"Autonomous Delievery Robot"

In [ ]:

In [7]:

```python
import numpy as np
import heapq
import tkinter as tk
from tkinter import messagebox, font, ttk
import customtkinter as ctk
import time
import threading
from typing import List, Tuple, Optional
import math
import random

# Configuration
GRID_SIZE = 20
CELL_SIZE = 30
ANIMATION_SPEED = 150  # milliseconds

# Enhanced Color Palette
COLORS = {
    'background': '#0a0a0a',
    'grid_bg': '#1a1a2e',
    'grid_line': '#2d3748',
    'empty_cell': '#f7fafc',
    'start': '#48bb78',
    'delivery': '#ed8936',
    'path_colors': ['#3182ce', '#9f7aea', '#38b2ac', '#e53e3e', '#d69e2e', '#38a169', '#805ad5'],
    'visited': '#a0aec0',
    'current': '#ff6b6b',
    'obstacle': '#2d3748',
    'hover': '#bee3f8',
    'selected': '#ffd700',
    'grid_numbers': '#4a5568'
}

class EnhancedPathfindingGrid:
    def __init__(self):
        self.grid = np.zeros((GRID_SIZE, GRID_SIZE))
        self.start_location = None
        self.delivery_locations = []
        self.obstacles = set()
        self.paths = {}  # Store paths between points with colors
        self.visited_cells = set()
        self.current_position = None
        self.delivery_order = []
        self.total_distance = 0

    def heuristic(self, pos1: Tuple[int, int], pos2: Tuple[int, int]) -> float:
        """Enhanced heuristic combining Manhattan and Euclidean distance"""
        manhattan = abs(pos1[0] - pos2[0]) + abs(pos1[1] - pos2[1])
        euclidean = math.sqrt((pos1[0] - pos2[0])**2 + (pos1[1] - pos2[1])**2)
        return manhattan + euclidean * 0.1

    def get_neighbors(self, pos: Tuple[int, int]) -> List[Tuple[int, int]]:
        """Get valid neighboring cells with different costs"""
        x, y = pos
        neighbors = []
        # 8-directional movement with different costs
        directions = [
            (0, 1, 1.0), (0, -1, 1.0), (1, 0, 1.0), (-1, 0, 1.0),  # Cardinal
            (1, 1, 1.4), (1, -1, 1.4), (-1, 1, 1.4), (-1, -1, 1.4)  # Diagonal
        ]

        for dx, dy, cost in directions:
            new_x, new_y = x + dx, y + dy
            if (0 <= new_x < GRID_SIZE and 0 <= new_y < GRID_SIZE and
                (new_x, new_y) not in self.obstacles):
                neighbors.append(((new_x, new_y), cost))
        return neighbors

    def a_star(self, start: Tuple[int, int], goal: Tuple[int, int]) -> List[Tuple[int, int]]:
        """Enhanced A* pathfinding algorithm with cost consideration"""
        if start == goal:
            return [start]

        open_set = [(0, start)]
        came_from = {}
        g_score = {start: 0}
        f_score = {start: self.heuristic(start, goal)}

        while open_set:
            current = heapq.heappop(open_set)[1]

            if current == goal:
                path = []
                while current in came_from:
                    path.append(current)
                    current = came_from[current]
                path.append(start)
                return path[::-1]

            for (neighbor, cost) in self.get_neighbors(current):
                tentative_g_score = g_score[current] + cost

                if neighbor not in g_score or tentative_g_score < g_score[neighbor]:
                    came_from[neighbor] = current
                    g_score[neighbor] = tentative_g_score
                    f_score[neighbor] = tentative_g_score + self.heuristic(neighbor, goal)
                    heapq.heappush(open_set, (f_score[neighbor], neighbor))

        return []

    def calculate_optimal_route(self):
        """Calculate optimal delivery route using nearest neighbor heuristic"""
        if not self.start_location or not self.delivery_locations:
            return []

        unvisited = self.delivery_locations.copy()
        route = [self.start_location]
        current = self.start_location

        while unvisited:
            nearest = min(unvisited, key=lambda x: self.heuristic(current, x))
            route.append(nearest)
            unvisited.remove(nearest)
            current = nearest

        return route

class DeliveryRobotApp(ctk.CTk):
    def __init__(self):
        super().__init__()

        # Initialize enhanced pathfinding grid
        self.pathfinding_grid = EnhancedPathfindingGrid()
        self.animation_running = False
