, 9, 5),
border_radius=2)
            pygame.draw.rect(surf, h_col2, (ox + w - 13, hl_y, 9, 5),
border_radius=2)

        return surf

    def reset(self):
        """Reset car to initial driving state."""
        self.__init__(self._home_x, self.y)


class SimulationManager:
    """
    Top-level simulation controller.
    Owns the RoadRenderer and Car, and exposes a single update() call.

    Usage:
        sim = SimulationManager()

        # Each frame:
        surf = sim.update(driver_state, drowsiness_score, distraction_score)
        # blit surf to screen bottom half
    """
```

```
def __init__(self, width: int = WINDOW_WIDTH, height: int = _PANEL_H):
    from simulation.road_renderer import RoadRenderer

    self.w = width
    self.h = height

    self._road   = RoadRenderer(width, height)
    self._car    = Car(
        start_x=width // 2,
        start_y=height * 2 // 3,
    )
    self._surface = pygame.Surface((width, height))

    pygame.font.init()
    self._font_sm  = pygame.font.SysFont("monospace", 13)
    self._font_md  = pygame.font.SysFont("monospace", 16, bold=True)

    # Road right edge (for pull-over targeting)
    road_x     = (width - int(width * 0.38)) // 2
    road_w     = int(width * 0.38)
    self._road_right_edge = road_x + road_w

# ── Public API ──────────────────────────────────────────────────────

def update(
    self,
    driver_state:     str   = "ALERT",
    drowsiness_score: float = 0.0,
    distraction_score:float = 0.0,
) -> pygame.Surface:
    """
    Advance simulation one frame and return the rendered surface.

    Args:
        driver_state:      "ALERT" | "DROWSY" | "SLEEPING"
        drowsiness_score:  0–1 (used for speed shading)
        distraction_score: 0–1 (used for minor swerve effect)

    Returns:
        pygame.Surface (width × height) ready to blit
    """
    # ── Update car physics ──────────────────────────────────────────
    scroll_speed = self._car.update(driver_state, self._road_right_edge)

    # ── Distraction swerve: subtle sinusoidal drift ─────────────────
    if distraction_score > 0.4 and self._car.state == "DRIVING":
        import math, time
        swerve = math.sin(time.time() * 2.5) * distraction_score * 18
        self._car.x = self._car._home_x + swerve

    # ── Scroll road (car X offset creates pull-over illusion) ───────
    car_x_offset = self._car.x - self._car._home_x
```

```
        self._road.scroll(scroll_speed)

        # ── Render road ──────────────────────────────────────────────
        road_surf = self._road.render(car_x_offset=car_x_offset)
        self._surface.blit(road_surf, (0, 0))

        # ── Render car ───────────────────────────────────────────────
        self._car.draw(self._surface)

        # ── Render HUD overlay ───────────────────────────────────────
        self._draw_sim_hud(driver_state, drowsiness_score, scroll_speed)

        return self._surface

    def reset(self):
        """Reset car to initial state (e.g. new driver session)."""
        self._car.reset()

    # ── HUD ──────────────────────────────────────────────────────────

    def _draw_sim_hud(
        self,
        driver_state: str,
        drowsiness_score: float,
        speed: float,
    ):
        surf = self._surface

        # Speed display (km/h proxy — normalized from px/frame)
        kmh = int(speed / CAR_NORMAL_SPEED * 120)
        speed_str = f"{kmh} km/h"
        speed_col = (
            (220, 40,  40) if driver_state == "SLEEPING" else
            (255, 180,  0) if driver_state == "DROWSY"   else
            (80,  220, 80)
        )
        spd_surf = self._font_md.render(speed_str, True, speed_col)
        surf.blit(spd_surf, (self.w - spd_surf.get_width() - 12, 10))

        # Car state label
        state_str = f"[ {self._car.state} ]"
        st_surf = self._font_sm.render(state_str, True, (160, 160, 180))
        surf.blit(st_surf, (12, 10))

        # STOPPED banner
        if self._car.state == "STOPPED":
            font_lg = pygame.font.SysFont("monospace", 24, bold=True)
            msg = font_lg.render("VEHICLE STOPPED — HAZARDS ON", True, C_HAZARD_ON)
            bx  = self.w // 2 - msg.get_width() // 2
            bg  = pygame.Surface((msg.get_width() + 20, msg.get_height() + 8),
pygame.SRCALPHA)
            bg.fill((0, 0, 0, 160))
```

```
            surf.blit(bg,  (bx - 10, self.h - 44))
            surf.blit(msg, (bx,      self.h - 40))

        # Drowsiness speed warning
        elif self._car.state == "SLOWING":
            warn = self._font_md.render("⚠ REDUCING SPEED", True, (255, 180, 0))
            surf.blit(warn, (self.w // 2 - warn.get_width() // 2, self.h - 36))

        # PULLING OVER banner
        elif self._car.state == "PULLING_OVER":
            font_lg = pygame.font.SysFont("monospace", 24, bold=True)
            msg = font_lg.render("PULLING OVER — DRIVER ASLEEP", True, (220, 40,
40))
            bx  = self.w // 2 - msg.get_width() // 2
            bg  = pygame.Surface((msg.get_width() + 20, msg.get_height() + 8),
pygame.SRCALPHA)
            bg.fill((0, 0, 0, 180))
            surf.blit(bg,  (bx - 10, self.h - 44))
            surf.blit(msg, (bx,      self.h - 40))
===
# ============================================================================
# simulation/car_simulation.py
#
# CarSimulation — a top-down car whose behavior is driven by AnalyticsState.
#
# Car states and transitions:
#
#   DRIVING  ──(DROWSY)──→  SLOWING  ──(SLEEPING)──→  PULLING_OVER
#            ←─(ALERT)──                                  │
#                                                         ▼
#                                                      STOPPED
#                          ←─(ALERT reset)──────────────┘
#
# Physics:
#   • Physics-based acceleration / deceleration (configurable rates)
#   • Pull-over: smooth S-curve (sigmoid) lateral trajectory
#   • Hazard lights: alternating left/right blink at 1Hz when stopped
#
# NPC Traffic:
#   • Ambient vehicles spawn at random intervals
#   • Scroll past the ego car at varied speeds
#   • Limited to NPC_MAX_CARS simultaneously
#
# Rendering:
#   • Top-down car sprite drawn with pygame primitives
#   • Speed indicator
#   • State label
#   • Hazard light flash overlay
# ============================================================================

import math
import time
```

```
import random
import pygame
from config import (
    WINDOW_WIDTH, WINDOW_HEIGHT,
    CAR_NORMAL_SPEED, CAR_DROWSY_SPEED, CAR_PULLOVER_SPEED,
    CAR_ACCELERATION, CAR_DECELERATION, PULLOVER_DURATION_SEC,
    NPC_SPAWN_INTERVAL, NPC_MIN_SPEED, NPC_MAX_SPEED, NPC_MAX_CARS,
)

_PANEL_H = WINDOW_HEIGHT // 2

# ── Palette ──────────────────────────────────────────────────────────────
C_CAR_BODY    = (220, 60,  60)    # Red car body
C_CAR_ROOF    = (180, 40,  40)
C_WINDOW      = (140, 200, 230)
C_TYRE        = (30,  30,  30)
C_HEADLIGHT   = (255, 255, 180)
C_TAILLIGHT   = (255, 60,  60)
C_HAZARD_ON   = (255, 180,  0)
C_HAZARD_OFF  = (120, 80,   0)
C_WHITE       = (255, 255, 255)
C_YELLOW      = (255, 210,  0)
C_DARK        = (20,  20,  30)

# NPC car colors (variety)
NPC_COLORS = [
    (60,  120, 200),   # Blue
    (200, 200, 210),   # Silver
    (40,  40,  50),    # Dark gray
    (180, 160,  40),   # Gold
    (80,  180,  80),   # Green
    (160, 60,  160),   # Purple
]

# Car geometry (pixels)
CAR_W = 36
CAR_H = 68

NPC_W = 32
NPC_H = 60


# ── NPC Car ──────────────────────────────────────────────────────────────

class NPCCar:
    """
    A simple ambient NPC vehicle that scrolls past the ego car.
    NPCs appear at the top of the screen and move downward (opposite lane)
    or appear at the bottom and move upward (same lane, slower pass).
    """

    def __init__(self, road_x: int, road_w: int, panel_h: int):
```

```
        self.color = random.choice(NPC_COLORS)
        self.speed = random.uniform(NPC_MIN_SPEED, NPC_MAX_SPEED)

        # Choose lane: left lane (oncoming) or right lane (same direction, passing)
        lane_half = road_w // 4
        if random.random() < 0.65:
            # Oncoming traffic (left lane) — moves downward
            self.x = float(road_x + lane_half)
            self.y = float(-NPC_H - random.randint(20, 120))
            self.direction = 1     # +1 = moving down screen
            self.relative_speed = self.speed + CAR_NORMAL_SPEED  # opposing
        else:
            # Same-direction traffic (right lane) — moves up slower
            self.x = float(road_x + road_w - lane_half)
            self.y = float(panel_h + NPC_H + random.randint(20, 120))
            self.direction = -1    # -1 = moving up screen
            self.relative_speed = max(0.5, self.speed - CAR_NORMAL_SPEED * 0.6)

        self.alive = True

    def update(self, ego_speed: float, car_x_offset: float = 0.0):
        """Advance NPC position, kill if off-screen."""
        if self.direction == 1:
            # Oncoming: scroll down at ego_speed + NPC speed
            self.y += ego_speed + self.speed
        else:
            # Same direction: relative motion = ego_speed - NPC speed
            self.y -= max(0.3, ego_speed - self.speed * 0.5)

        # Kill if off-screen
        if self.y > _PANEL_H + NPC_H + 50 or self.y < -NPC_H - 200:
            self.alive = False

    def draw(self, surf: pygame.Surface, car_x_offset: float = 0.0):
        """Draw a simple NPC car sprite."""
        cx = int(self.x + car_x_offset)
        cy = int(self.y)

        w, h = NPC_W, NPC_H
        half_w = w // 2

        # Body
        body_rect = (cx - half_w, cy - h // 2, w, h)
        pygame.draw.rect(surf, self.color, body_rect, border_radius=5)

        # Roof (darker)
        roof_color = tuple(max(0, c - 40) for c in self.color)
        roof_margin = 5
        pygame.draw.rect(surf, roof_color,
                         (cx - half_w + roof_margin, cy - h // 5,
                          w - 2 * roof_margin, h * 2 // 5),
                         border_radius=3)
```

```
        # Window
        win_margin = 7
        pygame.draw.rect(surf, C_WINDOW,
                         (cx - half_w + win_margin, cy - h // 5 + 3,
                          w - 2 * win_margin, h // 4),
                         border_radius=2)

        # Headlights / taillights
        if self.direction == 1:
            # Oncoming — show headlights at bottom (facing us)
            pygame.draw.rect(surf, C_HEADLIGHT,
                             (cx - half_w + 3, cy + h // 2 - 5, 7, 4),
border_radius=1)
            pygame.draw.rect(surf, C_HEADLIGHT,
                             (cx + half_w - 10, cy + h // 2 - 5, 7, 4),
border_radius=1)
        else:
            # Same direction — show taillights at bottom
            pygame.draw.rect(surf, C_TAILLIGHT,
                             (cx - half_w + 3, cy + h // 2 - 5, 7, 4),
border_radius=1)
            pygame.draw.rect(surf, C_TAILLIGHT,
                             (cx + half_w - 10, cy + h // 2 - 5, 7, 4),
border_radius=1)


# ── Ego Car ──────────────────────────────────────────────────────────

class Car:
    """
    Top-down car sprite with state-driven movement.

