

**SIMULASI KEMUDI OTOMATIS  
PADA PERMAINAN TRACKMANIA NATION MENGGUNAKAN  
TEKNIK VISI KOMPUTER**

**SKRIPSI**

Diajukan untuk memenuhi salah satu syarat  
memperoleh gelar Sarjana Komputer  
Program Studi Informatika



Disusun oleh :  
Marselinus Sandimus Jamlu  
NIM : 215314085

**PROGRAM STUDI INFORMATIKA  
FAKULTAS SAINS DAN TEKNOLOGI  
UNIVERSITAS SANATA DHARMA  
YOGYAKARTA**

**2025**

**SIMULATION OF AUTONOMOUS DRIVING IN THE TRACKMANIA  
NATION GAME USING  
COMPUTER VISION TECHNIQUES**

**THESIS**

Present as partial fulfillment of the requirements  
to obtain *Sarjana Komputer* Degree  
In Informatics Study Program



Author :

Marselinus Sandimus Jamlu

NIM : 215314085

**INFORMATICS STUDY PROGRAM  
FACULTY OF SCIENCE AND TECHNOLOGY  
SANATA DHARMA UNIVERSITY**

**2025**

**HALAMAN PERSETUJUAN PEMBIMBING**

**SKRIPSI**

**SIMULASI NAVIGASI OTONOM PADA PERMAINAN  
TRACKMANIA NATION MENGGUNAKAN TEKNIK  
VISI KOMPUTER**

Disusun oleh :

Marselinus Sandimus Jamlu

NIM : 215314085

Telah Disetujui Oleh:

Dose Pembimbing,

Ir. Kartono Pinaryanto, S.T., M.Cs.

Tanggal: .....

**HALAMAN PENGESAHAN**  
**SKRIPSI**  
**SIMULASI NAVIGASI OTONOM PADA PERMAINAN**  
**TRACKMANIA NATION MENGGUNAKAN TEKNIK**  
**VISI KOMPUTER**

Dipersiapkan dan ditulis oleh :

Marselinus Sandimus Jamlu

NIM : 215314085

**SUSUNAN DEWAN PENGUJI**

<b>JABATAN</b>	<b>NAMA LENGKAP</b>	<b>TANDA TANGAN</b>
Ketua	:	.....
Sekretaris	:	.....
Anggota	:	.....
Yogyakarta, .....		2025
Fakultas Sains dan Teknologi		
Universitas Sanata Dharma,		
Dekan,		
Drs. Haris Sriwindono M.Kom, Ph.D		

## **PERNYATAAN KEASLIAN KARYA**

Saya menyatakan dengan sesungguhnya bahwa skripsi yang saya tulis ini tidak memuat karya atau bagian dari karya orang lain, kecuali yang telah disebutkan dalam kutipan dan daftar pustaka dengan mengikuti ketentuan sebagaimana layaknya karya ilmiah.

Apabila di kemudian hari ditemukan indikasi plagiarisme dalam naskah ini, saya bersedia menanggung segala sanksi sesuai dengan peraturan perundang-undangan yang berlaku.

Yogyakarta, 9 April 2025

Penulis,

Marselinus Sandimus Jamlu

**LEMBAR PERNYATAAN PERSETUJUAN PUBLIKASI KARYA ILMIAH UNTUK  
KEPERLUAN AKADEMIS**

Yang bertanda tangan dibawah ini, saya mahasiswa Universitas Sanata Dharma:

Nama : Marselinus Sandimus Jamlu

NIM : 215314079

Demi pengembangan ilmu pengetahuan, saya memberikan kepada Perpustakaan Universitas Sanata Dharma karya ilmiaah saya yang berjudul:

**“SIMULASI NAVIGASI OTONOM PADA PERMAINAN  
TRACKMANIA NATION MENGGUNAKAN TEKNIK  
VISI KOMPUTER”**

beserta perangkat yang diperlukan (bila ada). Dengan demikian saya memberikan kepada Perpustakaan Universitas Sanata Dharma hak untuk menimpan dan mengalihkan dalam bentuk media lain, mengolah dalam bentuk pangkalan data, mendistribusikan secara terbatas, dan mempublikasikannya di internet ata media lain untuk kepentingan akademis tanpa perlu meminta ijin dai saya atau memberikan royaliti kepada saya selama tetap mencantumkan nama saya sebagai penulis.

Dengan demikian pernyataan ini saya buat dengan sebenarnya.

Dibuat di Yogyakarta  
Pada Tanggal : 9 April 2025  
Yang Menyatakan,

Marselinus Sandimus Jamlu

## KATA PENGANTAR

Puji dan syukur penulis panjatkan ke hadirat Tuhan Yang Maha Esa karena atas rahmat dan karunia-Nya, penulis dapat menyelesaikan tugas akhir yang berjudul "*Simulasi Navigasi Otonom pada Permainan TrackMania Nations Forever Menggunakan Teknik Visi Komputer*" sebagai salah satu syarat untuk memperoleh gelar Sarjana di Program Studi Teknik Informatika.

Penulisan tugas akhir ini tentunya tidak terlepas dari dukungan dan bantuan berbagai pihak. Oleh karena itu, pada kesempatan ini penulis ingin menyampaikan ucapan terima kasih yang sebesar-besarnya kepada:

1. Bapak Ir. Kartono Pinaryanto, S.T., M.Cs. selaku dosen pembimbing yang telah memberikan bimbingan, arahan, dan motivasi selama proses penyusunan tugas akhir ini.
2. Bapak/Ibu dosen di Program Studi Teknik Informatika yang telah memberikan ilmu dan wawasan yang sangat bermanfaat selama masa studi.
3. Keluarga tercinta yang selalu memberikan doa, dukungan moral, dan kepada penulis.
4. Kepada Palinus Alan Sanjaya Jamlu, S.Mat yang selalu memberikan dorongan penuh dalam menyelesaikan penelitian ini.
5. Terima kasih juga saya sampaikan kepada Aloy yang selalu menemani saya selama penyusunan skripsi.

Penulis menyadari bahwa tugas akhir ini masih jauh dari sempurna. Oleh karena itu, penulis terbuka terhadap kritik dan saran yang membangun demi perbaikan di masa mendatang. Semoga tugas akhir ini dapat memberikan manfaat bagi pembaca dan menjadi referensi yang bermanfaat untuk penelitian selanjutnya.

Yogyakarta, 9 April 2025

Marselinus Sandimus Jamlu

## **ABSTRAK**

Penelitian ini bertujuan untuk mengembangkan dan mengevaluasi sistem navigasi otomatis berbasis visi komputer pada lingkungan simulasi permainan menggunakan pendekatan *rule-based control*. Sistem ini terdiri dari dua komponen utama: deteksi jalur menggunakan metode *sliding window*, dan algoritma navigasi berbasis aturan untuk mengendalikan arah dan kecepatan kendaraan secara otomatis.

Proses deteksi jalur berhasil mencapai rata-rata *Intersection over Union* (IoU) sebesar 77% pada sepuluh lintasan berbeda, menunjukkan kemampuan deteksi yang cukup andal terhadap berbagai bentuk dan kondisi jalur. Namun, evaluasi navigasi menunjukkan bahwa kendaraan hanya berhasil menyelesaikan 55.56% lintasan sebelum mengalami tabrakan.

Keterbatasan utama terletak pada kendali berbasis keyboard, kecepatan kendaraan yang tinggi, serta latensi sistem yang signifikan akibat bottleneck pada pengiriman input melalui PyDirectInput. Profiling menunjukkan bahwa simulasi hanya mampu mencapai performa sekitar 5 FPS, yang menyebabkan penurunan responsivitas dalam pengambilan keputusan waktu nyata.

Penggunaan sumber daya komputasi menunjukkan beban CPU rata-rata sebesar 44.3% dan penggunaan memori yang stabil di kisaran 84.0–84.5%, menandakan efisiensi pemrosesan namun dengan keterbatasan pada aspek kecepatan dan real-time performance.

**Kata Kunci:** Visi Komputer, Kemudi Otomatis, *Sliding Window*, TrackMania Nations Forever.

## ABSTRACT

This study aims to develop and evaluate an automated navigation system based on computer vision within a simulated gaming environment using a rule-based control approach. The system consists of two main components: lane detection using the sliding window method, and a rule-based navigation algorithm to autonomously control the vehicle's steering and speed.

The lane detection process achieved an average Intersection over Union (IoU) of 77% across ten different tracks, demonstrating a reasonably robust ability to recognize various lane shapes and visual conditions. However, navigation evaluation showed that the vehicle only successfully completed 55.56% of the tracks before crashing.

The primary limitations stemmed from keyboard-based control, high vehicle speed, and significant system latency due to bottlenecks in input delivery via PyDirectInput. Profiling indicated that the simulation operated at only ~5 FPS, resulting in decreased responsiveness for real-time decision-making.

System resource usage showed moderate computational load, with average CPU utilization at 44.3% and memory usage remaining stable between 84.0–84.5%, suggesting efficient processing yet constrained real-time performance capabilities.

**Keywords:** Computer Vision, Autonomous Navigation, Lane-Following, Sliding Window, TrackMania Nations Forever.

## DAFTAR ISI

HALAMAN JUDUL BAHASA INDONESIA .....	i
HALAMAN JUDUL BAHASA INGGRIS .....	ii
HALAMAN PERSETUJUAN PEMBIMBING .....	iii
HALAMAN PENGESAHAN .....	iv
PERNYATAAN KEASLIAN KARYA .....	v
LEMBAR PERSETUJUAN PUBLIKASI KARYA ILMIAH .....	vi
KATA PENGANTAR .....	vii
ABSTRAK .....	viii
ABSTRACT .....	ix
DAFTAR ISI .....	x
DAFTAR TABEL .....	xv
DAFTAR GAMBAR .....	xvi
DAFTAR ISTILAH .....	xix
DAFTAR LAMPIRAN .....	xx
BAB I PENDAHULUAN .....	1
1.1 Latar Belakang .....	1
1.2 Rumusan Masalah .....	4
1.3 Tujuan Penelitian .....	5
1.4 Batasan Masalah .....	5
1.5 Manfaat Penelitian .....	6
1.6 Sistematika Penulisan .....	7
BAB II LANDASAN TEORI DAN STUDI TERKAIT .....	8
2.1 Penelitian Terkait .....	8
2.2 Landasan Teori .....	10
2.2.1 Visi Komputer .....	10

2.2.1.1	OpenCV.....	10
2.2.1.1.1	Grayscale .....	10
2.2.1.1.2	Gaussian Blurring.....	11
2.2.1.1.3	RoI.....	12
2.2.1.1.4	Thresholding.....	12
2.2.1.1.5	Image Segmentation.....	14
2.2.1.1.6	Deteksi Tepi Canny .....	14
2.2.1.1.7	HSV (Hue, Saturation, Value) .....	17
2.2.1.1.8	Perspective Transform.....	18
2.2.1.1.9	Hough Transform .....	19
2.2.1.1.10	Sliding Window.....	20
2.2.2	Evaluasi Deteksi Jalur .....	21
2.2.2.1	Intersection over Union (IoU) .....	21
2.2.2.2	Performa Kualitatif Konsistensi Deteksi Jalur .....	22
2.2.3	Evaluasi Navigasi Track.....	23
2.2.3.1	Track Berhasil Diselesaikan.....	24
2.2.3.2	Jarak Tempuh Sebelum Menabrak .....	24
2.2.4	Pustaka Automasi & Simulasi .....	25
2.2.4.1	PyAutoGUI.....	25
2.2.4.2	PyDirectInput .....	25
BAB III	METODOLOGI PENGEMBANGAN SIMULASI.....	26
3.1	Diagram Alur Penelitian.....	26
3.2	Pengumpulan Data .....	27
3.3	Pemrosesan Citra.....	29
3.3.1	Grayscale .....	29
3.3.2	Noise Reduction .....	31

3.3.3	Deteksi Tepi Canny .....	32
3.3.4	HSV .....	39
3.3.5	RoI .....	43
3.3.6	Perspective Transform .....	44
3.4	Diagram Alur Deteksi Jalur.....	45
3.4.1	Sliding Window.....	45
3.4.1.1	Histogram Intensitas Piksel.....	46
3.4.1.2	Deteksi Posisi Awal .....	46
3.4.1.3	Pergeseran Window .....	47
3.4.1.4	Menggambar Garis Polinomial .....	47
3.4.2	Perhitungan Radius Kelengkungan .....	49
3.5	Diagram Alur Algoritma Mengemudi Berdasarkan Jalur .....	51
3.6	Matrix Evaluasi Simulasi .....	54
3.6.1	Evaluasi Deteksi Jalur .....	54
3.6.1.1	IoU.....	54
3.6.1.2	Performa Kualitatif Konsistensi Deteksi Jalur .....	56
3.6.2	Evaluasi Navigasi Track.....	56
3.6.2.1	Track Berhasil Diselesaikan.....	56
3.6.2.2	Jarak Tempuh Sebelum Menabrak .....	56
BAB IV	HASIL SIMULASI DAN PEMBAHASAN .....	58
4.1	Pengumpulan Data .....	58
4.2	Pemrosesan Gambar .....	59
4.2.1	Deteksi Tepi.....	59
4.2.2	RoI.....	61
4.2.3	Perspective Transform .....	62
4.3	Deteksi Jalur.....	64

4.3.1	Metode Sliding Window.....	64
4.3.2	Debug dan Visualisasi .....	66
4.4	Implementasi Algoritma Mengemudi.....	67
4.4.1	Estimasi Kurva dan Ofset.....	68
4.4.1.1	Tipe Kurva Jalan .....	69
4.4.1.2	Posisi Mobil Terhadap Jalur .....	70
4.4.2	Algoritma Kontrol .....	73
4.4.2.1	Kontrol Kemudi.....	73
4.4.2.2	Kontrol Kecepatan.....	74
4.4.3	Integrasi PyDirectInput .....	75
4.5	Evaluasi Simulasi .....	76
4.5.1	Evaluasi Deteksi Jalur .....	77
4.5.1.1	IoU.....	77
4.5.1.2	Konsistensi Deteksi Jalur (Kualitatif) .....	79
4.5.2	Evaluasi Navigasi Track.....	81
4.5.2.1	Jumlah Track Sukses .....	81
4.5.2.2	Jarak Tempuh Sebelum Menabrak .....	83
4.6	Analisis Performa Simulasi.....	86
4.6.1	Analisis Performa Algoritma Deteksi Jalur.....	87
4.6.1.1	Kelebihan dan Kekurangan Algoritma Deteksi Jalur .....	87
4.6.1.2	Waktu Eksekusi (Profiling) .....	90
4.6.2	Analisis Performa Algoritma Kontrol Kemudi .....	91
4.6.3	Responsivitas Sistem & Latensi .....	94
4.6.4	Beban Komputasi Simulasi .....	95
4.7	Penyesuaian dan Tuning Parameter.....	96
4.7.1	RoI & Resolusi Frame Permainan.....	96

4.7.2	Sensitivitas Kemudi.....	98
4.7.3	Kecepatan Mobil .....	99
4.7.4	Optimasi Pipeline Simulasi .....	101
4.8	Capaian dan Keterbatasan .....	103
4.8.1	Capaian Simulasi.....	104
4.8.2	Keterbatasan dan Tantangan.....	104
BAB V	KESIMPULAN .....	107
5.1	Kesimpulan.....	107
5.2	Saran.....	108
DAFTAR PUSTAKA .....	109	
LAMPIRAN .....	113	

## DAFTAR TABEL

<b>Tabel 1.</b> Contoh gambar 2x3.....	39
<b>Tabel 2.</b> Normalisasi RGB .....	40
<b>Tabel 3.</b> Nilai V.....	40
<b>Tabel 4.</b> Nilai S .....	41
<b>Tabel 5.</b> Nilai Hue.....	42
<b>Tabel 6.</b> Hasil konversi HSV .....	42
<b>Tabel 7.</b> Visualisasi keseluruhan deteksi jalur .....	67
<b>Tabel 8.</b> Panjang dan layout track.....	77
<b>Tabel 9.</b> Intersection over Union .....	79
<b>Tabel 10.</b> Track berhasil diselesaikan .....	82
<b>Tabel 11.</b> Jarak tempuh sebelum menabrak .....	83
<b>Tabel 12.</b> Penurunan IoU <i>track</i> berwarna cokelat .....	88
<b>Tabel 13.</b> Profiling Deteksi Jalur .....	90
<b>Tabel 14.</b> Profiling Drive Loop.....	94
<b>Tabel 15.</b> Eksperimen koordinat RoI pada gambar .....	98

## DAFTAR GAMBAR

<b>Gambar 1.</b> Aplikasi gaussian blur pada noise .....	12
<b>Gambar 2.</b> Diagram metodologi pengembangan simulasi.....	26
<b>Gambar 3.</b> Diagram pengumpulan data .....	27
<b>Gambar 4.</b> Tangkapan layar disimpan ke dalam dua folder.....	27
<b>Gambar 5.</b> Flowchart pemrosesan gambar .....	29
<b>Gambar 6.</b> Konversi gambar RGB ke <i>grayscale</i> mengurangi kompleksitas citra .....	29
<b>Gambar 7.</b> Representasi RGB pada gambar 3x4 .....	30
<b>Gambar 8.</b> Representasi grayscale pada gambar 3x4 .....	30
<b>Gambar 9.</b> Noise pada gambar 5x5 (kiri) dan 3x3 (kanan) Gaussian kernel.....	31
<b>Gambar 10.</b> Hasil gambar 9 setelah menggunakan Gaussian <i>blur</i> .....	31
<b>Gambar 11.</b> Pengaplikasian <i>blurring</i> pada Gambar 6.....	32
<b>Gambar 12.</b> Penerapan Canny <i>edge detection</i> pada citra abu-abu.....	32
<b>Gambar 13.</b> Sobel filter horizontal (kiri) dan vertikal (kanan) .....	33
<b>Gambar 14.</b> Aplikasi <i>gradient magnitude</i> .....	33
<b>Gambar 15.</b> Hasil kalkulasi <i>gradient magnitude</i> .....	34
<b>Gambar 16.</b> Hasil kalkulasi <i>gradient direction</i> .....	34
<b>Gambar 17.</b> Aplikasi NMS pada hasil <i>gradient calculation</i> .....	35
<b>Gambar 18.</b> Non-Maximum Suppression .....	35
<b>Gambar 19.</b> Contoh NMS pada piksel (1,1) .....	36
<b>Gambar 20.</b> Empat arah gradien .....	36
<b>Gambar 21.</b> Hasil operasi NMS pada .....	36
<b>Gambar 22.</b> Double Thresholding .....	37
<b>Gambar 23.</b> Pemetaan NMS pada gambar asli .....	37
<b>Gambar 24.</b> Contoh Aplikasi <i>double thresholding</i> .....	38
<b>Gambar 25.</b> Eliminasi tepi yang lemah.....	38
<b>Gambar 26.</b> Distribusi tepi lemah dan kuat .....	39
<b>Gambar 27.</b> Derajat HSV .....	43
<b>Gambar 28.</b> Implementasi RoI pada hasil deteksi tepi Canny .....	43
<b>Gambar 29.</b> Transformasi RoI menjadi <i>Bird's eye view</i> .....	44
<b>Gambar 30.</b> Flowchart Sliding Window .....	45
<b>Gambar 31.</b> Histogram intensitas piksel untuk mengidentifikasi posisi awal jalur ...	46

<b>Gambar 32.</b> Sliding window pada hasil deteksi tepi.....	47
<b>Gambar 33.</b> Polinomial linear (kiri) dan polinomial kuadrat (kanan) .....	48
<b>Gambar 34.</b> Flowchart simulasi .....	51
<b>Gambar 35.</b> Kontrol kemudi berdasarkan jalur .....	52
<b>Gambar 36.</b> Representasi IoU .....	55
<b>Gambar 37.</b> Kode mengambil tangkapan layar pada posisi tertentu di monitor.....	58
<b>Gambar 38.</b> Proses penangkapan gambar dengan ukuran 800 x 600 pixel .....	59
<b>Gambar 39.</b> Fungsi deteksi tepi <i>Canny</i> menggunakan OpenCV .....	60
<b>Gambar 40.</b> Hasil deteksi deteksi tepi Canny .....	60
<b>Gambar 41.</b> Hasil deteksi batas jalan berwarna hijau menggunakan HSV .....	61
<b>Gambar 42.</b> Visualisasi posisi RoI .....	62
<b>Gambar 43.</b> Koordinat destinasi RoI .....	63
<b>Gambar 44.</b> Hasil <i>perspective warping</i> .....	63
<b>Gambar 45.</b> Lampiran sliding window .....	65
<b>Gambar 46.</b> Visualisasi <i>Sliding Window</i> .....	65
<b>Gambar 47.</b> Fungsi <i>debugging</i> dan visualisasi .....	66
<b>Gambar 48.</b> Kontrol permainan .....	68
<b>Gambar 49.</b> Implementasi perhitungan kurva jalan.....	68
<b>Gambar 50.</b> Kurva jalan lurus.....	69
<b>Gambar 51.</b> Kurva pada belokan ringan .....	70
<b>Gambar 52.</b> Kurva pada belokan tajam .....	70
<b>Gambar 53.</b> Menghitung ofset mobil.....	71
<b>Gambar 54.</b> Mobil berada di tengah jalan.....	72
<b>Gambar 55.</b> Mobil berada di pinggir kiri jalan .....	72
<b>Gambar 56.</b> Mobil berada di pinggir kanan jalan .....	73
<b>Gambar 57.</b> Implementasi fungsi kontrol kemudi .....	73
<b>Gambar 58</b> Implementasi fungsi kontrol kecepatan .....	74
<b>Gambar 59.</b> Implementasi fungsi <i>apply_control</i> .....	75
<b>Gambar 60.</b> Kode implementasi IoU .....	78
<b>Gambar 61.</b> Konsistensi deteksi jalur lurus .....	80
<b>Gambar 62.</b> Konsistensi deteksi belokan landai .....	80
<b>Gambar 63.</b> Konsistensi deteksi belokan tajam .....	80
<b>Gambar 64.</b> Konsistensi deteksi dengan kehadiran objek lain .....	81

<b>Gambar 65.</b> Distribusi Titik Tabrakan .....	85
<b>Gambar 66.</b> Heatmap Frekuensi Tabrakan.....	86
<b>Gambar 67.</b> Deteksi gagal pada batas jalan dengan warna selain hijau .....	88
<b>Gambar 68.</b> Perubahan perspektif pada kecepatan tinggi.....	89
<b>Gambar 69.</b> Deteksi jalur gagal saat mendekati tepi atau tabrakan .....	89
<b>Gambar 70.</b> RoI tidak menangkap seluruh jalur saat belokan .....	89
<b>Gambar 71.</b> Objek lain mengganggu deteksi jalur .....	90
<b>Gambar 72.</b> Penggunaan memory dan CPU .....	95
<b>Gambar 73.</b> Frame Skipping.....	101
<b>Gambar 74.</b> Pendekatan Producer-Consumer model .....	102
<b>Gambar 75.</b> Loop Producer.....	102
<b>Gambar 76.</b> Loop Consumer.....	103

## **DAFTAR ISTILAH**

<i>Computer Vision</i>	: Bagian dari AI ( <i>Artificial Intelligence</i> ) yang membuat komputer mampu memahami dan menginterpretasi data visual.
<i>Curves:</i>	: Ukuran kelengkungan jalan yang dihitung dari geometri jalur yang terdeteksi.
<i>Edge Detection</i>	: Teknik untuk mengidentifikasi tepi atau batas objek dalam sebuah gambar.
<i>Grayscale</i>	: Proses mengubah gambar berwarna menjadi citra abu-abu untuk menyederhanakan informasi visual.
<i>Offset</i>	: Posisi relatif mobil terhadap garis tengah jalur yang dilalui.
<i>OpenCV</i>	: Pustaka Python yang digunakan untuk keperluan visi komputer dan pengolahan citra.
<i>Pipeline</i>	: Serangkaian proses program berjalan secara otomatis.
<i>Region of Interest (RoI):</i>	: Area tertentu dalam gambar yang difokuskan untuk dianalisis lebih lanjut.
<i>Sliding Window :</i>	: Teknik pencarian pola dalam citra dengan cara menggeser jendela berukuran tetap di seluruh gambar.
<i>Track</i>	: Trek (jalur/lintasan)

## **DAFTAR LAMPIRAN**

<b>Lampiran 1</b> Performa Fungsi Penangkapan Layar Permainan .....	113
<b>Lampiran 2</b> Performa Fungsi Deteksi Jalur.....	113
<b>Lampiran 3</b> Performa Fungsi Mengemudi .....	114
<b>Lampiran 4</b> Performa Fungsi Deteksi Jalur.....	114
<b>Lampiran 5</b> Kelas Deteksi Tepi .....	115
<b>Lampiran 6</b> Kelas Deteksi Jalur .....	115
<b>Lampiran 7</b> Kelas Debugging.....	124
<b>Lampiran 8</b> Kelas Algoritma Mengemudi Real-Time.....	127

## **BAB I**

### **PENDAHULUAN**

#### **1.1 Latar Belakang**

Kecelakaan berkendara merupakan masalah serius yang dapat disebabkan oleh berbagai faktor. Salah satu penyebab utamanya adalah distraksi yang disebabkan oleh penggunaan *handphone* saat berkendara yang mendukung risiko kecelakaan bagi pengemudi remaja hingga dewasa (Klauer, Guo, Simons-Morton, Ouimet, dkk., 2014). Menurut Badan Pusat Statistik, kecelakaan jalan yang terjadi di Indonesia didominasi oleh sepeda motor, diikuti oleh pengendara mobil sedan, dan paling banyak terjadi pada kelompok usia muda, yaitu 15-29 tahun. Peningkatan jumlah kendaraan darat di Indonesia turut memperbesar potensi kecelakaan di jalan raya. Kementerian Komunikasi dan Informatika (Kominfo) menyatakan bahwa rata-rata tiga orang meninggal dunia setiap jamnya akibat kecelakaan lalu lintas di Indonesia. Salah satu upaya yang dapat dilakukan untuk mengurangi angka kecelakaan dan mengatasi distraksi akibat kesalahan manusia (*human error*) adalah dengan mengembangkan teknologi kendaraan otomatis (*autonomous driving*) (Klauer, Guo, Simons-Morton, Claude Ouimet, dkk., 2014; Pradityarahman dkk., 2021).

Kemudi otomatis atau mobil mengemudi otomatis (*self-driving*), telah dipelajari dan dikembangkan oleh berbagai universitas, pusat penelitian, perusahaan mobil, dan perusahaan di seluruh dunia sejak pertengahan tahun 1980 (Badue dkk., 2021). Penelitian ini bertujuan menciptakan kendaraan yang mampu mendeteksi, memahami, dan menavigasi lingkungannya tanpa bantuan manusia.

Minat akademis terhadap teknologi ini meningkat karena potensinya untuk mengurangi kecelakaan lalu lintas, meningkatkan efisiensi transportasi, dan mendukung inovasi di berbagai bidang terkait. Untuk mencapai tujuan ini, tugas yang paling penting merupakan mempelajari aturan-aturan dalam berkendara yang mampu memberikan luaran (*output*) berupa kontrol untuk mengemudi (gas, rem, belok kiri dan kanan, berhenti dsb.) berdasarkan masukkan (*input*) lingkungan sekitar (Pan dkk., 2017; Sallab dkk., 2017). Perkembangan teknologi komputasi seperti sensor, visi komputer (*computer vision*), pembelajaran mesin, dan *hardware acceleration*, serta berbagai jenis perangkat komunikasi untuk mendeteksi masukan pada beberapa tahun belakangan semakin menarik perhatian komunitas *automotif* serta akademisi (Liu dkk., 2021).

Minimnya akses ke *platform* pengujian yang aman dan terjangkau untuk pengembangan sistem mengemudi otomatis menjadi salah satu tantangan utama dalam penelitian dan pengembangan kendaraan otonom. Pengujian model yang komprehensif dan menyeluruh memiliki peran penting dalam melatih model mobil otonom untuk menangani berbagai skenario yang mungkin terjadi di jalan umum. Pelatihan dan pengujian fisik pada jalan umum sering kali tidak aman, membutuhkan biaya yang besar, dan tidak selalu dapat direproduksi secara

konsisten. Mengoperasikan mobil kemudi otomatis di dunia nyata membutuhkan biaya dan sumber daya manusia yang besar. Selain itu, satu unit mobil kemudi otomatis saja tidak cukup untuk mengumpulkan data yang mencakup berbagai skenario jalan yang dibutuhkan untuk proses pelatihan (*training*) dan validasi (*validation*) (Nezami dkk., 2020).

Untuk mengatasi keterbatasan tempat pelatihan yang aman serta biaya yang tinggi dalam memproduksi model mobil kemudi otomatis, pelatihan dan validasi model dapat dilakukan melalui simulasi dengan memberikan lingkungan yang aman dan terkendali sebelum diterapkan di dunia nyata (Dosovitskiy dkk., 2017; P. Kaur dkk., 2021; Liu dkk., 2021). Simulasi telah digunakan untuk melatih model mengemudi sejak masa awal penelitian mobil otonom (Pomerleau, 1988). Beberapa tahun terakhir, permainan simulasi balapan dan beberapa permainan komersial telah digunakan untuk mengumpulkan data, melatih, serta mengevaluasi baik model mobil kemudi otomatis maupun *perception system* (Authier-Carcelen & Zadourian, 2024; Bonyadi dkk., 2017; Chen dkk., 2015; Dosovitskiy dkk., 2017; Erdelyi, 2019), salah satunya adalah Trackmania. Dalam hal ini, visi komputer memiliki peran penting dalam *perception system* karena memungkinkan mobil otonom untuk memahami lingkungan di sekitarnya berdasarkan analisis gambar dan video. Melalui teknik-teknik seperti deteksi objek (*object detection*), segmentasi gambar, dan pelacakan objek (*object tracking*), visi komputer dapat membantu mobil otonom mendeteksi jalur, rambu lalu lintas, kendaraan lain, serta objek-objek di sekitar. Simulasi dalam permainan memungkinkan pengujian dan evaluasi teknik-teknik visi komputer ini dalam lingkungan yang aman dan terkendali (Dosovitskiy dkk., 2017; Punagin, 2020).

Salah satu pustaka yang populer dalam bidang visi komputer merupakan *Open Source Computer Vision* (OpenCV) yang menyediakan berbagai fungsi untuk kepentingan memecahkan masalah berkaitan dengan visi komputer (Pulli dkk., 2012). Dalam konteks simulasi mobil otonom, pustaka ini menyediakan fitur-fitur penting seperti deteksi tepi (*edge detection*) dan pengenalan jalur (*lane recognition*), yang memungkinkan simulasi untuk mengenali elemen-elemen di sekitar, khususnya garis-garis jalur (*lane lines*) (Rathore, 2008; Yang & Ling, 2015). Kemampuan ini sangat berguna dalam membantu mobil otonom memahami dan menavigasi lingkungannya di dalam simulasi secara *realtime* (Erdelyi, 2019; Rathore, 2008). Penelitian yang dilakukan oleh (Yang X & Ling Z, 2015) menunjukkan performa OpenCV dalam mendeteksi jalur secara *realtime* dengan akurasi 84.5% dan *false alarm rate* mencapai 12.0% yang memanfaatkan deteksi tepi serta Transformasi Hough.