        self.hover_cell = None
        self.selected_delivery = None
        self.path_colors = {}
        self.show_grid_numbers = True
        self.show_coordinates = True

        # Configure appearance
        ctk.set_appearance_mode("dark")
        ctk.set_default_color_theme("blue")

        self.setup_window()
        self.create_widgets()
        self.draw_grid()

    def setup_window(self):
        """Configure main window with enhanced styling"""
        self.title("Advanced Autonomous Delivery Robot - Multi-Path Planning System")
        self.geometry("1400x900")
        self.configure(fg_color=COLORS['background'])
        self.resizable(True, True)

        # Enhanced custom fonts
        self.title_font = ctk.CTkFont(family="Segoe UI", size=29, weight="bold")
        self.subtitle_font = ctk.CTkFont(family="Segoe UI", size=15)
        self.button_font = ctk.CTkFont(family="Segoe UI", size=11, weight="bold")
        self.info_font = ctk.CTkFont(family="Segoe UI", size=10)
        self.small_font = ctk.CTkFont(family="Segoe UI", size=8)

    def create_widgets(self):
        """Create and arrange enhanced GUI widgets"""
        # Main container with scrollable frame
        self.main_frame = ctk.CTkFrame(self, fg_color="transparent")
        self.main_frame.pack(fill="both", expand=True, padx=15, pady=15)

        # Title section
        self.create_enhanced_title_section()

        # Content frame
        self.content_frame = ctk.CTkFrame(self.main_frame, fg_color="transparent")
        self.content_frame.pack(fill="both", expand=True, pady=(15, 0))

        # Left panel (Scrollable Grid)
        self.create_scrollable_grid_panel()

        # Right panel (Enhanced Controls)
        self.create_enhanced_control_panel()

        # Bottom status and info bar
        self.create_enhanced_status_bar()

    def create_enhanced_title_section(self):
        """Create enhanced animated title section"""
        title_frame = ctk.CTkFrame(self.main_frame, height=100, fg_color=COLORS['grid_bg'])
        title_frame.pack(fill="x", pady=(0, 15))
        title_frame.pack_propagate(False)

        # Main title with gradient effect
        title_label = ctk.CTkLabel(
            title_frame,
            text="ADVANCED AUTONOMOUS DELIVERY ROBOT",
            font=self.title_font,
            text_color="#00e5ff"
        )
        title_label.pack(pady=(15, 5))

        # Subtitle
        subtitle_label = ctk.CTkLabel(
            title_frame,
            text="Multi-Path Planning & Optimization System with Real-time Visualization",
            font=self.subtitle_font,
            text_color="#b0bec5"
        )
        subtitle_label.pack()

        # Quick stats
        stats_frame = ctk.CTkFrame(title_frame, fg_color="transparent")
        stats_frame.pack(fill="x", padx=20, pady=(5, 10))

        self.quick_stats = ctk.CTkLabel(
            stats_frame,
            text="Grid: 20x20 | Algorithm: Enhanced A* | Status: Ready",
            font=self.small_font,
            text_color="#718096"
        )
        self.quick_stats.pack()

    def create_scrollable_grid_panel(self):
        """Create enhanced scrollable grid panel"""
        grid_main_frame = ctk.CTkFrame(self.content_frame, fg_color=COLORS['grid_bg'])
        grid_main_frame.pack(side="left", fill="both", expand=True, padx=(0, 15))

        # Grid controls header
        grid_header = ctk.CTkFrame(grid_main_frame, height=45, fg_color="transparent")
        grid_header.pack(fill="x", padx=15, pady=(15, 10))
        grid_header.pack_propagate(False)

        # Grid info and controls
        info_left = ctk.CTkFrame(grid_header, fg_color="transparent")
        info_left.pack(side="left", fill="both", expand=True)

        grid_info = ctk.CTkLabel(
            info_left,
            text="🎯 Left: Set Start/Delivery | Right: Add/Remove Obstacles | Scroll: Navigate Grid",
            font=self.info_font,
            text_color="#81c784"
        )
        grid_info.pack(anchor="w")

        # Grid options
        options_frame = ctk.CTkFrame(info_left, fg_color="transparent")
        options_frame.pack(anchor="w", pady=(1, 0))

        self.show_numbers_var = tk.BooleanVar(value=True)
        self.show_coords_var = tk.BooleanVar(value=True)