    The car is always drawn at a fixed screen position (centre lane).
    The ROAD scrolls past it to simulate forward motion.
    For the pull-over, the road shifts and the car moves right on screen.
    """

    def __init__(self, start_x: float, start_y: float):
        # Screen position (car centre)
        self.x = float(start_x)
        self.y = float(start_y)

        # Target X for pull-over (set when SLEEPING detected)
        self._target_x = start_x
        self._home_x   = start_x

        # Speed (pixels / frame of road scroll)
        self.speed        = CAR_NORMAL_SPEED
        self._target_speed = CAR_NORMAL_SPEED

        # State
```

```
        self.state = "DRIVING"   # DRIVING | SLOWING | PULLING_OVER | STOPPED

        # Hazard light state
        self._hazard_tick  = 0
        self._hazard_left  = False   # True when left hazard is lit this tick
        self._hazard_period = 18     # frames per half-cycle (~0.5s at 30fps)

        # Steering angle for visual tilt during pull-over (degrees)
        self.steer_angle = 0.0

        # Pull-over S-curve progress (0→1)
        self._pullover_progress = 0.0
        self._pullover_start_x  = start_x

        # Timestamp for pull-over initiation
        self._pullover_start_t = None

    # ── Public API ──────────────────────────────────────────────────────

    def update(self, driver_state: str, road_right_edge: float) -> float:
        """
        Update car physics for one frame.

        Args:
            driver_state:    "ALERT" | "DROWSY" | "SLEEPING"
            road_right_edge: X pixel of the road's right edge (for pull-over target)

        Returns:
            Current road scroll speed (pass to RoadRenderer.scroll())
        """
        self._update_state_machine(driver_state, road_right_edge)
        self._update_speed()
        self._update_position()
        self._update_hazards()

        return self.speed

    def draw(self, surf: pygame.Surface) -> None:
        """Draw the car at its current position onto surf."""
        cx, cy = int(self.x), int(self.y)
        angle  = self.steer_angle

        # Build car on a temp surface, then rotate and blit
        car_surf = self._build_car_surface()
        if abs(angle) > 0.5:
            car_surf = pygame.transform.rotate(car_surf, -angle)

        rect = car_surf.get_rect(center=(cx, cy))
        surf.blit(car_surf, rect)

        # Hazard flash overlay (full-screen tint when hazards active)
        if self.state == "STOPPED" and self._hazard_left:
```

```
            overlay = pygame.Surface((surf.get_width(), surf.get_height()),
pygame.SRCALPHA)
            overlay.fill((255, 160, 0, 18))
            surf.blit(overlay, (0, 0))

    # ── State Machine ──────────────────────────────────────────────────

    def _update_state_machine(self, driver_state: str, road_right_edge: float):
        if driver_state == "SLEEPING":
            if self.state not in ("PULLING_OVER", "STOPPED"):
                self.state = "PULLING_OVER"
                self._pullover_start_t = time.time()
                self._pullover_start_x = self.x
                # Target: shoulder position (right edge of road + 40px)
                self._target_x = road_right_edge - CAR_W // 2 - 10
                self._target_speed = CAR_PULLOVER_SPEED
                self._pullover_progress = 0.0

        elif driver_state == "DROWSY":
            if self.state == "DRIVING":
                self.state = "SLOWING"
                self._target_speed = CAR_DROWSY_SPEED

        elif driver_state == "ALERT":
            if self.state in ("SLOWING",):
                self.state = "DRIVING"
                self._target_speed = CAR_NORMAL_SPEED
                self._target_x     = self._home_x
            # Note: once STOPPED, car stays stopped until manual reset

        # Transition PULLING_OVER → STOPPED once speed is near zero
        if self.state == "PULLING_OVER":
            if abs(self.x - self._target_x) < 4 and self.speed < 0.3:
                self.state = "STOPPED"
                self.speed = 0.0
                self._target_speed = 0.0
                self.steer_angle = 0.0

    def _update_speed(self):
        """Physics-based acceleration/deceleration toward target speed."""
        diff = self._target_speed - self.speed
        if abs(diff) < 0.01:
            self.speed = self._target_speed
            return

        if diff > 0:
            # Accelerating
            self.speed += min(diff, CAR_ACCELERATION)
        else:
            # Decelerating
            self.speed += max(diff, -CAR_DECELERATION)
```

```
def _update_position(self):
    """
    Move car toward target X using S-curve (sigmoid) for pull-over.
    Produces a smooth, realistic lateral trajectory.
    """
    if self.state == "PULLING_OVER" and self._pullover_start_t is not None:
        # S-curve: sigmoid function maps t ∈ [0, duration] → progress ∈ [0, 1]
        elapsed = time.time() - self._pullover_start_t
        t_norm = min(1.0, elapsed / PULLOVER_DURATION_SEC)
        # Sigmoid: s(t) = 1 / (1 + e^(-12*(t-0.5)))
        sigmoid = 1.0 / (1.0 + math.exp(-12.0 * (t_norm - 0.5)))
        self._pullover_progress = sigmoid

        total_dx = self._target_x - self._pullover_start_x
        self.x = self._pullover_start_x + total_dx * sigmoid

        # Steering tilt proportional to derivative of sigmoid
        deriv = sigmoid * (1.0 - sigmoid) * 12.0
        self.steer_angle = max(-18.0, min(18.0, deriv * total_dx * 0.015))
    elif self.state not in ("PULLING_OVER", "STOPPED"):
        # Move back to home X if not pulling over
        dx = self._target_x - self.x
        if abs(dx) > 1.0:
            step = dx * 0.025
            self.x += step
            self.steer_angle = max(-18.0, min(18.0, step * 2.5))
        else:
            self.steer_angle *= 0.85   # straighten out

def _update_hazards(self):
    if self.state == "STOPPED":
        self._hazard_tick += 1
        if self._hazard_tick >= self._hazard_period:
            self._hazard_tick = 0
            self._hazard_left = not self._hazard_left
    else:
        self._hazard_left = False
        self._hazard_tick = 0

# ── Drawing ──────────────────────────────────────────────────────────

def _build_car_surface(self) -> pygame.Surface:
    """Construct the top-down car sprite onto a transparent surface."""
    surf = pygame.Surface((CAR_W + 10, CAR_H + 10), pygame.SRCALPHA)
    ox, oy = 5, 5   # offset so rotated surface has room

    w, h = CAR_W, CAR_H

    # ── Tyres (drawn first, behind body) ──────────────────────────
    tyre_w, tyre_h = 7, 14
    positions = [
        (ox - 3,         oy + 8),           # front-left
```

```
            (ox + w - tyre_w + 3, oy + 8),      # front-right
            (ox - 3,         oy + h - tyre_h - 8),  # rear-left
            (ox + w - tyre_w + 3, oy + h - tyre_h - 8),  # rear-right
        ]
        for tx, ty in positions:
            pygame.draw.rect(surf, C_TYRE, (tx, ty, tyre_w, tyre_h),
border_radius=2)

        # ── Body ──────────────────────────────────────────────────────────
        pygame.draw.rect(surf, C_CAR_BODY, (ox, oy, w, h), border_radius=6)

        # ── Roof (darker inner rectangle) ─────────────────────────────────
        roof_margin = 6
        pygame.draw.rect(surf, C_CAR_ROOF,
                         (ox + roof_margin, oy + h // 5,
                          w - 2 * roof_margin, h * 3 // 5),
                         border_radius=4)

        # ── Windows ───────────────────────────────────────────────────────
        win_margin = 9
        pygame.draw.rect(surf, C_WINDOW,
                         (ox + win_margin, oy + h // 5 + 4,
                          w - 2 * win_margin, h * 3 // 10),
                         border_radius=3)

        # ── Headlights (front = top of surface since car faces up) ────────
        hl_y = oy + 4
        pygame.draw.rect(surf, C_HEADLIGHT, (ox + 4,     hl_y, 9,  5),
border_radius=2)
        pygame.draw.rect(surf, C_HEADLIGHT, (ox + w - 13, hl_y, 9,  5),
border_radius=2)

        # ── Taillights ────────────────────────────────────────────────────
        tl_y = oy + h - 8
        pygame.draw.rect(surf, C_TAILLIGHT, (ox + 4,     tl_y, 9,  5),
border_radius=2)
        pygame.draw.rect(surf, C_TAILLIGHT, (ox + w - 13, tl_y, 9,  5),
border_radius=2)

        # ── Hazard lights (replace tail/head lights when active) ──────────
        if self.state == "STOPPED":
            h_col = C_HAZARD_ON if self._hazard_left else C_HAZARD_OFF
            pygame.draw.rect(surf, h_col, (ox + 4,     tl_y, 9, 5), border_radius=2)
            pygame.draw.rect(surf, h_col, (ox + 4,     hl_y, 9, 5), border_radius=2)
            h_col2 = C_HAZARD_ON if not self._hazard_left else C_HAZARD_OFF
            pygame.draw.rect(surf, h_col2, (ox + w - 13, tl_y, 9, 5),
border_radius=2)
            pygame.draw.rect(surf, h_col2, (ox + w - 13, hl_y, 9, 5),
border_radius=2)

        return surf
```

```
    def reset(self):
        """Reset car to initial driving state."""
        self.__init__(self._home_x, self.y)


# ── Traffic Manager ──────────────────────────────────────────────────

class TrafficManager:
    """
    Manages ambient NPC vehicles for the simulation.
    Spawns and despawns cars at random intervals.
    """

    def __init__(self, road_x: int, road_w: int, panel_h: int):
        self._road_x = road_x
        self._road_w = road_w
        self._panel_h = panel_h
        self._npcs: list[NPCCar] = []
        self._next_spawn_t = time.time() + random.uniform(*NPC_SPAWN_INTERVAL)

    def update(self, ego_speed: float, car_x_offset: float = 0.0):
        """Update all NPCs and spawn new ones if needed."""
        now = time.time()

        # Spawn new NPC if timer expired and below max
        if now >= self._next_spawn_t and len(self._npcs) < NPC_MAX_CARS:
            if ego_speed > 0.5:   # only spawn when car is moving
                npc = NPCCar(self._road_x, self._road_w, self._panel_h)
                self._npcs.append(npc)
            self._next_spawn_t = now + random.uniform(*NPC_SPAWN_INTERVAL)

        # Update and filter dead NPCs
        for npc in self._npcs:
            npc.update(ego_speed, car_x_offset)
        self._npcs = [npc for npc in self._npcs if npc.alive]

    def draw(self, surf: pygame.Surface, car_x_offset: float = 0.0):
        """Draw all NPC cars."""
        for npc in self._npcs:
            npc.draw(surf, car_x_offset)

    def reset(self):
        self._npcs.clear()
        self._next_spawn_t = time.time() + random.uniform(*NPC_SPAWN_INTERVAL)


# ── Simulation Manager ──────────────────────────────────────────────

class SimulationManager:
    """
    Top-level simulation controller.
    Owns the RoadRenderer, Car, and TrafficManager, and exposes a single update()
```

```
call.