Berdasarkan uraian tersebut, OpenCV memiliki performa yang sangat baik untuk memahami lingkungan di sekitar mobil otonom terutama mengenali jalur secara *realtime*. Penelitian ini akan memanfaatkan fungsi-fungsi yang disediakan oleh OpenCV untuk mengenali lingkungan pada permainan Trackmania Nations sebagai *platform* untuk mensimulasikan mobil otonom.

## 1.2 Rumusan Masalah

Bagaimana efektivitas metode deteksi jalur berbasis sliding window dan algoritma navigasi berbasis aturan dalam melakukan simulasi navigasi otonom pada Trackmania Nation?

### 1.3 Tujuan Penelitian

Penelitian ini memiliki tujuan untuk:

1. Mengevaluasi kinerja simulasi navigasi otonom berbasis visi komputer dan algoritma berbasis aturan pada permainan TrackMania Nations Forever.
2. Mengetahui sejauh mana pendekatan *sliding window* dan pengendalian berbasis aturan mampu mengikuti jalur secara *realtime* dalam lingkungan permainan.

### 1.4 Batasan Masalah

Pada penelitian ini terdapat beberapa batasan, yaitu:

1. Simulasi dilakukan dalam lingkungan permainan (*game environment*) yang tidak mencerminkan kondisi lalu lintas nyata, seperti tidak adanya kendaraan lain, rambu lalu lintas, ataupun variasi cuaca dan waktu. Oleh karena itu, hasil evaluasi hanya berlaku pada kondisi statis dan ideal.
2. Simulasi navigasi difokuskan pada pengendalian kendaraan secara otonom di jalur tunggal (*single lane*), tanpa mempertimbangkan manuver lantas jalur atau pengambilan keputusan kompleks lainnya.
3. Jalan yang digunakan tidak memiliki struktur persimpangan (*intersection*), sehingga penelitian ini tidak mencakup navigasi pada percabangan atau pengambilan arah.
4. Simulasi tidak mengimplementasikan fitur untuk menyalip (*overtaking*) kendaraan lain, karena lingkungan permainan tidak menyediakan skenario tersebut.

## 1.5 Manfaat Penelitian

Penelitian ini diharapkan dapat memberikan manfaat baik secara teoritis maupun secara praktis bagi beberapa pihak, dengan uraian sebagai berikut:

### 1. Manfaat Teoritis

- a. Menambah wawasan mengenai penerapan visi komputer dalam navigasi otonom.
- b. Memberikan referensi terkait integrasi teknik *sliding window* untuk deteksi jalur dalam lingkungan virtual.
- c. Dapat digunakan sebagai studi awal untuk pengembangan simulasi navigasi berbasis visi komputer

### 2. Manfaat Praktis

- a. Menyediakan simulasi yang dapat digunakan sebagai alternatif untuk eksperimen navigasi otonom tanpa memerlukan perangkat keras mahal.
- b. Menjadi dasar bagi pengembangan simulasi navigasi otonom berbasis visi komputer pada lingkungan yang lebih kompleks.
- c. Dapat digunakan sebagai alat pembelajaran bagi mahasiswa atau peneliti yang ingin memahami cara kerja simulasi navigasi berbasis visi komputer.

## **1.6 Sistematika Penulisan**

### **BAB I : PENDAHULUAN**

Menjelaskan tentang latar belakang, rumusan masalah, batasan masalah, tujuan penelitian, serta manfaat dari penelitian ini.

### **BAB II : LANDASAN TEORI DAN STUDI TERKAIT**

Pada bab ini mengulas teori-teori dan penelitian terdahulu yang relevan dengan deteksi jalur.

### **BAB III : METODOLOGI PENGEMBANGAN SIMULASI**

Menjelaskan metode yang digunakan dalam penelitian, termasuk dataset yang digunakan.

### **BAB IV : HASIL SIMULASI DAN PEMBAHASAN**

Memaparkan hasil eksperimen yang telah dilakukan. Analisis performa model, hasil uji akurasi akan dijelaskan di bab ini.

### **BAB V : KESIMPULAN DAN SARAN**

Berisi kesimpulan dan saran dari hasil penelitian.

### **DAFTAR PUSTAKA**

### **LAMPIRAN**

## **BAB II**

### **LANDASAN TEORI DAN STUDI TERKAIT**

#### **2.1 Penelitian Terkait**

Penelitian yang dilakukan oleh Aditya dalam menganalisis jalur jalan (*lane detection*) menggunakan pustaka OpenCV seperti Canny dan Hough Transform pada jalan terstruktur berhasil mendekripsi jalur jalan dengan cukup baik. Akan tetapi, penggunaan Hough Transform hanya terbatas pada jalur yang relatif lurus (*straight lane*). Oleh karena itu, algoritma ini akan kesulitan dalam mendekripsi jalur jalan dengan berbagai skenario seperti jalan yang berbelok (Rathore, 2008).

Aakash Punagin dan Sahana Punagin dalam penelitiannya juga menganalisis berbagai teknik deteksi jalur menggunakan OpenCV pada jalan terstruktur. Penelitian tersebut menggunakan teknik-teknik seperti Gaussian Blur untuk mengurangi *noise*, tiga jenis algoritma deteksi tepi seperti Sobel, Laplacian dan deteksi tepi Canny untuk mendekripsi tepi lintasan, serta transformasi Hough Line untuk mendekripsi garis. Berdasarkan tiga jenis algoritma deteksi tepi yang digunakan, Sobel dan Laplacian mendekripsi jalan dengan tidak sempurna karena tepi yang dihasilkan oleh kedua algoritma cenderung tebal. Sebaliknya, Canny merupakan metode yang paling efisien dalam mendekripsi tepi jalan dengan menghilangkan *noise* dan tekstur yang tidak relevan menggunakan *non-maximum suppression* (Punagin, 2020).

Wang dan Fan melakukan penelitian deteksi pada jalur yang lebih kompleks menggunakan algoritma *sliding window* dan *polynomial fitting* untuk mengekstrak garis pada jalur, serta menggunakan deteksi tepi dan *color filter* dari pustaka

OpenCV untuk mendeteksi yang mengindikasikan jalan. Penelitian ini menghasilkan mendapatkan performa yang jauh lebih akurat dibandingkan dengan Hough Transform serta lebih tahan terhadap berbagai skenario pada jalan (Wang dkk., 2021).

Christopher Erdelyi dalam penelitiannya mendemonstrasikan penggunaan fungsi-fungsi OpenCV untuk memproses gambar dari permainan balapan secara otomatis. Penelitian ini melibatkan pengubahan keluaran visual permainan menjadi gambar hitam putih, dengan batas-batas (*boundary*) lintasan ditandai menggunakan algoritma seperti deteksi tepi Canny dan Hough Line Transformation. Gambar hasil proses ini kemudian dievaluasi oleh algoritma mengemudi sederhana untuk menentukan apakah batas lintasan yang dihasilkan cukup membantu kendaraan dalam bernavigasi melalui jalur. Hasil penelitian menunjukkan bahwa algoritma berbasis OpenCV dapat menghasilkan data visual yang mendukung pengemudian otomatis. Dalam eksperimen, program berhasil menghindari batas lintasan dan menyelesaikan beberapa putaran tanpa kendali manusia. Hal ini dicapai dengan serangkaian penyempurnaan algoritma hingga 60 kali, mencatat keberhasilan kendaraan saat menghadapi belokan. Pada pengujian awal dengan *input* acak, kendaraan selalu menabrak sebelum mencapai belokan pertama. Sebaliknya, dengan algoritma OpenCV, kendaraan tidak pernah mengalami kecelakaan sebelum belokan pertama selama 30 kali percobaan (Erdelyi, 2019).

## 2.2 Landasan Teori

### 2.2.1 Visi Komputer

Visi komputer adalah bidang antar-disiplin yang berfokus pada bagaimana komputer dapat direkayasa untuk memperoleh pemahaman tingkat tinggi dari data berupa gambar digital maupun video. Visi komputer merupakan sub-bidang dari *Artificial Intelligence* yang bertujuan untuk mengotomatisasikan tugas-tugas yang dapat dilakukan oleh sistem penglihatan manusia, seperti pengenalan objek, analisis citra, dan pelacakan gerakan (Huang, 1990).

#### 2.2.1.1 OpenCV

OpenCV adalah pustaka perangkat lunak sumber terbuka yang dirancang untuk pengolahan citra secara *real-time*. Dikembangkan oleh Intel pada tahun 1999, OpenCV menyediakan berbagai algoritma yang mendukung penyelesaian masalah dalam bidang visi komputer. Pustaka ini mencakup algoritma pengolahan citra tingkat rendah, seperti deteksi tepi serta algoritma tingkat tinggi, seperti *face detection*, *feature matching*, dan *object tracking* (Pulli dkk., 2012).

##### 2.2.1.1.1 Grayscale

*Grayscale* adalah proses konversi gambar dari berbagai *color spaces* menjadi representasi dalam gradasi abu-abu. Warna abu-abu pada setiap piksel direpresentasikan dengan intensitas yang bervariasi dari 0 (hitam) hingga 255 (putih).

Adapun teknik *grayscale* dapat diekspresikan oleh rumus berikut ini:

$$\text{Average Grayscale} = \frac{(\text{red}+\text{green}+\text{blue})}{3} \quad (1)$$

Proses ini berguna untuk menyederhanakan pengolahan citra, terutama dalam tugas-tugas yang tidak membutuhkan informasi warna penuh, seperti deteksi tepi (Couprie dkk., 2001; Gonzalez & Woods, 2002).

#### 2.2.1.1.2 Gaussian Blurring

Gaussian Blurring adalah metode *smoothing* pada gambar menggunakan fungsi Gaussian. Teknik ini digunakan untuk mengurangi *noise* dan detail kecil pada gambar, sehingga mempermudah analisis lebih lanjut.

$$H_{ij} = \frac{1}{2\pi\rho^2 \exp\left(-\frac{(i-(k+1))^2 + (j-(k+1))^2}{2\sigma^2}\right)} \quad (2)$$

Dimana:

$H_{ij}$  : nilai dari *Gaussian Blur* pada posisi  $(i, j)$

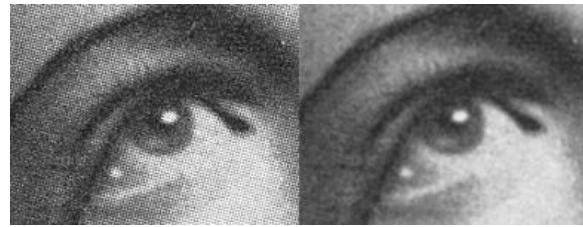
$k$  : ukuran dari kernel

$\rho$  : standar deviasi untuk *scalling factor*

$\sigma$  : standar deviasi

$\pi$  : nilai pi untuk normalisasi

Dalam pemrosesan citra digital, Gaussian Blur sering digunakan sebelum proses seperti deteksi tepi untuk menghasilkan *output* yang lebih akurat (Gonzalez & Woods, 2002).



**Gambar 1.** Aplikasi gaussian blur pada noise

#### 2.2.1.1.3 RoI

RoI adalah bagian tertentu yang diambil dari sebuah gambar untuk keperluan analisis. Proses ini memungkinkan fokus pada area yang relevan dalam sebuah gambar, sehingga pengolahan citra lebih efisien dan terarah (Bottazzi dkk., 2014).

#### 2.2.1.1.4 Thresholding

*Thresholding* atau nilai ambang batas adalah proses segmentasi citra yang digunakan untuk membagi gambar berdasarkan intensitas piksel. Teknik ini dapat menyederhanakan analisis citra dengan mengisolasi elemen-elemen tertentu dari latar belakang.

##### a. Simple Thresholding

Pada setiap piksel gambar, nilai intensitas dibandingkan dengan nilai ambang batas tertentu. Jika intensitas piksel lebih kecil dari nilai ambang, piksel akan diubah menjadi nol (hitam). Sebaliknya, jika lebih besar, piksel akan diubah menjadi nilai maksimum (putih). Teknik ini cocok untuk gambar dengan pencahayaan seragam.

b. Double Thresholding

*Double thresholding* digunakan untuk membedakan tepi yang kuat, lemah dan tidak relevan. *Threshold* ini biasanya digunakan pada algoritma deteksi tepi seperti deteksi tepi Canny.

strong edges: pixel Value >  $T_{high}$

weak edges:  $T_{Low} \leq$  pixel Value  $\leq T_{high}$

non – relevan: pixel Value <  $T_{low}$  (3)

Dimana:

*strong edges* : Tepi yang relevan

*weak edges* : Tepi yang lemah

*non-relevan* : Tepi yang tidak relevan

c. Adaptive Thresholding

*Adaptive thresholding* digunakan untuk gambar dengan variasi pencahayaan atau intensitas warna yang signifikan. Algoritma ini menentukan nilai ambang secara dinamis berdasarkan rata-rata atau median nilai piksel di sekitarnya. Dengan pendekatan ini, setiap area dalam gambar memiliki nilai ambang yang sesuai dengan kondisinya (Roy dkk., 2014).

#### 2.2.1.5 Image Segmentation

Dalam pemrosesan citra digital dan visi komputer, *image segmentation* (segmentasi citra) atau peruasan citra adalah proses pembagian citra digital ke dalam beberapa bagian. Segmentasi citra digunakan untuk menyederhanakan penggambaran citra ke dalam bentuk yang lebih bermakna dan lebih mudah dianalisis (Kaganami & Beiji, 2009). Tujuan dari segmentasi citra adalah untuk membagi sebuah gambar menjadi beberapa bagian/segmen yang memiliki fitur atau atribut yang sama (Gonzalez & Woods, 2002; D. Kaur & Kaur, 2014).

#### 2.2.1.6 Deteksi Tepi Canny

*Edge Detection* atau deteksi tepi adalah cara-cara matematis untuk mengenali titik-titik dalam citra digital yang kecerahannya berubah drastis atau, secara formal, memiliki diskontinuitas. Tepi dibentuk oleh perubahan intensitas atau warna dalam sebuah gambar. Dengan melakukan deteksi tepi, data dalam gambar akan dikurangi secara signifikan dan memberikan struktur dari gambar untuk keperluan pemrosesan citra (Canny, 1983; Gonzalez & Woods, 2002; Kaganami & Beiji, 2009; Punagin, 2020).

Salah satu algoritma deteksi tepi yang populer adalah deteksi tepi (*Canny Edge Detection*) diciptakan oleh John F. Canny pada tahun 1986. Algoritma ini memanfaatkan *multi-stage algorithm* yang memiliki tahapan-tahapan seperti *noise reduction*, *gradient calculation*, *Non-*

*maximum suppression, double threshold, dan edge tracking by hysteresis* untuk mendapatkan hasil deteksi yang maksimal (Canny, 1983).

### 1. Noise Reduction

Algoritma deteksi tepi sangat sensitif terhadap *noise* pada gambar karena bergantung pada derivatif untuk menghitung perubahan berdasarkan variasi *input*. *Noise* dapat menyebabkan algoritma deteksi tepi menginterpretasinya sebagai sebuah tepi. Salah satu cara untuk mengurangi *noise* adalah memanfaatkan Gaussian Blurring dengan memberikan filter pada elemen dengan frekuensi yang tinggi dalam sebuah gambar.

### 2. Gradient Calculation

Tahapan ini mendeteksi perubahan intensitas piksel yang signifikan pada gambar yang mengindikasikan tepi berdasarkan *gradient magnitude* dan *gradient direction*. Filter mendeteksi gambar pada sumbu x (horizontal) dan y (vertikal) kemudian menandai perubahan intensitas piksel pada kedua arah yang dapat dihitung berdasarkan kedua rumus berikut:

$$G = \sqrt{G_x^2 + G_y^2} \quad (4)$$

Dimana:

$G$  : *gradient magnitude*

$G_x$  : arah horizontal (sumbu x)

$G_y$  : arah vertikal (sumbu y)

$$\theta = \text{atan2}(G_y + G_x) \quad (5)$$

Dimana:

$\theta$  : gradient direction

$\text{atan2}(G_y, G_x)$  : menghitung arah dari gradien

### 3. Non-maximum suppression

Pada tahapan *gradient calculation*, intensitas tepi bervariasi dalam rentang nilai 0-255 menyebabkan beberapa tepi lebih tebal dari yang lain. Algoritma *Non-maximum suppression* digunakan untuk membuat hasil tepi lebih seimbang. Algoritma ini menelusuri intensitas setiap piksel dan membandingkannya dengan tetangga di sekitarnya, sehingga hanya tepi dengan nilai intensitas terbesar yang dipertahankan.

### 4. Double Threshold

Tahapan ini bertujuan untuk mengidentifikasi tepi dengan intensitas kecerahan kuat, lemah, dan tidak relevan. Piksel dengan intensitas kecerahan tinggi memiliki kontribusi paling kuat sebagai tepi, sehingga digunakan sebagai nilai-nilai ambang batas atas (*high threshold*). Sementara itu, piksel dengan intensitas rendah digunakan untuk sebagai ambang batas bawah (*low threshold*) untuk mengabaikan intensitas piksel yang tidak relevan.

## 5. Edge Tracking by Hysteresis

Tahapan ini bertujuan untuk mengubah piksel dengan intensitas rendah menjadi kuat jika piksel-piksel di sekitarnya terdapat piksel dengan intensitas kuat.

### 2.2.1.1.7 HSV (Hue, Saturation, Value)

HSV adalah model warna yang sering digunakan dalam pemrosesan citra karena memungkinkan pemisahan warna yang lebih jelas dibandingkan dengan model RGB atau *grayscale* (Sural dkk., 2002).

Dalam konteks deteksi marka jalan, penggunaan *grayscale* dapat menyebabkan kesulitan dalam menangkap perbedaan intensitas piksel, terutama ketika marka jalan memiliki warna yang mirip dengan latar belakangnya.

Konversi dari model warna RGB ke HSV dilakukan dengan langkah-langkah berikut:

#### 1. Normalisasi Nilai RGB

Setiap nilai kanal warna R (*Red*), G (*Green*), dan B (*Blue*) dinormalisasi ke rentang [0,1] dengan membagi nilai piksel dengan 255:

$$R' = \frac{R}{255}, \quad G' = \frac{G}{255}, \quad B' = \frac{B}{255} \quad (6)$$

Dimana:

$R'$ : kanal warna merah setelah dinormalisasi ke rentang [0,1]

$G'$ : kanal warna hijau setelah dinormalisasi ke rentang [0,1]

$B$  : kanal warna biru setelah dinormalisasi ke rentang [0,1]

## 2. Menghitung Nilai Value (V)

Nilai V (*Value*) merepresentasikan tingkat kecerahan suatu warna dan dihitung dengan mengambil nilai maksimum dari tiga kanal warna:

$$V = \max(R', G', B') \quad (7)$$

## 3. Menghitung Nilai Saturation (S)

Saturasi S (*Saturation*) mengukur intensitas atau kejemuhan warna, yang dihitung dengan rumus:

$$S = V - \frac{\min(R', G', B')}{V}, \text{ jika } V \neq 0 \quad (8)$$

Jika  $V=0$ , maka  $S=0$ .

## 4. Menghitung Nilai Hue (H)

*Hue* (H) menunjukkan rona warna dalam derajat (0°–360°) dan dihitung berdasarkan kanal warna yang memiliki nilai maksimum.

$$H = \begin{cases} 60 \times \frac{G' - B'}{V - \min(R', G', B')}, & \text{jika } V = R' \\ 60 \times (2 + \frac{G' - B'}{V - \min(R', G', B')}), & \text{jika } V = G' \\ 60 \times (4 + \frac{G' - B'}{V - \min(R', G', B')}), & \text{jika } V = B' \end{cases} \quad (9)$$

### 2.2.1.1.8 Perspective Transform

Transformasi perspektif (*perspective transform*) adalah teknik dalam pengolahan citra yang digunakan untuk mengubah sudut pandang suatu gambar, misalnya dari tampilan biasa menjadi tampilan "*bird's eye*

"view" (pandangan dari atas). Dalam konteks deteksi jalur, transformasi ini berguna untuk mengubah gambar jalan dari sudut pandang kamera mobil menjadi tampilan atas, sehingga jalur terlihat sejajar dan lebih mudah dianalisis. Proses ini melibatkan pemetaan titik-titik tertentu pada gambar asli ke titik-titik baru menggunakan matriks transformasi (Szeliski, 2022).

#### 2.2.1.1.9 Hough Transform

Hough Transform adalah teknik ekstraksi fitur yang digunakan dalam visi komputer, analisis citra, pengenalan pola, maupun pemrosesan citra digital. Teknik ini dirancang untuk menemukan bentuk tertentu, bahkan jika bentuk tersebut tidak sempurna atau memiliki gangguan. Salah satu aplikasi paling umum dari Hough Transform adalah deteksi garis lurus.

Secara dasar, garis lurus dapat direpresentasikan oleh persamaan:

$$y = mx + b \quad (10)$$

Dimana:

$y$  : sumbu vertikal

$x$  : sumbu horizontal

$m$  : *slope* (kemiringan)

$b$  : *y-intercept*

Untuk mengatasi masalah seperti kemiringan vertikal di mana  $mx$  menjadi tak terdefinisi, Hough Transform menggunakan representasi polar:

$$\rho = x \cos\theta + y \sin\theta \quad (11)$$

Dalam persamaan ini:

$\rho$  : jarak tegak lurus dari asal ke garis, dan

$\theta$  : sudut antara sumbu x dan garis tegak lurus dari asal ke garis yang dianalisis.

Pendekatan ini memungkinkan deteksi garis menjadi lebih stabil pada gambar berbasis piksel (Illingworth & Kittler, 1988).

#### 2.2.1.1.10 Sliding Window

Sliding Window adalah teknik untuk memecahkan masalah dengan menginisiasi sebuah jendela tetap (*fixed-window*) pada data input, kemudian menggeser jendela tersebut sepanjang data sambil melakukan operasi pada setiap posisi jendela. Teknik ini menggunakan dua penanda (*pointers*) untuk menunjuk awal dan akhir jendela..

Dalam konteks deteksi jalur, teknik ini digunakan untuk menemukan tepi dari garis jalur (*lane lines*). Jendela awal ditempatkan di bagian kiri dan kanan jalur pada area bawah gambar dan digeser ke atas hingga mencapai bagian atas gambar, mencari piksel yang menunjukkan jalur pada setiap posisi jendela. Ketika piksel jalur ditemukan, nilai piksel yang memiliki potensi sebagai jalur disimpan dalam dua daftar terpisah (kiri dan kanan). Nilai rata-rata (*mean*) dari posisi piksel ini kemudian digunakan untuk mengatur pusat jendela

berikutnya pada setiap langkah (Bhupathi & Ferdowsi, 2020; S. dkk., 2021).

### **2.2.2 Evaluasi Deteksi Jalur**

Deteksi jalur adalah komponen penting dalam simulasi mengemudi otomatis, yang berfungsi untuk mengenali dan mengikuti jalur atau garis yang ada di jalan. Teknik ini digunakan untuk membantu kendaraan tetap berada pada jalur yang benar dan menghindari penyimpangan. Berbagai metode dapat digunakan dalam deteksi jalur, mulai dari pengolahan citra dasar hingga penerapan algoritma canggih seperti jaringan saraf tiruan (*neural networks*). Dalam implementasi praktisnya, deteksi jalur menggunakan teknik-teknik seperti pemrosesan citra (*image processing*) untuk menemukan garis jalur yang terlihat di jalanan, dengan menggunakan algoritma seperti Canny *Edge Detection* dan *Hough Transform* (Bhupathi & Ferdowsi, 2020; Posch & Rask, 2017).

#### **2.2.2.1 Intersection over Union (IoU)**

IoU adalah metrik yang umum digunakan untuk mengukur akurasi deteksi jalur. IoU menghitung rasio antara area irisan (*intersection*) antara jalur yang terdeteksi dan jalur *ground truth* dengan total area gabungan (union) dari kedua jalur tersebut. Semakin tinggi nilai IoU (mendekati 1), semakin akurat simulasi dalam mendeteksi jalur.

$$\text{IoU} = \frac{|A \cap B|}{|A \cup B|} \quad (13)$$

Dimana:

$A$  = Area jalur *ground truth*

$B$  = Area jalur yang terdeteksi

Metrik ini mempertimbangkan tingkat tumpang tindih antara hasil deteksi dan *ground truth*, sehingga memberikan gambaran yang lebih akurat tentang kualitas deteksi jalur [33].

#### 2.2.2.2 Performa Kualitatif Konsistensi Deteksi Jalur

Konsistensi deteksi jalur mengukur sejauh mana simulasi dapat mendeteksi jalur secara stabil dan dapat diandalkan dalam berbagai kondisi jalan dan lingkungan (misalnya, perbedaan pencahayaan, cuaca, atau jenis jalan). Simulasi deteksi jalur yang konsisten harus mampu mengikuti jalur dengan akurat tanpa sering mengalami kesalahan atau gangguan, bahkan ketika tanda jalur di jalan kurang jelas.

Deteksi jalur tidak hanya bergantung pada akurasi dalam menemukan garis marka, tetapi juga pada kemampuannya mengikuti bentuk jalan, terutama pada belokan. Kelengkungan jalan dihitung menggunakan regresi *polinomial* orde dua pada koordinat jalur yang terdeteksi. Nilai ini menentukan seberapa tajam sebuah jalan berbelok dan mempengaruhi konsistensi deteksi jalur, karena algoritma harus mampu menyesuaikan estimasi jalur sesuai dengan bentuk lintasan.

$$R = \frac{(1+2(Ay+B)^2)^{3/2}}{|2A|} \quad (12)$$

Dimana:

$R$  : Radius kelengkungan (*Curvature radius*)

$A$  : Koefisien kuadrat dari hasil *fitting polinomial* bentuk  $y = ax^2 + bx + c$

$B$  : Koefisien linear dari *polinomial*

$Y$  : Posisi vertikal (sumbu y) pada gambar atau peta piksel tempat kelengkungan dihitung.

$|2A|$  : Nilai absolut dari dua kali koefisien A

### 2.2.3 Evaluasi Navigasi Track

Kelengkungan jalan memainkan peran penting dalam simulasi deteksi dan navigasi jalur. Nilai kelengkungan dihitung berdasarkan persamaan *polinomial* orde dua yang diperoleh dari koordinat jalur yang terdeteksi. Kelengkungan ini digunakan untuk menentukan tingkat kecepatan yang optimal serta menyesuaikan arah kendaraan, terutama pada belokan.

Dalam evaluasi jalur, kelengkungan dapat dikategorikan sebagai berikut:

1. Jalan Lurus: Kurvatur lebih dari atau sama dengan 1.000 m.
2. Belokan Ringan: Kurvatur antara 300 m hingga 1.000 m.
3. Belokan Tajam: Kurvatur kurang dari 300 m.

Metrik ini membantu mengukur seberapa baik algoritma mengemudi beradaptasi dengan kondisi jalan yang berbeda, terutama dalam menghadapi belokan yang membutuhkan koreksi arah yang lebih kompleks.

#### **2.2.3.1 Track Berhasil Diselesaikan**

Metrik ini mengukur kemampuan kendaraan untuk berhasil menavigasi setiap putaran atau belokan di sepanjang trek. Mengingat bahwa trek biasanya terdiri dari kombinasi jalur lurus dan belokan, kemampuan untuk menavigasi belokan dengan benar sangat penting. Metrik ini membantu menilai apakah simulasi dapat menyesuaikan arah kendaraan dengan baik ketika menghadapi perubahan jalur (Erdelyi, 2019).

#### **2.2.3.2 Jarak Tempuh Sebelum Menabrak**

Metrik ini mengukur sejauh mana kendaraan dapat melaju sebelum terjadi tabrakan atau kesalahan yang menyebabkan simulasi keluar jalur. Ini mencerminkan seberapa baik simulasi mendekripsi dan menghindari hambatan atau kesalahan. Semakin jauh jaraknya sebelum tabrakan, semakin baik kemampuan simulasi untuk mengikuti jalur dengan aman

Tingkat keberhasilan mengukur sejauh mana kendaraan dapat menyelesaikan trek tanpa kegagalan. Metrik ini merupakan indikator kinerja keseluruhan, yang mencakup kemampuan deteksi jalur, pengambilan keputusan, dan pemulihan dari kesalahan. Rasio

keberhasilan yang tinggi menunjukkan bahwa simulasi mampu mengikuti jalur dengan stabil dan menghindari masalah secara konsisten.

$$\text{Tingkat Keberhasilan} = \frac{\text{jarak berhasil ditempuh}}{\text{panjang jalur}} * 100 \quad (14)$$

## 2.2.4 Pustaka Automasi & Simulasi

### 2.2.4.1 PyAutoGUI

PyAutoGUI adalah pustaka Python yang memungkinkan pengguna untuk mengontrol mouse dan *keyboard* secara otomatis untuk berinteraksi dengan aplikasi-aplikasi yang berjalan di sistem operasi seperti Windows, macOS, dan Linux. Pustaka ini sangat berguna untuk mengotomatiskan tugas-tugas yang memerlukan interaksi manual, serta berjalan di kedua versi Python 2 dan 3 (Sweigart, 2021).

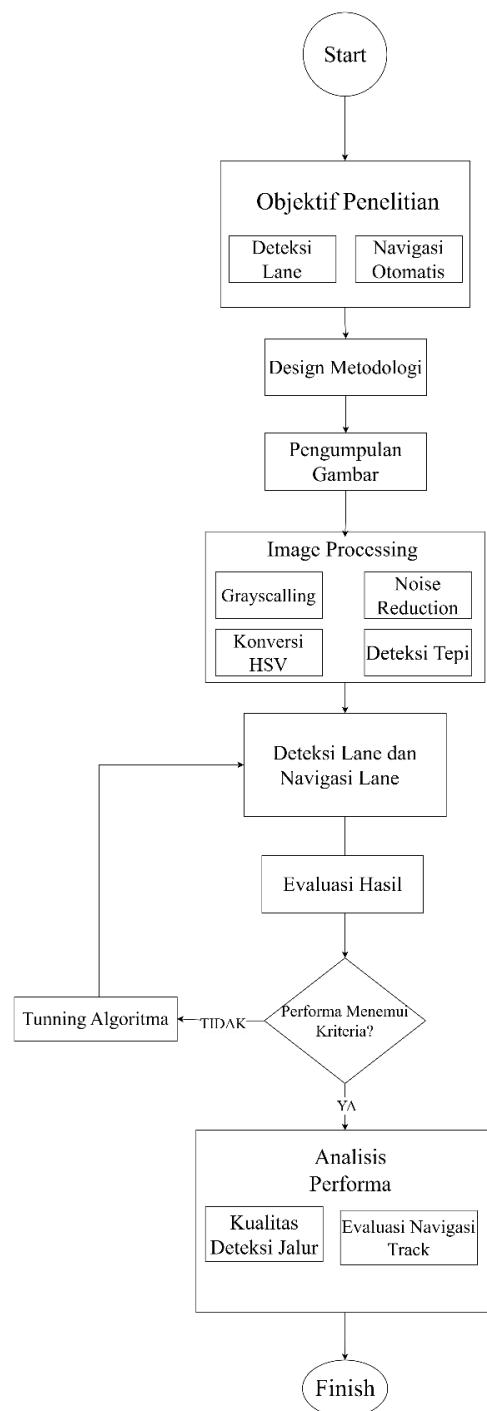
### 2.2.4.2 PyDirectInput

PyDirectInput adalah pustaka Python yang digunakan untuk mengirim input *keyboard* dan mouse secara langsung ke aplikasi, mengatasi masalah kompatibilitas dengan input standar di beberapa permainan atau aplikasi. Pustaka ini memungkinkan simulasi input perangkat keras lebih akurat, yang penting untuk aplikasi yang memerlukan kontrol presisi tinggi seperti dalam pengujian perangkat lunak atau aplikasi permainan.