        numbers_check = ctk.CTkCheckBox(
            options_frame, text="Grid Numbers", variable=self.show_numbers_var,
            command=self.toggle_grid_numbers, font=self.small_font
        )
        numbers_check.pack(side="left", padx=(0, 15))

        coords_check = ctk.CTkCheckBox(
            options_frame, text="Coordinates", variable=self.show_coords_var,
            command=self.toggle_coordinates, font=self.small_font
        )
        coords_check.pack(side="left")

        # Canvas container with scrollbars
        canvas_container = ctk.CTkFrame(grid_main_frame, fg_color="transparent")
        canvas_container.pack(expand=True, fill="both", padx=15, pady=(0, 15))

        # Create scrollable canvas
        self.create_scrollable_canvas(canvas_container)

    def create_scrollable_canvas(self, parent):
        """Create canvas with scrollbars"""
        # Canvas frame
        canvas_frame = tk.Frame(parent, bg=COLORS['grid_bg'])
        canvas_frame.pack(expand=True, fill="both")

        # Create scrollbars
        v_scrollbar = tk.Scrollbar(canvas_frame, orient="vertical", bg=COLORS['grid_bg'])
        h_scrollbar = tk.Scrollbar(canvas_frame, orient="horizontal", bg=COLORS['grid_bg'])

        # Create canvas
        self.canvas = tk.Canvas(
            canvas_frame,
            bg=COLORS['empty_cell'],
            highlightthickness=1,
            highlightbackground=COLORS['grid_line'],
            scrollregion=(0, 0, GRID_SIZE * CELL_SIZE + 100, GRID_SIZE * CELL_SIZE + 100),
            xscrollcommand=h_scrollbar.set,
            yscrollcommand=v_scrollbar.set
        )

        # Configure scrollbars
        v_scrollbar.config(command=self.canvas.yview)
        h_scrollbar.config(command=self.canvas.xview)

        # Pack scrollbars and canvas
        v_scrollbar.pack(side="right", fill="y")
        h_scrollbar.pack(side="bottom", fill="x")
        self.canvas.pack(side="left", expand=True, fill="both")

        # Bind events
        self.canvas.bind("<Button-1>", self.on_left_click)
        self.canvas.bind("<Button-3>", self.on_right_click)
        self.canvas.bind("<Motion>", self.on_mouse_motion)
        self.canvas.bind("<Leave>", self.on_mouse_leave)
        self.bind_mouse_wheel()

    def bind_mouse_wheel(self):
        """Enhanced mouse wheel scrolling"""
        def on_mouse_wheel(event):
            self.canvas.yview_scroll(int(-1 * (event.delta / 120)), "units")

        def on_shift_mouse_wheel(event):
            self.canvas.xview_scroll(int(-1 * (event.delta / 120)), "units")

        def on_ctrl_mouse_wheel(event):
            # Zoom functionality (future enhancement)
            pass

        self.canvas.bind("<MouseWheel>", on_mouse_wheel)
        self.canvas.bind("<Shift-MouseWheel>", on_shift_mouse_wheel)
        self.canvas.bind("<Control-MouseWheel>", on_ctrl_mouse_wheel)

        # Bind for different platforms
        self.canvas.bind("<Button-4>", lambda e: self.canvas.yview_scroll(-1, "units"))
        self.canvas.bind("<Button-5>", lambda e: self.canvas.yview_scroll(1, "units"))
        self.canvas.bind("<Shift-Button-4>", lambda e: self.canvas.xview_scroll(-1, "units"))
        self.canvas.bind("<Shift-Button-5>", lambda e: self.canvas.xview_scroll(1, "units"))

    def create_enhanced_control_panel(self):
        """Create enhanced control panel with multiple sections"""
        control_frame = ctk.CTkFrame(self.content_frame, width=350, fg_color=COLORS['grid_bg'])
        control_frame.pack(side="right", fill="y")
        control_frame.pack_propagate(False)

        # Scrollable control frame
        control_scroll = ctk.CTkScrollableFrame(control_frame, width=320)
        control_scroll.pack(fill="both", expand=True, padx=15, pady=15)

        # Control sections
        self.create_simulation_controls(control_scroll)
        self.create_path_visualization_controls(control_scroll)
        self.create_delivery_management(control_scroll)
        self.create_statistics_panel(control_scroll)
        self.create_algorithm_settings(control_scroll)

    def create_simulation_controls(self, parent):
        """Create simulation control section"""
        sim_frame = ctk.CTkFrame(parent, fg_color=COLORS['background'])
        sim_frame.pack(fill="x", pady=(0, 15))