    Usage:
        sim = SimulationManager()

        # Each frame:
        surf = sim.update(driver_state, drowsiness_score, distraction_score)
        # blit surf to screen bottom half
    """

    def __init__(self, width: int = WINDOW_WIDTH, height: int = _PANEL_H):
        from simulation.road_renderer import RoadRenderer

        self.w = width
        self.h = height

        self._road   = RoadRenderer(width, height)
        self._car    = Car(
            start_x=width // 2,
            start_y=height * 2 // 3,
        )
        self._surface = pygame.Surface((width, height))

        pygame.font.init()
        self._font_sm  = pygame.font.SysFont("monospace", 13)
        self._font_md  = pygame.font.SysFont("monospace", 16, bold=True)

        # Road geometry for pull-over targeting and traffic
        road_x     = (width - int(width * 0.38)) // 2
        road_w     = int(width * 0.38)
        self._road_right_edge = road_x + road_w
        self._road_x = road_x
        self._road_w = road_w

        # NPC traffic manager
        self._traffic = TrafficManager(road_x, road_w, height)

    # ── Public API ──────────────────────────────────────────────────────

    def update(
        self,
        driver_state:     str   = "ALERT",
        drowsiness_score: float = 0.0,
        distraction_score:float = 0.0,
    ) -> pygame.Surface:
        """
        Advance simulation one frame and return the rendered surface.

        Args:
            driver_state:      "ALERT" | "DROWSY" | "SLEEPING"
            drowsiness_score:  0–1 (used for speed shading)
            distraction_score: 0–1 (used for minor swerve effect)
```

```
    Returns:
        pygame.Surface (width × height) ready to blit
    """
    # ── Update car physics ──────────────────────────────────────────
    scroll_speed = self._car.update(driver_state, self._road_right_edge)

    # ── Distraction swerve: subtle sinusoidal drift ─────────────────
    if distraction_score > 0.4 and self._car.state == "DRIVING":
        swerve = math.sin(time.time() * 2.5) * distraction_score * 18
        self._car.x = self._car._home_x + swerve

    # ── Scroll road (car X offset creates pull-over illusion) ───────
    car_x_offset = self._car.x - self._car._home_x
    self._road.scroll(scroll_speed)

    # ── Update NPC traffic ──────────────────────────────────────────
    self._traffic.update(scroll_speed, car_x_offset)

    # ── Render road ─────────────────────────────────────────────────
    road_surf = self._road.render(car_x_offset=car_x_offset)
    self._surface.blit(road_surf, (0, 0))

    # ── Render NPC cars ─────────────────────────────────────────────
    self._traffic.draw(self._surface, car_x_offset)

    # ── Render ego car ──────────────────────────────────────────────
    self._car.draw(self._surface)

    # ── Render HUD overlay ──────────────────────────────────────────
    self._draw_sim_hud(driver_state, drowsiness_score, scroll_speed)

    return self._surface

def reset(self):
    """Reset car to initial state (e.g. new driver session)."""
    self._car.reset()
    self._traffic.reset()

# ── HUD ──────────────────────────────────────────────────────────────

def _draw_sim_hud(
    self,
    driver_state: str,
    drowsiness_score: float,
    speed: float,
):
    surf = self._surface

    # Speed display (km/h proxy — normalized from px/frame)
    kmh = int(speed / CAR_NORMAL_SPEED * 120)
    speed_str = f"{kmh} km/h"
```

```
        speed_col = (
            (220, 40,  40) if driver_state == "SLEEPING" else
            (255, 180,  0) if driver_state == "DROWSY"   else
            (80,  220, 80)
        )
        spd_surf = self._font_md.render(speed_str, True, speed_col)
        surf.blit(spd_surf, (self.w - spd_surf.get_width() - 12, 10))

        # Car state label
        state_str = f"[ {self._car.state} ]"
        st_surf = self._font_sm.render(state_str, True, (160, 160, 180))
        surf.blit(st_surf, (12, 10))

        # STOPPED banner
        if self._car.state == "STOPPED":
            font_lg = pygame.font.SysFont("monospace", 24, bold=True)
            msg = font_lg.render("VEHICLE STOPPED — HAZARDS ON", True, C_HAZARD_ON)
            bx  = self.w // 2 - msg.get_width() // 2
            bg  = pygame.Surface((msg.get_width() + 20, msg.get_height() + 8),
pygame.SRCALPHA)
            bg.fill((0, 0, 0, 160))
            surf.blit(bg,  (bx - 10, self.h - 44))
            surf.blit(msg, (bx,      self.h - 40))

        # Drowsiness speed warning
        elif self._car.state == "SLOWING":
            warn = self._font_md.render("⚠ REDUCING SPEED", True, (255, 180, 0))
            surf.blit(warn, (self.w // 2 - warn.get_width() // 2, self.h - 36))

        # PULLING OVER banner
        elif self._car.state == "PULLING_OVER":
            font_lg = pygame.font.SysFont("monospace", 24, bold=True)
            msg = font_lg.render("PULLING OVER — DRIVER ASLEEP", True, (220, 40,
40))
            bx  = self.w // 2 - msg.get_width() // 2
            bg  = pygame.Surface((msg.get_width() + 20, msg.get_height() + 8),
pygame.SRCALPHA)
            bg.fill((0, 0, 0, 180))
            surf.blit(bg,  (bx - 10, self.h - 44))
            surf.blit(msg, (bx,      self.h - 40))
            surf.blit(msg, (bx,      self.h - 40))
```

Key additions:

- **NPCCar class** — ambient vehicles in opposing/same lane with varied colors
- **TrafficManager** — spawns up to 3 NPCs at random intervals when car is moving
- **Physics-based speed** — real acceleration/deceleration rates instead of flat ease
- **S-curve pull-over** — sigmoid function for smooth lateral trajectory

---

2. Visual Telemetry & AR Overlays

hud_renderer.py

```
# ============================================================================
# ui/hud_renderer.py
#
# HUDRenderer — all PyGame drawing primitives for the DMS top-half feed.
#
# Responsibilities:
#   • Convert OpenCV BGR frame → PyGame surface
#   • Draw face mesh tessellation overlay
#   • Draw YOLO bounding boxes
#   • Render telemetry panel (EAR, pose, gaze, blink)
#   • Render score bars (drowsiness, distraction, PERCLOS)
#   • Render state badges and alert banners
#   • Render gaze crosshair
# ============================================================================

import cv2
import numpy as np
import pygame
import mediapipe as mp

from config import (
    CAMERA_WIDTH, CAMERA_HEIGHT,
    COLOR_GREEN, COLOR_YELLOW, COLOR_RED, COLOR_WHITE,
    COLOR_BLACK, COLOR_DARK_GRAY, COLOR_CYAN, COLOR_ORANGE,
    HUD_FONT_SIZE, ALERT_FONT_SIZE,
)
from dms_engine.data_structures import AnalyticsState

# MediaPipe face mesh connections
_FACE_CONTOURS = list(mp.solutions.face_mesh.FACEMESH_CONTOURS)
_FACE_TESSELATION = list(mp.solutions.face_mesh.FACEMESH_TESSELATION)

# State → badge color mapping
_DRIVER_STATE_COLORS = {
    "ALERT":    (40,  200,  80),
    "DROWSY":   (255, 180,   0),
    "SLEEPING": (220,  40,  40),
}
_ATTENTION_STATE_COLORS = {
    "FOCUSED":    (40,  200,  80),
    "DISTRACTED": (220,  40,  40),
}

# Emotion → color
_EMOTION_COLORS = {
    "happy":    (40,  200,  80),
    "neutral":  (180, 180, 180),
    "angry":    (220,  40,  40),
    "sad":      (100, 140, 220),
    "surprise": (40,  210, 210),
    "fear":     (180,  60, 200),
```

```
    "disgust":  (60,  160,  80),
}


def _bgr_frame_to_surface(frame_bgr: np.ndarray, target_w: int, target_h: int) ->
pygame.Surface:
    """Convert a BGR numpy frame to a pygame Surface, resized to target
dimensions."""
    frame_rgb = cv2.cvtColor(frame_bgr, cv2.COLOR_BGR2RGB)
    if frame_rgb.shape[1] != target_w or frame_rgb.shape[0] != target_h:
        frame_rgb = cv2.resize(frame_rgb, (target_w, target_h))
    # PyGame expects (width, height, channels) with shape[1], shape[0]
    surface = pygame.surfarray.make_surface(frame_rgb.swapaxes(0, 1))
    return surface


class HUDRenderer:
    """
    Renders all HUD elements onto a PyGame surface.

    Usage:
        renderer = HUDRenderer(screen_width, panel_height)
        renderer.render(screen, state, frame_bgr, y_offset=0)
    """

    def __init__(self, panel_width: int, panel_height: int):
        self.pw = panel_width
        self.ph = panel_height

        # Feed area: left 2/3 of panel
        self.feed_w = int(panel_width * 0.62)
        self.feed_h = panel_height

        # Telemetry panel: right 1/3
        self.telem_x = self.feed_w
        self.telem_w = panel_width - self.feed_w

        pygame.font.init()
        self._font_sm  = pygame.font.SysFont("monospace", 13)
        self._font_md  = pygame.font.SysFont("monospace", HUD_FONT_SIZE)
        self._font_lg  = pygame.font.SysFont("monospace", ALERT_FONT_SIZE,
bold=True)
        self._font_badge = pygame.font.SysFont("monospace", 16, bold=True)

        # Pre-create a dark surface for the telemetry panel background
        self._telem_bg = pygame.Surface((self.telem_w, panel_height),
pygame.SRCALPHA)
        self._telem_bg.fill((18, 18, 28, 210))

        # Score bar background
        self._bar_bg_color = (50, 50, 60)
```

```
# ── Public API ──────────────────────────────────────────────────────

def render(
    self,
    screen: pygame.Surface,
    state: AnalyticsState,
    frame_bgr: np.ndarray,
    y_offset: int = 0,
) -> None:
    """
    Render the complete top-half DMS panel.