## BAB III

### METODOLOGI PENGEMBANGAN SIMULASI

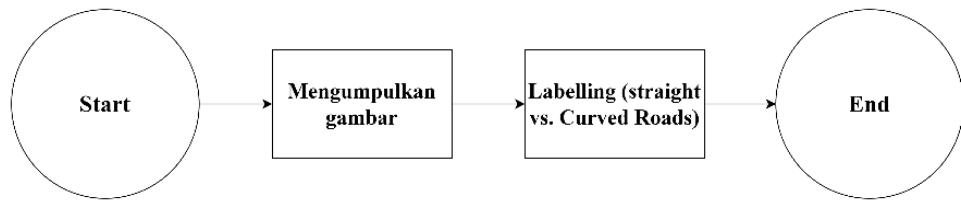
#### 3.1 Diagram Alur Penelitian



**Gambar 2.** Diagram metodologi pengembangan simulasi

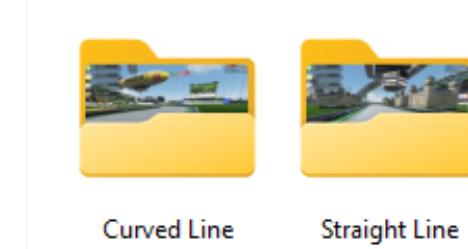
Pada bab ini, akan diuraikan mengenai rancangan sistem untuk mendeteksi jalur dan mengemudikan mobil sepanjang jalur dalam permainan Trackmania Nation. Rancangan tersebut mencakup aspek pengumpulan data gambar, pengelolaan gambar, mendeteksi jalur pada permainan, menggunakan informasi jalur yang terdeteksi untuk mengendalikan mobil serta mengevaluasi kualitas simulasi secara keseluruhan.

### 3.2 Pengumpulan Data



**Gambar 3.** Diagram pengumpulan data

Data dikumpulkan dengan merekam layar permainan. Proses ini melibatkan program Python untuk mengambil tangkapan layar dari permainan secara *real-time* dengan resolusi layar permainan 1080p (*windowed*) dan menyimpan gambar-gambar tersebut ke dalam folder khusus. Setiap gambar diberi ID unik untuk mempermudah pengelolaan dan identifikasi di tahap selanjutnya.



**Gambar 4.** Tangkapan layar disimpan ke dalam dua folder

Gambar hasil tangkapan layar dikelompokkan ke dalam dua folder berdasarkan jenis jalan (lihat Gambar 4).

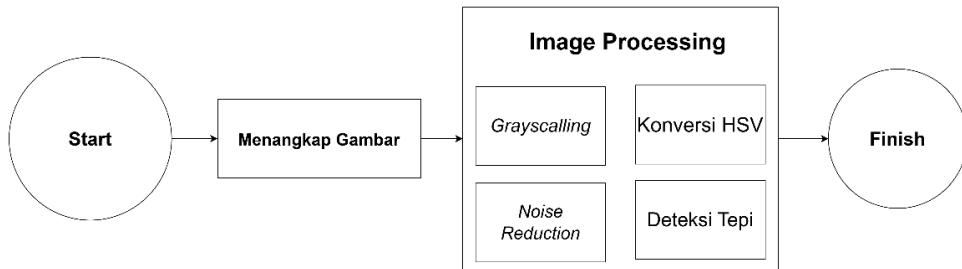
1. Jalan lurus (*straight line*): Pola piksel lebih teratur, sehingga algoritma lebih mudah mengenali tepi dan menentukan jalur.
2. Jalan berbelok (*curved line*): Mengandung lebih banyak variasi bentuk dan intensitas, sehingga membutuhkan pendekatan yang lebih kompleks dalam pendekatan jalur

Pengumpulan gambar dilakukan secara *real-time* yaitu untuk memastikan bahwa gambar yang didapatkan mampu merepresentasikan kondisi jalur pada permainan yang sebenarnya.

Data yang diperoleh diambil pada kondisi jalan yang terlihat dengan jelas tanpa adanya hadangan seperti kendaraan lain, rambu lalu lintas maupun objek yang umumnya terdapat di jalan seperti rambu-rambu lalu lintas. Karena kondisi permainan yang tidak memungkinkan untuk mengatur waktu (siang atau malam), maka data yang diolah terdiri dari satu kondisi waktu saja.

Posisi jalan pada gambar yang diambil juga dapat mempengaruhi kualitas dari hasil simulasi. Semakin jelas gambar jalan, maka hasil deteksi jalur juga akan lebih akurat. Oleh karena itu, posisi pengambilan gambar yang diambil pada simulasi ini adalah dari bagian jalan yang paling mendekati kamera hingga ujung dari jalan sepanjang sumbu y.

### 3.3 Pemrosesan Citra



**Gambar 5.** Flowchart pemrosesan gambar

Setelah data terkumpul, gambar-gambar tersebut dimanipulasi untuk mempersiapkan tahap pemrosesan lebih lanjut seperti yang dapat dilihat pada Gambar 5. Proses ini memanfaatkan pustaka OpenCV, yang menyediakan berbagai fungsi pemrosesan citra secara efisien. Tahapan manipulasi gambar dijelaskan sebagai berikut:

#### 3.3.1 Grayscale

Tahap pertama adalah konversi gambar ke skala abu-abu (*grayscale*). Tujuannya adalah untuk mengurangi kompleksitas gambar dengan menyederhanakan informasi warna. Hal ini mempermudah segmentasi antara jalur jalan dan objek lainnya yang dapat dilihat pada Gambar 6.



**Gambar 6.** Konversi gambar RGB ke *grayscale* mengurangi kompleksitas citra

Setiap piksel RGB dikonversi menjadi nilai intensitas Tunggal (0-255) dengan rata-rata piksel pada tiga kanal warna (R, G, B). Untuk setiap piksel pada masing-masing *channel* akan dikenai algoritma *grayscale* dengan menghitung rata-rata piksel. Contoh pada gambar 3x4 yang direpresentasikan pada Gambar 7.

			255		
	100				
255					255
				255	
			100		

**Gambar 7.** Representasi RGB pada gambar 3x4

Setelah dikenai *grayscale* akan menghasilkan gambar *single channel* yang terdiri dari piksel dengan intensitasi 0-255 dan menghasilkan Gambar 8.

203		
		203

**Gambar 8.** Representasi grayscale pada gambar 3x4

Gambar 8. memiliki jumlah piksel yang jauh lebih sedikit dan memiliki kompleksitas gambar yang jauh lebih rendah jika dibandingkan dengan gambar berwarna (Gambar 7). Oleh karena itu, *Grayscale* akan mengurangi komputasi simulasi dengan cukup signifikan.

### 3.3.2 Noise Reduction

*Noise* adalah piksel-piksel dengan intensitas yang berbeda secara signifikan dibandingkan dengan lingkungan sekitarnya, yang dapat menyebabkan kesalahan deteksi tepi (*false edge*). Gambar 9 merupakan contoh gambar 5x5 pixel dengan distribusi piksel yang tidak seimbang (*noise*) yang akan dikenai operasi pengurangan *noise*.

100	120	150	130	100
110	140	160	140	110
130	150	200	150	120
110	130	150	130	100
90	100	120	110	90

0,166667	0,333333	0,166667
0,333333	0,666667	0,333333
0,166667	0,333333	0,166667

**Gambar 9.** Noise pada gambar 5x5 (kiri) dan 3x3

(kanan) Gaussian kernel

Salah satu metode yang digunakan untuk mengurangi *noise* adalah menggunakan *Gaussian blur* yang memanfaatkan *Gaussian kernel* pada Gambar 9 (kanan) yang berfungsi untuk menghaluskan perubahan intensitas piksel dengan rata-rata bergantung pada area sekitar piksel target.

Setelah melakukan operasi konvolusi, yaitu melakukan perkalian *kernel* Gausian terhadap Gambar 9, maka hasil yang didapatkan berupa gambar yang memiliki distribusi piksel yang jauh lebih seimbang yang dapat dilihat pada Gambar 10.

167	255	283	262	170
242	370	408	372	240
255	388	427	383	245
232	347	378	345	223
152	223	243	228	152

**Gambar 10.** Hasil gambar 9 setelah menggunakan

*Gaussian blur*

Dengan menerapkan operasi Gaussian *blur* pada Gambar 9, hasil yang didapatkan berupa gambar dengan distribusi piksel yang lebih seimbang, sehingga gambar memiliki *noise* yang minim dan nantinya akan memudahkan melakukan operasi deteksi tepi.

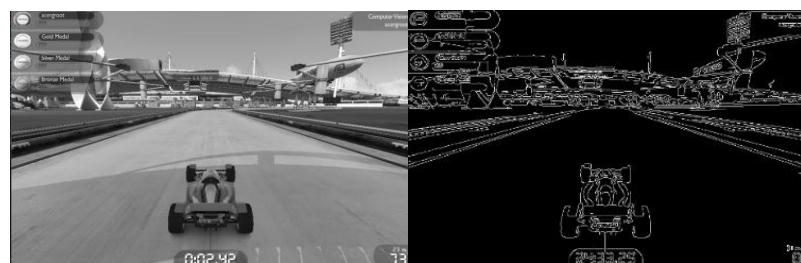


**Gambar 11.** Pengaplikasian *blurring* pada Gambar 6

Penggunaan *blur* mampu membuat distribusi piksel yang jauh lebih seimbang sehingga memudahkan algoritma seperti deteksi tepi *Canny* untuk mendeteksi perubahan intensitas piksel.

### 3.3.3 Deteksi Tepi Canny

Tahap ini menggunakan algoritma deteksi tepi *Canny*, yang merupakan salah satu metode deteksi tepi yang paling populer dan efektif. Deteksi tepi berguna untuk mengidentifikasi marka jalan pada permainan seperti yang terlihat pada Gambar 12.



**Gambar 12.** Penerapan Canny *edge detection* pada citra abu-abu

Tepi didefinisikan sebagai area dengan perubahan intensitas yang signifikan. Algoritma ini menggunakan gradien intensitas untuk mendeteksi tepi secara persisi.

### 1. Gradient Calculation

Algoritma menghitung gradien menggunakan operator Sobel untuk menentukan perubahan intensitas pada gambar yang terdiri dari matriks 3x3 yang dapat dilihat pada Gambar 13 untuk mendeteksi tepi horizontal (x) dan deteksi tepi vertikal (y).

-1	0	1
-2	0	2
-1	0	1

1	2	1
0	0	0
-1	-2	-1

**Gambar 13.** Sobel filter horizontal (kiri)

dan vertikal (kanan)

Untuk setiap pixel pada gambar akan dihitung:

- a. *Gradient magnitude* mengidentifikasi tingkat atau kedalaman perubahan intensitas dalam gambar.



**Gambar 14.** Aplikasi *gradient magnitude*

sumber: Understanding and Implementing Canny Edge Detection  
in Native Python | by Pasan Kalansooriya | Medium

Gambar 14 menunjukkan aplikasi *gradient magnitude* pada sebuah gambar, yang membantu mendeteksi perubahan intensitas pada piksel secara signifikan. *Gradient magnitude* merupakan komponen penting dalam proses deteksi tepi, karena memberikan informasi mengenai kekuatan perubahan intensitas.

422,5416	501,6263	540,6543	502,14	427,0264
497,0609	288,9258	205,6335	281,9921	495,9728
528,1875	235,7468	25,99579	245,7148	524,002
483,2523	283,739	222,514	283,3408	478,4823
408,3695	478,8928	515,0061	474,2692	404,9576

**Gambar 15.** Hasil kalkulasi *gradient magnitude*

Gambar 15 di atas ini menunjukkan hasil kalkulasi gradien pada Gambar 10. Setiap nilai dalam tabel merepresentasikan tingkat perubahan intensitas pada piksel terkait. Nilai tinggi menunjukkan adanya perubahan intensitas yang tajam, sedangkan nilai rendah mengindikasikan perubahan yang lebih halus.

- b. *Gradient direction* untuk menentukan arah perubahan intensitas.

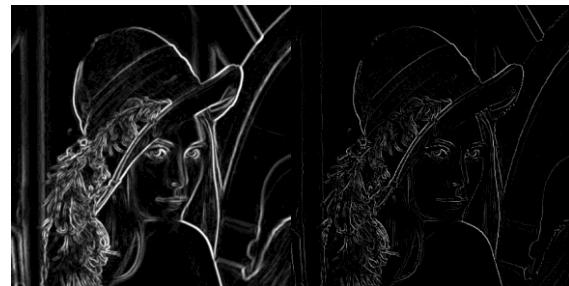
2,341551	2,860041	3,131188	-2,86552	-2,33342
1,808151	2,282707	3,132474	-2,24311	-1,78154
1,555413	1,491178	-0,1691	-1,46632	-1,53978
1,32519	0,794744	-0,01124	-0,8447	-1,34419
0,794056	0,257324	0,004854	-0,26671	-0,81159

**Gambar 16.** Hasil kalkulasi *gradient direction*

Gambar 16 di atas ini menunjukkan hasil kalkulasi *gradient direction* pada Gambar 10. Setiap nilai dalam tabel merepresentasikan orientasi (vertikal maupun horizontal) tepi yang terdeteksi dalam Gambar 6.

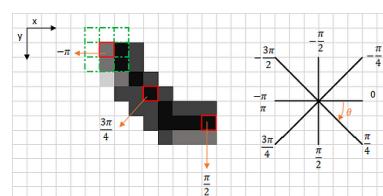
## 2. Non-Maximum Suppression (NMS)

Operasi NMS mempertahankan piksel dengan gradien tertinggi di sekitar suatu area, sehingga garis tepi menjadi lebih tipis dan presisi.



**Gambar 17.** Aplikasi NMS pada hasil *gradient calculation* Sumber: Understanding and Implementing Canny Edge Detection in Native Python | by Pasan Kalansooriya | Medium

Berdasarkan *gradient magnitude* dan *gradient direction* yang diperoleh, setiap piksel akan dibandingkan dengan dua piksel terdekat sesuai arah gradiennya berdasarkan yang dapat dilihat pada Gambar 17. Hasilnya adalah tepi yang lebih bersih dan akurat.



**Gambar 18.** Non-Maximum Suppression

Berdasarkan *gradient magnitude* dan *gradient direction* yang diperoleh pada Gambar 15 dan Gambar 16, operasi NMS berlaku secara lokal dimana, setiap piksel akan dibandingkan dengan dua piksel terdekat berdasarkan nilai *radiance*.

```
pos (1,1) with value 288.926
dir is 2.28271
rad = 130.79   aprox. 135 degree (diagonal left)
compare with top (528.187) and bottom (408.369). So, the current pixel is suppressed
```

**Gambar 19.** Contoh NMS pada piksel (1,1)

Untuk menentukan piksel tetangga yang akan dikenai perbandingan, bulatkan *angle* ke  $45^\circ$  terdekat yang dapat dilihat pada Gambar 19.

- $0^\circ$ : horizontal (approx.  $-22.5^\circ$  to  $+22.5^\circ$ ).
- $45^\circ$ : diagonal (approx.  $+22.5^\circ$  to  $+67.5^\circ$ ).
- $90^\circ$ : vertical (approx.  $+67.5^\circ$  to  $+112.5^\circ$ ).
- $135^\circ$ : diagonal (approx.  $+112.5^\circ$  to  $+157.5^\circ$ ).

**Gambar 20.** Empat arah gradien

Sumber: *Sobel Operator Gradient Calculation*

Maka, tepi akhir yang didapatkan dapat dilihat pada Gambar 21:

0	0	540.654	0	0
497.061	0	0	0	495.973
528.187	0	0	0	524.002
0	0	0	0	0
0	478.893	515.006	474.269	0

**Gambar 21.** Hasil operasi NMS pada

Setiap nilai pada Gambar 21 merupakan intensitas dari piksel pada Gambar 6. Semakin tinggi nilai intensitas pada suatu koordinat, maka semakin jelas perbedaan intensitas piksel pada koordinat tersebut jika dibandingkan dengan piksel tetangganya.

### 3. Double Thresholding

Dua ambang batas (*upper* dan *lower*) digunakan untuk mengidentifikasi tepi kuat, tepi lemah dan tepi tidak relevan. Piksel tepi yang tidak memenuhi kriteria ambang akan dihilangkan.

Menggunakan Gambar 21, *double thresholding* dapat diterapkan dengan menetapkan dua nilai ambang atas dan bawah (perhatikan Gambar 22).

0	0	540.654	0	0
497.061	0	0	0	495.973
528.187	0	0	0	524.002
0	0	0	0	0
0	478.893	515.006	474.269	0
high thres	500			
low thresh	300			

**Gambar 22.** Double Thresholding

Untuk setiap piksel pada Gambar 22 yang memiliki intensitas melebihi batas atas (500) dan tidak kurang dari batas bawah (300) akan dianggap sebagai tepi. Nilai yang tidak memenuhi ambang batas akan dianggap tepi yang tidak relevan (*weak edges*).

0	0	96.25	0	0
82.5	0	0	0	82.5
88.75	0	0	0	85.625
0	0	0	0	0
0	83.75	91.25	85.625	0

**Gambar 23.** Pemetaan NMS pada gambar asli

Hasil yang didapatkan pada Gambar 23 merupakan klasifikasi gradien berdasarkan nilai *threshold*. Untuk setiap piksel yang tidak nol akan dipetakan kembali pada gambar aslinya.

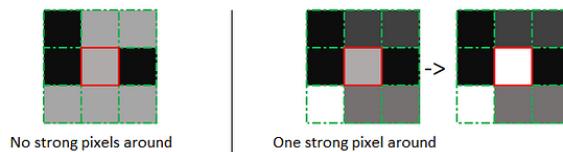


**Gambar 24.** Contoh Aplikasi *double thresholding*

Gambar 24 adalah contoh aplikasi *double thresholding* pada sebuah gambar hasil perhitungan NMS (kiri) dan hasil eliminasi tepi tidak relevan (kanan).

#### 4. Edge Tracking by Hysteresis

Hasil dari *thresholding* akan diproses lebih lanjut jika dan hanya jika terdapat minimal satu piksel yang kuat di sekitar piksel yang sedang di proses.



**Gambar 25.** Eliminasi tepi yang lemah

Sumber: Canny Edge Detection Step by Step in Python — Computer Vision | by Sofiane Sahir | Towards Data Science

Tahapan ini akan menghasilkan representasi yang hanya menunjukkan tepi-tepi signifikan dari jalur jalan.

0	0	high	0	0
low	0	0	0	low
high	0	0	0	high
0	0	0	0	0
0	low	high	low	0

**Gambar 26.** Distribusi tepi lemah dan kuat

Gambar 26 merepresentasikan distribusi tepi yang kuat dan lemah pada Gambar 24. Informasi ini nantinya dapat digunakan untuk mengambil keputusan apakah sebuah tepi pada koordinat tertentu relevan atau tidak.

### 3.3.4 HSV

Pada simulasi ini, marka jalan berwarna hijau sementara permukaan jalan berwarna abu-abu, sehingga metode *grayscale* kurang efektif untuk deteksi tepi. Dengan mengonversi citra ke HSV dan mengisolasi komponen warna hijau, marka jalan dapat diekstrak dengan lebih akurat, meningkatkan keandalan proses deteksi.

Dalam proses deteksi marka jalan, simulasi menggunakan model warna HSV untuk meningkatkan akurasi deteksi warna. Berikut ini adalah langkah-langkah manual dalam mengonversi nilai RGB menjadi HSV dengan menggunakan contoh gambar 2x3.

1. Tabel RGB 2x3 awal

Tabel 1 merupakan contoh gambar 2x3 yang akan dikenai operasi konversi HSV.

**Tabel 1.** Contoh gambar 2x3

Pixel	R	G	B
<b>P1</b>	255	0	0
<b>P2</b>	0	255	0
<b>P3</b>	0	0	255
<b>P4</b>	255	255	0
<b>P5</b>	0	255	255
<b>P6</b>	255	0	255

## 2. Normalisasi RGB

Setiap nilai RGB pada citra dinormalisasi dengan cara membagi masing-masing komponen warna dengan 255, sehingga nilainya berada dalam rentang 0 hingga 1.

**Tabel 2.** Normalisasi RGB

Pixel	R_norm	G_norm	B_norm
P1	1.00	0.00	0.00
P2	0.00	1.00	0.00
P3	0.00	0.00	1.00
P4	1.00	1.00	0.00
P5	0.00	1.00	1.00
P6	1.00	0.00	1.00

## 3. Menghitung Nilai V

Nilai V pada ruang warna HSV dihitung dengan mengambil nilai maksimum dari tiga kanal warna hasil normalisasi, yaitu R', G', dan B'. Nilai ini merepresentasikan tingkat kecerahan (*brightness*) suatu warna.

**Tabel 3.** Nilai V

Pixel	V
P1	1.00
P2	1.00
P3	1.00
P4	1.00
P5	1.00
P6	1.00

## 4. Menghitung Nilai S

Nilai S mengukur tingkat kejemuhan warna, yaitu seberapa "murni" warna tersebut dibandingkan dengan warna abu-abu. Rumus yang digunakan untuk menghitung nilai S adalah:

Nilai minimum dari ketiga kanal ( $R'$ ,  $G'$ ,  $B'$ ) dikurangkan dari nilai maksimum ( $V$ ), kemudian hasilnya dibagi dengan  $V$  untuk mendapatkan nilai kejemuhan. Karena pada tabel sebelumnya nilai  $V$  semuanya adalah 1.00 dan salah satu kanal bernilai 0, maka perhitungan  $S$  akan selalu menghasilkan 1.00.

**Tabel 4.** Nilai S

Pixel	S
P1	1.00
P2	1.00
P3	1.00
P4	1.00
P5	1.00
P6	1.00

##### 5. Menghitung Nilai H

Rumus *hue* bergantung pada nilai maksimum:

$$H = \begin{cases} 60 \times \frac{G' - B'}{V - \min(R', G', B')}, & \text{jika } V = R' \\ 60 \times (2 + \frac{G' - B'}{V - \min(R', G', B')}), & \text{jika } V = G' \\ 60 \times (4 + \frac{G' - B'}{V - \min(R', G', B')}), & \text{jika } V = B' \end{cases}$$

Jika hasil perhitungan menghasilkan nilai negatif, maka  $360^\circ$  akan ditambahkan agar nilai berada dalam rentang  $0^\circ$  hingga  $360^\circ$ . Nilai Hue ini dinyatakan dalam derajat dan menentukan spektrum warna (misalnya  $0^\circ$  untuk merah,  $120^\circ$  untuk hijau,  $240^\circ$  untuk biru, dan seterusnya).

Tabel berikut menampilkan hasil perhitungan nilai Hue dari masing-masing piksel berdasarkan rumus di atas:

**Tabel 5.** Nilai Hue

Pixel	H (derajat)
P1	0°
P2	120°
P3	240°
P4	60°
P5	180°
P6	300°

#### 6. Hasil akhir dalam format HSV

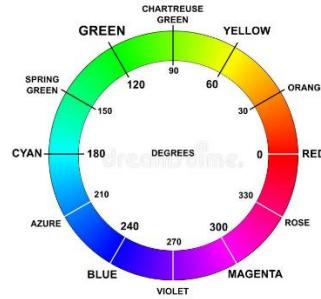
Derajat pada nilai Hue dalam model HSV merepresentasikan warna dalam spektrum 360°.

**Tabel 6.** Hasil konversi HSV

Pixel	H	S	V
P1	0°	1.00	1.00
P2	120°	1.00	1.00
P3	240°	1.00	1.00
P4	60°	1.00	1.00
P5	180°	1.00	1.00
P6	300°	1.00	1.00

- 0° (Merah): Warna merah terletak di awal spektrum HSV.
- 60° (Kuning): Warna yang dihasilkan dari perpaduan merah dan hijau.
- 120° (Hijau): Warna hijau dominan.
- 180° (Cyan): Warna hasil perpaduan hijau dan biru.
- 240° (Biru): Warna biru dominan.
- 300° (Magenta): Warna hasil perpaduan biru dan merah.

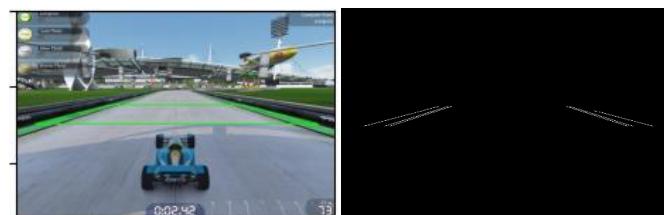
Derajat ini menunjukkan posisi warna pada spektrum warna melingkar, di mana perubahan kecil dalam nilai *Hue* dapat menghasilkan perbedaan warna yang signifikan.



**Gambar 27.** Derajat HSV  
sumber: <https://redrainkim.github.io/assets/images/color-colors-wheel-names-degrees-rgb-hsb-hsv-hue->

### 3.3.5 RoI

RoI digunakan untuk mengisolasi area jalan dari objek lain pada gambar. Proses ini mencakup menentukan area berbentuk *trapezoid* atau poligon yang mencakup jalur jalan dan mengabaikan piksel di luar area ini.



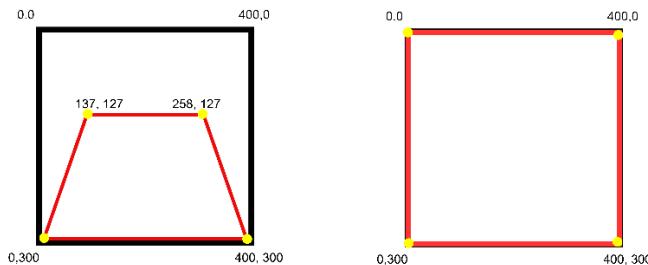
**Gambar 28.** Implementasi RoI pada hasil deteksi tepi Canny

Isolasi area jalan membantu meningkatkan akurasi proses berikutnya, seperti pendekripsi garis pada *sliding window*. Dengan hanya memproses jalur jalan, simulasi dapat fokus pada objek yang benar-benar penting untuk pengemudi otomatis.

### 3.3.6 Perspective Transform

Transformasi perspektif (*perspective transform*) adalah jenis transformasi geometri yang memetakan titik-titik dari satu perspektif (misalnya, tampilan kamera menghadap ke depan) ke perspektif lain (misalnya, tampilan *bird's-eye view* atau tampilan dari atas).

Transformasi ini direpresentasikan oleh matriks transformasi  $3 \times 3$  dan sering dihitung menggunakan homografi (perhatikan contoh Gambar 29).



**Gambar 29.** Transformasi RoI menjadi *Bird's eye view*

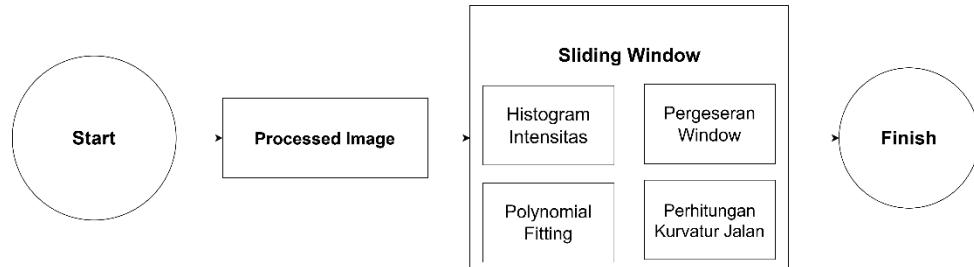
Dalam deteksi jalur, kamera yang menghadap ke depan menangkap jalan dari sudut pandang yang membuat garis-garis jalur terlihat menyatu di kejauhan (akibat sudut pandang kamera).

Transformasi perspektif dapat:

1. Menghilangkan distorsi perspektif: Mengubah gambar menjadi tampilan *bird's-eye view*, di mana garis-garis jalur yang sejajar terlihat paralel.
2. Menyederhanakan deteksi jalur: Memudahkan pendekripsi garis-garis jalur.

3. Meningkatkan akurasi: Memberikan tampilan yang lebih jelas tentang tata letak jalan untuk pemrosesan lebih lanjut.

### 3.4 Diagram Alur Deteksi Jalur



**Gambar 30.** Flowchart Sliding Window

Algoritma utama dalam simulasi ini adalah mendeteksi piksel pada permainan Trackmania yang mengindikasikan jalur. Informasi ini nantinya akan digunakan sebagai patokan untuk mengemudikan mobil sepanjang jalur jalan yang terdeteksi.

Adapun tahapan-tahapan yang dilakukan untuk mendeteksi jalur jalan yaitu melakukan mengirimkan gambar yang telah diproses secara *real-time* kedalam algoritma *Sliding Window* untuk melakukan ekstraksi jalur jalan.

#### 3.4.1 Sliding Window

Tahapan ini menggunakan algoritma *sliding window* untuk mendeteksi jalur jalan pada gambar yang sudah melalui proses ROI. *Sliding window* membantu menelusuri dan menentukan posisi garis secara bertahap dengan pendekatan *window-based*. *Sliding window* membagi area gambar menjadi beberapa *window* kecil, yang kemudian dipindahkan (*slid*) secara bertahap sepanjang jalur jalan untuk mendeteksi garis.

Tahapan penerapan sliding window untuk menentukan jalur jalan adalah sebagai berikut:

#### 3.4.1.1 Histogram Intensitas Piksel

Histogram intensitas piksel digunakan untuk menentukan posisi awal garis. Bagian bawah gambar diambil untuk membuat histogram intensitas horizontal, yang menunjukkan distribusi intensitas piksel di sepanjang lebar gambar. Pada bagian puncak histogram menunjukkan area dengan konsentrasi intensitas tinggi, yang mengindikasikan lokasi jalur.

#### 3.4.1.2 Deteksi Posisi Awal

Posisi awal garis ditentukan berdasarkan histogram intensitas piksel pada bagian bawah gambar. Puncak histogram menunjukkan area dengan intensitas piksel tinggi, yang sering kali merupakan garis jalan.



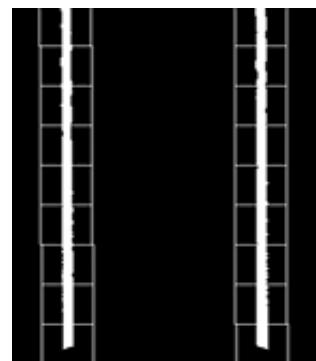
**Gambar 31.** Histogram intensitas piksel untuk mengidentifikasi posisi awal jalur

Sumber: 50220200616-82467-1kw0uel-libre.pdf, hal: 8

Pada Gambar 31, intensitas jalur kiri dan kanan akan memiliki histogram yang jauh lebih tinggi dibandingkan piksel lainnya yang akan digunakan sebagai posisi awal *window*.

### 3.4.1.3 Pergeseran Window

*Window* digeser ke atas berdasarkan deteksi piksel yang termasuk dalam garis. Jika terdapat banyak piksel dalam sebuah *window* yang sesuai dengan jalur, posisi *window* diperbarui.



**Gambar 32.** Sliding window pada hasil deteksi tepi

Pada kasus tertentu jika piksel jalan tidak ditemukan, posisi terakhir dianggap sebagai titik referensi untuk *window* berikutnya.