        ctk.CTkLabel(
            sim_frame,
            text="SIMULATION CONTROLS",
            font=ctk.CTkFont(size=16, weight="bold"),
            text_color="#00e5ff"
        ).pack(pady=(15, 10))

        button_configs = [
            ("START SIMULATION", self.start_enhanced_simulation, "#4caf50"),
            ("PAUSE/RESUME", self.toggle_pause, "#ff9800"),
            ("STOP SIMULATION", self.stop_simulation, "#f44336"),
            ("RESET ALL", self.reset_grid, "#9c27b0"),
            ("CLEAR PATHS", self.clear_paths, "#2196f3"),
        ]

        for text, command, color in button_configs:
            btn = ctk.CTkButton(
                sim_frame,
                text=text,
                command=command,
                height=35,
                font=self.button_font,
                fg_color=color,
                hover_color=self.darken_color(color)
            )
            btn.pack(pady=5, padx=15, fill="x")

    def create_path_visualization_controls(self, parent):
        """Create path visualization controls"""
        viz_frame = ctk.CTkFrame(parent, fg_color=COLORS['background'])
        viz_frame.pack(fill="x", pady=(0, 15))

        ctk.CTkLabel(
            viz_frame,
            text="PATH VISUALIZATION",
            font=ctk.CTkFont(size=16, weight="bold"),
            text_color="#00e5ff"
        ).pack(pady=(15, 10))

        # Animation speed control
        speed_frame = ctk.CTkFrame(viz_frame, fg_color="transparent")
        speed_frame.pack(fill="x", padx=15, pady=5)

        ctk.CTkLabel(speed_frame, text="Animation Speed:", font=self.info_font).pack(anchor="w")
        self.speed_slider = ctk.CTkSlider(
            speed_frame, from_=50, to=500, number_of_steps=45,
            command=self.update_animation_speed
        )
        self.speed_slider.set(ANIMATION_SPEED)
        self.speed_slider.pack(fill="x", pady=5)

        # Path style options
        style_frame = ctk.CTkFrame(viz_frame, fg_color="transparent")
        style_frame.pack(fill="x", padx=15, pady=5)

        self.path_style = ctk.CTkOptionMenu(
            style_frame,
            values=["Colored Lines"],
            command=self.change_path_style
        )
        self.path_style.pack(fill="x", pady=5)

    def create_delivery_management(self, parent):
        """Create delivery management section"""
        delivery_frame = ctk.CTkFrame(parent, fg_color=COLORS['background'])
        delivery_frame.pack(fill="x", pady=(0, 15))

        ctk.CTkLabel(
            delivery_frame,
            text="DELIVERY MANAGEMENT",
            font=ctk.CTkFont(size=16, weight="bold"),
            text_color="#00e5ff"
        ).pack(pady=(15, 10))

        # Delivery list
        list_frame = ctk.CTkFrame(delivery_frame, fg_color="transparent", height=120)
        list_frame.pack(fill="x", padx=15, pady=5)
        list_frame.pack_propagate(False)

        ctk.CTkLabel(list_frame, text="Delivery Points:", font=self.info_font).pack(anchor="w")

        self.delivery_listbox = tk.Listbox(
            list_frame, height=4, bg=COLORS['empty_cell'], fg="#2d3748",
            selectbackground=COLORS['selected'], font=("Segoe UI", 14)
        )
        self.delivery_listbox.pack(fill="both", expand=True, pady=5)
        self.delivery_listbox.bind("<Double-Button-1>", self.select_delivery_point)

        # Management buttons
        mgmt_buttons = ctk.CTkFrame(delivery_frame, fg_color="transparent")
        mgmt_buttons.pack(fill="x", padx=15, pady=5)

        ctk.CTkButton(
            mgmt_buttons, text="Remove Selected", command=self.remove_selected_delivery,
            height=30, font=self.info_font
        ).pack(side="left", padx=(0, 5), expand=True, fill="x")

        ctk.CTkButton(
            mgmt_buttons, text="Optimize Route", command=self.optimize_route,
            height=30, font=self.info_font
        ).pack(side="right", padx=(5, 0), expand=True, fill="x")

    def create_statistics_panel(self, parent):
        """Create enhanced statistics panel"""
        stats_frame = ctk.CTkFrame(parent, fg_color=COLORS['background'])
        stats_frame.pack(fill="x", pady=(0, 15))

        ctk.CTkLabel(
            stats_frame,
            text="REAL-TIME STATISTICS",
            font=ctk.CTkFont(size=16, weight="bold"),