    Args:
        screen:    The main PyGame display surface
        state:     Latest AnalyticsState from DMSCore
        frame_bgr: Raw BGR camera frame
        y_offset:  Vertical offset on screen (0 for top half)
    """
    # ── 1. Camera feed ──────────────────────────────────────────────
    feed_surf = _bgr_frame_to_surface(frame_bgr, self.feed_w, self.feed_h)
    screen.blit(feed_surf, (0, y_offset))

    # ── 2. Face mesh overlay on feed ────────────────────────────────
    geo = state.geometry
    if geo.face_detected and geo.landmarks is not None:
        self._draw_face_mesh(screen, geo.landmarks, y_offset)
        self._draw_gaze_point(screen, geo, y_offset)

    # ── 3. YOLO bounding boxes ──────────────────────────────────────
    self._draw_yolo_boxes(screen, state.detection, y_offset)

    # ── 4. Telemetry panel background ───────────────────────────────
    screen.blit(self._telem_bg, (self.telem_x, y_offset))

    # ── 5. Telemetry content ────────────────────────────────────────
    self._draw_telemetry(screen, state, y_offset)

    # ── 6. Score bars (bottom of feed) ──────────────────────────────
    self._draw_score_bars(screen, state, y_offset)

    # ── 7. State badges ─────────────────────────────────────────────
    self._draw_state_badges(screen, state, y_offset)

    # ── 8. Alert banner (if critical) ───────────────────────────────
    self._draw_alert_banner(screen, state, y_offset)

    # ── 9. Panel border ─────────────────────────────────────────────
    pygame.draw.rect(screen, (60, 60, 80),
                     (0, y_offset, self.pw, self.ph), 2)
    pygame.draw.line(screen, (60, 60, 80),
                     (self.telem_x, y_offset),
                     (self.telem_x, y_offset + self.ph), 2)
```

```
# ── Face Mesh ──────────────────────────────────────────────────────

def _draw_face_mesh(self, screen, landmarks, y_offset):
    lm = landmarks
    scale_x = self.feed_w
    scale_y = self.feed_h

    # Tessellation (very light, thin lines)
    for i, j in _FACE_TESSELATION[::3]:   # draw every 3rd for performance
        if i < len(lm) and j < len(lm):
            x1 = int(lm[i][0] * scale_x)
            y1 = int(lm[i][1] * scale_y) + y_offset
            x2 = int(lm[j][0] * scale_x)
            y2 = int(lm[j][1] * scale_y) + y_offset
            pygame.draw.line(screen, (40, 80, 40), (x1, y1), (x2, y2), 1)

    # Contours (slightly brighter)
    for i, j in _FACE_CONTOURS:
        if i < len(lm) and j < len(lm):
            x1 = int(lm[i][0] * scale_x)
            y1 = int(lm[i][1] * scale_y) + y_offset
            x2 = int(lm[j][0] * scale_x)
            y2 = int(lm[j][1] * scale_y) + y_offset
            pygame.draw.line(screen, (60, 160, 60), (x1, y1), (x2, y2), 1)

    # Iris landmarks
    from config import LEFT_IRIS_IDX, RIGHT_IRIS_IDX
    for idx in LEFT_IRIS_IDX + RIGHT_IRIS_IDX:
        if idx < len(lm):
            x = int(lm[idx][0] * scale_x)
            y = int(lm[idx][1] * scale_y) + y_offset
            pygame.draw.circle(screen, (0, 220, 200), (x, y), 2)

def _draw_gaze_point(self, screen, geo, y_offset):
    gx, gy = geo.gaze.gaze_point_px
    # Scale to feed dimensions
    gx = int(gx * self.feed_w / CAMERA_WIDTH)
    gy = int(gy * self.feed_h / CAMERA_HEIGHT) + y_offset
    pygame.draw.circle(screen, (0, 220, 255), (gx, gy), 7)
    pygame.draw.circle(screen, (0, 160, 200), (gx, gy), 11, 2)
    # Crosshair
    pygame.draw.line(screen, (0, 200, 200), (gx - 15, gy), (gx + 15, gy), 1)
    pygame.draw.line(screen, (0, 200, 200), (gx, gy - 15), (gx, gy + 15), 1)

# ── YOLO Boxes ─────────────────────────────────────────────────────

def _draw_yolo_boxes(self, screen, detection, y_offset):
    for box in detection.boxes:
        x1, y1, x2, y2 = box.bbox
        # Scale to feed dimensions
        sx = self.feed_w / CAMERA_WIDTH
```

```
        sy = self.feed_h / CAMERA_HEIGHT
        rx1, ry1 = int(x1 * sx), int(y1 * sy) + y_offset
        rx2, ry2 = int(x2 * sx), int(y2 * sy) + y_offset

        color = COLOR_RED if box.label == "phone" else COLOR_CYAN
        pygame.draw.rect(screen, color, (rx1, ry1, rx2 - rx1, ry2 - ry1), 2)

        label = f"{box.label} {box.confidence:.2f}"
        label_surf = self._font_sm.render(label, True, COLOR_BLACK)
        label_bg   = pygame.Surface((label_surf.get_width() + 4, 16))
        label_bg.fill(color)
        screen.blit(label_bg,   (rx1, ry1 - 16))
        screen.blit(label_surf, (rx1 + 2, ry1 - 15))

# ── Telemetry Panel ──────────────────────────────────────────────────

def _draw_telemetry(self, screen, state: AnalyticsState, y_offset):
    geo = state.geometry
    tx  = self.telem_x + 10
    line_h = 20

    def txt(text, row, color=(180, 220, 180), bold=False):
        font = self._font_md
        surf = font.render(text, True, color)
        screen.blit(surf, (tx, y_offset + 8 + row * line_h))

    def section(label, row):
        surf = self._font_sm.render(f"── {label} ──", True, (100, 120, 200))
        screen.blit(surf, (tx, y_offset + 8 + row * line_h))

    # ── Head Pose ────────────────────────────────────────────────────
    section("HEAD POSE", 0)
    if geo.face_detected and geo.head_pose.valid:
        hp = geo.head_pose
        txt(f"Yaw  : {hp.yaw:+6.1f}°", 1)
        txt(f"Pitch: {hp.pitch:+6.1f}°", 2)
        txt(f"Roll : {hp.roll:+6.1f}°",  3)
        txt(f"Z    : {hp.z_mm:6.0f}mm",  4)
    else:
        txt("No face detected", 1, (160, 80, 80))

    # ── Eyes ─────────────────────────────────────────────────────────
    section("EYES", 6)
    if geo.face_detected:
        le = geo.left_eye
        re = geo.right_eye
        le_col = COLOR_RED if le.is_closed else (180, 220, 180)
        re_col = COLOR_RED if re.is_closed else (180, 220, 180)
        txt(f"L EAR: {le.ear:.3f} ({le.ear_percentile:.0f}%)", 7,  le_col)
        txt(f"R EAR: {re.ear:.3f} ({re.ear_percentile:.0f}%)", 8,  re_col)
        txt(f"Mean : {geo.mean_ear:.3f}", 9)
```

```
        # ── Blink ──────────────────────────────────────────────────────────
        section("BLINK", 11)
        if geo.face_detected:
            bk = geo.blink
            txt(f"Rate : {bk.blinks_per_second:.2f}/s", 12)
            txt(f"Total: {bk.total_blinks}",             13)

        # ── Gaze ───────────────────────────────────────────────────────────
        section("GAZE", 15)
        if geo.face_detected:
            gz = geo.gaze
            txt(f"H: {gz.horizontal:+.2f}  V: {gz.vertical:+.2f}", 16)
            txt(f"Dev: {gz.deviation:.3f}", 17)

        # ── Emotion ────────────────────────────────────────────────────────
        section("EMOTION", 19)
        emo   = state.fer.emotion_label
        e_col = _EMOTION_COLORS.get(emo, (180, 180, 180))
        txt(f"{emo.upper()} ({state.fer.confidence*100:.0f}%)", 20, e_col)

        # ── Action ─────────────────────────────────────────────────────────
        section("ACTION", 22)
        act_label = state.action.action_label.replace("_", " ").upper()
        txt(f"{act_label}", 23, (180, 220, 255))

    # ── Score Bars ─────────────────────────────────────────────────────────

    def _draw_score_bars(self, screen, state: AnalyticsState, y_offset):
        bar_y   = y_offset + self.ph - 72
        bar_h   = 14
        bar_w   = self.feed_w - 20
        bar_x   = 10

        def draw_bar(bx, by, bw, bh, score, label, color):
            # Background
            pygame.draw.rect(screen, self._bar_bg_color, (bx, by, bw, bh))
            # Fill
            fill = int(bw * max(0.0, min(1.0, score)))
            if fill > 0:
                pygame.draw.rect(screen, color, (bx, by, fill, bh))
            # Border
            pygame.draw.rect(screen, (100, 100, 110), (bx, by, bw, bh), 1)
            # Label
            lbl = self._font_sm.render(f"{label}: {score*100:.0f}%", True, (210,
210, 210))
            screen.blit(lbl, (bx, by - 14))

        # Drowsiness
        d_col = _DRIVER_STATE_COLORS.get(state.driver_state, COLOR_GREEN)
        draw_bar(bar_x, bar_y, bar_w, bar_h,
                 state.drowsiness_score, "DROWSINESS", d_col)
```

```
        # Distraction
        a_col = _ATTENTION_STATE_COLORS.get(state.attention_state, COLOR_GREEN)
        draw_bar(bar_x, bar_y + 26, bar_w, bar_h,
                 state.distraction_score, "DISTRACTION", a_col)

        # PERCLOS (thin accent bar)
        draw_bar(bar_x, bar_y + 46, bar_w, 6,
                 state.perclos, "PERCLOS", (180, 100, 255))

    # ── State Badges ──────────────────────────────────────────────────

    def _draw_state_badges(self, screen, state: AnalyticsState, y_offset):
        def badge(text, bx, by, color):
            surf = self._font_badge.render(text, True, COLOR_BLACK)
            w, h = surf.get_width() + 12, surf.get_height() + 6
            pygame.draw.rect(screen, color, (bx, by, w, h), border_radius=4)
            screen.blit(surf, (bx + 6, by + 3))

        d_col = _DRIVER_STATE_COLORS.get(state.driver_state, COLOR_GREEN)
        a_col = _ATTENTION_STATE_COLORS.get(state.attention_state, COLOR_GREEN)

        badge(state.driver_state,    10, y_offset + 10, d_col)
        badge(state.attention_state, 10, y_offset + 38, a_col)

    # ── Alert Banner ──────────────────────────────────────────────────

    def _draw_alert_banner(self, screen, state: AnalyticsState, y_offset):
        if not (state.alarm_drowsiness or state.alarm_distraction or
state.alarm_obstruction):
            return

        messages = []
        if state.driver_state == "SLEEPING":
            messages.append(("DRIVER SLEEPING — PULLING OVER", (220, 30, 30)))
        elif state.alarm_drowsiness:
            messages.append(("⚠  DROWSINESS DETECTED", (255, 160, 0)))
        if state.alarm_distraction:
            messages.append(("⚠  DRIVER DISTRACTED", (255, 160, 0)))
        if state.alarm_obstruction:
            messages.append(("⚠  CAMERA OBSTRUCTED", (220, 30, 30)))

        for i, (msg, color) in enumerate(messages):
            surf = self._font_lg.render(msg, True, color)
            bx = self.feed_w // 2 - surf.get_width() // 2
            by = y_offset + self.ph // 2 - 20 + i * 38
            # Semi-transparent backing
            bg = pygame.Surface((surf.get_width() + 20, surf.get_height() + 8),
pygame.SRCALPHA)
            bg.fill((0, 0, 0, 160))
            screen.blit(bg, (bx - 10, by - 4))
            screen.blit(surf, (bx, by))
===
```