### 3.4.1.4 Menggambar Garis Polinomial

Garis jalan digambar berdasarkan posisi *window* yang terdeteksi pada setiap langkah berdasarkan koordinat yang diperoleh dari fungsi *polinominal* kuadrat.

$$x = ay^2 + by + c \quad (15)$$

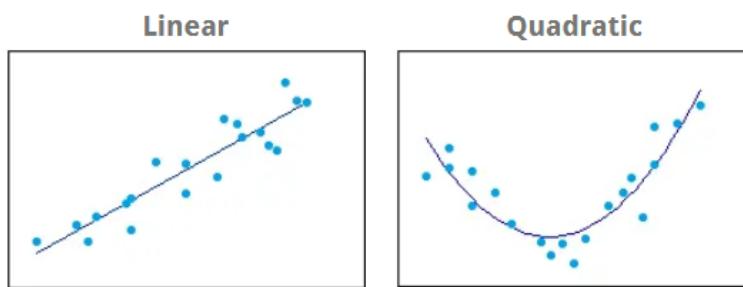
Dimana:

*a*: Mendeteksi *curvature* pada garis

*b*: Representasi *slope* dari garis

*c*: *y-intercept* (dimana garis dimulai pada *y-axis* pada bagian bawah gambar)

*Sliding window* akan mengembalikan nilai berupa koordinat x dan y untuk masing-masing garis (kiri dan kanan). *Sliding window* membantu melacak jalur jalan secara akurat meskipun terdapat lengkungan atau perubahan arah. Algoritma ini juga efektif dalam mengurangi *noise* karena hanya fokus pada area dalam *window*.



**Gambar 33.** Polinomial linear (kiri) dan polinomial kuadrat (kanan)

Sumber: [https://sklc-tinymce-2021.s3.amazonaws.com/2020/09/mceclip1\\_1600436474.png](https://sklc-tinymce-2021.s3.amazonaws.com/2020/09/mceclip1_1600436474.png)

Dari gambar di atas, terlihat bahwa *polinomial* kuadrat (kanan) lebih mampu mengikuti bentuk jalur yang melengkung dibandingkan dengan *polinomial* linear (kiri) yang hanya cocok untuk jalur lurus. Dalam konteks deteksi jalur kendaraan, jalur di dunia nyata sering kali tidak sepenuhnya lurus dan memiliki lengkungan. Oleh karena itu, penggunaan *polinomial* kuadrat lebih disukai karena mampu memodelkan bentuk jalur

dengan lebih akurat, mengikuti kurva jalan, serta meningkatkan ketepatan dalam estimasi lintasan kendaraan.

### 3.4.2 Perhitungan Radius Kelengkungan

Untuk menentukan bentuk jalan, radius kelengkungan dihitung berdasarkan deteksi marka jalan yang telah diperoleh. Nilai ini memberikan indikasi apakah jalan tersebut lurus, berbelok ringan, atau tajam. Ini dilakukan dengan menghitung radius kelengkungan menggunakan persamaan:

$$R = \frac{(1 + (2 \times \text{fit}[0] \times y_{\text{eval}} + \text{fit}[1])^2)^{1.5}}{|2 \times \text{fit}[0]|} \quad (16)$$

Di mana:

$A, B$  : adalah koefisien dari *polinomial* hasil *fitting* garis marka jalan.

$y$  : adalah posisi vertikal pada gambar yang dievaluasi.

Hasil perhitungan dilakukan konversi dari piksel ke meter dengan skala:

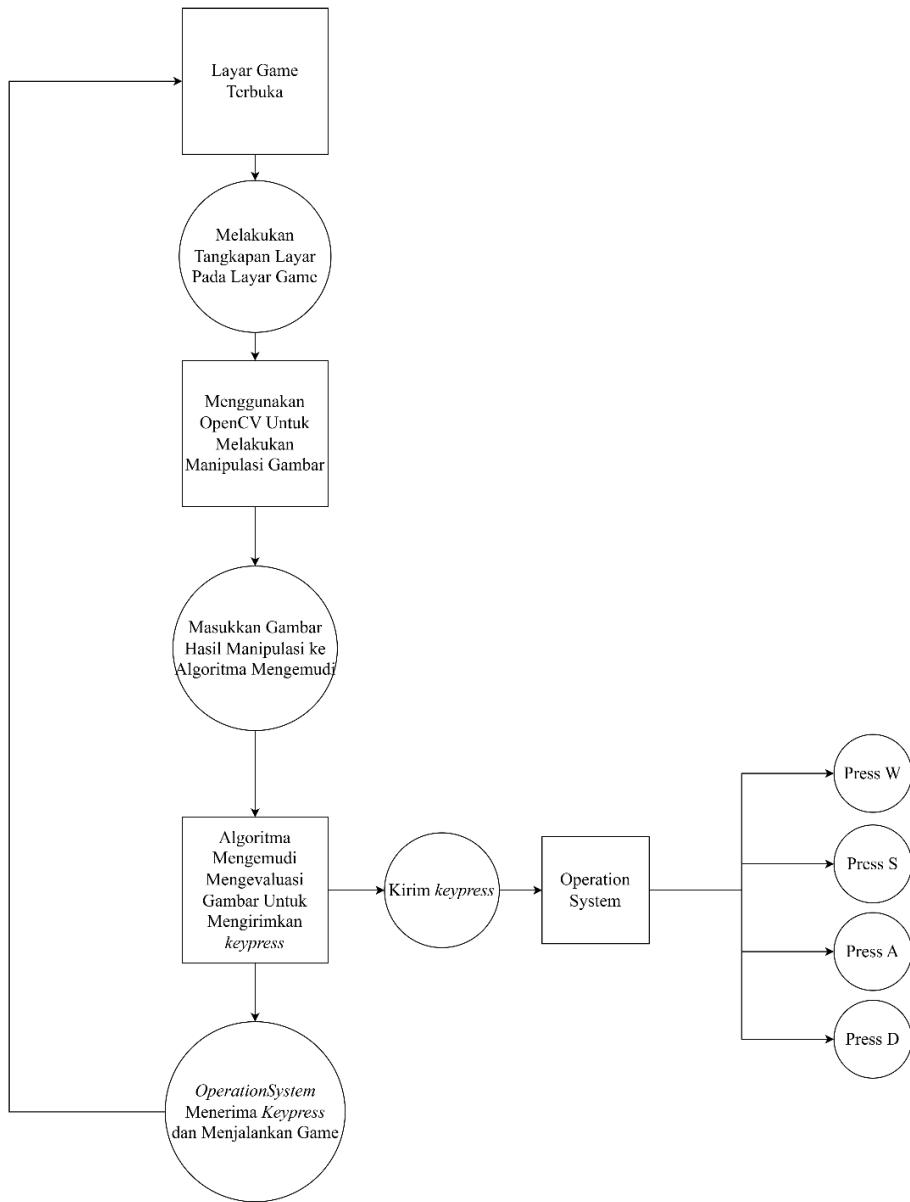
- $\text{XM\_PER\_PIX} = 2 / 400$  (lebar jalan ~2 meter, diwakili oleh 400 piksel).
- $\text{YM\_PER\_PIX} = 10 / 300$  (jarak 10 meter ke depan, diwakili oleh 300 piksel).

Berdasarkan hasil konversi tersebut, radius kelengkungan dikategorikan sebagai berikut:

1. Jalan lurus:  $R \geq 1000$  m
2. Belokan ringan :  $300 \text{ m} \leq R < 1000$  m
3. Belokan tajam:  $R < 300$  m

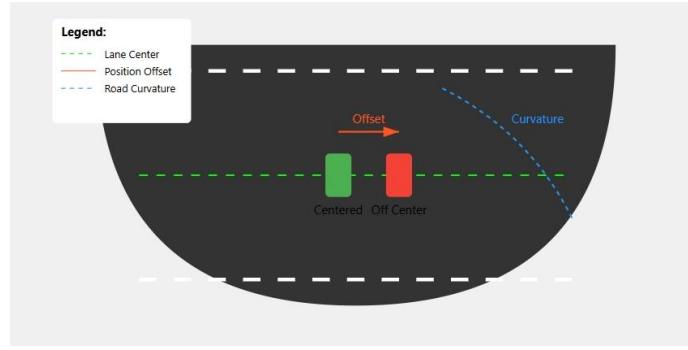
Dengan informasi ini, kendaraan dapat menyesuaikan kecepatan atau algoritma kemudi untuk mengantisipasi perubahan bentuk jalan.

### 3.5 Diagram Alur Algoritma Mengemudi Berdasarkan Jalur



**Gambar 34.** Flowchart simulasi

Tahapan ini merupakan inti dari pengemudian otomatis berbasis jalur. Algoritma ini menggunakan informasi dari jalur yang terdeteksi untuk menentukan aksi pengemudian dengan mengirimkan *keypress* berupa tombol-tombol pada *keyboard*.



**Gambar 35.** Kontrol kemudi berdasarkan jalur

Algoritma akan dirancang untuk menavigasi kendaraan dengan mempertimbangkan posisi jalur relatif terhadap mobil. Jalur digunakan sebagai panduan untuk menentukan arah dan kecepatan pengemudian.

Tahapan ini meliputi:

1. Mengidentifikasi Posisi Jalur

Posisi jalur relatif terhadap kendaraan dihitung berdasarkan koordinat hasil *Sliding Window* dengan menghitung rata-rata kurvatur. Simulasi menggunakan pendekatan kuantitatif untuk menerjemahkan posisi jalur:

- a. *Off-center* ke kanan (nilai *offset* positif)

Jika mobil mendekati jalur bagian kanan maka belok ke kiri.

- b. *Off-center* ke kiri (nilai *offset* negatif)

Jika mobil mendekati jalur bagian kiri maka belok ke kanan.

- c. Mendekati jalur lurus (rata-rata kurvatur  $\geq 1.000$ ):

- Mobil bergerak lurus
- Minimal koreksi kemudi
- Mengirim *key press* tombol W pada *keyboard*.

- d. Mendekati belok kiri atau kanan (rata-rata kurvatur < 800)
  - Belok ringan ke kiri atau ke kanan (sudut kecil)
  - Koreksi gradual untuk menjaga posisi di tengah jalur
  - Mengirim *key press* tombol keyboard A (kiri) dan B (kanan) pada *keyboard*.
- e. Kurvatur ekstrem: Jika melebihi nilai *threshold* yang ditetapkan, mobil diarahkan untuk belok tajam dengan penyesuaian kecepatan untuk keamanan.

## 2. Logika Pengambilan Keputusan

Algoritma mengimplementasikan hierarki keputusan berbasis prioritas:

- a. Prioritas Utama: Menjaga mobil di dalam jalur
- b. Prioritas Kedua: Minimalisasi koreksi kemudi
- c. Prioritas Ketiga: Mempertahankan kecepatan stabil

## 3. Eksekusi dengan PyAutoGUI dan PyDirectInput

Pustaka Python seperti PyAutoGUI dan PyDirectInput digunakan untuk mengirim input kontrol ke permainan secara otomatis. Algoritma mengirimkan perintah berupa *keypress* berdasarkan jalur sebagai berikut:

- a. Mengirim perintah keyboard
- b. Menyimulasikan input *steering*
- c. Mengontrol akselerasi

Dengan pendekatan ini, kendaraan dapat bergerak secara mandiri sesuai dengan jalur yang terdeteksi, baik pada jalur lurus maupun melengkung.

Implementasi ini memastikan navigasi yang presisi dengan penyesuaian *real-time* terhadap kondisi jalur.

### 3.6 Matrix Evaluasi Simulasi

Evaluasi kinerja algoritma dilakukan berdasarkan matriks yang telah disebutkan pada Bab 2. Matriks ini mencakup pengukuran kualitatif dan kuantitatif dari hasil deteksi jalur serta performa navigasi kendaraan pada simulasi TrackMania.

#### 3.6.1 Evaluasi Deteksi Jalur

Evaluasi ini bertujuan untuk mengetahui performa algoritma deteksi jalur dalam berbagai kondisi jalan pada permainan.

##### 3.6.1.1 IoU

IoU digunakan untuk mengukur akurasi deteksi jalur dengan membandingkan area tumpang tindih antara jalur yang terdeteksi dan jalur *ground truth*. IoU dihitung dengan rumus berikut:

$$\text{IoU} = \frac{|A \cap B|}{|A \cup B|}$$

Misalkan hasil deteksi jalur dengan data sebagai berikut:

1. Jalur *ground truth*: 500 piksel persegi
2. Jalur yang terdeteksi: 520 piksel persegi
3. Area tumpang tindih: 450 piksel persegi

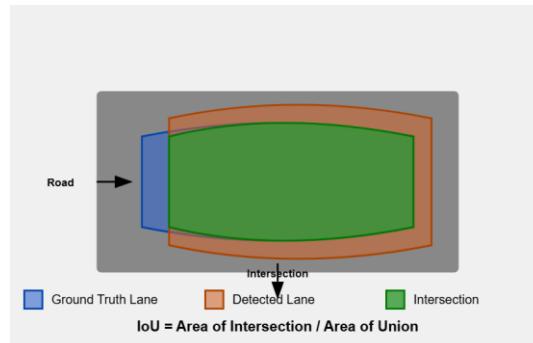
Menggunakan rumus IoU:

$$\text{IoU} = \frac{450}{500 + 520 - 450} = \frac{450}{570} \approx 0.79$$

Karena nilai IoU berada di antara 0 dan 1, interpretasi hasilnya adalah sebagai berikut:

1.  $\text{IoU} > 0.75$  : Deteksi jalur sangat akurat
2.  $0.5 \leq \text{IoU} \leq 0.75$  : Deteksi jalur cukup baik, tetapi masih bisa diperbaiki
3.  $\text{IoU} < 0.5$  : Deteksi jalur kurang akurat dan perlu diperbaiki

Berdasarkan hasil di atas, simulasi memiliki akurasi cukup baik dengan nilai IoU 0.79, menunjukkan bahwa jalur yang terdeteksi memiliki tumpang tindih yang tinggi dengan *ground truth*.



**Gambar 36.** Representasi IoU

Gambar 36 merepresentasikan nilai IoU, di mana warna biru menunjukkan *ground truth*, sedangkan warna oranye merupakan jalur yang terdeteksi oleh simulasi. Area *intersection* ditandai dengan warna hijau, yang menunjukkan bagian jalur terdeteksi yang tumpang tindih (*overlapping*) dengan *ground truth*.

### **3.6.1.2 Performa Kualitatif Konsistensi Deteksi Jalur**

Konsistensi deteksi jalur dinilai dengan menghitung seberapa sering algoritma berhasil mendeteksi jalur secara kontinu dalam berbagai kondisi, termasuk tikungan, jalur lurus, dan area dengan *noise* tinggi. Evaluasi dilakukan dengan menganalisis jumlah *frame* di mana jalur terdeteksi dibandingkan dengan total *frame* dalam satu putaran simulasi.

### **3.6.2 Evaluasi Navigasi Track**

Evaluasi ini bertujuan untuk mengetahui performa algoritma mengemudi berbasis aturan untuk melakukan navigasi pada berbagai kondisi jalan dalam permainan.

#### **3.6.2.1 Track Berhasil Diselesaikan**

Metode evaluasi ini menghitung jumlah *track* yang berhasil diselesaikan kendaraan tanpa keluar jalur atau mengalami tabrakan. Matriks ini digunakan untuk menilai stabilitas algoritma pada navigasi jalur.

#### **3.6.2.2 Jarak Tempuh Sebelum Menabrak**

Jarak tempuh dihitung sebagai total panjang jalur yang dilalui kendaraan sebelum terjadi tabrakan atau kendaraan keluar dari jalur. Evaluasi ini memberikan gambaran tentang efektivitas algoritma dalam menjaga kendaraan tetap pada jalur dalam kondisi simulasi yang berbeda.

Tingkat keberhasilan didefinisikan sebagai persentase dari total simulasi di mana kendaraan berhasil menyelesaikan putaran tanpa

tabrakan atau keluar jalur. Tingkat keberhasilan memberikan indikasi keseluruhan efektivitas algoritma pada berbagai skenario.

## BAB IV

### HASIL SIMULASI DAN PEMBAHASAN

#### 4.1 Pengumpulan Data

Data dikumpulkan dengan cara mengambil tangkapan layar pada *frame* permainan secara otomatis menggunakan pustaka PyAutoGUI yang bisa dilihat pada Gambar 37.

```
def capture_screenshots(num_screenshots, delay=1, save_dir="screenshots"):
    # Create the directory if it doesn't exist
    if not os.path.exists(save_dir):
        os.makedirs(save_dir)

    for i in range(num_screenshots):
        # Capture screenshot
        screenshot = py.screenshot(region=(50,50, 800, 600))
        # {"top": 50, "left": 50, "width": 800, "height": 600}
        # capture fullscreen
        # screenshot = py.screenshot()

        # Save the screenshot to the specified directory with a unique filename
        save_path = os.path.join(save_dir, f'screenshot_{i+1}.png')
        screenshot.save(save_path)

        print(f'Screenshot {i+1} saved at {save_path}')

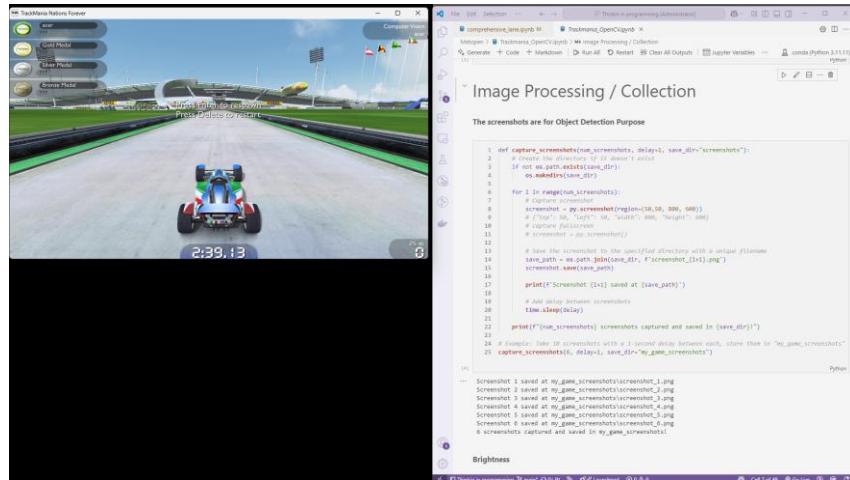
        # Add delay between screenshots
        time.sleep(delay)

    print(f'{num_screenshots} screenshots captured and saved in {save_dir}!')

# Example: Take 10 screenshots with a 1-second delay between each, store them in "my_game_screenshots"
capture_screenshots(6, delay=1, save_dir="my_game_screenshots")
```

**Gambar 37.** Kode mengambil tangkapan layar pada posisi tertentu di monitor

Untuk menyimpan hasil tangkapan layar, kode tersebut membuat folder bernama *screenshots* dan melakukan perulangan sebanyak parameter *num\_screenshots*, yang merupakan jumlah tangkapan yang akan diambil. Setelah itu, gambar ditempatkan di layar monitor (1920 x 1080). Posisi gambar di sumbu x dan y juga dihitung, serta tinggi (600) dan lebar (800).



**Gambar 38.** Proses penangkapan gambar dengan ukuran 800 x 600 pixel

Kode pada Gambar 37 dijalankan bersamaan dengan layar permainan seperti yang terlihat pada Gambar 38. Setelah ditangkap, gambar akan disimpan sesuai ukuran yang telah ditentukan.

## 4.2 Pemrosesan Gambar

### 4.2.1 Deteksi Tepi

Setelah mengumpulkan gambar, langkah berikutnya adalah menggunakan operasi deteksi tepi Canny selama simulasi yang berlangsung secara *real-time*. Operasi ini membutuhkan gambar yang hanya terdiri dari satu *channel*, misalnya abu-abu, dan melalui tahap blur menggunakan kernel Blur Gaussinan.

Kernel adalah matriks kecil yang digunakan dalam operasi konvolusi untuk memproses gambar, dan blur mengacu pada efek pelembutan gambar untuk mengurangi *noise* sebelum mendeteksi tepi. Ini karena mendeteksi tepi hanya memerlukan intensitas piksel. Gambar 39

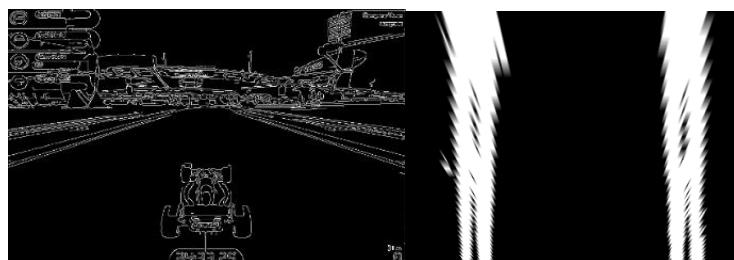
menunjukkan implementasi deteksi tepi Canny menggunakan OpenCV, termasuk penerapan ambang batas dan konversi ke skala abu-abu.

```
def canny_edge_detection(img):
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY )
    kernel = 5
    blur = cv.GaussianBlur(gray, (kernel, kernel),0)
    canny = cv.Canny(blur, 50, 150)

    return canny
```

**Gambar 39.** Fungsi deteksi tepi *Canny* menggunakan OpenCV

Pada tahap awal pengembangan simulasi, deteksi tepi Canny digunakan untuk mengidentifikasi marka jalan. Proses ini dimulai dengan mengonversi citra ke skala abu-abu menggunakan *cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)*, kemudian menerapkan algoritma Canny dengan parameter ambang batas bawah dan atas (*cv2.Canny(gray, 50, 150)*). Untuk memperjelas hasil deteksi, dilakukan dilasi menggunakan *cv2.dilate(edges, None, iterations=1)* agar tepi yang terdeteksi menjadi lebih tegas.



**Gambar 40.** Hasil deteksi deteksi tepi Canny

Berdasarkan eksperimen, metode *grayscale* memiliki keterbatasan dalam mendeteksi marka jalan berwarna hijau karena warna jalan yang dominan abu-abu menyebabkan kontras yang rendah antara marka dan latar belakangnya. Hal ini membuat deteksi tepi Canny kurang efektif dalam

menangkap perbedaan yang halus antara kedua area tersebut seperti yang terlihat pada Gambar 40.

Sebagai solusi, model HSV digunakan untuk meningkatkan akurasi deteksi marka jalan. Dengan memanfaatkan HSV, warna hijau pada marka jalan dapat diisolasi secara lebih efektif, sehingga hasil deteksi menjadi lebih jelas dan konsisten dibandingkan metode *grayscale*.



**Gambar 41.** Hasil deteksi batas jalan berwarna hijau menggunakan HSV

Berdasarkan eksperimen, penggunaan HSV menghasilkan deteksi marka jalan yang lebih konsisten dibandingkan *grayscale*.

#### 4.2.2 RoI

Untuk meningkatkan efisiensi pemrosesan citra dan mengurangi gangguan dari area di luar marka jalan, RoI ditetapkan agar hanya bagian jalan yang relevan yang diproses. RoI didefinisikan untuk membatasi area pemrosesan citra hanya pada bagian jalan yang relevan, sehingga mengurangi gangguan dari area lain di luar marka jalan. Dengan isolasi ini, algoritma deteksi dapat bekerja lebih efisien dan akurat.



**Gambar 42.** Visualisasi posisi RoI

Resolusi gambar juga dikurangi menjadi 400 x 300 untuk meningkatkan efisiensi pemrosesan. Akibatnya, koordinat RoI juga diubah. Untuk memastikan bahwa hanya area jalan yang diproses, RoI ini divisualisasikan sebagai poligon pada citra menggunakan *poliline cv2*. Bagian eksperimen parameter akan menjelaskan lebih lanjut tentang perubahan parameter ini dan bagaimana hal itu berdampak pada kinerja simulasi (subbab 4.7).

#### 4.2.3 Perspective Transform

Perspektif transformasi digunakan untuk mengubah tampilan citra dari sudut pandang kamera menjadi tampilan, sehingga mempermudah deteksi marka jalan dengan menghilangkan distorsi perspektif.

Proses ini dilakukan menggunakan fungsi *cv2.warpPerspective*, yang menerapkan matriks transformasi berdasarkan titik-titik koordinat RoI. Titik destinasi RoI telah ditetapkan sebagaimana ditunjukkan pada Gambar 43, menghasilkan citra keluaran berukuran 400×300 piksel.

```

self.desired_roi_points = np.float32([
    [50, 0],           # Top-Left
    [350, 0],          # Top-right
    [350, 300],         # Bottom-right
    [50, 300]          # Bottom-Left
])
self.transformation_matrix = cv2.getPerspectiveTransform(self.roi_points, self.desired_roi_points)
self.inv_transformation_matrix = cv2.getPerspectiveTransform(self.desired_roi_points, self.roi_points)

```

```
self.warped_frame = cv2.warpPerspective(
    frame, self.transformation_matrix, self.orig_image_size, flags=(cv2.INTER_LINEAR))
```

**Gambar 43.** Koordinat destinasi RoI

Untuk menjaga kualitas citra setelah transformasi, interpolasi *cv2.INTER\_LINEAR* digunakan agar tepi marka jalan tetap jelas dan minim distorsi. Dengan perspektif yang telah dikoreksi, marka jalan menjadi sejajar dalam hasil transformasi, sehingga mempermudah proses perhitungan histogram intensitas dan deteksi menggunakan metode *sliding window*. Hasil dari perspektif *warping* dapat dilihat pada Gambar 44.



**Gambar 44.** Hasil *perspective warping*

Untuk memvisualisasikan hasilnya secara langsung di atas layar permainan, deteksi jalur pada gambar perspektif *bird's eye* harus dikembalikan ke perspektif awal. Fungsi *cv2.warpPerspective* dengan matriks invers dari transformasi awal juga digunakan dalam proses ini. Oleh karena itu, garis jalur yang telah ditemukan dapat ditampilkan dengan benar mengikuti kontur jalan dalam permainan.

## 4.3 Deteksi Jalur

### 4.3.1 Metode Sliding Window

Metode *sliding window* digunakan untuk mendeteksi marka jalan secara akurat setelah citra mengalami transformasi perspektif yang dilampirkan pada Gambar 45. Proses ini diawali dengan perhitungan histogram intensitas piksel pada citra hasil *perspective warping*, yang berfungsi untuk mengidentifikasi posisi awal marka jalan kiri dan kanan. Berdasarkan histogram tersebut, area dengan kepadatan piksel tertinggi ditetapkan sebagai titik awal deteksi marka jalan.

```

def sliding_window(img_warped):
    histogram = np.sum(img_warped[int(img_warped.shape[0]//2):,:], axis=0)
    # find the peak of the left and right halves of the histogram
    # this will be the starting points for the left and right lane lines
    output_img = np.zeros_like(img_warped)

    midpoint = np.int32(histogram.shape[0]//2)
    leftx_base = np.argmax(histogram[:midpoint])
    rightx_base = np.argmax(histogram[midpoint:]) + midpoint

    # Number of sliding windows
    nwindows = 10

    # Set height of windows
    window_height = np.int32(img_warped.shape[0]//nwindows)

    # Identify the x and y positions of all nonzero pixels in the image.
    nonzero = img_warped.nonzero()
    nonzeroy = np.array(nonzero[0])
    nonzerox = np.array(nonzero[1])

    # Current position to be updated for each window
    leftx_current = leftx_base
    rightx_current = rightx_base

    # set the width of the window
    margin = 100

    # set the minimum number of pixels found to recenter window
    minpix = 50

    # create empty lists to receive left and right lane pixels indices
    left_lane_inds = []
    right_lane_inds = []

    # step through the windows one by one
    for window in range(nwindows):
        # Identify window boundaries in x and y
        win_y_low = img_warped.shape[0] - (window+1)*window_height
        win_y_high = img_warped.shape[0] - window*window_height

        win_xleft_low = leftx_current - margin
        win_xleft_high = leftx_current + margin

        win_xright_low = rightx_current - margin
        win_xright_high = rightx_current + margin

```

```

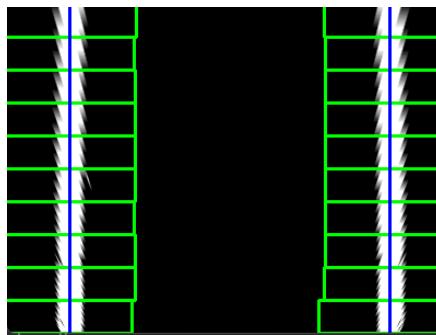
# Identify the nonzero pixels in x and y within the current window
good_left_inds = ((nonzeroy >= win_y_low) & (nonzeroy < win_y_high) &
                  (nonzerox >= win_xleft_low) & (nonzerox < win_xleft_high)).nonzero()[0]
good_right_inds = ((nonzeroy >= win_y_low) & (nonzeroy < win_y_high) &
                   (nonzerox >= win_xright_low) & (nonzerox < win_xright_high)).nonzero()[0]

# append these indices to the list
left_lane_inds.append(good_left_inds)
right_lane_inds.append(good_right_inds)

```

**Gambar 45.** Lampiran sliding window

*Sliding window* diterapkan secara bertahap dari bagian bawah citra ke atas untuk melacak bentuk marka jalan seperti yang dilampirkan pada Gambar 46. Hasil deteksi pada setiap langkah digunakan untuk membentuk model *polinomial* yang merepresentasikan kurva marka jalan, dengan parameter *left\_fit* untuk sisi kiri dan *right\_fit* untuk sisi kanan.



**Gambar 46.** Visualisasi *Sliding Window*

Untuk meningkatkan stabilitas deteksi antar *frame*, diterapkan teknik *smoothing* dengan pendekatan eksponensial, di mana parameter baru dihitung sebagai kombinasi dari deteksi sebelumnya dan deteksi saat ini. Teknik ini membantu mengurangi fluktuasi yang dapat disebabkan oleh variasi pencahayaan (pada jalan di dunia nyata) atau ketidak sempurnaan deteksi. Selain itu, untuk memastikan kontinuitas pada marka jalan yang berbelok, metode *get\_lane\_line\_previous\_window* digunakan untuk memperbarui posisi jendela berdasarkan informasi deteksi dari *frame* sebelumnya.

### 4.3.2 Debug dan Visualisasi

*Debug* digunakan untuk memverifikasi hasil deteksi marka jalan.

Citra-citra seperti deteksi jalur, *ROI Image*, dan *Warped Frame* ditampilkan menggunakan *cv2.imshow*. Citra *Warped Frame* menunjukkan marka jalan yang telah disajarkan, sedangkan fungsi *ROI Image* menampilkan poligon ROI yang digambar pada citra asli.

```
def real_time_lane_detection():
    sct = mss()
    monitor = {"top": 300, "left": 100, "width": 400, "height": 300}
    lane_detector = LaneRealTime((600, 800, 3))

    while True:
        screenshot = sct.grab(monitor)
        frame = np.array(screenshot)
        frame = cv2.cvtColor(frame, cv2.COLOR_BGRA2BGR)

        processed_frame = lane_detector.process_frame(frame)
        debug_log = lane_detector.print_log(True)
        if debug_log:
            print(debug_log)

        cv2.imshow("Lane Detection", processed_frame)
        cv2.imshow("ROI Image", lane_detector.roi_image)
        cv2.imshow("Warped Frame", lane_detector.warped_frame)
        # print(f"Offset: {lane_detector.center_offset:.1f} cm, Radius: {lane_detector.radius:.1f} cm")

        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

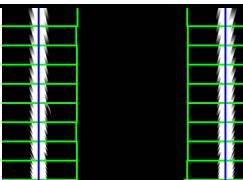
    cv2.destroyAllWindows()

if __name__ == "__main__":
    real_time_lane_detection()
```

**Gambar 47.** Fungsi *debugging* dan visualisasi

Visualisasi ini membantu mengidentifikasi masalah seperti ROI yang terlalu sempit atau deteksi marka yang tidak akurat.