            text_color="#00e5ff"
        ).pack(pady=(15, 10))

        self.stats_labels = {}
        stats_data = [
            ("Start Location", "Not Set", "#48bb78"),
            ("Obstacles", "0", "#2d3748"),
            ("Current Status", "Ready", "#9f7aea"),
        ]

        for label, value, color in stats_data:
            stat_frame = ctk.CTkFrame(stats_frame, fg_color="transparent")
            stat_frame.pack(fill="x", padx=15, pady=2)

            ctk.CTkLabel(
                stat_frame, text=f"{label}:", font=self.info_font,
                text_color="#b0bec5", anchor="w"
            ).pack(side="left")

            self.stats_labels[label] = ctk.CTkLabel(
                stat_frame, text=value, font=self.info_font,
                text_color=color, anchor="e"
            )
            self.stats_labels[label].pack(side="right")

        self.update_stats()

    def create_algorithm_settings(self, parent):
        """Create algorithm settings section"""
        algo_frame = ctk.CTkFrame(parent, fg_color=COLORS['background'])
        algo_frame.pack(fill="x", pady=(0, 15))

        ctk.CTkLabel(
            algo_frame,
            text="ALGORITHM SETTINGS",
            font=ctk.CTkFont(size=16, weight="bold"),
            text_color="#00e5ff"
        ).pack(pady=(15, 10))

        # Algorithm selection
        algo_select_frame = ctk.CTkFrame(algo_frame, fg_color="transparent")
        algo_select_frame.pack(fill="x", padx=15, pady=5)

        ctk.CTkLabel(algo_select_frame, text="Pathfinding:", font=self.info_font).pack(anchor="w")
        self.algorithm_menu = ctk.CTkOptionMenu(
            algo_select_frame,
            values=["Enhanced A*", "Dijkstra", "A* (Simple)"],
            command=self.change_algorithm
        )
        self.algorithm_menu.pack(fill="x", pady=5)

        # Heuristic weight
        weight_frame = ctk.CTkFrame(algo_frame, fg_color="transparent")
        weight_frame.pack(fill="x", padx=15, pady=5)

        ctk.CTkLabel(weight_frame, text="Heuristic Weight:", font=self.info_font).pack(anchor="w")
        self.heuristic_slider = ctk.CTkSlider(
            weight_frame, from_=0.1, to=2.0, number_of_steps=19
        )
        self.heuristic_slider.set(1.0)
        self.heuristic_slider.pack(fill="x", pady=5)

    def create_enhanced_status_bar(self):
        """Create enhanced status bar with multiple info sections"""
        status_main = ctk.CTkFrame(self.main_frame, height=50, fg_color=COLORS['grid_bg'])
        status_main.pack(fill="x", pady=(15, 0))
        status_main.pack_propagate(False)

        # Left status
        status_left = ctk.CTkFrame(status_main, fg_color="transparent")
        status_left.pack(side="left", fill="both", expand=True, padx=15, pady=10)

        self.status_label = ctk.CTkLabel(
            status_left,
            text="🔓 Ready - Click on grid to set start location",
            font=self.info_font,
            text_color="#81c784",
            anchor="w"
        )
        self.status_label.pack(side="left")

        # Right status (coordinates and info)
        status_right = ctk.CTkFrame(status_main, fg_color="transparent")
        status_right.pack(side="right", padx=15, pady=10)

        self.coord_label = ctk.CTkLabel(
            status_right,
            text="Mouse: (0, 0) | Grid: 20x20",
            font=self.small_font,
            text_color="#718096"
        )
        self.coord_label.pack()

    # Enhanced interaction methods
    def get_cell_from_coords(self, x, y):
        """Convert canvas coordinates to grid coordinates"""
        return (y // CELL_SIZE, x // CELL_SIZE)

    def on_left_click(self, event):
        """Enhanced left click handling"""
        canvas_x = self.canvas.canvasx(event.x)
        canvas_y = self.canvas.canvasy(event.y)
        cell = self.get_cell_from_coords(canvas_x, canvas_y)

        if not (0 <= cell[0] < GRID_SIZE and 0 <= cell[1] < GRID_SIZE):
            return

        if cell in self.pathfinding_grid.obstacles:
            self.update_status("❌ Cannot place on obstacle")
            return

        if self.pathfinding_grid.start_location is None:
            self.pathfinding_grid.start_location = cell
            self.update_status("🏠 Start location set. Click to add delivery points.")