```
# ============================================================================
# ui/hud_renderer.py
#
# HUDRenderer — all PyGame drawing primitives for the DMS top-half feed.
#
# Responsibilities:
#   • Convert OpenCV BGR frame → PyGame surface
#   • Draw face mesh tessellation overlay
#   • Draw YOLO bounding boxes
#   • Draw 3D head pose axis arrows (RGB: R=X, G=Y, B=Z)
#   • Draw eye-gaze direction arrows from iris centers
#   • Render reference-style telemetry panel with all metrics
#   • Render score bars (drowsiness, distraction, PERCLOS)
#   • Render state badges and alert banners
#   • Render gaze crosshair
# ============================================================================

import cv2
import math
import numpy as np
import pygame
import mediapipe as mp

from config import (
    CAMERA_WIDTH, CAMERA_HEIGHT,
    COLOR_GREEN, COLOR_YELLOW, COLOR_RED, COLOR_WHITE,
    COLOR_BLACK, COLOR_DARK_GRAY, COLOR_CYAN, COLOR_ORANGE,
    HUD_FONT_SIZE, ALERT_FONT_SIZE,
    LEFT_IRIS_IDX, RIGHT_IRIS_IDX,
    LEFT_EYE_EAR_IDX, RIGHT_EYE_EAR_IDX,
)
from dms_engine.data_structures import AnalyticsState

# MediaPipe face mesh connections
_FACE_CONTOURS = list(mp.solutions.face_mesh.FACEMESH_CONTOURS)
_FACE_TESSELATION = list(mp.solutions.face_mesh.FACEMESH_TESSELATION)

# State → badge color mapping
_DRIVER_STATE_COLORS = {
    "ALERT":    (40,  200,  80),
    "DROWSY":   (255, 180,   0),
    "SLEEPING": (220,  40,  40),
}
_ATTENTION_STATE_COLORS = {
    "FOCUSED":    (40,  200,  80),
    "DISTRACTED": (220,  40,  40),
}

# Emotion → color
_EMOTION_COLORS = {
    "happy":    (40,  200,  80),
    "neutral":  (180, 180, 180),
```

```
    "angry":    (220,  40,  40),
    "sad":      (100, 140, 220),
    "surprise": (40,  210, 210),
    "fear":     (180,  60, 200),
    "disgust":  (60,  160,  80),
}


def _bgr_frame_to_surface(frame_bgr: np.ndarray, target_w: int, target_h: int) ->
pygame.Surface:
    """Convert a BGR numpy frame to a pygame Surface, resized to target
dimensions."""
    frame_rgb = cv2.cvtColor(frame_bgr, cv2.COLOR_BGR2RGB)
    if frame_rgb.shape[1] != target_w or frame_rgb.shape[0] != target_h:
        frame_rgb = cv2.resize(frame_rgb, (target_w, target_h))
    # PyGame expects (width, height, channels) with shape[1], shape[0]
    surface = pygame.surfarray.make_surface(frame_rgb.swapaxes(0, 1))
    return surface


class HUDRenderer:
    """
    Renders all HUD elements onto a PyGame surface.

    Usage:
        renderer = HUDRenderer(screen_width, panel_height)
        renderer.render(screen, state, frame_bgr, y_offset=0)
    """

    def __init__(self, panel_width: int, panel_height: int):
        self.pw = panel_width
        self.ph = panel_height

        # Feed area: left 2/3 of panel
        self.feed_w = int(panel_width * 0.62)
        self.feed_h = panel_height

        # Telemetry panel: right 1/3
        self.telem_x = self.feed_w
        self.telem_w = panel_width - self.feed_w

        pygame.font.init()
        self._font_xs  = pygame.font.SysFont("monospace", 11)
        self._font_sm  = pygame.font.SysFont("monospace", 13)
        self._font_md  = pygame.font.SysFont("monospace", HUD_FONT_SIZE)
        self._font_lg  = pygame.font.SysFont("monospace", ALERT_FONT_SIZE,
bold=True)
        self._font_badge = pygame.font.SysFont("monospace", 16, bold=True)
        self._font_title = pygame.font.SysFont("monospace", 20, bold=True)

        # Pre-create a dark surface for the telemetry panel background
        self._telem_bg = pygame.Surface((self.telem_w, panel_height),
```

```
pygame.SRCALPHA)
        self._telem_bg.fill((12, 14, 22, 230))

        # Score bar background
        self._bar_bg_color = (50, 50, 60)

    # ── Public API ──────────────────────────────────────────────────────

    def render(
        self,
        screen: pygame.Surface,
        state: AnalyticsState,
        frame_bgr: np.ndarray,
        y_offset: int = 0,
    ) -> None:
        """
        Render the complete top-half DMS panel.

        Args:
            screen:    The main PyGame display surface
            state:     Latest AnalyticsState from DMSCore
            frame_bgr: Raw BGR camera frame
            y_offset:  Vertical offset on screen (0 for top half)
        """
        # ── 1. Camera feed ──────────────────────────────────────────────
        feed_surf = _bgr_frame_to_surface(frame_bgr, self.feed_w, self.feed_h)
        screen.blit(feed_surf, (0, y_offset))

        # ── 2. Face mesh overlay on feed ────────────────────────────────
        geo = state.geometry
        if geo.face_detected and geo.landmarks is not None:
            self._draw_face_mesh(screen, geo.landmarks, y_offset)
            self._draw_gaze_point(screen, geo, y_offset)
            self._draw_gaze_arrows(screen, geo, y_offset)

        # ── 3. 3D head pose axis arrows ─────────────────────────────────
        if geo.face_detected and geo.head_pose.valid:
            self._draw_3d_axis_arrows(screen, geo, y_offset)

        # ── 4. YOLO bounding boxes ──────────────────────────────────────
        self._draw_yolo_boxes(screen, state.detection, y_offset)

        # ── 5. Telemetry panel background ───────────────────────────────
        screen.blit(self._telem_bg, (self.telem_x, y_offset))

        # ── 6. Telemetry content ────────────────────────────────────────
        self._draw_telemetry(screen, state, y_offset)

        # ── 7. Score bars (bottom of feed) ──────────────────────────────
        self._draw_score_bars(screen, state, y_offset)

        # ── 8. State badges ─────────────────────────────────────────────
```

```
    self._draw_state_badges(screen, state, y_offset)

    # ── 9. Alert banner (if critical) ──────────────────────────────
    self._draw_alert_banner(screen, state, y_offset)

    # ── 10. Panel border ──────────────────────────────────────────
    pygame.draw.rect(screen, (60, 60, 80),
                     (0, y_offset, self.pw, self.ph), 2)
    pygame.draw.line(screen, (60, 60, 80),
                     (self.telem_x, y_offset),
                     (self.telem_x, y_offset + self.ph), 2)

# ── Face Mesh ──────────────────────────────────────────────────────

def _draw_face_mesh(self, screen, landmarks, y_offset):
    lm = landmarks
    scale_x = self.feed_w
    scale_y = self.feed_h

    # Tessellation (very light, thin lines)
    for i, j in _FACE_TESSELATION[::3]:   # draw every 3rd for performance
        if i < len(lm) and j < len(lm):
            x1 = int(lm[i][0] * scale_x)
            y1 = int(lm[i][1] * scale_y) + y_offset
            x2 = int(lm[j][0] * scale_x)
            y2 = int(lm[j][1] * scale_y) + y_offset
            pygame.draw.line(screen, (40, 80, 40), (x1, y1), (x2, y2), 1)

    # Contours (slightly brighter)
    for i, j in _FACE_CONTOURS:
        if i < len(lm) and j < len(lm):
            x1 = int(lm[i][0] * scale_x)
            y1 = int(lm[i][1] * scale_y) + y_offset
            x2 = int(lm[j][0] * scale_x)
            y2 = int(lm[j][1] * scale_y) + y_offset
            pygame.draw.line(screen, (60, 160, 60), (x1, y1), (x2, y2), 1)

    # Iris landmarks (bright cyan dots)
    for idx in LEFT_IRIS_IDX + RIGHT_IRIS_IDX:
        if idx < len(lm):
            x = int(lm[idx][0] * scale_x)
            y = int(lm[idx][1] * scale_y) + y_offset
            pygame.draw.circle(screen, (0, 220, 200), (x, y), 3)

def _draw_gaze_point(self, screen, geo, y_offset):
    gx, gy = geo.gaze.gaze_point_px
    # Scale to feed dimensions
    gx = int(gx * self.feed_w / CAMERA_WIDTH)
    gy = int(gy * self.feed_h / CAMERA_HEIGHT) + y_offset
    pygame.draw.circle(screen, (0, 220, 255), (gx, gy), 7)
    pygame.draw.circle(screen, (0, 160, 200), (gx, gy), 11, 2)
    # Crosshair
```

```
    pygame.draw.line(screen, (0, 200, 200), (gx - 15, gy), (gx + 15, gy), 1)
    pygame.draw.line(screen, (0, 200, 200), (gx, gy - 15), (gx, gy + 15), 1)

# ── 3D Head Pose Axis Arrows (RGB) ──────────────────────────────────

def _draw_3d_axis_arrows(self, screen, geo, y_offset):
    """
    Draw RGB axis arrows showing 3D head orientation.
    R=X (right), G=Y (down), B=Z (forward) — projected using the
    head pose rotation vector onto the image plane.
    Placed in upper-right of the feed.
    """
    hp = geo.head_pose
    if hp.rvec is None or hp.tvec is None:
        return

    # Axis origin in the upper-right corner of the feed
    origin_x = self.feed_w - 70
    origin_y = 60 + y_offset

    # Build rotation matrix from rvec
    R, _ = cv2.Rodrigues(hp.rvec)

    # 3D axis unit vectors (scaled for visibility)
    axis_len = 40.0
    axes = {
        (255, 60,  60):  np.array([axis_len, 0, 0]),   # X = Red
        (60,  255, 60):  np.array([0, axis_len, 0]),   # Y = Green
        (60,  60,  255): np.array([0, 0, axis_len]),   # Z = Blue
    }

    for color, axis_3d in axes.items():
        # Rotate axis by head rotation
        rotated = R @ axis_3d

        # Simple orthographic projection (ignore Z for display)
        end_x = int(origin_x + rotated[0])
        end_y = int(origin_y + rotated[1])

        # Draw thick arrow line
        pygame.draw.line(screen, color, (origin_x, origin_y), (end_x, end_y), 3)

        # Arrow head (small triangle)
        dx = end_x - origin_x
        dy = end_y - origin_y
        length = math.sqrt(dx*dx + dy*dy)
        if length > 5:
            ux, uy = dx / length, dy / length
            # Perpendicular
            px, py = -uy, ux
            head_size = 8
            p1 = (end_x, end_y)
```

```
                p2 = (int(end_x - head_size * ux + head_size * 0.4 * px),
                      int(end_y - head_size * uy + head_size * 0.4 * py))
                p3 = (int(end_x - head_size * ux - head_size * 0.4 * px),
                      int(end_y - head_size * uy - head_size * 0.4 * py))
                pygame.draw.polygon(screen, color, [p1, p2, p3])