**Tabel 7.** Visualisasi keseluruhan deteksi jalur

Metode	Visualisasi
Gambar Asli	
RoI	
HSV	
<i>Bird's-eye view</i>	
<i>Sliding Window</i>	
Jalur yang terdeteksi	

#### 4.4 Implementasi Algoritma Mengemudi

Perintah kontrol kendaraan menggunakan PyDirectInput harus disesuaikan dengan *keybinding* dalam permainan. Dalam contoh Gambar 48, tombol 'W' digunakan untuk akselerasi, tombol 'S' untuk rem, tombol 'A' digunakan untuk belok kiri dan 'D' untuk belok kanan.

Actions	Keys
Menu	Esc
Give up	Backspace , Delete
Respawn	Right Ctrl , Enter
Accelerate	W
Brake	S
Accelerate (analog)	<None>
Steer left	A
Steer right	D

**Gambar 48.** Kontrol permainan

Algoritma pengemudi menggunakan posisi relatif (offset) kendaraan dari jalur dan radius kurva jalan untuk mengontrol kemudi dan kecepatan mobil.

#### 4.4.1 Estimasi Kurva dan Ofset

Untuk menghitung perhitungan kurva jalan, fungsi *calculate\_curvature* digunakan seperti yang ditunjukan pada gambar 49. Fungsi ini menghitung kelengkungan berdasarkan *polinomial* yang disesuaikan pada jalur kiri dan kanan. Dengan cara ini, kelengkungan jalan dapat secara efektif dihitung.

```
def calculate_curvature(self):
    y_eval = self.ploty[-1]
    left_curvem = ((1 + (2 * self.left_fit[0] * y_eval + self.left_fit[1]) ** 2) ** 1.5) / np.abs(2 * self.left_fit[0])
    right_curvem = ((1 + (2 * self.right_fit[0] * y_eval + self.right_fit[1]) ** 2) ** 1.5) / np.abs(2 * self.right_fit[0])
    return (left_curvem + right_curvem) / 2
```

**Gambar 49.** Implementasi perhitungan kurva jalan

Evaluasi kelengkungan jalan dilakukan pada titik terendah pada gambar permainan menggunakan:

$$y_{eval} = \text{ploty}[-1]$$

menghasilkan estimasi kelengkungan yang lebih realistik untuk kondisi jalan.

Untuk memastikan perhitungan kelengkungan dapat mencerminkan kondisi nyata, skala dari piksel ke meter juga diterapkan dengan faktor konversi seperti:

- Sumbu X :  $M_{PER\_PIX} = 2/400$ , dan
- Sumbu Y :  $M_{PER\_PIX} = 100/300$

Hasil perhitungan memberikan informasi mengenai radius kelengkungan jalan, di mana jalan lurus memiliki radius lebih besar atau sama dengan 1.000 meter dan tikungan tajam memiliki radius kurang dari 300 meter. Dengan menggunakan rumus kelengkungan ini, simulasi dapat memperkirakan kelengkungan jalan yang akan ditempuh dan membantu dalam pengendalian kecepatan serta pengemudian mobil.

#### **4.4.1.1 Tipe Kurva Jalan**

- a. Jalan lurus: Jalan dengan radius lebih dari atau sama dengan 1.000 m



**Gambar 50.** Kurva jalan lurus

- b. Belokan Ringan: Radius kelengkungan 100-800 m.



**Gambar 51.** Kurva pada belokan ringan

- c. Belokan tajam: Tikungan yang lebih ekstrim ditunjukkan oleh radius kelengkungan yang kurang dari 100 meter.



**Gambar 52.** Kurva pada belokan tajam

#### 4.4.1.2 Posisi Mobil Terhadap Jalur

Untuk menghitung posisi relatif mobil terhadap pusat jalur, fungsi *calculate\_car\_position* digunakan, yang dilampirkan pada Gambar 53.

```

def calculate_car_position(self, print_to_terminal=False):
    """
    Calculate the position of the car relative to the center

    :param: print_to_terminal Display data to console if True
    :return: Offset from the center of the lane
    """

    # Assume the camera is centered in the image.
    # Get position of car in centimeters
    # In warped space (800x600), car is assumed at bottom center of ROI
    car_location = (self.desired_roi_points[2][0] + self.desired_roi_points[3][0]) / 2

    # Lane bottoms in warped space (green lines)
    height = self.warped_frame.shape[0] # 600
    bottom_left = self.left_fitx[-1] # Last point of left green line
    bottom_right = self.right_fitx[-1] # Last point of right green line

    # center of the lane in warped space
    center_lane = (bottom_right + bottom_left)/2

    # offset in pixels
    center_offset_px = car_location - center_lane

    # Convert to centimeters using XM_PER_PIX (adjusted for game)
    center_offset = center_offset_px * self.XM_PER_PIX * 100

    lane_width_px = bottom_right - bottom_left
    lane_width_m = lane_width_px * self.XM_PER_PIX * 100

```

**Gambar 53.** Menghitung ofset mobil

Bagian tengah jalur dapat dihitung dengan menggunakan rumus berikut:

$$\text{lane_center} = \frac{\text{self.left_fitx}[-1] + \text{self.right_fitx}[-1]}{2}$$

Kemudian nilai *lane\_center* dibandingkan dengan posisi pusat mobil pada citra terproyeksi, yang dapat dihitung dengan menggunakan rumus berikut:

$$\text{car_center} = \text{warped}_{\text{frame}}. \text{shape}[1]/2$$

Ofset mobil dihitung sebagai perbedaan antara *lane\_center* dan *car\_center*. Hasilnya kemudian dikonversi ke dalam satuan sentimeter dengan menggunakan faktor skala *XM\_PER\_PIX*.

Untuk mengurangi fluktuasi yang disebabkan oleh variasi posisi mobil yang cepat, teknik *smoothing* diterapkan pada perhitungan ofset sebagai berikut:

$$\text{smoothed}_{\text{offset}} = 0.8 \times \text{last_center}_{\text{offset}} + 0.2 \times \text{center}_{\text{offset}}$$

Perhitungan ini menghasilkan variasi offset, misalnya -10.0 cm ketika mobil berada di sisi kiri jalur dan 10.0 cm ketika mobil berada di sisi kanan jalur.

Kategori posisi mobil berdasarkan offset dapat dilihat sebagai berikut:

- a. Mobil berada di Tengah jalan

Offset mendekati nol, menunjukkan bahwa mobil berada di posisi optimal.



**Gambar 54.** Mobil berada di tengah jalan

- b. Mobil di pinggir kiri jalan

Offset negatif, mengindikasikan bahwa mobil lebih condong ke sisi kiri jalur.



**Gambar 55.** Mobil berada di pinggir kiri jalan

c. Mobil berada di pinggir kanan jalan

Offset positif menunjukkan bahwa mobil lebih dekat ke sisi kanan jalur.



**Gambar 56.** Mobil berada di pinggir kanan jalan

#### 4.4.2 Algoritma Kontrol

##### 4.4.2.1 Kontrol Kemudi

Fungsi `control_steering` digunakan untuk menghitung nilai kemudi kendaraan berdasarkan posisi relatif kendaraan terhadap pusat jalur serta tingkat kelengkungan jalan. Nilai kemudi yang dihasilkan mempertimbangkan kestabilan arah kendaraan dan responya terhadap perubahan kondisi jalan secara adaptif.

```
def control_steering(self, center_offset, curve_radius):
    raw_steering = center_offset / 100.0 * self.steering_sensitivity
    steering_value = 0.7 * self.last_steering + 0.3 * raw_steering
    self.last_steering = steering_value
    if abs(center_offset) < self.center_threshold:
        steering_value = steering_value * 0.4
    if curve_radius < 500:
        curve_factor = min(1.0, 300 / max(curve_radius, 50)) ** 2
        steering_value += curve_factor * 0.3 * (-1 if center_offset < 0 else 1)
    return max(-1.0, min(1.0, steering_value))
```

**Gambar 57.** Implementasi fungsi kontrol kemudi

Nilai kemudi awal dihitung berdasarkan seberapa jauh kendaraan menyimpang dari pusat jalur berdasarkan rumus berikut:

$$\text{raw\_steering} = \frac{\text{center\_offset}}{100.0} \times \text{steering\_sensitivity}$$

Untuk menghindari perubahan mengemudi mendadak, digunakan teknik *smoothing* dengan rumus:

$$\text{steering\_value} = 0.7 \times \text{last\_steering} + 0.1 \times \text{raw\_steering}$$

Nilai *steering\_value* tersebut merupakan kombinasi dari kemudi sebelumnya (*last\_steering*) dan kemudi baru (*raw\_steering*) dengan bobot masing-masing 70% dan 30%. Teknik ini membantu menjaga kestabilan arah kendaraan.

Jika kendaraan sedang berada di bagian jalan yang melengkung (ditandai dengan radius kurva yang kecil), nilai kemudi diperkuat dengan faktor tambahan *curve\_factor*, yang dikalikan dengan bobot 0.2 untuk meningkatkan responsivitas kemudi terhadap tingkat kelengkungan jalan.

#### 4.4.2.2 Kontrol Kecepatan

Kecepatan mobil dapat disesuaikan dengan kondisi jalan, terutama saat melintasi tikungan tajam, dengan menggunakan fungsi kontrol kecepatan, yang dapat dilihat pada Gambar 58.

```
def control_speed(self, curve_radius, center_offset):
    speed_value = self.speed_control
    if curve_radius < 500:
        speed_factor = min(1.0, curve_radius / 300) ** 2
        speed_value *= speed_factor
    if abs(center_offset) > 50:
        speed_value *= 0.8
    return max(0.05, speed_value)
```

**Gambar 58** Implementasi fungsi kontrol kecepatan

Nilai kecepatan dihitung berdasarkan radius kelengkungan menggunakan rumus berikut:

$$\text{speed factor} = \min\left(1.0, \frac{\text{curve\_radius}}{300}\right)^2$$

Dengan menggunakan rumus ini, mobil akan melambat secara signifikan pada radius tikungan yang kecil. Misalnya, jika radiusnya kurang dari 300, kecepatan mobil dapat turun hingga faktor 0.05. Saat digunakan dalam simulasi, kecepatan terendah dicapai saat menekan tombol "w" selama 0,5 detik, atau sekitar 20 mil per jam.

#### 4.4.3 Integrasi PyDirectInput

Dengan menggunakan pustaka Python PyDirectInput, fungsi *apply\_control* akan mengirimkan input keyboard ke permainan. Gambar 59 menunjukkan lampiran fungsi *apply\_control*.

```
def apply_controls(self, steering_value, speed_value):
    if time.time() - self.last_control_time < self.control_delay:
        return
    self.last_control_time = time.time()
    self.reset_controls()
    press_duration = min(0.05, abs(steering_value) * 0.4) # Increased multiplier
    if steering_value < -0.1:
        pydirectinput.keyDown(self.keys['left'])
        time.sleep(press_duration)
        pydirectinput.keyUp(self.keys['left'])
        self.current_steering = -1
    elif steering_value > 0.1:
        pydirectinput.keyDown(self.keys['right'])
        time.sleep(press_duration)
        pydirectinput.keyUp(self.keys['right'])
        self.current_steering = 1
    else:
        self.current_steering = 0
    if speed_value > 0.05:
        pydirectinput.keyDown(self.keys['forward'])
        self.current_speed = speed_value
    elif self.failure_count > 5:
        pydirectinput.keyDown(self.keys['brake'])
        time.sleep(0.1)
        pydirectinput.keyUp(self.keys['brake'])
        self.current_speed = 0
```

**Gambar 59.** Implementasi fungsi *apply\_control*

Perintah arah kemudi dikirim dengan menekan tombol kiri (*keyDown('a')*) atau kanan (*keyDown('d')*) tergantung nilai *steering\_value*, dengan durasi penekanan ditentukan oleh:

$$\text{press\_duration} = \min(0.05, |\text{steering\_value}| \times 0.3)$$

Implementasi ini memungkinkan simulasi kontrol yang lebih realistik dengan respons yang halus terhadap perubahan posisi dan arah jalan.

#### 4.5 Evaluasi Simulasi

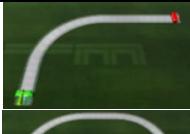
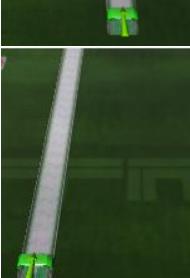
Bagian ini menjelaskan metode yang digunakan untuk mengevaluasi performa simulasi kemudi otomatis. Evaluasi dibagi menjadi dua fokus utama:

1. Evaluasi Deteksi Jalur, yang mencakup konsistensi dan ketepatan hasil deteksi marka jalur pada berbagai kondisi.
2. Performa Simulasi Pengemudian, yang menilai efektivitas kontrol kecepatan dan kemampuan simulasi dalam mengarahkan mobil secara otomatis.

#### 4.5.1 Evaluasi Deteksi Jalur

Beberapa contoh jalan yang akan diuji pada penelitian dapat dilihat pada Tabel 8.

**Tabel 8.** Panjang dan layout track

Track	Panjang (m)	Layout
1	538	
2	575	
3	405	
4	409	

##### 4.5.1.1 IoU

IoU digunakan untuk mengukur akurasi deteksi jalur dengan membandingkan area tumpang tindih antara jalur yang terdeteksi dan jalur *ground truth*.

```

def get_detected_lane_box(self):
    """Convert detected lane lines to a bounding box [x_min, y_min, x_max, y_max]."""
    if hasattr(self, 'left_fitx') and hasattr(self, 'right_fitx'):
        x_min = int(np.min(self.left_fitx))
        x_max = int(np.max(self.right_fitx))
        y_min = 0 # Top of warped frame
        y_max = self.warped_frame.shape[0] - 1 # Bottom of warped frame
        return [x_min, y_min, x_max, y_max]
    return [0, 0, 0, 0] # Default if no detection

def intersection_over_union(self, boxA, boxB):
    # Compute the intersection area
    xA = max(boxA[0], boxB[0])
    yA = max(boxA[1], boxB[1])
    xB = min(boxA[2], boxB[2])
    yB = min(boxA[3], boxB[3])

    interArea = max(0, xB - xA + 1) * max(0, yB - yA + 1)

    # Compute the area of both rectangles
    boxAArea = (boxA[2] - boxA[0] + 1) * (boxA[3] - boxA[1] + 1)
    boxBArea = (boxB[2] - boxB[0] + 1) * (boxB[3] - boxB[1] + 1)

    # Compute the intersection over union
    iou = interArea / float(boxAArea + boxBArea - interArea)
    return iou

```

**Gambar 60.** Kode implementasi IoU

Pada Gambar 60, *bounding box* jalur yang terdeteksi dihitung menggunakan fungsi *get\_detected\_lane\_box()*, yang mengambil nilai minimum dan maksimum koordinat x dari jalur kiri dan kanan yang terdeteksi. Nilai *y\_min* diatur ke nol karena jalur dimulai dari bagian atas *frame* yang telah ditransformasi, sedangkan *y\_max* ditentukan oleh tinggi layar permainan.

Setelah mendapatkan *bounding box* jalur yang terdeteksi, perhitungan IoU dilakukan dengan menggunakan fungsi *intersection\_over\_union()*. Fungsi ini menentukan area tumpang tindih (*intersection*) antara *bounding box* hasil deteksi dan *bounding box ground truth*. Kemudian, luas area gabungan (*union*) dihitung, dan rasio antara keduanya menjadi nilai IoU.

Tabel 9 berikut menunjukkan hasil evaluasi IoU pada sepuluh *track*:

**Tabel 9.** Intersection over Union

<b>Track</b>	<b>Mean IoU (%)</b>
1	0.846
2	0.799
3	0.783
4	0.732
5	0.808
6	0.809
7	0.704
8	0.796
9	0.711
10	0.732
<b>rata-rata</b>	<b>0.770</b>

Nilai IoU yang tinggi menunjukkan bahwa hasil deteksi jalur sangat mendekati jalur *ground truth*. Penurunan nilai IoU pada *track* 2 dan 3 menunjukkan bahwa ada beberapa faktor yang memengaruhi akurasi, seperti tingkat kelengkungan jalur serta kompleksitas *track*.

#### 4.5.1.2 Konsistensi Deteksi Jalur (Kualitatif)

Konsistensi deteksi merujuk pada sejauh mana algoritma mampu terus mendeksi jalur meskipun terdapat perubahan bentuk jalur serta gangguan visual. Evaluasi dilakukan dengan mengamati hasil visual dari deteksi jalur pada berbagai bagian track, sebagai berikut;

### 1. Jalur Lurus

Jalur terdeteksi secara konsisten di sepanjang segmen lurus tanpa deviasi signifikan.



**Gambar 61.** Konsistensi deteksi jalur lurus

### 2. Belokan Landai

Algoritma mampu mengikuti perubahan gradual pada geometri jalan, mempertahankan bentuk jalur yang terdeteksi.



**Gambar 62.** Konsistensi deteksi belokan landai

### 3. Belokan Tajam

Deteksi masih terjadi, namun akurasi menurun di titik belokan paling tajam karena RoI tidak mampu menangkap bentuk jalur.



**Gambar 63.** Konsistensi deteksi belokan tajam

#### 4. Gangguan Visual

Deteksi jalur mengalami kegagalan ketika tertutup dengan objek lain seperti bayangan dari objek yang berada pada lintasan.



**Gambar 64.** Konsistensi deteksi dengan  
kehadiran objek lain

#### 4.5.2 Evaluasi Navigasi Track

##### 4.5.2.1 Jumlah Track Sukses

Evaluasi ini mengukur seberapa baik algoritma mengemudi dapat menyelesaikan putaran pada setiap *track* dengan mempertimbangkan jenis belokan yang ada. Belokan dikategorikan menjadi belokan ringan dan belokan tajam, di mana keberhasilan melewati belokan menjadi indikator efektivitas simulasi kemudi dan kontrol kecepatan.

Tabel 10 adalah rangkuman percobaan yang dilakukan pada berbagai jenis jalur pada permainan.

**Tabel 10.** Track berhasil diselesaikan

<i>Track</i>	Jumlah Belokan Ringan	Jumlah Belokan Tajam	Belokan Ringan Berhasil Dilalui	Belokan Tajam Berhasil Dilalui
<b>1</b>	1	0	1	0
<b>2</b>	1	1	1	0
<b>3</b>	2	1	2	0
<b>4</b>	1	0	1	0
<b>5</b>	0	2	0	0
<b>6</b>	3	0	2	0
<b>7</b>	0	1	0	0
<b>8</b>	2	1	1	0
<b>9</b>	1	1	1	0
<b>10</b>	2	2	1	0

Berdasarkan hasil pengujian pada sepuluh *track* yang berbeda, dapat diamati bahwa navigasi menunjukkan performa yang cukup baik dalam menghadapi belokan ringan, namun masih menghadapi kesulitan signifikan pada belokan tajam. Dari total 14 belokan ringan yang diuji, kendaraan berhasil melakukan navigasi sebanyak 11 belokan ringan.

Sebaliknya, dari 11 belokan tajam yang diuji, tidak terdapat satu pun yang berhasil dilalui tanpa kegagalan. Hal ini menunjukkan bahwa simulasi saat ini belum memiliki kemampuan yang memadai dalam melakukan manuver pada belokan tajam.

Meskipun demikian, keberhasilan ini tidak sepenuhnya mencerminkan manuver yang stabil atau efisien, karena dalam beberapa kasus, kendaraan masih mengalami tabrakan dengan pembatas jalur sebelum melakukan koreksi kemudi dan melanjutkan navigasi.

Secara keseluruhan, meskipun simulasi mampu melanjutkan pergerakan pada sebagian besar belokan ringan, tingginya frekuensi tabrakan mengindikasikan perlunya peningkatan pada aspek sensitifitas kemudi kecepatan dari mobil, serta mengurangi latensi *pipeline* simulasi.

#### 4.5.2.2 Jarak Tempuh Sebelum Menabrak

Evaluasi ini bertujuan untuk mengukur sejauh mana kendaraan dapat melaju di sepanjang *track* sebelum mengalami tabrakan atau keluar jalur. Pengukuran dilakukan dengan mencatat total panjang lintasan serta jarak yang berhasil ditempuh sebelum terjadi insiden.

**Tabel 11.** Jarak tempuh sebelum menabrak

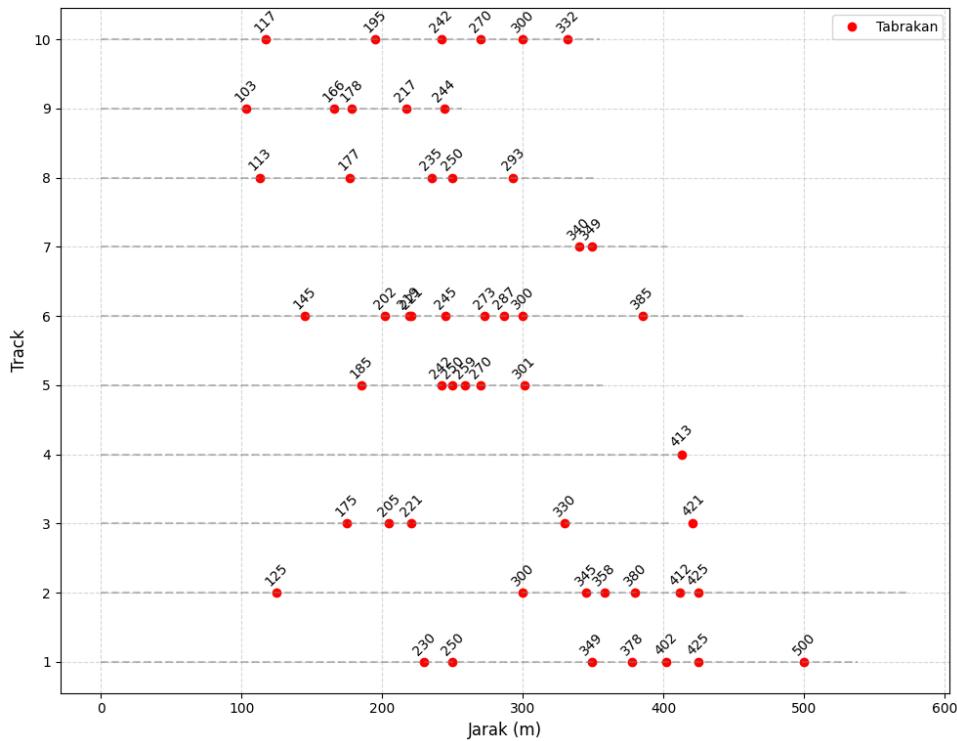
<b>Track</b>	<b>Panjang <i>Track</i> (m)</b>	<b>Jarak Tempuh Sebelum Menabrak (m)</b>	<b>Persentase Keberhasilan (%)</b>
1	538	230	42.75
2	575	300	52.17
3	405	205	50.61
4	413	413	100.0
5	357	185	51.82
6	457	145	31.72
7	403	340	84.36
8	351	177	50.42
9	257	166	64.59
10	355	195	54.92
<b>Rata-rata keberhasilan</b>			<b>55.56</b>

Tabel 11 menunjukkan jarak tempuh sebelum terjadi tabrakan untuk sepuluh *track* yang diuji. Dari seluruh *track*, hanya *track* 4 yang berhasil dilalui sepenuhnya tanpa tabrakan, dengan persentase keberhasilan 100%. Pada *track* lainnya, kendaraan mengalami insiden sebelum mencapai garis akhir, dengan persentase keberhasilan yang

berkisar antara 31.72% hingga 84.36%. Secara keseluruhan, dari total 10 *track*, rata-rata persentase keberhasilan navigasi sebelum akhirnya mobil menabrak adalah 55.56%.

Sebagian besar tabrakan terjadi saat kendaraan memasuki belokan pertama, yang mengindikasikan bahwa simulasi saat ini belum dapat menangani manuver awal dengan cukup baik. Meskipun kendaraan mampu menempuh sebagian besar lintasan pada beberapa *track*, tidak satu pun (kecuali *track* 4) dapat dilalui secara utuh tanpa tabrakan pada setiap tikungan.

Gambar 65 menunjukkan distribusi titik tabrakan yang terjadi pada masing-masing *track*. Titik-titik tabrakan tercatat pada jarak yang berbeda-beda, dengan sejumlah *track* mengalami lebih dari tiga tabrakan sebelum mencapai garis akhir. Hal ini menunjukkan bahwa faktor kompleksitas lintasan, terutama pada belokan tajam dan belokan ringan, sangat mempengaruhi kemampuan kendaraan dalam menyelesaikan *track* tanpa insiden.

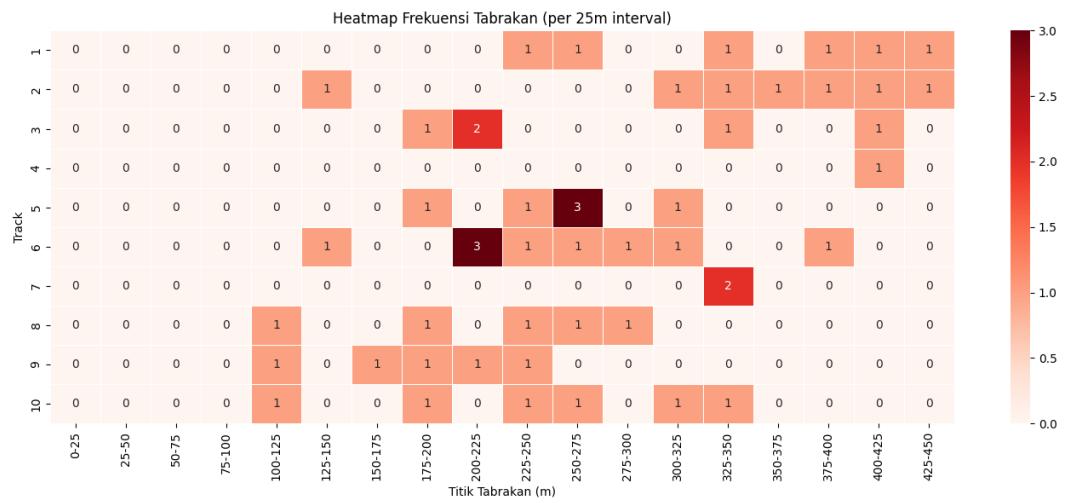


**Gambar 65.** Distribusi Titik Tabrakan

Pada *track* dengan jumlah belokan yang lebih banyak seperti *track* 1, 2, 5, dan 10, kendaraan mengalami lebih banyak tabrakan, terutama pada belokan pertama atau kedua serta kondisi jalan dengan belokan beruntun.

Sebaliknya pada *track* dengan kompleksitas rendah seperti 3,4 dan 7, mobil lebih stabil dalam melakukan navigasi. Sebagai contoh, *track* 7 yang memiliki dua belokan tajam tercatat memiliki dua titik tabrakan pada jarak 340 m dan 349 m, sementara *track* 4, yang hanya memiliki satu belokan ringan, mampu dilalui tanpa tabrakan.

Gambar 66 menunjukkan total tabrakan pada setiap *track* dengan rentang 25 meter, dimana *track* yang memiliki jumlah tabrakan terbanyak adalah *track* 2, 5, dan 6, serta *track* dengan tabrakan yang minim terjadi pada *track* 4 dan 7.



**Gambar 66.** Heatmap Frekuensi Tabrakan

Hasil ini mengindikasikan bahwa performa simulasi masih belum optimal, terutama dalam menangani jalur yang mengandung tikungan tajam. Faktor-faktor seperti kecepatan, sensitivitas kemudi, efektivitas algoritma pengendalian, dan akurasi deteksi jalur juga berkontribusi besar terhadap performa keseluruhan kendaraan.

#### 4.6 Analisis Performa Simulasi

Bagian ini menyajikan hasil dari implementasi algoritma deteksi jalur dan algoritma mengemudi, termasuk performa waktu eksekusi dan efektivitas visualisasi jalur yang terdeteksi serta kemampuan navigasi mobil. Analisis lebih lanjut juga membahas kekuatan dan keterbatasan simulasi.

#### 4.6.1 Analisis Performa Algoritma Deteksi Jalur

*Pipeline* deteksi jalur terbukti efektif dalam mengidentifikasi garis jalur, seperti yang ditunjukkan oleh *frame* yang diproses yang ditampilkan selama *debugging*. Simulasi ini dengan tepat menampilkan jalur yang terdeteksi di atas rekaman permainan, dengan nilai kurva dan offset yang ditampilkan di layar serta akurasi yang konsisten di semua jenis jalur.

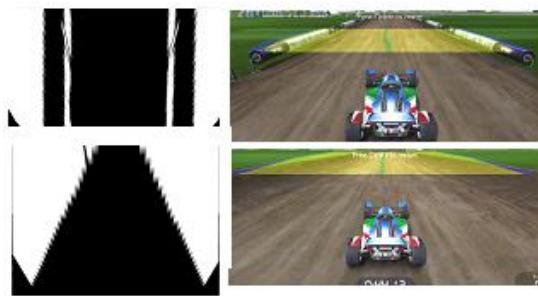
##### 4.6.1.1 Kelebihan dan Kekurangan Algoritma Deteksi Jalur

Algoritma deteksi jalur yang diterapkan menggunakan metode *sliding window* terbukti efektif dalam mengidentifikasi garis jalur pada semua lintasan yang diuji. Dengan nilai rata-rata IoU sebesar 77%, simulasi mampu mendeteksi garis jalur dengan tingkat akurasi yang konsisten. Selain itu, algoritma ini juga mampu secara akurat mengidentifikasi kurva serta menghitung offset kendaraan terhadap pusat jalur, yang divisualisasikan secara langsung di atas permainan.

Meskipun algoritma deteksi jalur menunjukkan performa yang baik pada sebagian besar kondisi, terdapat beberapa keterbatasan penting:

###### a. Ketergantungan terhadap Warna Track

Konversi warna ke ruang HSV mengandalkan batas jalan berwarna hijau. Pada *track* dengan warna berbeda, deteksi tepi gagal mengisolasi jalur dengan baik.



**Gambar 67.** Deteksi gagal pada batas jalan dengan warna selain hijau

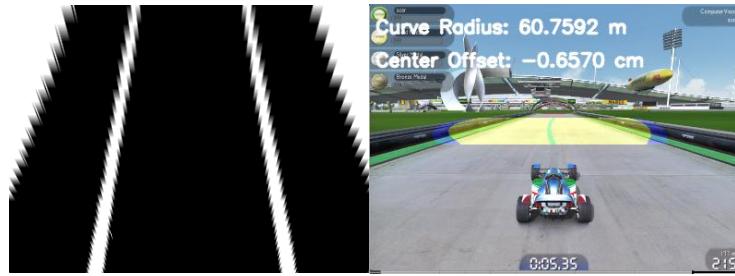
Kegagalan ini ditandai dengan penurunan akurasi IoU (perhatikan Tabel 12 ) pada jenis *track* ditutupi oleh tanah yang didominasi oleh warna cokelat.

**Tabel 12.** Penurunan IoU *track* berwarna cokelat

<i>Track</i>	IoU
1	0.666
2	0.666
3	0.669
4	0.671
5	0.675
<b>Rata-rata IoU</b>	<b>0,669391</b>

#### b. Perubahan Perspektif pada Kecepatan Tinggi

Algoritma deteksi jalur akan kesusahan pada saat terjadi perubahan perspektif jalur yang semakin menyempit pada permainan ketika mobil bergerak pada kecepatan tinggi ( $> 60$  mph).