        else:
            if cell not in self.pathfinding_grid.delivery_locations and cell != self.pathfinding_grid.start_loc
                self.pathfinding_grid.delivery_locations.append(cell)
                self.update_delivery_list()
                self.update_status(f"📦 Delivery point added at {cell}. Total: {len(self.pathfinding_grid.delive
            elif cell in self.pathfinding_grid.delivery_locations:
                self.selected_delivery = cell
                self.update_status(f"📍 Selected delivery point at {cell}")

        self.draw_grid()
        self.update_stats()

    def on_right_click(self, event):
        """Enhanced right click for obstacles"""
        canvas_x = self.canvas.canvasx(event.x)
        canvas_y = self.canvas.canvasy(event.y)
        cell = self.get_cell_from_coords(canvas_x, canvas_y)

        if not (0 <= cell[0] < GRID_SIZE and 0 <= cell[1] < GRID_SIZE):
            return

        if (cell != self.pathfinding_grid.start_location and
            cell not in self.pathfinding_grid.delivery_locations):
            if cell in self.pathfinding_grid.obstacles:
                self.pathfinding_grid.obstacles.remove(cell)
                self.update_status(f"□ Obstacle removed at {cell}")
            else:
                self.pathfinding_grid.obstacles.add(cell)
                self.update_status(f"🚧 Obstacle added at {cell}")
        else:
            self.update_status("❌ Cannot place obstacle on start/delivery location")

        self.draw_grid()
        self.update_stats()

    def on_mouse_motion(self, event):
        """Enhanced mouse motion with coordinate display"""
        canvas_x = self.canvas.canvasx(event.x)
        canvas_y = self.canvas.canvasy(event.y)
        cell = self.get_cell_from_coords(canvas_x, canvas_y)

        # Update coordinate display
        self.coord_label.configure(text=f"Mouse: ({int(canvas_x)}, {int(canvas_y)}) | Cell: {cell} | Grid: {GRI

        if (0 <= cell[0] < GRID_SIZE and 0 <= cell[1] < GRID_SIZE and
            cell != self.hover_cell):
            self.hover_cell = cell
            self.draw_grid()

    def on_mouse_leave(self, event):
        """Handle mouse leaving canvas"""
        self.hover_cell = None
        self.coord_label.configure(text=f"Grid: {GRID_SIZE}x{GRID_SIZE}")
        self.draw_grid()

    def draw_grid(self):
        """Enhanced grid drawing with lines instead of blocks for paths"""
        self.canvas.delete("all")

        # Draw grid cells
        for i in range(GRID_SIZE):
            for j in range(GRID_SIZE):
                x1, y1 = j * CELL_SIZE, i * CELL_SIZE
                x2, y2 = x1 + CELL_SIZE, y1 + CELL_SIZE

                # Determine cell color
                cell_color = COLORS['empty_cell']
                if (i, j) == self.hover_cell:
                    cell_color = COLORS['hover']
                elif (i, j) in self.pathfinding_grid.obstacles:
                    cell_color = COLORS['obstacle']
                elif (i, j) == self.pathfinding_grid.start_location:
                    cell_color = COLORS['start']
                elif (i, j) in self.pathfinding_grid.delivery_locations:
                    cell_color = COLORS['delivery']
                elif (i, j) == self.selected_delivery:
                    cell_color = COLORS['selected']
                elif (i, j) == self.pathfinding_grid.current_position:
                    cell_color = COLORS['current']

                # Draw cell
                self.canvas.create_rectangle(
                    x1, y1, x2, y2,
                    fill=cell_color,
                    outline=COLORS['grid_line'],
                    width=1
                )

                # Add grid numbers if enable

                # Add grid numbers if enabled
                if self.show_grid_numbers:
                    self.canvas.create_text(
                        x1 + CELL_SIZE // 2, y1 + CELL_SIZE // 2,
                        text=f"{i},{j}",
                        font=("Arial", 6),
                        fill=COLORS['grid_numbers']
                    )

        # Draw paths as lines