    # ── Eye-Gaze Direction Arrows ──────────────────────────────────

    def _draw_gaze_arrows(self, screen, geo, y_offset):
        """
        Draw gaze direction arrows from each iris center, showing where
        the driver is looking. Red arrows emanate from each eye.
        """
        if geo.landmarks is None:
            return

        lm = geo.landmarks
        scale_x = self.feed_w
        scale_y = self.feed_h

        # Gaze direction (normalized -1 to 1)
        gh = geo.gaze.horizontal
        gv = geo.gaze.vertical

        arrow_len = 35.0  # pixels

        for iris_idx in [LEFT_IRIS_IDX, RIGHT_IRIS_IDX]:
            # Iris center in screen coords
            iris_pts = np.array([[lm[i][0], lm[i][1]] for i in iris_idx if i <
len(lm)])
            if len(iris_pts) == 0:
                continue
            cx = int(np.mean(iris_pts[:, 0]) * scale_x)
            cy = int(np.mean(iris_pts[:, 1]) * scale_y) + y_offset

            # Direction vector
            dx = int(gh * arrow_len)
            dy = int(-gv * arrow_len)  # invert Y for screen coords

            end_x = cx + dx
            end_y = cy + dy

            # Draw arrow line (red)
            pygame.draw.line(screen, (255, 60, 60), (cx, cy), (end_x, end_y), 2)

            # Arrow head
            line_len = math.sqrt(dx*dx + dy*dy)
            if line_len > 5:
                ux, uy = dx / line_len, dy / line_len
                px, py = -uy, ux
                hs = 6
                p1 = (end_x, end_y)
```

```
                p2 = (int(end_x - hs * ux + hs * 0.4 * px),
                      int(end_y - hs * uy + hs * 0.4 * py))
                p3 = (int(end_x - hs * ux - hs * 0.4 * px),
                      int(end_y - hs * uy - hs * 0.4 * py))
                pygame.draw.polygon(screen, (255, 60, 60), [p1, p2, p3])

    # ── YOLO Boxes ──────────────────────────────────────────────────────

    def _draw_yolo_boxes(self, screen, detection, y_offset):
        for box in detection.boxes:
            x1, y1, x2, y2 = box.bbox
            # Scale to feed dimensions
            sx = self.feed_w / CAMERA_WIDTH
            sy = self.feed_h / CAMERA_HEIGHT
            rx1, ry1 = int(x1 * sx), int(y1 * sy) + y_offset
            rx2, ry2 = int(x2 * sx), int(y2 * sy) + y_offset

            color = COLOR_RED if box.label == "phone" else COLOR_CYAN
            pygame.draw.rect(screen, color, (rx1, ry1, rx2 - rx1, ry2 - ry1), 2)

            label = f"{box.label} {box.confidence:.2f}"
            label_surf = self._font_sm.render(label, True, COLOR_BLACK)
            label_bg   = pygame.Surface((label_surf.get_width() + 4, 16))
            label_bg.fill(color)
            screen.blit(label_bg,   (rx1, ry1 - 16))
            screen.blit(label_surf, (rx1 + 2, ry1 - 15))

    # ── Telemetry Panel (Reference-Style) ──────────────────────────────

    def _draw_telemetry(self, screen, state: AnalyticsState, y_offset):
        geo = state.geometry
        tx  = self.telem_x + 8
        tw  = self.telem_w - 16    # usable width
        line_h = 17

        def txt(text, row, color=(180, 220, 180), font=None):
            f = font or self._font_sm
            surf = f.render(text, True, color)
            screen.blit(surf, (tx, y_offset + 6 + row * line_h))

        def section_icon(icon_char, label, row):
            """Draw a section header with icon character."""
            surf = self._font_sm.render(f"{icon_char} {label}", True, (100, 180,
255))
            screen.blit(surf, (tx, y_offset + 6 + row * line_h))

        def draw_pct_bar(bx, by, bw, bh, score, lo_color, hi_color):
            """Draw a thin percentage bar with gradient."""
            pygame.draw.rect(screen, (35, 35, 45), (bx, by, bw, bh))
            fill = int(bw * max(0.0, min(1.0, score)))
            if fill > 0:
                # Interpolate color based on score
```

```
                t = max(0.0, min(1.0, score))
                r = int(lo_color[0] + (hi_color[0] - lo_color[0]) * t)
                g = int(lo_color[1] + (hi_color[1] - lo_color[1]) * t)
                b = int(lo_color[2] + (hi_color[2] - lo_color[2]) * t)
                pygame.draw.rect(screen, (r, g, b), (bx, by, fill, bh))
            pygame.draw.rect(screen, (70, 70, 80), (bx, by, bw, bh), 1)

        row = 0

        # ── Driver ID ──────────────────────────────────────────────────
        # Icon placeholder
        pygame.draw.circle(screen, (60, 80, 60), (tx + 10, y_offset + 14), 10, 2)
        txt("Driver", 0, (200, 200, 200), self._font_title)
        row = 1

        # ── Separator ──────────────────────────────────────────────────
        pygame.draw.line(screen, (50, 50, 70),
                         (tx, y_offset + 6 + row * line_h + 4),
                         (tx + tw, y_offset + 6 + row * line_h + 4), 1)
        row += 1

        # ── Distraction Level ──────────────────────────────────────────
        dist_pct = int(state.distraction_score * 100)
        dist_col = (40, 200, 80) if dist_pct < 30 else (255, 180, 0) if dist_pct <
60 else (220, 40, 40)
        txt(f"DISTRACTION LEVEL", row, (160, 160, 180))
        row += 1
        bar_y = y_offset + 6 + row * line_h
        draw_pct_bar(tx, bar_y, tw, 14, state.distraction_score, (40, 200, 80),
(220, 40, 40))
        pct_surf = self._font_badge.render(f"{dist_pct}%", True, dist_col)
        screen.blit(pct_surf, (tx + tw // 2 - pct_surf.get_width() // 2, bar_y - 1))
        row += 1

        # ── Drowsy Level ──────────────────────────────────────────
        drow_pct = int(state.drowsiness_score * 100)
        drow_col = (40, 200, 80) if drow_pct < 30 else (255, 180, 0) if drow_pct <
60 else (220, 40, 40)
        txt(f"DROWSY LEVEL", row, (160, 160, 180))
        row += 1
        bar_y = y_offset + 6 + row * line_h
        draw_pct_bar(tx, bar_y, tw, 14, state.drowsiness_score, (40, 200, 80), (220,
40, 40))
        pct_surf = self._font_badge.render(f"{drow_pct}%", True, drow_col)
        screen.blit(pct_surf, (tx + tw // 2 - pct_surf.get_width() // 2, bar_y - 1))
        row += 1

        # ── Action State ──────────────────────────────────────────
        d_state = state.driver_state
        d_col = _DRIVER_STATE_COLORS.get(d_state, (180, 180, 180))
        action_label = d_state
        if state.attention_state == "DISTRACTED":
```

```
            action_label = "DISTRACTED"
            d_col = (220, 40, 40)
        txt(f"ACTION: {action_label}", row, d_col)
        row += 1

        # ── Expression ──────────────────────────────────────────────
        section_icon("☺", "EXPRESSION", row)
        row += 1
        emo = state.fer.emotion_label
        e_col = _EMOTION_COLORS.get(emo, (180, 180, 180))
        txt(f"  {emo.upper()}", row, e_col)
        row += 1

        # ── Eye Openness ────────────────────────────────────────────
        section_icon("◉", "EYE OPENNESS", row)
        row += 1
        if geo.face_detected:
            le = geo.left_eye
            re = geo.right_eye
            le_pct = int(le.ear_percentile)
            re_pct = int(re.ear_percentile)
            le_col = (220, 40, 40) if le.is_closed else (40, 200, 80)
            re_col = (220, 40, 40) if re.is_closed else (40, 200, 80)
            txt(f"  L: {le_pct:3d}%    R: {re_pct:3d}%", row, (180, 220, 180))
        else:
            txt("  -- N/A --", row, (100, 80, 80))
        row += 1

        # ── Eye Blink ───────────────────────────────────────────────
        section_icon("◉", "EYE BLINK", row)
        row += 1
        if geo.face_detected:
            bps = geo.blink.blinks_per_second
            # Blink rate bar: green (0.2/s) to red (0.8/s)
            blink_norm = max(0.0, min(1.0, bps / 1.0))
            bar_y = y_offset + 6 + row * line_h
            draw_pct_bar(tx, bar_y, tw, 10, blink_norm, (40, 200, 80), (220, 40,
40))
            txt(f"  {bps:.1f}/s", row, (180, 220, 180))
        row += 1

        # ── HEAD LOC (mm) ───────────────────────────────────────────
        section_icon("⊞", "HEAD LOC (mm)", row)
        row += 1
        if geo.face_detected and geo.head_pose.valid:
            hp = geo.head_pose
            txt(f"  {hp.x_mm:+7.0f} {hp.y_mm:+7.0f} {hp.z_mm:+7.0f}", row, (180,
220, 180))
        else:
            txt("  -- N/A --", row, (100, 80, 80))
        row += 1
```

```
        # ── EYE LOC (mm) ──────────────────────────────────────────────────
        section_icon("⊞", "EYE LOC (mm)", row)
        row += 1
        if geo.face_detected:
            le = geo.left_eye
            re = geo.right_eye
            # Scale iris center from normalized to approximate mm
            le_x, le_y, le_z = [int(v * 1000) for v in le.iris_center]
            re_x, re_y, re_z = [int(v * 1000) for v in re.iris_center]
            txt(f"  L: {le_x:4d} {le_y:4d} {le_z:4d}", row, (180, 220, 180))
            row += 1
            txt(f"  R: {re_x:4d} {re_y:4d} {re_z:4d}", row, (180, 220, 180))
        else:
            txt("  -- N/A --", row, (100, 80, 80))
        row += 1

        # ── HEAD DIR (PYR) ────────────────────────────────────────────────
        section_icon("⤴", "HEAD DIR (PYR)", row)
        row += 1
        if geo.face_detected and geo.head_pose.valid:
            hp = geo.head_pose
            txt(f"  {hp.pitch:+5.0f}°  {hp.yaw:+5.0f}°  {hp.roll:+5.0f}°", row,
(180, 220, 180))
        else:
            txt("  -- N/A --", row, (100, 80, 80))
        row += 1

        # ── GAZE DIR (PY) ─────────────────────────────────────────────────
        section_icon("⤴", "GAZE DIR (PY)", row)
        row += 1
        if geo.face_detected:
            gz = geo.gaze
            # Convert normalized gaze to approximate degrees
            gz_pitch = gz.vertical * 20.0
            gz_yaw   = gz.horizontal * 30.0
            txt(f"  {gz_pitch:+5.0f}°     {gz_yaw:+5.0f}°", row, (180, 220, 180))
        else:
            txt("  -- N/A --", row, (100, 80, 80))
        row += 1

        # ── GAZE ZONE ─────────────────────────────────────────────────────
        section_icon("◉", "GAZE ZONE", row)
        row += 1
        gz_zone = geo.gaze.gaze_zone if geo.face_detected else "UNKNOWN"
        zone_col = (40, 200, 80) if gz_zone == "FRONT_WINDSHIELD" else (255, 180, 0)
        txt(f"  {gz_zone}", row, zone_col)
        row += 1