**Gambar 68.** Perubahan perspektif pada kecepatan tinggi

c. Kegagalan di Tepi Jalan atau Saat Tabrakan

Ketika mobil mendekati tepi jalan, algoritma deteksi jalur akan gagal dalam menangkap jalur.



**Gambar 69.** Deteksi jalur gagal saat mendekati tepi atau tabrakan

d. Keterbatasan ROI Statis di Belokan Tajam

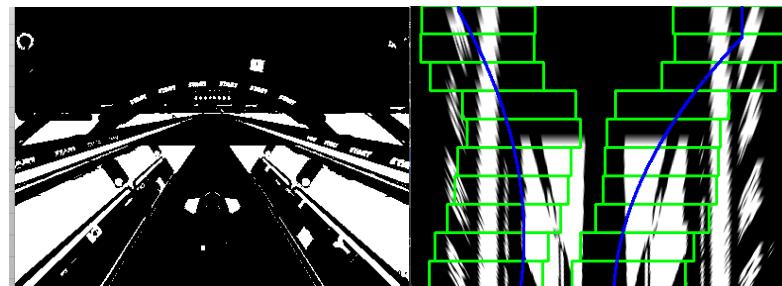
Penggunaan ROI statis membuat simulasi tidak fleksibel dalam mengikuti perubahan posisi jalur di tikungan, terutama pada belokan tajam.



**Gambar 70.** ROI tidak menangkap seluruh jalur saat belokan

### e. Gangguan dari Objek Lain di Jalan

Kehadiran objek lain di atas jalur, seperti kendaraan atau rintangan, menyebabkan gangguan visual yang menurunkan akurasi deteksi.



**Gambar 71.** Objek lain mengganggu deteksi jalur

#### 4.6.1.2 Waktu Eksekusi (Profiling)

Data *profiling* (Tabel 13.) menunjukkan bahwa deteksi jalur berjalan dengan efisien, memakan waktu sekitar 1.830 detik untuk 213 *frame*, atau 0.009 detik per *frame*.

**Tabel 13.** Profiling Deteksi Jalur

Fungsi	Panggilan	Waktu per Panggilan (s)	Total Waktu (s)
<i>overlay_lane_lines</i>	213	0.003	0.666
<i>get_lane_line_indices_sliding_windows</i>	213	0.002	0.467
<i>warpPerspective</i>	426	0.001	0.240

Berdasarkan data pada Tabel 13, fungsi *overlay\_lane\_lines* merupakan komponen dengan waktu eksekusi tertinggi, yaitu 0.666 detik secara total untuk 213 *frame*, diikuti oleh *get\_lane\_line\_indices\_sliding\_windows* dan *warpPerspective*. Meskipun fungsi-fungsi ini dominan dalam penggunaan waktu, rata-rata waktu per

*frame* tetap di bawah 0.01 detik, menunjukkan performa yang cukup efisien untuk aplikasi *real-time*.

Jumlah pemanggilan fungsi *warpPerspective* mencapai 426, berbeda dengan fungsi lainnya yang hanya 213 *frame* per detik, karena fungsi ini digunakan baik dalam tahap pemrosesan gambar seperti transformasi *bird's eye view* maupun pasca-pemrosesan (pengembalian hasil ke perspektif asli).

#### 4.6.2 Analisis Performa Algoritma Kontrol Kemudi

Fungsi *control\_steering* mengandalkan hasil dari deteksi jalur untuk menghitung posisi ofset mobil dari tengah jalur dan kelengkungan jalur. Nilai ini kemudian diterjemahkan menjadi input arah ('a' untuk kiri, 'd' untuk kanan) melalui *apply\_controls*.

Meskipun algoritma deteksi jalur bekerja cukup baik dalam mengenali posisi jalur, hasilnya tidak sepenuhnya dapat dimanfaatkan oleh simulasi pengemudian yang diterapkan.

Berikut ini adalah beberapa alasan yang menjelaskan mengapa algoritma pengemudian masih belum optimal:

- a. Kontrol Biner pada Simulasi yang Memerlukan Kendali Analog

Algoritma pengemudian mengandalkan input *keyboard* (tombol 'a' dan 'd') yang bersifat biner (*ON/OFF*) pada permainan yang memerlukan kendali yang lebih halus dan presisi. Hal ini membatasi kemampuan algoritma mengemudi untuk menyesuaikan arah mobil secara bertahap dan akurat, terutama saat menghadapi tikungan atau perubahan arah yang kompleks.

b. Tidak Terdapat Mekanisme Umpang Balik

Fungsi *control\_steering* hanya memperhitungkan kondisi offset saat ini dari posisi mobil terhadap pusat jalur, tanpa memperhatikan hasil dari kontrol yang telah diberikan sebelumnya. Tidak terdapat mekanisme umpan balik atau pengontrol seperti PID (*Proportional-Inegral-Derivative*) yang mampu mengevaluasi dan mengoreksi posisi kendaraan secara *real-time*, sehingga arah kendaraan sulit distabilkan.

c. Latensi Eksekusi yang Tinggi dalam Pipeline Simulasi

Salah satu kendala utama dalam performa pengemudian otomatis adalah tingginya latensi dalam siklus kendali (control loop) simulasi. Berdasarkan hasil profiling pada fungsi-fungsi utama dalam loop pengemudian, ditemukan bahwa sebagian besar waktu eksekusi dihabiskan pada bagian PyDirectInput dan *apply\_controls*, dengan rata-rata durasi per panggilan mencapai lebih dari 0.1 detik. Latensi ini mengakibatkan jeda antara saat informasi visual diproses dan saat aksi pengendalian diberikan pada simulasi.

Dengan kata lain, mobil dapat saja sudah berada dalam kondisi yang berbeda ketika perintah belok atau percepatan akhirnya dijalankan. Hal ini membuat sistem tidak responsif terhadap perubahan kondisi jalan secara *real-time*, menyebabkan pengambilan keputusan yang tertunda, serta mengurangi kemampuan algoritma untuk menyesuaikan arah dan kecepatan secara efektif.

d. Kontrol Berdasarkan Waktu yang Rentan terhadap Latensi

Durasi penekanan tombol arah ditentukan secara *hardcode* (misalnya 5ms), tanpa mempertimbangkan kondisi aktual pada layar atau kecepatan *frame* permainan. Simulasi ini sendiri memiliki latensi signifikan dalam pengiriman input melalui PyDirectInput, seperti yang ditunjukkan pada hasil *profiling* (Tabel 13). Hasil ini menyebabkan perintah tidak dieksekusi secara konsisten, terutama saat permainan mengalami *lag* atau fluktuasi performa.

e. Tidak Adanya Prediksi Gerakan atau Memori Historis

Algoritma tidak mempertimbangkan arah gerakan atau perubahan posisi dalam beberapa *frame* sebelumnya. Tanpa prediksi atau pemrosesan temporal, kontrol pengemudi menjadi terlalu reaktif dan sering terlambat dalam menanggapi perubahan kondisi jalan.

f. Keterbatasan Platform Permainan

Pada dasarnya, algoritma mengemudi ini bertujuan untuk melakukan koreksi kendaraan sehingga mobil tetap berada pada jalur dengan mengirimkan tombol *keyboard* selama rentang waktu tertentu. Pendekatan ini menjadi tidak efisien jika dihadapkan dengan kondisi jalan dalam permainan yang lebih kompleks.

#### 4.6.3 Responsivitas Sistem & Latensi

Hasil profiling dari loop pengendalian mobil (Tabel 14) menunjukkan bahwa hampir seluruh waktu eksekusi didominasi oleh fungsi `apply_controls`, yang menggunakan modul PyDirectInput untuk mengirimkan sinyal input ke permainan. Fungsi ini menyumbang lebih dari 99% total waktu eksekusi, dengan rata-rata waktu eksekusi per panggilan sekitar 0.202 detik.

**Tabel 14.** Profiling Drive Loop

Fungsi	Panggilan	Per Panggilan (detik)	Total Waktu (detik)
<code>apply_controls</code>	158	0.202	31.885
<code>pydirectinput.wrapper</code>	316	0.101	31.854
<code>pydirectinput._handlePause</code>	316	0.100	31.706
<code>cv2.waitKey</code>	166	0.013	2.101

Ketika dikonversi ke dalam *frame rate*, sistem hanya mampu menjalankan sekitar 5 input per detik, atau sekitar 5 FPS secara praktis.

$$FPS = \frac{\text{panggilan}}{\text{total waktu}} = 158/31.885 \approx 4.96$$

Namun, karena `apply_controls` bersifat *blocking* dan membutuhkan waktu  $\pm 200$  ms per siklus, respons terhadap perubahan lintasan menjadi terlambat. Inilah yang membentuk *latensi* yang tinggi dalam konteks kendali real-time.

*Bottleneck* ini bukan berasal dari komputasi visi atau logika kontrol, melainkan dari cara sinyal input disampaikan ke permainan. Optimalisasi seperti penggunaan metode input lain (misalnya *direct memory manipulation*, jika diperbolehkan oleh simulator), atau penggantian modul

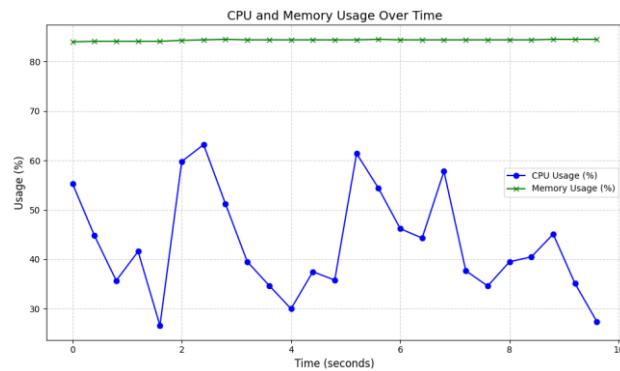
PyDirectInput dengan pendekatan yang lebih efisien, berpotensi secara signifikan meningkatkan responsivitas sistem.

#### 4.6.4 Beban Komputasi Simulasi

Simulasi dijalankan pada perangkat Windows PC dengan spesifikasi:

- Prosesor: Intel(R) Core(TM) i5-13420H (13th Gen)
- Memori: 8 GB RAM

Meskipun perangkat memiliki GPU NVIDIA GeForce RTX 2050 6 GB, simulasi tidak menggunakan akselerasi GPU sehingga pengujian berfokus pada performa CPU dan penggunaan memori (RAM).



**Gambar 72.** Penggunaan memory dan CPU

Selama simulasi dijalankan selama 10 detik, dilakukan pencatatan penggunaan CPU dan memori sistem. Hasilnya ditampilkan pada grafik performa (perhatikan Gambar 72):

1. Penggunaan CPU berkisar antara 26.6% hingga 63.2%, dengan rata-rata berkisar 44.3%. Hal ini menunjukkan beban komputasi sedang selama eksekusi simulasi.

2. Penggunaan Memori relatif stabil, berada di kisaran 84.0% hingga 84.5%. Ini menandakan bahwa simulasi menggunakan memori secara konsisten tanpa terjadi lonjakan signifikan atau kebocoran memori.

## 4.7 Penyesuaian dan Tuning Parameter

Parameter disesuaikan untuk meningkatkan kinerja dan efisiensi komputasi simulasi kemudi otomatis. Beberapa rangkaian eksperimen dilakukan untuk menentukan nilai parameter yang untuk faktor-faktor berikut:

- a. Akurasi deteksi
- b. Stabilitas kemudi
- c. Kecepatan mobil
- d. Efisiensi pipeline

Proses ini melibatkan pengujian iteratif untuk memastikan simulasi dapat menangani berbagai kondisi jalan, termasuk tikungan, sambil meminimalkan latensi dalam *pipeline* simulasi.

### 4.7.1 RoI & Resolusi Frame Permainan

Pada tahap awal, RoI ditetapkan untuk citra dengan resolusi 800x600 piksel dengan koordinat layar permainan sebagai berikut:

- (274, 254) (kiri atas),
- (517, 254) (kanan atas),
- (748, 313) (kanan bawah),
- (13, 313) (kiri bawah).

Setelah dilakukan evaluasi, RoI tersebut tidak sepenuhnya efektif dalam menangkap marka jalan pada tikungan, sehingga diperlukan penyesuaian untuk meningkatkan akurasi deteksi.

Untuk mengurangi beban komputasi, jumlah piksel yang diproses dikurangi sebesar 75%, dari 480.000 menjadi 120.000, dengan menurunkan resolusi gambar menjadi 400x300 piksel. Akibatnya, koordinat RoI diperbarui menjadi:

- (137, 127) (kiri atas),
- (258, 127) (kanan atas),
- (374, 156) (kanan bawah),
- (6, 156) (kiri bawah).

Koordinat-koordinat ini membentuk poligon yang membatasi pemrosesan gambar pada area jalan yang relevan. Tabel 15 merangkum eksperimen pada koordinat RoI, ukuran bingkai, dan dampaknya terhadap latensi pemrosesan.

**Tabel 15.** Eksperimen koordinat RoI pada gambar

<b>Frame</b>	<b>Kiri</b>	<b>Kanan</b>	<b>Kanan</b>	<b>Kiri</b>	<b>Latensi</b>
<b>Permainan</b>	<b>Atas</b>	<b>Atas</b>	<b>Bawah</b>	<b>Bawah</b>	<b>(ms)</b>
	<b>(x,y)</b>	<b>(x,y)</b>	<b>(x,y)</b>	<b>(x,y)</b>	
<b>800x600</b>	274, 254	517, 254	748, 313	13, 313	1000 555
	200, 254	600, 254	790, 313)	10, 313	
<b>400x300</b>	137, 127	258, 127	374, 156	6, 156	200

Dengan penyesuaian ini, simulasi dapat mendeteksi marka jalan pada tikungan dengan lebih baik. Namun, karena pemrosesan tambahan pada area RoI yang lebih luas sedikit meningkatkan latensi. Mengurangi resolusi gambar menjadi 400 x 300 piksel membantu mengurangi beban komputasi. Pengurangan ini kemudian diimbangi dengan pengoptimalan tambahan seperti teknik *skipping frame* yang dijelaskan pada sub bab 4.7.4.

#### 4.7.2 Sensitivitas Kemudi

Parameter *steering\_sensitivity* mengontrol seberapa responsif kemudi mobil terhadap perubahan offset dari pusat jalur. Nilai awal *steering\_sensitivity* di atur ke nilai 0.7 untuk memastikan mobil dapat bereaksi cepat terhadap deviasi dari jalur, semakin tinggi nilai *steering\_sensitivity* kemudi juga akan lebih responsi terhadap perubahan offset atau kurvatur jalan.

Nilai tersebut menyebabkan koreksi kemudi yang berlebihan, di mana mobil sering kali berbelok terlalu tajam, terutama pada jalan lurus, sehingga mengurangi stabilitas. Setelah beberapa iterasi pengujian, *steering\_sensitivity* diturunkan menjadi 0.4. Penurunan ini mengurangi koreksi kemudi secara signifikan, memungkinkan mobil untuk berbelok dengan lebih halus dan stabil, terutama pada tikungan dengan radius kecil ( $<300$  m).

Selain itu, penerapan pemulusan pada nilai kemudi untuk memastikan transisi yang lebih mulus antar *frame*. Rumus pemulusan yang digunakan adalah:

$$\text{steering\_value} = 0.9 \times \text{last\_steering} + 0.1 \times \text{raw\_steering}$$

Pendekatan ini membantu mengurangi fluktuasi (berubah terlalu cepat atau terlalu besar dalam waktu singkat) tiba-tiba pada kemudi, yang sebelumnya menyebabkan mobil menabrak batas jalan. Penyesuaian untuk tikungan dengan faktor *curve\_factor*\*0.2, yang memungkinkan mobil untuk menyesuaikan kemudi secara dinamis berdasarkan radius kelengkungan jalan. Hasilnya, mobil dapat menavigasi tikungan dengan lebih baik.

#### 4.7.3 Kecepatan Mobil

Setelah jalur berhasil terdeteksi, kecepatan mobil disesuaikan untuk menjaga kestabilan deteksi dan memastikan mobil tetap berada pada jalur. Fungsi *apply\_controls* digunakan untuk mengirimkan input keyboard ke permainan menggunakan pustaka PyDirectInput.

Kecepatan target ditetapkan antara 30–40 mph, karena pengujian menunjukkan bahwa kecepatan yang lebih tinggi menyebabkan distorsi perspektif, yang secara langsung menurunkan akurasi deteksi jalur (sebagaimana dijelaskan pada sub bagian 4.5.1.1). Untuk menjaga kontrol kecepatan yang stabil, digunakan strategi pemulusan berupa penekanan tombol 'w' dalam durasi singkat (awalnya 20ms, dikurangi menjadi 10ms), diikuti dengan periode pelepasan selama 50ms. Pendekatan ini menjaga agar pergerakan mobil tidak terlalu cepat sekaligus memungkinkan deteksi jalur tetap konsisten.

Menyesuaikan parameter *control\_delay* dari nilai awal 0.1 menjadi 0.05 untuk meningkatkan responsivitas simulasi terhadap perubahan kondisi jalan. Parameter *press\_duration\_multiplier* juga dikurangi menjadi 0.3, memastikan bahwa perintah kemudi dan kecepatan diterapkan dengan durasi yang lebih singkat, sehingga mengurangi latensi.

*Smoothing* juga diterapkan pada ofset mobil untuk stabilitas, menggunakan rumus.

$$\text{smoothed\_ofset} = 0.8 \times \text{last\_center\_ofset} + 0.2 \times \text{center\_ofset}$$

Penyesuaian ini memastikan bahwa mobil dapat bergerak dengan lebih halus, baik pada jalan lurus maupun tikungan, dengan kecepatan yang sesuai untuk setiap kondisi.

#### 4.7.4 Optimasi Pipeline Simulasi

##### a. Frame Skipping

Untuk meningkatkan efisiensi pengolahan, ukuran citra dikurangi dari 800x600 menjadi 400x300, mengurangi jumlah piksel yang diproses sebesar 75% (dari 480.000 piksel menjadi 120.000 piksel). Hal ini secara signifikan mempercepat operasi seperti konversi HSV dan transformasi perspektif. Selain itu, teknik *skipping frames* diterapkan dengan parameter *skip\_frames* = 6, sehingga hanya setiap *frame* ke-7 yang diproses.

```
self.frame_count = 0
self.skip_frames = 6 # Increased
self.processing_scale = 0.15
```

**Gambar 73.** Frame Skipping

Gambar 73 merupakan lampiran dari teknik *skipping frames*, di mana hanya *frame* tertentu yang diproses untuk mengurangi beban komputasi. Pendekatan ini memungkinkan simulasi untuk tetap mempertahankan stabilitas deteksi marka jalan, meskipun latensi pada *pipeline* simulasi masih tinggi.

##### b. Komputasi Parallel dengan Producer-Consumer Model

Dalam optimasi *pipeline*, digunakan model *Producer-Consumer* (Gambar 74) untuk meningkatkan efisiensi pemrosesan data secara paralel. Model ini membagi proses utama menjadi dua bagian: *capturing* (pengambilan *frame* dari layar) dan *detection* (pemrosesan citra untuk deteksi jalur).

```
self.frame_queue = queue.Queue(maxsize=5)
self.result_queue = queue.Queue(maxsize=5)
```

**Gambar 74.** Pendekatan Producer-Consumer model

Dengan memisahkan tugas ini ke dalam dua perulangan independen, simulasi dapat berjalan lebih cepat dibandingkan dengan pendekatan sekuensial. Dua antrian (*queue*) digunakan untuk menangani pertukaran data antara proses:

- *frame\_queue*: Menyimpan *frame* yang diambil dari layar sebelum diproses.
- *result\_queue*: Menyimpan hasil deteksi sebelum diterapkan pada algoritma mengemudi.

Penggunaan *queue* dengan batas ukuran (*maxsize*) memastikan bahwa tidak ada *bottleneck* akibat terlalu banyak data tertunda.

```
def capture_loop(self):
    while self.is_running:
        self.capture_profiler.enable()
        start_time = time.time()
        frame = self.capture_screen()
        print(f"Capture time: {time.time() - start_time:.3f}s")
        try:
            self.frame_queue.put_nowait(frame)
        except queue.Full:
            pass
```

**Gambar 75.** Loop Producer

Gambar 75 menunjukkan fungsi *capture\_loop* yang bertugas menangkap *frame* dari layar dan memasukannya ke *frame\_queue*. Jika antrian penuh, *frame* terbaru akan diabaikan untuk menjaga kelancaran simulasi.

```

def detection_loop(self):
    while self.is_running:
        self.detection_profiler.enable()
        start_time = time.time()
        try:
            frame = self.frame_queue.get(timeout=0.1)
        except queue.Empty:
    
```

**Gambar 76.** Loop Consumer

Gambar 76 menunjukkan fungsi *detection\_loop* yang mengambil *frame* dari *frame\_queue*, menjalankan deteksi jalur, lalu menyimpan hasilnya ke *result\_queue*.

Keuntungan model *Producer-Consumer*:

- a. Mengurangi Latensi: Pemrosesan dapat berjalan secara asinkron tanpa menunggu proses sebelumnya selesai.
- b. Meningkatkan Performa: Dengan memisahkan tugas *capture* dan *detection*, waktu eksekusi per *frame* lebih optimal.
- c. Mencegah *Bottleneck*: Dengan simulasi antrian terbatas, *frame* atau hasil yang berlebih dapat dikontrol agar tidak membebani simulasi.

#### 4.8 Capaian dan Keterbatasan

Simulasi kemudi otomatis mencapai sebagian besar tujuan awal, namun tetap menghadapi batasan teknis yang signifikan. Berikut adalah pencapaian dan kekurangan yang ditemukan selama proses implementasi:

#### 4.8.1 Capaian Simulasi

1. Deteksi Marka Jalur yang Andal pada Kondisi Ideal

Simulasi mampu mendeteksi marka jalur dengan baik pada kondisi lintasan yang memiliki warna hijau yang kontras terhadap latar belakang. Penggunaan transformasi perspektif berhasil mengubah tampilan kamera menjadi *bird's-eye view* yang mempermudah deteksi jalur.

2. Implementasi Deteksi Jalur Secara Paralel

Dengan arsitektur *producer-consumer*, simulasi mampu menjalankan deteksi citra dan kontrol kendaraan secara paralel. Ini mengurangi *bottleneck* CPU dan meningkatkan respons simulasi secara keseluruhan.

3. Mobil Dapat Bergerak Secara Otomatis

Melalui *tuning parameter* pada fungsi *apply\_controls* dan *control\_speed*, mobil dapat bergerak ke depan dengan stabil dalam kecepatan yang telah ditentukan (30–40 mph), menjaga performa deteksi jalur tetap optimal tanpa mengorbankan akurasi.

#### 4.8.2 Keterbatasan dan Tantangan

1. Kinerja Deteksi Jalur Terbatas pada Warna Tertentu

Simulasi bergantung pada HSV untuk mengisolasi warna hijau, sehingga jalur dengan warna lain akan sulit terdeteksi.

## 2. Pengaruh Perspektif pada Kecepatan Tinggi

Pada kecepatan >60 mph, perubahan perspektif menjadi signifikan dan menyebabkan kesalahan dalam estimasi posisi jalur, karena RoI tidak menyesuaikan secara adaptif.

## 3. Kelemahan pada Kontrol Belokan

Pengemudian masih belum responsif dan sering gagal menangani tikungan. Penyebabnya antara lain:

- RoI statis tidak mengikuti bentuk jalan yang dinamis.
- Simulasi hanya menggunakan input keyboard biner (hanya bisa ON/OFF), sehingga tidak bisa memberikan kontrol bertahap layaknya input analog.
- Latensi yang cukup tinggi dimana simulasi hanya mampu mengoperasikan 5 input per detik.
- Tidak adanya sistem umpan balik (*feedback*) seperti PID, sehingga kontrol tidak mampu mengoreksi kesalahan posisi secara otomatis dan berkelanjutan.

## 4. Kompleksitas Track

Mobil masih mengalami kesulitan dalam menavigasi *track* dengan kompleksitas yang menengah maupun tinggi seperti. Terbukti pada Tabel 10 dan Gambar 66 dimana mobil selalu mengalami tabrakan pada setiap tikungan.

##### 5. Minimnya Respons terhadap Lingkungan Sekitar

Simulasi belum mampu mengadaptasi dengan kehadiran objek lain di jalan atau perubahan mendadak pada kondisi lintasan, karena tidak ada komponen pemrosesan konteks atau perencanaan jalur.

Secara keseluruhan, simulasi yang dikembangkan telah menunjukkan potensi untuk diimplementasikan dalam lingkungan simulasi, meskipun masih terdapat beberapa keterbatasan yang perlu diperbaiki untuk aplikasi yang lebih kompleks atau *real-time*.

## **BAB V**

### **KESIMPULAN**

#### **5.1 Kesimpulan**

Berdasarkan hasil evaluasi, metode deteksi jalur berbasis *sliding window* menunjukkan efektivitas yang cukup tinggi dengan rata-rata *Intersection over Union* (IoU) sebesar 77% pada sepuluh lintasan yang diuji. Hal ini membuktikan bahwa metode ini mampu mengenali bentuk jalur secara andal dalam beragam kompleksitas lingkungan permainan.

Sementara itu, algoritma navigasi berbasis aturan menunjukkan efektivitas yang terbatas, dengan rata-rata keberhasilan kendaraan dalam menyelesaikan lintasan sebesar 55.56%. Faktor utama penyebab keterbatasan ini meliputi kontrol berbasis keyboard, kecepatan kendaraan yang relatif tinggi, serta latensi sistem. *Profiling* menunjukkan bahwa algoritma mengemudi hanya mampu beroperasi pada  $\sim 5$  FPS (5 input per detik), yang memberikan jeda signifikan antara pengambilan keputusan dan aksi aktual dalam permainan.

Penggunaan CPU berkisar antara 26.6% dan 63.2%, dengan rata-rata 44.3%, menunjukkan beban komputasi sedang selama simulasi dijalankan. Penggunaan Memori relatif stabil, berada di kisaran 84.0% hingga 84.5% menunjukkan bahwa simulasi menggunakan memori secara konsisten tanpa lonjakan atau kebocoran memori yang signifikan.

Dengan demikian, dapat disimpulkan bahwa kombinasi metode deteksi jalur dan algoritma navigasi yang digunakan memiliki tingkat efektivitas menengah dalam mensimulasikan navigasi otomatis pada *TrackMania Nations Forever*.

Penelitian ini memberikan kontribusi awal yang menjanjikan dan dapat dikembangkan lebih lanjut untuk mencapai kinerja yang lebih optimal.

## 5.2 Saran

1. Optimalisasi Algoritma: Diperlukan peningkatan efisiensi pada proses deteksi dan pengambilan keputusan untuk memperbaiki performa simulasi dan mengurangi latensi.
2. Penambahan Variasi Lingkungan: Penelitian selanjutnya dapat mencakup variasi lingkungan permainan, seperti kondisi malam hari, persimpangan, atau kehadiran kendaraan lain.
3. Menerapkan umpan-balik ketika kendaraan menabrak batas jalur.
4. Integrasi Model Pembelajaran Mesin: Penggunaan model pembelajaran seperti CNN maupun *reinforcement learning* dapat menjadi alternatif untuk meningkatkan kemampuan navigasi kendaraan secara otonom.

## DAFTAR PUSTAKA

- Authier-Carcelen, J.-B., & Zadourian, R. (2024). *A Driving Model in the Realistic 3D Game Trackmania Using Deep Reinforcement Learning.* <https://doi.org/10.20944/preprints202409.0778.v1>
- Badue, C., Guidolini, R., Carneiro, R. V., Azevedo, P., Cardoso, V. B., Forechi, A., Jesus, L., Berriel, R., Paixão, T. M., Mutz, F., de Paula Veronese, L., Oliveira-Santos, T., & De Souza, A. F. (2021). Self-driving cars: A survey. Dalam *Expert Systems with Applications* (Vol. 165). Elsevier Ltd. <https://doi.org/10.1016/j.eswa.2020.113816>
- Bhupathi, K. C., & Ferdowsi, H. (2020). An Augmented Sliding Window Technique to Improve Detection of Curved Lanes in Autonomous Vehicles. *IEEE International Conference on Electro Information Technology, 2020-July*, 522–527. <https://doi.org/10.1109/EIT48999.2020.9208278>
- Bonyadi, M. R., Michalewicz, Z., Nallaperuma, S., & Neumann, F. (2017). *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES I Ahura: A heuristic-based racer for the open racing car simulator.* <http://cig.dei.polimi.it/>
- Bottazzi, V. S., Borges, P. V. K., Stantic, B., & Jo, J. (2014). Adaptive regions of interest based on HSV histograms for lane marks detection. *Advances in Intelligent Systems and Computing*, 274, 677–687. [https://doi.org/10.1007/978-3-319-05582-4\\_58](https://doi.org/10.1007/978-3-319-05582-4_58)
- Canny, J. (1983). *A VARIATIONAL APPROACH TO EDGE DETECTION.* [www.aaai.org](http://www.aaai.org)
- Chen, C., Seff, A., Kornhauser, A., & Xiao, J. (2015). *DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving.* <http://arxiv.org/abs/1505.00256>

- Couplie, M., Nivando Bezerra, F., Bertrand, G., Bertrand Topological, G., & Bertrand Laboratoire, G. A. (2001). operators for grayscale image processing. Dalam *Journal of Electronic Imaging* (Vol. 10, Nomor 4). [www.esiee.fr/coupliem/Sdi](http://www.esiee.fr/coupliem/Sdi)
- Dosovitskiy, A., Ros, G., Codevilla, F., López, A., & Koltun, V. (2017). *CARLA: An Open Urban Driving Simulator*.
- Erdelyi, C. (2019). *Using Computer Vision Techniques to Play an Existing Video Game*. <https://scholarworks.calstate.edu/downloads/7w62f8544>
- Gonzalez, R. C. ., & Woods, R. E. . (2002). *Digital image processing*. Prentice Hall.
- Huang, T. S. (1990). *Computer Vision: Evolution and Promise*.
- Illingworth, J., & Kittler, J. (1988). SURVEY A Survey of the Hough Transform. Dalam *COMPUTER VISION, GRAPHICS, AND IMAGE PROCESSING* (Vol. 44).
- Kaganami, H. G., & Beiji, Z. (2009). Region-based segmentation versus edge detection. *IIH-MSP 2009 - 2009 5th International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, 1217–1221. <https://doi.org/10.1109/IIH-MSP.2009.13>
- Kaur, D., & Kaur, Y. (2014). International Journal of Computer Science and Mobile Computing Various Image Segmentation Techniques: A Review. Dalam *International Journal of Computer Science and Mobile Computing* (Vol. 3, Nomor 5). [www.ijcsmc.com](http://www.ijcsmc.com)
- Kaur, P., Taghavi, S., Tian, Z., & Shi, W. (2021). *A Survey on Simulators for Testing Self-Driving Cars*. <http://arxiv.org/abs/2101.05337>
- Klauer, S. G., Guo, F., Simons-Morton, B. G., Claude Ouimet, M., E. Lee, S., & A. Dingus, T. (2014). *The Royal Society for the Prevention of Accidents Road Safety Factsheet*. [www.rospa.com](http://www.rospa.com)

- Klauer, S. G., Guo, F., Simons-Morton, B. G., Ouimet, M. C., Lee, S. E., & Dingus, T. A. (2014). Distracted Driving and Risk of Road Crashes among Novice and Experienced Drivers. *New England Journal of Medicine*, 370(1), 54–59. <https://doi.org/10.1056/nejmsa1204142>
- Liu, L., Lu, S., Zhong, R., Wu, B., Yao, Y., Zhang, Q., & Shi, W. (2021). Computing Systems for Autonomous Driving: State of the Art and Challenges. *IEEE Internet of Things Journal*, 8(8), 6469–6486. <https://doi.org/10.1109/JIOT.2020.3043716>
- Nezami, F. N., Wächter, M. A., Pipa, G., & König, P. (2020). Project Westdrive: Unity City With Self-Driving Cars and Pedestrians for Virtual Reality Studies. *Frontiers in ICT*, 7. <https://doi.org/10.3389/fict.2020.00001>
- Pan, X., You, Y., Wang, Z., & Lu, C. (2017). *Virtual to Real Reinforcement Learning for Autonomous Driving*.
- Pomerleau, D. A. (1988). *ALVINN: AN AUTONOMOUS LAND VEHICLE IN A NEURAL NETWORK*.
- Posch, D., & Rask, J. (2017). *A model based approach to lane detection and lane positioning using OpenCV*.
- Pradityarahman, Y., Hestiwi, D. I., Al-Mustaqim, F., & Hakim, M. L. (2021). *Prototype Smart Autonomous Car berbasis Deep Learning dengan Sistem Pencegah Kecelakaan*. <https://journal.uny.ac.id/index.php/jee>
- Pulli, K., Baksheev, A., Korniyakov, K., & Eruhimov, V. (2012). Real-Time Computer Vision with openCV. *Communications of the ACM*, 55(6), 61–69. <https://doi.org/10.1145/2184319.2184337>
- Punagin, A. (2020). Analysis of Lane Detection Techniques on Structured Roads using OpenCV. *International Journal for Research in Applied Science and Engineering Technology*, 8(5), 2994–3003. <https://doi.org/10.22214/ijraset.2020.5502>

- Rathore, A. S. (2008). Lane Detection for Autonomous Vehicles using OpenCV Library. *International Research Journal of Engineering and Technology*, 1326. [www.irjet.net](http://www.irjet.net)
- Roy, P., Dutta, S., Dey, N., Dey, G., Chakraborty, S., & Ray, R. (2014). Adaptive thresholding: A comparative study. *2014 International Conference on Control, Instrumentation, Communication and Computational Technologies, ICCICCT 2014*, 1182–1186. <https://doi.org/10.1109/ICCICCT.2014.6993140>
- S., S., S., V. K., Z., M. R. H., K., S. K., S., D., & P. D., R. (2021). Advanced Driver Assistant System. *Indian Journal of Computer Science*, 6(3–4), 35. <https://doi.org/10.17010/ijcs/2021/v6/i3-4/165410>
- Sallab, A. El, Abdou, M., Perot, E., & Yogamani, S. (2017). *Deep Reinforcement Learning framework for Autonomous Driving*.
- Sural, S., Qian, G., & Pramanik, S. (2002). Segmentation and histogram generation using the HSV color space for image retrieval. *IEEE International Conference on Image Processing*, 2. <https://doi.org/10.1109/icip.2002.1040019>
- Sweigart, A. (2021). *PyAutoGUI Documentation*.
- Szeliski, R. (2022). *Computer Vision: Algorithms and Applications*. [https://docs.opencv.org/3.4/da/d6e/tutorial\\_py\\_geometric\\_transformations.html](https://docs.opencv.org/3.4/da/d6e/tutorial_py_geometric_transformations.html)
- Wang, L., Han, M., Li, X., Zhang, N., & Cheng, H. (2021). Review of Classification Methods on Unbalanced Data Sets. *IEEE Access*, 9, 64606–64628. <https://doi.org/10.1109/ACCESS.2021.3074243>
- Yang, X., & Ling, Z. (2015). *RESEARCH ON LANE DETECTION TECHNOLOGY BASED ON OPENCV*.