        self.draw_paths()

        # Draw special markers
        self.draw_special_markers()

    def draw_paths(self):
        """Draw paths as colored lines between points"""
        color_index = 0
        for path_key, path in self.pathfinding_grid.paths.items():
            if len(path) > 1:
                color = COLORS['path_colors'][color_index % len(COLORS['path_colors'])]

                for i in range(len(path) - 1):
                    start_cell = path[i]
                    end_cell = path[i + 1]

                    # Calculate center points
                    start_x = start_cell[1] * CELL_SIZE + CELL_SIZE // 2
                    start_y = start_cell[0] * CELL_SIZE + CELL_SIZE // 2
                    end_x = end_cell[1] * CELL_SIZE + CELL_SIZE // 2
                    end_y = end_cell[0] * CELL_SIZE + CELL_SIZE // 2

                    # Draw line
                    self.canvas.create_line(
                        start_x, start_y, end_x, end_y,
                        fill=color, width=3, capstyle="round"
                    )

                color_index += 1

    def draw_special_markers(self):
        """Draw special markers for start and delivery points"""
        # Start location marker
        if self.pathfinding_grid.start_location:
            start = self.pathfinding_grid.start_location
            x = start[1] * CELL_SIZE + CELL_SIZE // 2
            y = start[0] * CELL_SIZE + CELL_SIZE // 2
            self.canvas.create_text(x, y, text="🏠", font=("Arial", 12))

        # Delivery location markers
        for i, delivery in enumerate(self.pathfinding_grid.delivery_locations):
            x = delivery[1] * CELL_SIZE + CELL_SIZE // 2
            y = delivery[0] * CELL_SIZE + CELL_SIZE // 2
            self.canvas.create_text(x, y, text="📦", font=("Arial", 10))

            # Add delivery number
            self.canvas.create_text(
                x + 8, y - 8, text=str(i + 1),
                font=("Arial", 8), fill="white"
            )

    def update_delivery_list(self):
        """Update the delivery points listbox"""
        self.delivery_listbox.delete(0, tk.END)
        for i, location in enumerate(self.pathfinding_grid.delivery_locations):
            self.delivery_listbox.insert(tk.END, f"{i+1}. Point {location}")

    def select_delivery_point(self, event):
        """Handle delivery point selection from listbox"""
        selection = self.delivery_listbox.curselection()
        if selection:
            index = selection[0]
            if index < len(self.pathfinding_grid.delivery_locations):
                self.selected_delivery = self.pathfinding_grid.delivery_locations[index]
                self.draw_grid()

    def remove_selected_delivery(self):
        """Remove selected delivery point"""
        if self.selected_delivery and self.selected_delivery in self.pathfinding_grid.delivery_locations:
            self.pathfinding_grid.delivery_locations.remove(self.selected_delivery)
            self.selected_delivery = None
            self.update_delivery_list()
            self.draw_grid()
            self.update_stats()
            self.update_status("📦 Delivery point removed")

    def optimize_route(self):
        """Optimize delivery route"""
        if len(self.pathfinding_grid.delivery_locations) < 2:
            self.update_status("❌ Need at least 2 delivery points to optimize")
            return

        route = self.pathfinding_grid.calculate_optimal_route()
        self.update_status(f"🔄 Route optimized: {len(route)} points")

    def start_enhanced_simulation(self):
        """Start the enhanced pathfinding simulation"""
        if not self.pathfinding_grid.start_location:
            messagebox.showwarning("Warning", "Please set a start location first!")
            return

        if not self.pathfinding_grid.delivery_locations:
            messagebox.showwarning("Warning", "Please add at least one delivery location!")
            return

        if self.animation_running:
            return

        self.animation_running = True
        self.update_status("🚀 Starting simulation...")