        # ── HEAD ZONE ─────────────────────────────────────────────────────
        section_icon("⊙", "HEAD ZONE", row)
        row += 1
        hd_zone = geo.head_pose.head_zone if (geo.face_detected and
```

```
geo.head_pose.valid) else "UNKNOWN"
        zone_col = (40, 200, 80) if hd_zone == "FRONT_WINDSHIELD" else (255, 180, 0)
        txt(f"  {hd_zone}", row, zone_col)

    # ── Score Bars ──────────────────────────────────────────────────────

    def _draw_score_bars(self, screen, state: AnalyticsState, y_offset):
        bar_y   = y_offset + self.ph - 72
        bar_h   = 14
        bar_w   = self.feed_w - 20
        bar_x   = 10

        def draw_bar(bx, by, bw, bh, score, label, color):
            # Background
            pygame.draw.rect(screen, self._bar_bg_color, (bx, by, bw, bh))
            # Fill
            fill = int(bw * max(0.0, min(1.0, score)))
            if fill > 0:
                pygame.draw.rect(screen, color, (bx, by, fill, bh))
            # Border
            pygame.draw.rect(screen, (100, 100, 110), (bx, by, bw, bh), 1)
            # Label
            lbl = self._font_sm.render(f"{label}: {score*100:.0f}%", True, (210,
210, 210))
            screen.blit(lbl, (bx, by - 14))

        # Drowsiness
        d_col = _DRIVER_STATE_COLORS.get(state.driver_state, COLOR_GREEN)
        draw_bar(bar_x, bar_y, bar_w, bar_h,
                 state.drowsiness_score, "DROWSINESS", d_col)

        # Distraction
        a_col = _ATTENTION_STATE_COLORS.get(state.attention_state, COLOR_GREEN)
        draw_bar(bar_x, bar_y + 26, bar_w, bar_h,
                 state.distraction_score, "DISTRACTION", a_col)

        # PERCLOS (thin accent bar)
        draw_bar(bar_x, bar_y + 46, bar_w, 6,
                 state.perclos, "PERCLOS", (180, 100, 255))

    # ── State Badges ────────────────────────────────────────────────────

    def _draw_state_badges(self, screen, state: AnalyticsState, y_offset):
        def badge(text, bx, by, color):
            surf = self._font_badge.render(text, True, COLOR_BLACK)
            w, h = surf.get_width() + 12, surf.get_height() + 6
            pygame.draw.rect(screen, color, (bx, by, w, h), border_radius=4)
            screen.blit(surf, (bx + 6, by + 3))

        d_col = _DRIVER_STATE_COLORS.get(state.driver_state, COLOR_GREEN)
        a_col = _ATTENTION_STATE_COLORS.get(state.attention_state, COLOR_GREEN)
```

```
        badge(state.driver_state,    10, y_offset + 10, d_col)
        badge(state.attention_state, 10, y_offset + 38, a_col)

    # ── Alert Banner ──────────────────────────────────────────────────────

    def _draw_alert_banner(self, screen, state: AnalyticsState, y_offset):
        if not (state.alarm_drowsiness or state.alarm_distraction or
state.alarm_obstruction):
            return

        messages = []
        if state.driver_state == "SLEEPING":
            messages.append(("DRIVER SLEEPING — PULLING OVER", (220, 30, 30)))
        elif state.alarm_drowsiness:
            messages.append(("⚠  DROWSINESS DETECTED", (255, 160, 0)))
        if state.alarm_distraction:
            messages.append(("⚠  DRIVER DISTRACTED", (255, 160, 0)))
        if state.alarm_obstruction:
            messages.append(("⚠  CAMERA OBSTRUCTED", (220, 30, 30)))

        for i, (msg, color) in enumerate(messages):
            surf = self._font_lg.render(msg, True, color)
            bx = self.feed_w // 2 - surf.get_width() // 2
            by = y_offset + self.ph // 2 - 20 + i * 38
            # Semi-transparent backing
            bg = pygame.Surface((surf.get_width() + 20, surf.get_height() + 8),
pygame.SRCALPHA)
            bg.fill((0, 0, 0, 160))
            screen.blit(bg, (bx - 10, by - 4))
            screen.blit(surf, (bx, by))
            screen.blit(surf, (bx, by))
```

Key additions:

- **3D RGB axis arrows** — R=X, G=Y, B=Z projected from head rotation in upper-right of feed
- **Eye-gaze direction arrows** — red arrows from each iris center showing look direction
- **Reference-style telemetry panel** — DISTRACTION/DROWSY level bars, EXPRESSION, EYE OPENNESS L/R %, BLINK rate bar, HEAD LOC (mm), EYE LOC (mm), HEAD DIR (PYR), GAZE DIR (PY), GAZE ZONE, HEAD ZONE

data_structures.py

- Added `GazeState.gaze_zone: str` and `HeadPoseState.head_zone: str`

geometry_tracker.py

- Added _classify_zone() — angle-threshold-based zone classifier
- Populates `gaze_zone` and `head_zone` using head pose + gaze offset

---

3. Sensor Sensitivity & Stability

config.py

```
# ============================================================================
# config.py — Central Configuration for Intelligent DMS
# All tunable parameters live here. Never hardcode values in modules.
# ============================================================================

import os

# ── Paths ──────────────────────────────────────────────────────────────────
BASE_DIR    = os.path.dirname(os.path.abspath(__file__))
MODELS_DIR  = os.path.join(BASE_DIR, "models")
ASSETS_DIR  = os.path.join(BASE_DIR, "assets")
FONTS_DIR   = os.path.join(ASSETS_DIR, "fonts")
LOGS_DIR    = os.path.join(BASE_DIR, "logs")
os.makedirs(LOGS_DIR, exist_ok=True)
os.makedirs(MODELS_DIR, exist_ok=True)

# ── Camera ─────────────────────────────────────────────────────────────────
CAMERA_INDEX        = 0          # Webcam device index
CAMERA_WIDTH        = 640
CAMERA_HEIGHT       = 480
CAMERA_FPS          = 30

# ── Window / UI ────────────────────────────────────────────────────────────
WINDOW_TITLE        = "Intelligent Driver Monitoring System"
WINDOW_WIDTH        = 1280
WINDOW_HEIGHT       = 960        # Top half: 480px feed | Bottom half: 480px sim
HUD_FONT_SIZE       = 18
ALERT_FONT_SIZE     = 28

# Colors (R, G, B)
COLOR_GREEN         = (0,   255, 100)
COLOR_YELLOW        = (255, 220,   0)
COLOR_RED           = (255,  50,  50)
COLOR_WHITE         = (255, 255, 255)
COLOR_BLACK         = (0,     0,   0)
COLOR_DARK_GRAY     = (30,   30,  40)
COLOR_CYAN          = (0,   220, 255)
COLOR_ORANGE        = (255, 140,   0)

# ── MediaPipe Face Mesh ────────────────────────────────────────────────────
MP_MAX_FACES            = 1
MP_REFINE_LANDMARKS     = True     # Enables iris landmarks (468–477)
MP_MIN_DETECTION_CONF   = 0.7
MP_MIN_TRACKING_CONF    = 0.7

# 3D reference face model landmarks (indices into the 478-point mesh)
# Used for solvePnP head pose estimation
# Indices: Nose tip, Chin, Left eye corner, Right eye corner, Left mouth, Right
mouth
FACE_MODEL_LANDMARK_IDS = [1, 152, 263, 33, 287, 57]
```

```
# Approximate 3D coordinates of the above landmarks in mm (canonical face model)
FACE_3D_MODEL_POINTS = [
    [ 0.0,    0.0,    0.0  ],   # Nose tip
    [ 0.0,  -63.6,  -12.5 ],   # Chin
    [-43.3,   32.7,  -26.0],   # Left eye corner (from camera perspective)
    [ 43.3,   32.7,  -26.0],   # Right eye corner
    [-28.9,  -28.9,  -24.1],   # Left mouth corner
    [ 28.9,  -28.9,  -24.1],   # Right mouth corner
]

# ── Eye / EAR ──────────────────────────────────────────────────────────
# MediaPipe iris landmark indices
LEFT_IRIS_IDX           = [468, 469, 470, 471, 472]
RIGHT_IRIS_IDX          = [473, 474, 475, 476, 477]

# EAR landmark indices (6 points per eye: p1..p6)
# Left eye
LEFT_EYE_EAR_IDX        = [362, 385, 387, 263, 373, 380]
# Right eye
RIGHT_EYE_EAR_IDX       = [33,  160, 158,  133, 153, 144]

EAR_BLINK_THRESHOLD     = 0.21    # Below this → eye considered closed
EAR_OPEN_BASELINE       = 0.35    # Typical open-eye EAR (for percentile calc)
BLINK_CONSEC_FRAMES     = 2       # Frames below threshold to count as blink

# ── Drowsiness ─────────────────────────────────────────────────────────
PERCLOS_WINDOW_FRAMES   = 90      # ~3 seconds at 30fps
DROWSY_EAR_THRESHOLD    = 0.25    # EAR below this counts toward PERCLOS
DROWSY_PERCLOS_THRESH   = 0.20    # >20% eye closure in window → DROWSY warning
SLEEPING_PERCLOS_THRESH = 0.40    # >40% → SLEEPING state
DROWSY_SCORE_WEIGHTS    = {
    "perclos":      0.50,
    "ear":          0.25,
    "blink_rate":   0.10,
    "pitch":        0.15,
}

# ── Distraction ────────────────────────────────────────────────────────
# Head pose deviation thresholds (degrees) from neutral (0, 0, 0)
YAW_DISTRACT_THRESH     = 20.0    # Looking left/right
PITCH_DISTRACT_THRESH   = 15.0    # Looking up/down
GAZE_DISTRACT_THRESH    = 0.30    # Normalized gaze deviation

DISTRACTION_SCORE_WEIGHTS = {
    "yaw":          0.35,
    "pitch":        0.25,
    "gaze":         0.25,
    "action":       0.15,
}

# ── State Machine ──────────────────────────────────────────────────────
# Hysteresis: require N consecutive frames to confirm a state transition
```

```
STATE_CONFIRM_FRAMES    = 15      # ~0.5s at 30fps
ALERT_LEVEL_THRESHOLDS  = {
    "ALERT":        (0.0,  0.40),
    "DROWSY":       (0.40, 0.70),
    "SLEEPING":     (0.70, 1.01),
}

# ── Kalman Filter ──────────────────────────────────────────────────────
KALMAN_PROCESS_NOISE    = 1e-3
KALMAN_MEASUREMENT_NOISE= 1e-1

# ── Deep Learning Inference ────────────────────────────────────────────
YOLO_MODEL_PATH         = os.path.join(MODELS_DIR, "yolov8n.pt")
YOLO_CONFIDENCE         = 0.45
YOLO_IOU_THRESHOLD      = 0.45
DL_INFERENCE_FPS        = 15     # Background thread target FPS for DL modules

# Action class names (must match training label order)
ACTION_CLASSES          = [
    "safe_driving", "phone_right", "phone_left",
    "texting_right", "texting_left", "radio",
    "drinking", "reaching_back", "hair_makeup", "talking_passenger"
]

# FER class names
FER_CLASSES             = ["angry", "disgust", "fear", "happy",
                           "sad", "surprise", "neutral"]

# ── Camera Obstruction ─────────────────────────────────────────────────
OBSTRUCTION_VARIANCE_THRESH = 100   # Frame variance below this → obstructed
OBSTRUCTION_CONFIRM_FRAMES  = 10

# ── Simulation ─────────────────────────────────────────────────────────
SIM_FPS                 = 60
CAR_NORMAL_SPEED        = 3.5     # pixels/frame
CAR_DROWSY_SPEED        = 1.5
CAR_PULLOVER_SPEED      = 0.5
ROAD_SCROLL_SPEED       = CAR_NORMAL_SPEED
ALARM_SOUND_PATH        = os.path.join(ASSETS_DIR, "alarm.wav")
===
# ============================================================================
# config.py — Central Configuration for Intelligent DMS
# All tunable parameters live here. Never hardcode values in modules.
# ============================================================================

import os

# ── Paths ──────────────────────────────────────────────────────────────
BASE_DIR    = os.path.dirname(os.path.abspath(__file__))
MODELS_DIR  = os.path.join(BASE_DIR, "models")
ASSETS_DIR  = os.path.join(BASE_DIR, "assets")
FONTS_DIR   = os.path.join(ASSETS_DIR, "fonts")
```

```
LOGS_DIR    = os.path.join(BASE_DIR, "logs")
os.makedirs(LOGS_DIR, exist_ok=True)
os.makedirs(MODELS_DIR, exist_ok=True)

# ── Camera ──────────────────────────────────────────────────────────────
CAMERA_INDEX        = 0          # Webcam device index
CAMERA_WIDTH        = 640
CAMERA_HEIGHT       = 480
CAMERA_FPS          = 30

# ── Window / UI ─────────────────────────────────────────────────────────
WINDOW_TITLE        = "Intelligent Driver Monitoring System"
WINDOW_WIDTH        = 1280
WINDOW_HEIGHT       = 960        # Top half: 480px feed | Bottom half: 480px sim
HUD_FONT_SIZE       = 18
ALERT_FONT_SIZE     = 28

# Colors (R, G, B)
COLOR_GREEN         = (0,   255, 100)
COLOR_YELLOW        = (255, 220,   0)
COLOR_RED           = (255,  50,  50)
COLOR_WHITE         = (255, 255, 255)
COLOR_BLACK         = (0,     0,   0)
COLOR_DARK_GRAY     = (30,   30,  40)
COLOR_CYAN          = (0,   220, 255)
COLOR_ORANGE        = (255, 140,   0)

# ── MediaPipe Face Mesh ─────────────────────────────────────────────────
MP_MAX_FACES            = 1
MP_REFINE_LANDMARKS     = True     # Enables iris landmarks (468–477)
MP_MIN_DETECTION_CONF   = 0.7
MP_MIN_TRACKING_CONF    = 0.7

# 3D reference face model landmarks (indices into the 478-point mesh)
# Used for solvePnP head pose estimation
# Indices: Nose tip, Chin, Left eye corner, Right eye corner, Left mouth, Right
mouth
FACE_MODEL_LANDMARK_IDS = [1, 152, 263, 33, 287, 57]

# Approximate 3D coordinates of the above landmarks in mm (canonical face model)
FACE_3D_MODEL_POINTS = [
    [ 0.0,    0.0,    0.0  ],   # Nose tip
    [ 0.0,  -63.6,  -12.5 ],   # Chin
    [-43.3,   32.7,  -26.0],   # Left eye corner (from camera perspective)
    [ 43.3,   32.7,  -26.0],   # Right eye corner
    [-28.9,  -28.9,  -24.1],   # Left mouth corner
    [ 28.9,  -28.9,  -24.1],   # Right mouth corner
]

# ── Eye / EAR ───────────────────────────────────────────────────────────
# MediaPipe iris landmark indices
LEFT_IRIS_IDX           = [468, 469, 470, 471, 472]
```

```
RIGHT_IRIS_IDX          = [473, 474, 475, 476, 477]

# EAR landmark indices (6 points per eye: p1..p6)
# Left eye
LEFT_EYE_EAR_IDX        = [362, 385, 387, 263, 373, 380]
# Right eye
RIGHT_EYE_EAR_IDX       = [33,  160, 158,  133, 153, 144]

EAR_BLINK_THRESHOLD     = 0.18    # Below this → eye considered closed (tuned down
from 0.21)
EAR_OPEN_BASELINE       = 0.35    # Typical open-eye EAR (for percentile calc)
BLINK_CONSEC_FRAMES     = 2       # Frames below threshold to count as blink

# ── Drowsiness ──────────────────────────────────────────────────────────
PERCLOS_WINDOW_FRAMES   = 90      # ~3 seconds at 30fps
DROWSY_EAR_THRESHOLD    = 0.20    # EAR below this counts toward PERCLOS (tuned down
from 0.25)
DROWSY_PERCLOS_THRESH   = 0.20    # >20% eye closure in window → DROWSY warning
SLEEPING_PERCLOS_THRESH = 0.40    # >40% → SLEEPING state
DROWSY_SCORE_WEIGHTS    = {
    "perclos":      0.50,
    "ear":          0.25,
    "blink_rate":   0.10,
    "pitch":        0.15,
}

# ── Distraction ─────────────────────────────────────────────────────────
# Head pose deviation thresholds (degrees) from neutral (0, 0, 0)
YAW_DISTRACT_THRESH     = 20.0    # Looking left/right
PITCH_DISTRACT_THRESH   = 15.0    # Looking up/down
GAZE_DISTRACT_THRESH    = 0.30    # Normalized gaze deviation

DISTRACTION_SCORE_WEIGHTS = {
    "yaw":          0.35,
    "pitch":        0.25,
    "gaze":         0.25,
    "action":       0.15,
}

# ── State Machine ───────────────────────────────────────────────────────
# Hysteresis: require N consecutive frames to confirm a state transition
STATE_CONFIRM_FRAMES    = 20      # ~0.67s at 30fps (increased for stability)
ALERT_LEVEL_THRESHOLDS  = {
    "ALERT":        (0.0,  0.40),
    "DROWSY":       (0.40, 0.70),
    "SLEEPING":     (0.70, 1.01),
}

# ── Kalman Filter ───────────────────────────────────────────────────────
KALMAN_PROCESS_NOISE    = 5e-4    # Smoother state transitions (was 1e-3)
KALMAN_MEASUREMENT_NOISE= 2e-1    # Trust raw measurements less (was 1e-1)
```

```
# ── Deep Learning Inference ──────────────────────────────────────────
YOLO_MODEL_PATH         = os.path.join(MODELS_DIR, "yolov8n.pt")
YOLO_CONFIDENCE         = 0.45
YOLO_IOU_THRESHOLD      = 0.45
DL_INFERENCE_FPS        = 15     # Background thread target FPS for DL modules

# Action class names (must match training label order)
ACTION_CLASSES          = [
    "safe_driving", "phone_right", "phone_left",
    "texting_right", "texting_left", "radio",
    "drinking", "reaching_back", "hair_makeup", "talking_passenger"
]

# FER class names
FER_CLASSES             = ["angry", "disgust", "fear", "happy",
                           "sad", "surprise", "neutral"]

# ── Camera Obstruction ──────────────────────────────────────────────
OBSTRUCTION_VARIANCE_THRESH = 100   # Frame variance below this → obstructed
OBSTRUCTION_CONFIRM_FRAMES  = 10

# ── Simulation ──────────────────────────────────────────────────────
SIM_FPS                 = 60
CAR_NORMAL_SPEED        = 3.5     # pixels/frame
CAR_DROWSY_SPEED        = 1.5
CAR_PULLOVER_SPEED      = 0.5
ROAD_SCROLL_SPEED       = CAR_NORMAL_SPEED
ALARM_SOUND_PATH        = os.path.join(ASSETS_DIR, "alarm.wav")

# NPC traffic
NPC_SPAWN_INTERVAL      = (2.0, 5.0)   # seconds min/max between spawns
NPC_MIN_SPEED           = 2.0           # pixels/frame
NPC_MAX_SPEED           = 5.0           # pixels/frame
NPC_MAX_CARS            = 3             # max simultaneous NPCs

# Physics model
CAR_ACCELERATION        = 0.06          # speed change per frame (accel)
CAR_DECELERATION        = 0.03          # speed change per frame (decel)
PULLOVER_DURATION_SEC   = 3.0           # S-curve pull-over duration

# Analytics smoothing
DROWSINESS_EMA_ALPHA    = 0.15          # EMA factor for drowsiness score
DISTRACTION_EMA_ALPHA   = 0.15          # EMA factor for distraction score
EAR_CALIBRATION_FRAMES  = 60            # Frames for adaptive EAR baseline (~2s)

# Gaze / Head zone thresholds (degrees)
GAZE_ZONE_MAP           = {
    "FRONT_WINDSHIELD": {"yaw": (-15, 15),  "pitch": (-10, 10)},
    "LEFT_MIRROR":      {"yaw": (-60, -15), "pitch": (-15, 10)},
    "RIGHT_MIRROR":     {"yaw": (15, 60),   "pitch": (-15, 10)},
    "CENTER_CONSOLE":   {"yaw": (-15, 15),  "pitch": (-40, -10)},
    "REARVIEW_MIRROR":  {"yaw": (-10, 10),  "pitch": (10, 30)},
```

```
    "LEFT_WINDOW":      {"yaw": (-90, -60), "pitch": (-20, 20)},
    "RIGHT_WINDOW":     {"yaw": (60, 90),   "pitch": (-20, 20)},
}
```

Parameter	Before	After	Why
`EAR_BLINK_THRESHOLD`	0.21	0.18	Reduce false blink detections
`DROWSY_EAR_THRESHOLD`	0.25	0.20	Require more closure for PERCLOS
`KALMAN_PROCESS_NOISE`	1e-3	5e-4	Smoother transitions
`KALMAN_MEASUREMENT_NOISE`	1e-1	2e-1	Trust raw measurements less
`STATE_CONFIRM_FRAMES`	15	20	More hysteresis (~0.67s)

analytics.py

- **EMA smoothing**: `score = α × raw + (1−α) × prev` with α=0.15 for both drowsiness and distraction

geometry_tracker.py

- **Adaptive EAR baseline**: Collects first 60 open-eye EAR samples, computes median as per-user baseline

---

Verification

All 7 modified files pass `py_compile` and import verification:

```
✅ config.py
✅ dms_engine/data_structures.py
✅ dms_engine/analytics.py
✅ dms_engine/geometry_tracker.py
✅ simulation/road_renderer.py
✅ simulation/car_simulation.py
✅ ui/hud_renderer.py
```

How to Test

```
# Test simulation (keyboard: 1=ALERT, 2=DROWSY, 3=SLEEPING, R=Reset)
python test_simulation.py

# Test telemetry panel + gaze arrows
python test_ui.py

# Test false-positive reduction
python test_analytics.py

# Full system
python main.py
```