## LAMPIRAN

### Lampiran 1 Performa Fungsi Penangkapan Layar Permainan

Performa dihitung berdasarkan jumlah pemanggilan *frame* per detik.

```
*** Capture Profiling ***
168526 function calls in 9.035 seconds

Ordered by: cumulative time
List reduced from 98 to 20 due to restriction <20>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    939    0.015    0.000    8.935    0.010 C:\Users\acer\AppData\Local\Temp\ipykernel_10064\265601963.py:142(capture_screen)
    939    0.006    0.000    8.375    0.009 d:\ThInKIn in programming\.\conda\lib\site-packages\mss\base.py:82(grab)
    939    8.317    0.009    8.368    0.009 d:\ThInKIn in programming\.\conda\lib\site-packages\mss\windows.py:196(_grab_impl)
    939    0.016    0.000    0.369    0.000 d:\ThInKIn in programming\.\conda\lib\site-packages\mss\factory.py:12(mss)
    939    0.051    0.000    0.344    0.000 d:\ThInKIn in programming\.\conda\lib\site-packages\mss\windows.py:98(__init__)
    939    0.009    0.000    0.113    0.000 d:\ThInKIn in programming\.\conda\lib\site-packages\mss\windows.py:147(_set_dpi_awareness)
    939    0.010    0.000    0.111    0.000 d:\ThInKIn in programming\.\conda\lib\site-packages\mss\windows.py:137(_set_cfunctions)
    939    0.103    0.000    0.103    0.000 {built-in method sys.getwindowsversion}
11268    0.014    0.000    0.101    0.000 d:\ThInKIn in programming\.\conda\lib\site-packages\mss\base.py:247(_cfactory)
11274    0.013    0.000    0.086    0.000 {built-in method builtinsgetattr}
    939    0.005    0.000    0.078    0.000 {built-in method builtins.print}
11268    0.011    0.000    0.073    0.000 d:\ThInKIn in programming\.\conda\lib\ctypes\__init__.py:386(__getattr__)
1878    0.019    0.000    0.073    0.000 C:\Users\acer\AppData\Roaming\Python\Python311\site-packages\ipykernel\iostream.py:655(write)
    939    0.067    0.000    0.067    0.000 {cvtColor}
1878    0.011    0.000    0.066    0.000 d:\ThInKIn in programming\.\conda\lib\ctypes\__init__.py:342(__init__)
    939    0.062    0.000    0.066    0.000 {built-in method numpy.array}
11268    0.056    0.000    0.059    0.000 d:\ThInKIn in programming\.\conda\lib\ctypes\__init__.py:393(__getitem__)
    939    0.002    0.000    0.044    0.000 d:\ThInKIn in programming\.\conda\lib\site-packages\mss\base.py:59(__exit__)
1878    0.041    0.000    0.044    0.000 {built-in method builtins.__build_class__}
    939    0.043    0.000    0.043    0.000 d:\ThInKIn in programming\.\conda\lib\site-packages\mss\windows.py:123(close)
```

### Lampiran 2 Performa Fungsi Deteksi Jalur

Performa dihitung berdasarkan jumlah pemrosesan deteksi jalur per detik.

```
*** Detection Profiling ***
168526 function calls in 9.035 seconds

Ordered by: cumulative time
List reduced from 98 to 20 due to restriction <20>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    939    0.015    0.000    8.935    0.010 C:\Users\acer\AppData\Local\Temp\ipykernel_10064\265601963.py:142(capture_screen)
    939    0.006    0.000    8.375    0.009 d:\ThInKIn in programming\.\conda\lib\site-packages\mss\base.py:82(grab)
    939    8.317    0.009    8.368    0.009 d:\ThInKIn in programming\.\conda\lib\site-packages\mss\windows.py:196(_grab_impl)
    939    0.016    0.000    0.369    0.000 d:\ThInKIn in programming\.\conda\lib\site-packages\mss\factory.py:12(mss)
    939    0.051    0.000    0.344    0.000 d:\ThInKIn in programming\.\conda\lib\site-packages\mss\windows.py:98(__init__)
    939    0.009    0.000    0.113    0.000 d:\ThInKIn in programming\.\conda\lib\site-packages\mss\windows.py:147(_set_dpi_awareness)
    939    0.103    0.000    0.103    0.000 {built-in method sys.getwindowsversion}
11268    0.014    0.000    0.101    0.000 d:\ThInKIn in programming\.\conda\lib\site-packages\mss\base.py:247(_cfactory)
11274    0.013    0.000    0.086    0.000 {built-in method builtinsgetattr}
    939    0.005    0.000    0.078    0.000 {built-in method builtins.print}
11268    0.011    0.000    0.073    0.000 d:\ThInKIn in programming\.\conda\lib\ctypes\__init__.py:386(__getattr__)
1878    0.019    0.000    0.073    0.000 C:\Users\acer\AppData\Roaming\Python\Python311\site-packages\ipykernel\iostream.py:655(write)
    939    0.067    0.000    0.067    0.000 {cvtColor}
1878    0.011    0.000    0.066    0.000 d:\ThInKIn in programming\.\conda\lib\ctypes\__init__.py:342(__init__)
    939    0.062    0.000    0.066    0.000 {built-in method numpy.array}
11268    0.056    0.000    0.059    0.000 d:\ThInKIn in programming\.\conda\lib\ctypes\__init__.py:393(__getitem__)
    939    0.002    0.000    0.044    0.000 d:\ThInKIn in programming\.\conda\lib\site-packages\mss\base.py:59(__exit__)
1878    0.041    0.000    0.044    0.000 {built-in method builtins.__build_class__}
    939    0.043    0.000    0.043    0.000 d:\ThInKIn in programming\.\conda\lib\site-packages\mss\windows.py:123(close)
```

### Lampiran 3 Performa Fungsi Mengemudi

Performa dihitung berdasarkan jumlah pemanggilan fungsi mengemudi per detik.

```
*** Drive Loop Profiling ***
168526 function calls in 9.035 seconds

Ordered by: cumulative time
List reduced from 98 to 20 due to restriction <20>

ncalls  tottime   percall  cumtime  percall filename:lineno(function)
    939    0.015    0.000   8.935    0.010  c:\users\acer\appdata\local\temp\ipykernel_10064\265601963.py:142(capture_screen)
    939    0.006    0.000   8.375    0.009  d:\thinkin\programming\.conda\lib\site-packages\mss\base.py:82(grab)
    939    8.317    0.009   8.368    0.009  d:\thinkin\programming\.conda\lib\site-packages\mss\windows.py:196(_grab_impl)
    939    0.016    0.000   0.369    0.000  d:\thinkin\programming\.conda\lib\site-packages\mss\factory.py:12(msc)
    939    0.051    0.000   0.344    0.000  d:\thinkin\programming\.conda\lib\site-packages\mss\windows.py:98(__init__)
    939    0.009    0.000   0.113    0.000  d:\thinkin\programming\.conda\lib\site-packages\mss\windows.py:147(_set_dpi_awareness)
    939    0.010    0.000   0.111    0.000  d:\thinkin\programming\.conda\lib\site-packages\mss\windows.py:137(_set_cfunctions)
    939    0.103    0.000   0.103    0.000  {built-in method sys.getwindowsversion}
11268    0.014    0.000   0.101    0.000  d:\thinkin\programming\.conda\lib\site-packages\mss\base.py:247(_cfactory)
11274    0.013    0.000   0.086    0.000  {built-in method builtinsgetattr}
    939    0.005    0.000   0.078    0.000  {built-in method builtins.print}
11268    0.011    0.000   0.073    0.000  d:\thinkin\programming\.conda\lib\ctypes\__init__.py:386(__getattr__)
1878    0.019    0.000   0.073    0.000  c:\users\acer\appdata\roaming\python\python311\site-packages\ipykernel\iostream.py:655(write)
    939    0.067    0.000   0.067    0.000  {cvtcColor}
1878    0.011    0.000   0.066    0.000  d:\thinkin\programming\.conda\lib\ctypes\__init__.py:342(__init__)
    939    0.062    0.000   0.066    0.000  {built-in method numpy.array}
11268    0.056    0.000   0.059    0.000  d:\thinkin\programming\.conda\lib\ctypes\__init__.py:393(__getitem__)
    939    0.002    0.000   0.044    0.000  d:\thinkin\programming\.conda\lib\site-packages\mss\base.py:59(__exit__)
1878    0.041    0.000   0.044    0.000  {built-in method builtins._build_class_}
    939    0.043    0.000   0.043    0.000  d:\thinkin\programming\.conda\lib\site-packages\mss\windows.py:123(close)
```

### Lampiran 4 Performa Fungsi Debugging Deteksi Jalur

Performa dihitung berdasarkan jumlah pemanggilan berbagai fungsi deteksi jalur serta menampilkan hasilnya per detik.

```
*** Drive Loop Profiling ***
362050 function calls in 2.355 seconds

Ordered by: cumulative time
List reduced from 152 to 20 due to restriction <20>

ncalls  tottime   percall  cumtime  percall filename:lineno(function)
    234    0.185    0.001   0.758    0.003  c:\users\acer\appdata\local\temp\ipykernel_10064\3845042713.py:374(get_lane_line_indices_sliding_windows)
    234    0.212    0.001   0.724    0.003  c:\users\acer\appdata\local\temp\ipykernel_10064\3845042713.py:468(overlay_lane_lines)
1404    0.146    0.000   0.708    0.001  c:\users\acer\appdata\roaming\python\python311\site-packages\numpy\lib\polynomial.py:453(polyfit)
    234    0.110    0.000   0.489    0.002  c:\users\acer\appdata\local\temp\ipykernel_10064\3845042713.py:272(get_lane_line_previous_window)
209898    0.277    0.000   0.277    0.000  {line}
    468    0.255    0.001   0.255    0.001  {warpPerspective}
1404    0.151    0.000   0.248    0.000  c:\users\acer\appdata\roaming\python\python311\site-packages\numpy\lib\twodim_base.py:534(vander)
1404    0.196    0.000   0.214    0.000  c:\users\acer\appdata\roaming\python\python311\site-packages\numpy\linalg\linalg.py:2191(lstsq)
    3276    0.202    0.000   0.202    0.000  {method 'nonzero' of 'numpy.ndarray' objects}
    234    0.006    0.000   0.146    0.001  c:\users\acer\appdata\local\temp\ipykernel_10064\3845042713.py:239(display_curvature_offset)
    468    0.140    0.000   0.140    0.000  {putText}
4878    0.129    0.000   0.129    0.000  {method 'reduce' of 'numpy.ufunc' objects}
1404    0.002    0.000   0.098    0.000  {method 'sum' of 'numpy.ndarray' objects}
1404    0.001    0.000   0.096    0.000  c:\users\acer\appdata\roaming\python\python311\site-packages\numpy\core\_methods.py:47(_sum)
1404    0.004    0.000   0.094    0.000  {method 'accumulate' of 'numpy.ufunc' objects}
    234    0.004    0.000   0.037    0.000  c:\users\acer\appdata\local\temp\ipykernel_10064\3845042713.py:186(calculate_curvature)
    2772    0.004    0.000   0.031    0.000  c:\users\acer\appdata\roaming\python\python311\site-packages\numpy\core\fromnumeric.py:3385(mean)
    468    0.003    0.000   0.029    0.000  c:\users\acer\appdata\local\temp\ipykernel_10064\3845042713.py:212(calculate_histogram)
    2772    0.009    0.000   0.028    0.000  c:\users\acer\appdata\roaming\python\python311\site-packages\numpy\core\_methods.py:101(_mean)
    702    0.003    0.000   0.027    0.000  c:\users\acer\appdata\roaming\python\python311\site-packages\numpy\core\fromnumeric.py:71(_wrapreduction)
```

## Lampiran 5 Kelas Deteksi Tepi

Kelas deteksi tepi digunakan untuk mendeteksi piksel dengan intensitas yang berbeda. Tujuannya adalah mengekstrak piksel jalan dalam permainan.

```
class EdgeDetection:
    def threshold(self, img, thresh=(0, 255)):
        binary = np.zeros_like(img)
        binary[(img >= thresh[0]) & (img <= thresh[1])] = 255
        return True, binary

    def blur_gaussian(self, img, ksize=3):
        return cv2.GaussianBlur(img, (ksize, ksize), 0)

    def mag_thresh(self, img, sobel_kernel=3, thresh=(0, 255)):
        sobelx = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=sobel_kernel)
        sobely = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=sobel_kernel)
        gradmag = np.sqrt(sobelx**2 + sobely**2)
        scale_factor = np.max(gradmag)/255
        gradmag = (gradmag/scale_factor).astype(np.uint8)
        binary_output = np.zeros_like(gradmag)
        binary_output[(gradmag >= thresh[0]) & (gradmag <= thresh[1])] = 255
        return binary_output
```

## Lampiran 6 Kelas Deteksi Jalur

Kelas ini terdiri dari berbagai fungsi utama seperti memetakan ROI, kalkulasi posisi mobil terhadap pusat jalur, kurvatur jalan, transformasi perspektif, kalkulasi histogram, menentukan posisi *window* pertama pada piksel yang terdeteksi, serta *sliding window*. Beberapa fungsi tambahan lainnya bertujuan untuk menampilkan *lane* yang berhasil dideteksi untuk kepentingan analisis.

```

class Lane:
"""
    Represents a lane on a road.
"""

def __init__(self, orig_frame):
    """
        Default constructor

    :param orig_frame: Original camera image (i.e. frame)
    """
    self.orig_frame = orig_frame

    # This will hold an image with the lane lines
    self.lane_line_markings = None

    # Previous polynomial coefficients
    self.prev_left_fit = None
    self.prev_right_fit = None

    # This will hold the image after perspective transformation
    self.warped_frame = None
    self.transformation_matrix = None
    self.inv_transformation_matrix = None

    # (Width, Height) of the original video frame (or image)
    self.orig_image_size = self.orig_frame.shape[::-1][1:]

    width = self.orig_image_size[0]
    height = self.orig_image_size[1]
    self.width = width
    self.height = height

    """ Static ROI points.
    These ROI are based on the screen dimensions of 1920 x 1080 pixels.
    The roi must satisfy the game screen dimensions.

    ...
    # self.roi_points = np.float32([
    #     (634, 457), # Top-left
    #     (1266, 457), # Top-right
    #     (1700, 600), # Bottom-right
    #     (200, 600) # Bottom-left
    # ])
    """

    # ... for width": 1270, "height": 813 screen resolution ...
    # self.roi_points = np.float32([
    #     (419, 344), # Top-left
    #     (837, 344), # Top-right
    #     (1124, 452), # Bottom-right
    #     (132, 452) # Bottom-left
    # ])

    """ for 800 x 600 screen resolution """
    self.roi_points = np.float32([
        (266, 200), # Top-left
        (532, 200), # Top-right
        (700, 300), # Bottom-right
        (100, 300) # Bottom-left
    ])

    """
    Calculate ROI points based on screen dimensions
    Returns points in clockwise order: top-left, top-right, bottom-right, bottom-left
    """

    # # # Define points as percentages of width and height
    self.roi_points = np.float32([
        (self.width * 0.25, self.height * 0.5), # Top-left (25% from left, 50% down)
        (self.width * 0.75, self.height * 0.5), # Top-right (75% from left, 50% down)
        (self.width * 0.85, self.height * 0.9), # Bottom-right (85% from left, 90% down)
        (self.width * 0.15, self.height * 0.9) # Bottom-left (15% from left, 90% down)
    ])

    # The desired corner locations of the region of interest
    # after we perform perspective transformation.
    # Assume image width of 600, padding == 150.
    self.padding = int(0.25 * width) # padding from side of the image in pixels

    self.desired_roi_points = np.float32([
        [self.padding, 0], # Top-left corner
        [self.padding, self.orig_image_size[1]], # Bottom-left corner
        [self.orig_image_size[
            0]-self.padding, self.orig_image_size[1]], # Bottom-right corner
        [self.orig_image_size[0]-self.padding, 0] # Top-right corner
    ])

```

```

# Histogram that shows the white pixel peaks for lane line detection
self.histogram = None

# Sliding window parameters
# no of windows
self.no_of_windows = 10

# window size
self.margin = int((1/12) * width) # Window width is +/- margin
self.minpix = int((1/24) * width) # Min no. of pixels to recenter window

# Best fit polynomial lines for left line and right line of the lane
self.left_fit = None
self.right_fit = None
self.left_lane_inds = None
self.right_lane_inds = None
self.ploty = None
self.left_fitx = None
self.right_fitx = None
self.leftx = None
self.rightx = None
self.lefty = None
self.righty = None

# Pixel parameters for x and y dimensions
self.YM_PER_PIX = 10.0 / 1000 # meters per pixel in y dimension
self.XM_PER_PIX = 3.7 / 781 # meters per pixel in x dimension

# Radii of curvature and offset
self.left_curvem = None
self.right_curvem = None
self.center_offset = None

# Pixel parameters for x and y dimensions
self.YM_PER_PIX = 10.0 / 1000 # meters per pixel in y dimension
self.XM_PER_PIX = 3.7 / 781 # meters per pixel in x dimension

# Radii of curvature and offset
self.left_curvem = None
self.right_curvem = None
self.center_offset = None

def calculate_car_position(self, print_to_terminal=False):
    """
    Calculate the position of the car relative to the center

    :param: print_to_terminal Display data to console if True
    :return: Offset from the center of the lane
    """
    # Assume the camera is centered in the image.
    # Get position of car in centimeters
    # In warped space (800x600), car is assumed at bottom center of ROI
    car_location = (self.desired_roi_points[2][0] + self.desired_roi_points[3][0]) / 2

    # Lane bottoms in warped space (green lines)
    height = self.warped_frame.shape[0] # 600
    bottom_left = self.left_fitx[-1] # Last point of left green line
    bottom_right = self.right_fitx[-1] # Last point of right green line

    # center of the lane in warped space
    center_lane = (bottom_right + bottom_left)/2

    # offset in pixels
    center_offset_px = car_location - center_lane

    # Convert to centimeters using XM_PER_PIX (adjusted for game)
    center_offset = center_offset_px * self.XM_PER_PIX * 100

    lane_width_px = bottom_right - bottom_left
    lane_width_m = lane_width_px * self.XM_PER_PIX * 100

    # Assume car width ~0.5m, green lines ~0.1m each
    offset_adjustment = (0.5 / lane_width_m) * (bottom_right - bottom_left) * self.XM_PER_PIX * 100 / 2
    center_offset -= offset_adjustment if center_offset > 0 else -offset_adjustment

```

```

    if print_to_terminal == True:
        | print(str(center_offset) + 'cm')

    self.center_offset = center_offset

    return center_offset

def calculate_curvature(self, print_to_terminal=False):
    """
    Calculate the road curvature in meters.

    :param: print_to_terminal Display data to console if True
    :return: Radii of curvature
    """

    ym_per_pix = 10/600 # meters per pixel in y dimension
    xm_per_pix = 2/800 #adjust for 800 x 600 screen resolution
    ploty = self.ploty
    left_fitx = self.left_fitx
    right_fitx = self.right_fitx

    # Define y-value where we want radius of curvature
    left_fit_cr = np.polyfit(ploty * ym_per_pix, left_fitx * xm_per_pix, 2)
    right_fit_cr = np.polyfit(ploty * ym_per_pix, right_fitx * xm_per_pix, 2)

    y_eval = np.max(ploty)
    left_curverad = ((1 + (2 * left_fit_cr[0] * y_eval * ym_per_pix + left_fit_cr[1])**2)**1.5) / np.absolute(2 * left_fit_cr[0])
    right_curverad = ((1 + (2 * right_fit_cr[0] * y_eval * ym_per_pix + right_fit_cr[1])**2)**1.5) / np.absolute(2 * right_fit_cr[0])

    self.left_curvem = left_curverad
    self.right_curvem = right_curverad

    return left_curverad, right_curverad

def calculate_histogram(self, frame=None, plot=True):
    """
    Calculate the image histogram to find peaks in white pixel count

    :param frame: The warped image
    :param plot: Create a plot if True
    """

    if frame is None:
        | frame = self.warped_frame

    # Generate the histogram
    self.histogram = np.sum(frame[int(frame.shape[0]/2):,:,:], axis=0)

    if plot == True:

        # Draw both the image and the histogram
        figure, (ax1, ax2) = plt.subplots(2,1) # 2 row, 1 columns
        figure.set_size_inches(10, 5)
        ax1.imshow(frame, cmap='gray')
        ax1.set_title("Warped Binary Frame")
        ax2.plot(self.histogram)
        ax2.set_title("Histogram Peaks")
        plt.show()

        return self.histogram

def display_curvature_offset(self, frame=None, plot=False):
    """
    Display curvature and offset statistics on the image

    :param: plot Display the plot if True
    :return: Image with lane lines and curvature
    """

    image_copy = None
    if frame is None:
        | image_copy = self.orig_frame.copy()
    else:
        | image_copy = frame

    cv2.putText(image_copy, 'Curve Radius: '+str((
        self.left_curvem+self.right_curvem)/2)[:7]+' m', (int((5/600)*self.width), int((20/338)*self.height)), cv2.FONT_HERSHEY_SIMPLEX, (float((0.5/600)*self.width)),(255,255,255),2,cv2.LINE_AA)
    cv2.putText(image_copy, 'Center Offset: '+str(
        self.center_offset)[:7]+' cm', (int((5/600)*self.width), int((40/338)*self.height)), cv2.FONT_HERSHEY_SIMPLEX, (float((0.5/600)*self.width)),(255,255,255),2,cv2.LINE_AA)
    # debug

    if plot==True:
        | cv2.imshow("Image with Curvature and Offset", image_copy)

    return image_copy

```

```

def get_lane_line_previous_window(self, left_fit, right_fit, plot=False):
    """
    Use the lane line from the previous sliding window to get the parameters
    for the polynomial line for filling in the lane line
    :param: left_fit Polynomial function of the left lane line
    :param: right_fit Polynomial function of the right lane line
    :param: plot To display an image or not

    """
    if self.prev_left_fit is not None and self.prev_right_fit is not None:
        left_fit = 0.7 * self.prev_left_fit + 0.3 * left_fit
        right_fit = 0.7 * self.prev_right_fit + 0.3 * right_fit
        self.prev_left_fit = left_fit
        self.prev_right_fit = right_fit
        # margin is a sliding window parameter
        margin = self.margin

    # Find the x and y coordinates of all the nonzero
    # (i.e. white) pixels in the frame.
    nonzero = self.warped_frame.nonzero()
    nonzeroy = np.array(nonzero[0])
    nonzerox = np.array(nonzero[1])

    # Store left and right lane pixel indices
    left_lane_inds = ((nonzeroy > (left_fit[0]*(nonzeroy**2) + left_fit[1]*nonzeroy + left_fit[2] - margin)) &
                      (nonzeroy < (left_fit[0]*(nonzeroy**2) + left_fit[1]*nonzeroy + left_fit[2] + margin)))
    right_lane_inds = ((nonzeroy > (right_fit[0]*(nonzeroy**2) + right_fit[1]*nonzeroy + right_fit[2] - margin)) &
                       (nonzeroy < (right_fit[0]*(nonzeroy**2) + right_fit[1]*nonzeroy + right_fit[2] + margin)))
    self.left_lane_inds = left_lane_inds
    self.right_lane_inds = right_lane_inds

    # Get the left and right lane line pixel locations
    leftx = nonzerox[left_lane_inds]
    lefty = nonzeroy[left_lane_inds]
    rightx = nonzerox[right_lane_inds]
    righty = nonzeroy[right_lane_inds]

    self.leftx = leftx
    self.rightx = rightx
    self.lefty = lefty
    self.righty = righty

    # Fit a second order polynomial curve to each lane line
    left_fit = np.polyfit(lefty, leftx, 2)
    right_fit = np.polyfit(righty, rightx, 2)
    self.left_fit = left_fit
    self.right_fit = right_fit

    # Create the x and y values to plot on the image
    ploty = np.linspace(
        0, self.warped_frame.shape[0]-1, self.warped_frame.shape[0])
    left_fitx = left_fit[0]*ploty**2 + left_fit[1]*ploty + left_fit[2]
    right_fitx = right_fit[0]*ploty**2 + right_fit[1]*ploty + right_fit[2]
    self.ploty = ploty
    self.left_fitx = left_fitx
    self.right_fitx = right_fitx

    if plot==True:
        # Generate images to draw on
        out_img = np.dstack((self.warped_frame, self.warped_frame, (
            self.warped_frame)))*255
        window_img = np.zeros_like(out_img)

        # Add color to the left and right line pixels
        out_img[nonzeroy[left_lane_inds], nonzerox[left_lane_inds]] = [255, 0, 0]
        out_img[nonzeroy[right_lane_inds], nonzerox[right_lane_inds]] = [
            0, 255, 0, 255]

        # Create a polygon to show the search window area, and recast
        # the x and y points into a usable format for cv2.fillPoly()
        margin = self.margin
        left_line_window1 = np.array([np.transpose(np.vstack([
            left_fitx-margin, ploty]))])
        left_line_window2 = np.array([np.flipud(np.transpose(np.vstack([
            left_fitx+margin, ploty])))])
        left_line_pts = np.hstack((left_line_window1, left_line_window2))
        right_line_window1 = np.array([np.transpose(np.vstack([
            right_fitx-margin, ploty]))])
        right_line_window2 = np.array([np.flipud(np.transpose(np.vstack([
            right_fitx+margin, ploty])))])
        right_line_pts = np.hstack((right_line_window1, right_line_window2))

        # Draw the lane onto the warped blank image
        cv2.fillPoly(window_img, np.int_(left_line_pts), (0,255, 0))
        cv2.fillPoly(window_img, np.int_(right_line_pts), (0,255, 0))
        result = cv2.addWeighted(out_img, 1, window_img, 0.3, 0)

```

```

# Plot the figures
figure, (ax1, ax2, ax3) = plt.subplots(3,1) # 3 rows, 1 column
figure.set_size_inches(10, 10)
figure.tight_layout(pad=3.0)
ax1.imshow(cv2.cvtColor(self.orig_frame, cv2.COLOR_BGR2RGB))
ax2.imshow(self.warped_frame, cmap='gray')
ax3.imshow(result)
ax3.plot(leftx_fity, ploty, color='yellow')
ax3.plot(rightx_fity, ploty, color='yellow')
ax1.set_title("Original Frame")

def get_lane_line_indices_sliding_windows(self, plot=False):

    # if self.prev_left_fit is not None and self.prev_right_fit is not None:
    #     return self.prev_left_fit, self.prev_right_fit # skip if we have previous fits
    histogram = self.calculate_histogram(plot=False)
    midpoint = int(histogram.shape[0] / 2)
    leftx_base = np.argmax(histogram[:midpoint])
    rightx_base = np.argmax(histogram[midpoint:]) + midpoint

    # If we don't have previous fits, start with base points
    if self.prev_left_fit is None:
        leftx_current = leftx_base
        rightx_current = rightx_base
    # histogram = self.histogram
    # midpoint = int(histogram.shape[0] / 2)
    # leftx_base = np.argmax(histogram[:midpoint])
    # rightx_base = np.argmax(histogram[midpoint:]) + midpoint

    nwindows = 6
    window_height = int(self.warped_frame.shape[0] / nwindows)
    nonzero = self.warped_frame.nonzero()
    nonzeroy = np.array(nonzero[0])
    nonzerox = np.array(nonzero[1])

    leftx_current = leftx_base
    rightx_current = rightx_base
    margin = 100
    minpix = 50

    left_lane_inds = []
    right_lane_inds = []

    for window in range(nwindows):
        win_y_low = self.warped_frame.shape[0] - (window + 1) * window_height
        win_y_high = self.warped_frame.shape[0] - window * window_height

        if self.prev_left_fit is not None and window > 0:
            leftx_current = int(self.prev_left_fit[0] * (win_y_high**2) + self.prev_left_fit[1] * win_y_high + self.prev_left_fit[2])
            rightx_current = int(self.prev_right_fit[0] * (win_y_high**2) + self.prev_right_fit[1] * win_y_high + self.prev_right_fit[2])

        win_xleft_low = leftx_current - margin
        win_xleft_high = leftx_current + margin
        win_xright_low = rightx_current - margin
        win_xright_high = rightx_current + margin

        good_left_inds = ((nonzeroy >= win_y_low) & (nonzeroy < win_y_high) &
                          (nonzerox >= win_xleft_low) & (nonzerox < win_xleft_high)).nonzero()[0]
        good_right_inds = ((nonzeroy >= win_y_low) & (nonzeroy < win_y_high) &
                           (nonzerox >= win_xright_low) & (nonzerox < win_xright_high)).nonzero()[0]

        left_lane_inds.append(good_left_inds)
        right_lane_inds.append(good_right_inds)

        if len(good_left_inds) > minpix:
            leftx_current = int(np.mean(nonzerox[good_left_inds]))
        if len(good_right_inds) > minpix:
            rightx_current = int(np.mean(nonzerox[good_right_inds]))

        left_lane_inds = np.concatenate(left_lane_inds)
        right_lane_inds = np.concatenate(right_lane_inds)

        leftx = nonzerox[left_lane_inds]
        lefty = nonzeroy[left_lane_inds]
        rightx = nonzerox[right_lane_inds]
        righty = nonzeroy[right_lane_inds]

        left_fit = np.polyfit(lefty, leftx, 2)
        right_fit = np.polyfit(righty, rightx, 2)

    return left_fit, right_fit

```

```

def get_lane_markings(self, frame=None):
    if frame is None:
        frame = self.orig_frame
    # Faster edge detection (adjust thresholds as needed)
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    _, binary = cv2.threshold(gray, 120, 255, cv2.THRESH_BINARY)
    self.lane_line_markings = cv2.GaussianBlur(binary, (5, 5), 0)
    return self.lane_line_markings

def histogram_peak(self):
    """
    Get the left and right peak of the histogram

    Return the x coordinate of the left histogram peak and the right histogram
    peak.
    """
    midpoint = int(self.histogram.shape[0]/2)
    leftx_base = np.argmax(self.histogram[:midpoint])
    rightx_base = np.argmax(self.histogram[midpoint:]) + midpoint

    # (x coordinate of left peak, x coordinate of right peak)
    return leftx_base, rightx_base

def overlay_lane_lines(self, plot=False, left_color=(255, 0, 0),
                      right_color=(255, 0, 0), center_color=(0, 255, 0)):
    """
    Overlay detected lane lines on the original frame with custom colors and a center line.

    :param plot: If True, return the overlay plot (optional)
    :param left_color: BGR tuple for left lane (default green)
    :param right_color: BGR tuple for right lane (default red)
    :param center_color: BGR tuple for center line (default blue)
    :return: Frame with overlaid lanes
    """
    # Generate an image to draw the lane lines on
    warp_zero = np.zeros_like(self.warped_frame).astype(np.uint8)
    color_warp = np.dstack((warp_zero, warp_zero, warp_zero))

    # Recast the x and y points into usable format for cv2.fillPoly()
    pts_left = np.array([np.transpose(np.vstack([self.left_fitx, self.ploty]))])

    pts_right = np.array([np.flipud(np.transpose(np.vstack([self.right_fitx, self.ploty])))])
    pts = np.hstack((pts_left, pts_right))

    # Draw lane on the warped blank image
    cv2.fillPoly(color_warp, np.int_(pts), (0,255, 255))

    # draw left and right lane lines
    for i in range(len(self.ploty)-1):
        cv2.line(color_warp,
                  (int(self.left_fitx[i]), int(self.ploty[i])),
                  (int(self.left_fitx[i+1]), int(self.ploty[i+1])),
                  left_color, 16)
        cv2.line(color_warp,
                  (int(self.right_fitx[i]), int(self.ploty[i])),
                  (int(self.right_fitx[i+1]), int(self.ploty[i+1])),
                  right_color, 16)

    # Draw center line
    center_x = (self.left_fitx + self.right_fitx) / 2
    for i in range(len(self.ploty)-1):
        cv2.line(color_warp,
                  (int(center_x[i]), int(self.ploty[i])),
                  (int(center_x[i+1]), int(self.ploty[i+1])),
                  center_color, 4)

    # Warp the blank back to original image space using inverse perspective
    # matrix (Minv)
    # Warp back to original image space
    newwarp = cv2.warpPerspective(color_warp, self.inv_transformation_matrix,
                                   (self.orig_frame.shape[1], self.orig_frame.shape[0]))

    # Combine with original image
    result = cv2.addWeighted(self.orig_frame, 1, newwarp, 0.3, 0)

    if plot:
        # Optional plotting code
        pass

    return result

```

```

def perspective_transform(self, frame=None, plot=False):
    """
    Perform the perspective transform.
    :param: frame Current frame
    :param: plot Plot the warped image if True
    :return: Bird's eye view of the current lane
    """
    if frame is None:
        frame = self.lane_line_markings

    # Calculate the transformation matrix
    self.transformation_matrix = cv2.getPerspectiveTransform(
        self.roi_points, self.desired_roi_points)

    # Calculate the inverse transformation matrix
    self.inv_transformation_matrix = cv2.getPerspectiveTransform(
        self.desired_roi_points, self.roi_points)

    # Perform the transform using the transformation matrix
    self.warped_frame = cv2.warpPerspective(
        frame, self.transformation_matrix, self.orig_image_size, flags=(
        cv2.INTER_NEAREST))

    # Convert image to binary
    (thresh, binary_warped) = cv2.threshold(
        self.warped_frame, 127, 255, cv2.THRESH_BINARY)
    self.warped_frame = binary_warped

    # Display the perspective transformed (i.e. warped) frame
    if plot == True:
        warped_copy = self.warped_frame.copy()
        warped_plot = cv2.polylines(warped_copy, np.int32([
            [self.roi_points], self.desired_roi_points]), True, (147,20,255), 3)

        # Display the image
        while(1):
            cv2.imshow('Warped Image', warped_plot)

            # Press any key to stop
            if cv2.waitKey(0):
                break

        cv2.destroyAllWindows()

    return self.warped_frame

```

```

def plot_roi(self, frame=None, plot=False):
    """
    Plot the region of interest on an image.
    :param: frame The current image frame
    :param: plot Plot the roi image if True
    """
    if plot == False:
        return

    if frame is None:
        frame = self.orig_frame.copy()

    # Overlay trapezoid on the frame
    this_image = cv2.polylines(frame, np.int32([
        self.roi_points]), True, (147,20,255), 3)

    # Display the image
    while(1):
        cv2.imshow('ROI Image', this_image)

        # Press any key to stop
        if cv2.waitKey(0):
            break

    cv2.destroyAllWindows()

```

```

def main():

    # Load a frame (or image)
    original_frame = cv2.imread(filename)

    # Create a Lane object
    lane_obj = Lane(orig_frame=original_frame)

    # Perform thresholding to isolate lane lines
    lane_line_markings = lane_obj.get_lane_line_markings()

    # Plot the region of interest on the image
    lane_obj.plot_roi(plot=False)

    # Perform the perspective transform to generate a bird's eye view
    # If Plot == True, show image with new region of interest
    warped_frame = lane_obj.perspective_transform(plot=False)

    # Generate the image histogram to serve as a starting point
    # for finding lane line pixels
    histogram = lane_obj.calculate_histogram(plot=False)

    # Find lane line pixels using the sliding window method
    left_fit, right_fit = lane_obj.get_lane_line_indices_sliding_windows(
        plot=False)

    # Fill in the lane line
    lane_obj.get_lane_line_previous_window(left_fit, right_fit, plot=False)

    # Overlay lines on the original frame
    frame_with_lane_lines = lane_obj.overlay_lane_lines(plot=False)

    # Calculate lane line curvature (left and right lane lines)
    lane_obj.calculate_curvature(print_to_terminal=False)

    # Calculate center offset
    lane_obj.calculate_car_position(print_to_terminal=False)

    # Display curvature and center offset on image
    frame_with_lane_lines2 = lane_obj.display_curvature_offset(
        frame=frame_with_lane_lines, plot=True)

    # Create the output file name by removing the '.jpg' part
    size = len(filename)
    new_filename = filename[:size - 4]
    new_filename = new_filename + '_thresholded.jpg'

    # Save the new image in the working directory
    #cv2.imwrite(new_filename, lane_line_markings)

    # Display the image
    #cv2.imshow("Image", lane_line_markings)

    # Display the window until any key is pressed
    cv2.waitKey(0)

    # Close all windows
    cv2.destroyAllWindows()

main()

```

## Lampiran 7 Kelas Debugging

Kelas ini bertujuan untuk menampilkan keseluruhan proses deteksi jalur serta melakukan evaluasi terhadapa performa dari masing-masing algoritma deteksi jalur.

```

import cv2
import numpy as np
from mss import mss
import time
import numpy as np

class Debugging(Lane):
    def __init__(self, frame_shape):
        dummy_frame = np.zeros(frame_shape, dtype=np.uint8)
        super().__init__(dummy_frame)
        self.last_print_time = time.time()

        # Default ROI for 400x300 screen resolution
        self.default_roi_points = np.float32([
            (137, 127), # Top-left
            (258, 127), # Top-right
            (374, 156), # Bottom-right
            (6, 156)     # Bottom-left
        ])
        self.roi_points = self.default_roi_points.copy()
        self.desired_roi_points = np.float32([
            [50, 0],      # Top-left
            [350, 0],     # Top-right
            [350, 300],   # Bottom-right
            [50, 300]     # Bottom-left
        ])

        self.XM_PER_PIX = 2 / 800
        self.YM_PER_PIX = 10 / 600
        self.transformation_matrix = cv2.getPerspectiveTransform(self.roi_points, self.desired_roi_points)
        self.inv_transformation_matrix = cv2.getPerspectiveTransform(self.desired_roi_points, self.roi_points)
        self.roi_image = None
        self.prev_left_fit = None
        self.prev_right_fit = None
        self.debug = True
        self.ground_truth_box = [100, 0, 300, 300]

        # ROI adjustment parameters
        self.roi_curve_threshold = 600 # Curve radius threshold for adjustment
        self.roi_max_widening = 100   # Maximum widening in pixels
        self.roi_top_scale = 4       # Scale factor for top width adjustment
        self.roi_bottom_scale = 0.5  # Scale factor for bottom width adjustment
        self.roi_height_scale = 0.3  # Scale factor for height adjustment
    
```

```

def adjust_roi_for_turn(self, curve_radius=500):
    # Reset to default ROI
    target_roi = self.default_roi_points.copy()

    # Calculate curve severity - more aggressive scaling
    curve_severity = 0
    if curve_radius < 600:
        curve_severity = (600 - curve_radius) / 400 # Steeper scaling
        curve_severity = min(curve_severity, 1.0) # Cap at 1.0
        curve_severity = max(curve_severity, 0.1) # Ensure minimum widening for any curve

    # More aggressive base widening factor - up to 120 pixels for sharp curves
    widening = curve_severity * 120

    # Apply widening to ROI points
    target_roi[0, 0] -= widening # Top-left x
    target_roi[1, 0] += widening # Top-right x

    # Also adjust bottom points
    target_roi[2, 0] += widening * 0.5 # Bottom-right x
    target_roi[3, 0] -= widening * 0.5 # Bottom-left x

    # Ensure the points stay within the frame
    target_roi[:, 0] = np.clip(target_roi[:, 0], 0, 400) # Frame width
    target_roi[:, 1] = np.clip(target_roi[:, 1], 0, 300) # Frame height

    # Less smoothing for more immediate response
    if hasattr(self, 'prev_roi_points') and self.prev_roi_points is not None:
        self.roi_points = self.roi_points * 0.4 + target_roi * 0.6 # More weight on new values
    else:
        self.roi_points = target_roi

    self.prev_roi_points = self.roi_points.copy()

    # Update transformation matrices
    self.transformation_matrix = cv2.getPerspectiveTransform(self.roi_points, self.desired_roi_points)
    self.inv_transformation_matrix = cv2.getPerspectiveTransform(self.desired_roi_points, self.roi_points)

def process_frame(self, frame, curve_radius=500):
    self.orig_frame = frame

    # Adjust ROI based on turn
    self.adjust_roi_for_turn(curve_radius) # Adding default steering_value of 0

    # HSV green detection
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
    lower_green = np.array([40, 50, 50])
    upper_green = np.array([180, 255, 255])
    green_mask = cv2.inRange(hsv, lower_green, upper_green)
    self.lane_line_markings = green_mask

    if self.debug:
        cv2.imshow("Green Mask", self.lane_line_markings)
        debug_frame = frame.copy()
        cv2.polyline(debug_frame, [np.int32(self.default_roi_points)], isClosed=True, color=(0, 255, 0), thickness=2)
        cv2.polyline(debug_frame, [np.int32(self.roi_points)], isClosed=True, color=(0, 0, 255), thickness=2)
        cv2.imshow("ROI Adjustment", debug_frame)

    # Warp
    self.warped_frame = cv2.warpPerspective(
        self.lane_line_markings,
        self.transformation_matrix,
        (400, 300),
        flags=cv2.INTER_LINEAR
    )

    self.calculate_histogram(plot=False)
    try:
        left_fit, right_fit = self.get_lane_line_indices_sliding_windows(plot=False)
        if self.prev_left_fit is not None:
            left_fit = 0.8 * self.prev_left_fit + 0.2 * left_fit
            right_fit = 0.8 * self.prev_right_fit + 0.2 * right_fit
        self.prev_left_fit = left_fit
        self.prev_right_fit = right_fit
        self.get_lane_line_previous_window(left_fit, right_fit, plot=False)
        self.ploty = np.linspace(0, self.warped_frame.shape[0] - 1, self.warped_frame.shape[0])
        self.left_fitx = np.clip(left_fit[0] * self.ploty**2 + left_fit[1] * self.ploty + left_fit[2], 50, 350)
        self.right_fitx = np.clip(right_fit[0] * self.ploty**2 + right_fit[1] * self.ploty + right_fit[2], 50, 350)
        self.curve_radius = self.calculate_curvature()
        self.calculate_car_position()
        result = self.overlay_lane_lines(plot=False)
        result = self.display_curvature_offset(result, plot=False)
    
```

```

detected_box = self.get_detected_lane_box()
iou_score = self.intersection_over_union(detected_box, self.ground_truth_box)
if self.debug:
    print(f"IoU Score: {iou_score:.3f}")

except Exception as e:
    print(f"Lane detection failed: {e}")
    result = frame.copy()

self.roi_image = frame.copy()
cv2.polylines(self.roi_image, [np.int32(self.roi_points)], isClosed=True, color=(0, 0, 255), thickness=2)
return result

def get_detected_lane_box(self):
    """Convert detected lane lines to a bounding box [x_min, y_min, x_max, y_max]."""
    if hasattr(self, 'left_fitx') and hasattr(self, 'right_fitx'):
        x_min = int(np.min(self.left_fitx))
        x_max = int(np.max(self.right_fitx))
        y_min = 0 # Top of warped frame
        y_max = self.warped_frame.shape[0] - 1 # Bottom of warped frame
        return [x_min, y_min, x_max, y_max]
    return [0, 0, 0] # Default if no detection

def intersection_over_union(self, boxA, boxB):
    # Compute the intersection area
    xA = max(boxA[0], boxB[0])
    yA = max(boxA[1], boxB[1])
    xB = min(boxA[2], boxB[2])
    yB = min(boxA[3], boxB[3])

    interArea = max(0, xB - xA + 1) * max(0, yB - yA + 1)

    # Compute the area of both rectangles
    boxAArea = (boxA[2] - boxA[0] + 1) * (boxA[3] - boxA[1] + 1)
    boxBArea = (boxB[2] - boxB[0] + 1) * (boxB[3] - boxB[1] + 1)

    # Compute the intersection over union
    iou = interArea / float(boxAArea + boxBArea - interArea)
    return iou

def print_log(self, print_to_terminal=False, interval=1):
    current_time = time.time()
    if current_time - self.last_print_time < interval:
        return

    try:
        if hasattr(self, "curve_radius") and hasattr(self, "center_offset"):
            self.last_print_time = current_time
            print_curves = self.calculate_curvature()
            car_offset = self.calculate_car_position()
            detected_box = self.get_detected_lane_box()
            iou_score = self.intersection_over_union(detected_box, self.ground_truth_box)
            return f'Offset: {car_offset:.1f} cm, Radius: {print_curves:.1f} m, IoU: {iou_score:.3f}'
        return "No Lane Detected"
    except Exception as e:
        print(f"Error: {e}")
        return "Error: No Lane Detected"

# Rest of your code (real_time_lane_detection) remains unchanged

def real_time_lane_detection():
    sct = mss()
    monitor = {"top": 300, "left": 100, "width": 400, "height": 300} # 800x600 window
    lane_detector = Debugging((600, 800, 3))

    while True:
        screenshot = sct.grab(monitor)
        frame = np.array(screenshot)
        frame = cv2.cvtColor(frame, cv2.COLOR_BGRA2BGR)

        processed_frame = lane_detector.process_frame(frame)
        debug_log = lane_detector.print_log(True)
        if debug_log:
            print(debug_log)

        cv2.imshow("Lane Detection", processed_frame)
        cv2.imshow("ROI Image", lane_detector.roi_image)
        cv2.imshow("Warped Frame", lane_detector.warped_frame)
        # print(f'Offset: {lane_detector.center_offset:.1f} cm, Radius: {lane_detector.curve_radius:.1f} m')

        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    cv2.destroyAllWindows()

if __name__ == "__main__":
    real_time_lane_detection()

```

## Lampiran 8 Kelas Algoritma Mengemudi Real-Time

Kelas ini merupakan keseluruhan dari simulasi yang menggabungkan semua algoritma seperti *processing*, *lane detection* dan menggunakan informasi yang berhasil diekstrak untuk mengemudikan mobil secara otomatis.

```

import cv2
import numpy as np
import mss
import time
import pydirectinput
import pyautogui
import threading
import queue
import cProfile
import pstats
import io

class LaneRealTime(Lane):
    def __init__(self, frame_shape):
        dummy_frame = np.zeros(frame_shape, dtype=np.uint8)
        super().__init__(dummy_frame)
        # Adjusted ROI points based on the screenshot (400x300 resolution)
        self.roi_points = np.float32([
            (137, 127), # Top-left
            (258, 127), # Top-right
            (374, 156), # Bottom-right
            (6, 156) # Bottom-left
        ])
        self.desired_roi_points = np.float32([
            [50, 0], # Top-left
            [350, 0], # Top-right
            [350, 300], # Bottom-right
            [50, 300] # Bottom-left
        ])
        self.transformation_matrix = cv2.getPerspectiveTransform(self.roi_points, self.desired_roi_points)
        self.inv_transformation_matrix = cv2.getPerspectiveTransform(self.desired_roi_points, self.roi_points)
        self.roi_image = None
        self.frame_count = 0
        self.skip_frames = 4
        self.processing_scale = 0.5

        # Profiler setup
        self.profiler = cProfile.Profile()
        self.stats_output = io.StringIO()

    def profile_stats(self):
        self.profiler.disable() # Ensure profiler is disabled before collecting stats
        stats = pstats.Stats(self.profiler, stream=self.stats_output)
        stats.sort_stats('cumulative')
        stats.print_stats(20) # Top 20 functions by cumulative time
        return self.stats_output.getvalue()

```

```

def process_frame(self, frame):
    self.frame_count += 1
    if self.frame_count % self.skip_frames != 0:
        return None, None, None

    # Enable profiler for this method
    self.profiler.enable()

    start_time = time.time()
    self.orig_frame = frame
    small_frame = cv2.resize(frame, None,
                           fx=self.processing_scale,
                           fy=self.processing_scale,
                           interpolation=cv2.INTER_AREA)

    hsv = cv2.cvtColor(small_frame, cv2.COLOR_BGR2HSV)
    lower_green = np.array([40, 50, 50])
    upper_green = np.array([80, 255, 255])
    green_mask = cv2.inRange(hsv, lower_green, upper_green)

    green_mask = cv2.resize(green_mask,
                           (frame.shape[1], frame.shape[0]),
                           interpolation=cv2.INTER_NEAREST)

    self.lane_line_markings = green_mask

    self.warped_frame = cv2.warpPerspective(
        self.lane_line_markings,
        self.transformation_matrix,
        (400, 300),
        flags=cv2.INTER_LINEAR
    )

    try:
        self.calculate_histogram(plot=False)
        left_fit, right_fit = self.get_lane_line_indices_sliding_windows(plot=False)
        self.get_lane_line_previous_window(left_fit, right_fit, plot=False)

        self.ploty = np.linspace(0, self.warped_frame.shape[0] - 1, self.warped_frame.shape[0])
        self.left_fixt = np.clip(left_fit[0] * self.ploty**2 + left_fit[1] * self.ploty + left_fit[2], 50, 350)
        self.right_fixt = np.clip(right_fit[0] * self.ploty**2 + right_fit[1] * self.ploty + right_fit[2], 50, 350)

        self.calculate_curvature()
        center_offset = self.calculate_car_position()
        curve_radius = (self.left_curvem + self.right_curvem) / 2
    
```

```

        result = self.overlay_lane_lines(plot=False, left_color=(128, 0, 128), right_color=(128, 0, 128), center_color=(0, 0, 255))
        result = self.display_curvature_offset(result, plot=False)
    except Exception as e:
        print(f"Lane detection failed: {e}")
        self.profiler.disable()
        return None, None, frame.copy()

    self.roi_image = frame.copy()
    cv2.polylines(self.roi_image, [np.int32(self.roi_points)], isClosed=True, color=(0, 0, 255), thickness=2)

    print(f"Detection time: {time.time() - start_time:.3f}s")
    self.profiler.disable() # Disable profiler after method completes
    return center_offset, curve_radius, result

```

```
class AutoDriver:  
    def __init__(self, screen_region):  
        self.screen_region = screen_region  
        self.steering_sensitivity = 1.0  
        self.speed_control = 0.02  
        self.center_threshold = 15  
        self.is_running = False  
        self.current_speed = 0  
        self.current_steering = 0  
        self.last_control_time = time.time()  
        self.control_delay = 0.05  
        self.keys = {'forward': 'w', 'left': 'a', 'right': 'd', 'brake': 'space'}  
        self.debug = True  
        self.last_center_offset = 0  
        self.last_steering = 0  
        self.failure_count = 0  
        self.frame_count = 0 # For FPS calculation  
        self.lane_detector = LaneRealTime((300, 400, 3))  
        self.frame_queue = queue.Queue(maxsize=5)  
        self.result_queue = queue.Queue(maxsize=5)  
  
        # Profiler setup  
        self.capture_profiler = cProfile.Profile()  
        self.detection_profiler = cProfile.Profile()  
        self.drive_profiler = cProfile.Profile()  
        self.stats_output = io.StringIO() # Initialize stats output  
  
    def print_profiling_stats(self, profiler):  
        stats = pstats.Stats(profiler, stream=self.stats_output)  
        stats.sort_stats('cumulative')  
        stats.print_stats(20)  
        return self.stats_output.getvalue()  
  
    def capture_screen(self):  
        try:  
            with mss.mss() as sct:  
                sct_img = sct.grab(self.screen_region)  
                return cv2.cvtColor(np.array(sct_img), cv2.COLOR_BGRA2BGR)  
        except Exception as e:  
            print(f"Screen capture error: {e}")  
            return np.zeros((300, 400, 3), dtype=np.uint8)
```

```

def capture_loop(self):
    while self.is_running:
        self.capture_profiler.enable()
        start_time = time.time()
        frame = self.capture_screen()
        print(f"Capture time: {time.time() - start_time:.3f}s")
        try:
            self.frame_queue.put_nowait(frame)
        except queue.Full:
            pass
    self.capture_profiler.disable()
    time.sleep(0.03)

def detection_loop(self):
    while self.is_running:
        self.detection_profiler.enable()
        start_time = time.time()
        try:
            frame = self.frame_queue.get(timeout=0.1)
        except queue.Empty:
            self.detection_profiler.disable()
            continue
        center_offset, curve_radius, processed_frame = self.lane_detector.process_frame(frame)
        print(f"Detection time: {time.time() - start_time:.3f}s")
        if center_offset is not None and curve_radius is not None:
            try:
                self.result_queue.put_nowait((center_offset, curve_radius, processed_frame))
            except queue.Full:
                pass
    self.frame_queue.task_done()
    self.detection_profiler.disable()

```

```

def control_steering(self, center_offset, curve_radius):
    raw_steering = center_offset / 100.0 * self.steering_sensitivity
    steering_value = 0.7 * self.last_steering + 0.3 * raw_steering
    self.last_steering = steering_value
    if abs(center_offset) < self.center_threshold:
        steering_value *= 0.4
    if curve_radius < 500:
        curve_factor = min(0.5, min(1.0, 300 / max(curve_radius, 50)) ** 2)
        steering_value += curve_factor * 0.3 * (-1 if center_offset < 0 else 1)
    return max(-1.0, min(1.0, steering_value))

def control_speed(self, curve_radius, center_offset):
    speed_value = self.speed_control
    if self.failure_count > 5:
        speed_value *= 0.5
    if curve_radius < 500:
        speed_factor = min(1.0, curve_radius / 500) ** 2
        speed_value *= speed_factor
    if abs(center_offset) > 50:
        speed_value *= 0.8
    return max(0.005, speed_value)

def apply_controls(self, steering_value, speed_value):
    if time.time() - self.last_control_time < self.control_delay:
        return
    self.last_control_time = time.time()

    apply_start = time.time()

    # Reset steering keys only if needed
    steering_reset_start = time.time()
    if self.current_steering != 0:
        pydirectinput.keyUp(self.keys['left'])
        pydirectinput.keyUp(self.keys['right'])
    print(f"Steering reset time: {time.time() - steering_reset_start:.3f}s")

```

```

# Steering with minimal sleep
steering_start = time.time()
if abs(steering_value - self.current_steering) > 0.05:
    if steering_value < -0.1:
        pydirectinput.keyDown(self.keys['left'])
        time.sleep(0.005)
        pydirectinput.keyUp(self.keys['left'])
        self.current_steering = steering_value
    elif steering_value > 0.1:
        pydirectinput.keyDown(self.keys['right'])
        time.sleep(0.005)
        pydirectinput.keyUp(self.keys['right'])
        self.current_steering = steering_value
    else:
        self.current_steering = 0
print(f"Steering apply time: {time.time() - steering_start:.3f}s")

# Speed control with pulsing - Modified for slower speed
speed_start = time.time()
if self.failure_count > 5:
    if self.current_speed != 0:
        pydirectinput.keyUp(self.keys['forward'])
        pydirectinput.keyDown(self.keys['brake'])
        time.sleep(0.001)
        pydirectinput.keyUp(self.keys['brake'])
    self.current_speed = 0
else:
    # Reduced press time and increased release time for slower speed
    press_time = speed_value * 0.01 # Max 2.5ms press (reduced from 5ms)
    release_time = (1 - speed_value) * 0.2 # Max 190ms release (increased from 95ms)

    # Add speed limiting for curves and steering
    if abs(steering_value) > 0.3: # Reduce speed in sharp turns
        press_time *= 0.5
        release_time *= 1.5

    pydirectinput.keyDown(self.keys['forward'])
    time.sleep(press_time)
    pydirectinput.keyUp(self.keys['forward'])
    time.sleep(release_time)
    self.current_speed = speed_value
print(f"Speed apply time: {time.time() - speed_start:.3f}s")
print(f"Total apply controls time: {time.time() - apply_start:.3f}s")

```

```

def reset_controls(self):
    # Only called on stop, not every frame
    for key in self.keys.values():
        pydirectinput.keyUp(key)

def drive_loop(self):
    self.frame_count = 0
    start_time = time.time()
    while self.is_running:
        self.drive_profiler.enable()

        control_start = time.time()
        try:
            center_offset, curve_radius, processed_frame = self.result_queue.get(timeout=0.1)
            self.last_center_offset = center_offset
            self.failure_count = 0
        except queue.Empty:
            center_offset = self.last_center_offset
            curve_radius = 500
            processed_frame = self.capture_screen()
            self.failure_count += 1
            self.drive_profiler.disable()
            continue

        steering_value = self.control_steering(center_offset, curve_radius)
        speed_value = self.control_speed(curve_radius, center_offset)
        apply_start = time.time()
        self.apply_controls(steering_value, speed_value)
        print(f"Apply controls time: {time.time() - apply_start:.3f}s")
        print(f"Total control time: {time.time() - control_start:.3f}s")

        display_start = time.time()
        if self.debug and self.frame_count % 20 == 0:
            cv2.putText(processed_frame, f"Steering: {steering_value:.2f}", (10, 30),
                       cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255, 255, 255), 2)
            cv2.putText(processed_frame, f"Speed: {speed_value:.2f}", (10, 60),
                       cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255, 255, 255), 2)
            cv2.putText(processed_frame, f"Offset: {center_offset:.1f} cm", (10, 90),
                       cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255, 255, 255), 2)
            cv2.putText(processed_frame, f"Curve: {curve_radius:.1f} m", (10, 120),
                       cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255, 255, 255), 2)
            cv2.imshow("Lane Detection", processed_frame)
            cv2.waitKey(1)
        print(f"Display time: {time.time() - display_start:.3f}s")
        # print(f"Distance covered :{self.lane_detector.distance_covered:.2f} m")
    
```

```

        self.frame_count += 1
        if self.frame_count % 10 == 0:
            fps = 10 / (time.time() - start_time)
            print(f"FPS: {fps:.2f}, Steering: {steering_value:.2f}, Speed: {speed_value:.2f}")
            start_time = time.time()

        if cv2.waitKey(1) & 0xFF == ord('q'):
            self.stop()
            break
        self.result_queue.task_done()

        self.drive_profiler.disable()

    def start(self):
        if not self.is_running:
            self.is_running = True
            self.capture_thread = threading.Thread(target=self.capture_loop)
            self.capture_thread.daemon = True
            self.capture_thread.start()
            self.detection_thread = threading.Thread(target=self.detection_loop)
            self.detection_thread.daemon = True
            self.detection_thread.start()
            print("Autonomous driving started")
            self.drive_loop()

    def stop(self):
        if self.is_running:
            self.is_running = False
            print("\n==== Capture Profiling ===")
            print(self.print_profiling_stats(self.capture_profiler))
            print("\n==== Detection Profiling ===")
            print(self.print_profiling_stats(self.detection_profiler))
            print("\n==== Drive Loop Profiling ===")
            print(self.print_profiling_stats(self.drive_profiler))
            print("\n==== Lane Detection Profiling ===")
            print(self.lane_detector.profile_stats())
            self.reset_controls()
            if self.capture_thread:
                self.capture_thread.join(timeout=1.0)
            if self.detection_thread:
                self.detection_thread.join(timeout=1.0)
            cv2.destroyAllWindows()
            print("Autonomous driving stopped")

```

```

def main():
    screen_region = {"top": 300, "left": 100, "width": 400, "height": 300}
    driver = AutoDriver(screen_region)
    print("Starting in 3 seconds...")
    for i in range(3, 0, -1):
        print(f"{i}...")
        time.sleep(1)
    driver.start()
    try:
        while driver.is_running:
            time.sleep(0.1)
    except KeyboardInterrupt:
        driver.stop()

if __name__ == "__main__":
    main()

```