        # Start simulation in separate thread
        threading.Thread(target=self.run_simulation, daemon=True).start()

    def run_simulation(self):
        """Run the pathfinding simulation"""
        try:
            # Calculate paths between all points
            route = self.pathfinding_grid.calculate_optimal_route()

            for i in range(len(route) - 1):
                if not self.animation_running:
                    break

                start_point = route[i]
                end_point = route[i + 1]

                path = self.pathfinding_grid.a_star(start_point, end_point)
                if path:
                    self.pathfinding_grid.paths[f"{start_point}-{end_point}"] = path

                    # Animate path traversal
                    for cell in path:
                        if not self.animation_running:
                            break

                        self.pathfinding_grid.current_position = cell
                        self.after(0, self.draw_grid)
                        time.sleep(ANIMATION_SPEED / 1000.0)

            self.after(0, lambda: self.update_status("✅ Simulation completed!"))

        except Exception as e:
            self.after(0, lambda: self.update_status(f"❌ Simulation error: {str(e)}"))
        finally:
            self.animation_running = False

    def toggle_pause(self):
        """Toggle simulation pause/resume"""
        # Implementation for pause/resume functionality
        pass

    def stop_simulation(self):
        """Stop the running simulation"""
        self.animation_running = False
        self.pathfinding_grid.current_position = None
        self.draw_grid()
        self.update_status("⏹ Simulation stopped")

    def reset_grid(self):
        """Reset the entire grid"""
        self.pathfinding_grid = EnhancedPathfindingGrid()
        self.animation_running = False
        self.selected_delivery = None
        self.path_colors = {}
        self.update_delivery_list()
        self.draw_grid()
        self.update_stats()
        self.update_status("🔄 Grid reset - Click to set start location")

    def clear_paths(self):
        """Clear all calculated paths"""
        self.pathfinding_grid.paths.clear()
        self.pathfinding_grid.visited_cells.clear()
        self.pathfinding_grid.current_position = None
        self.draw_grid()
        self.update_status("□ Paths cleared")

    def update_animation_speed(self, value):
        """Update animation speed"""
        global ANIMATION_SPEED
        ANIMATION_SPEED = int(value)

    def change_path_style(self, style):
        """Change path visualization style"""
        self.update_status(f"🎨 Path style changed to: {style}")

    def change_algorithm(self, algorithm):
        """Change pathfinding algorithm"""
        self.update_status(f"⚙ Algorithm changed to: {algorithm}")

    def toggle_grid_numbers(self):
        """Toggle grid number display"""
        self.show_grid_numbers = self.show_numbers_var.get()
        self.draw_grid()

    def toggle_coordinates(self):
        """Toggle coordinate display"""
        self.show_coordinates = self.show_coords_var.get()

    def update_stats(self):
        """Update statistics display"""
        stats = {
            "Start Location": str(self.pathfinding_grid.start_location) if self.pathfinding_grid.start_location
            "Delivery Points": str(len(self.pathfinding_grid.delivery_locations)),
            "Obstacles": str(len(self.pathfinding_grid.obstacles)),
            "Total Distance": f"{self.pathfinding_grid.total_distance:.1f}",
            "Current Status": "Running" if self.animation_running else "Ready",
            "Completion": "0%"
        }

        for key, value in stats.items():
            if key in self.stats_labels:
                self.stats_labels[key].configure(text=value)

    def update_status(self, message):
        """Update status bar message"""
        self.status_label.configure(text=message)

    def darken_color(self, color):
        """Darken a hex color for hover effect"""
        # Simple darkening by reducing RGB values
        color = color.lstrip('#')
        rgb = tuple(int(color[i:i+2], 16) for i in (0, 2, 4))
        darkened = tuple(max(0, int(c * 0.8)) for c in rgb)
        return f"#{darkened[0]:02x}{darkened[1]:02x}{darkened[2]:02x}"

# Main application entry point
if __name__ == "__main__":
    app = DeliveryRobotApp()
    app.mainloop()
```

In [ ]: