Analisis Perbandingan Performa dan Visual *Ray Casting* dengan Operasi Vektor Menggunakan DDA dan Tanpa DDA

Fajar Kurniawan, and 13523027[1,2]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13523027@mahasiswa.itb.ac.id, 13523027@std.stei.itb.ac.id

***Abstrak*—*Ray casting* merupakan metode *rendering* yang memanfaatkan pancaran sinar virtual dari pengamat ke objek untuk mengukur jarak dari pengamat ke objek sehingga bisa mentransformasikan gambar 2D menjadi ruangan 3D melalui perbedaan ukuran benda yang ditampilkan. Algortima *ray casting* menggunakan komputasi yang banyak dan berat sehingga memerlukan optimisasi. Salah satu optimisasi yang dapat digunakan adalah algoritma DDA. Pada makalah ini akan dianalisis perbedaan dari *ray casting* dengan DDA dan tanpa DDA baik dari segi visual maupun performa.**

***Kata Kunci*—Ray Casting, DDA, vektor, FPS, Optimisasi, rendering, proyeksi.**

I. PENDAHULUAN

Ray casting adalah istilah yang pertama kali digunakan oleh Scott Roth pada makalah berjudul "Ray Casting for Modeling Solids". *Ray casting* merupakan sebuah metode *rendering* yang digunakan untuk menampilkan visual 3D dari sebuah gambar 2D. Fakta itulah yang membuat beberapa orang menyebut visual yang dihasilkan sebagai pseudo-3D, bukan 3D yang sebenarnya.

Implementasi *ray casting* memanfaatkan vektor di ruang Euclidean dengan melakukan berbagai operasi pada vektor *ray* atau sinar yang digunakan dalam *rendering*. Operasi yang dilakukan bisa menjadi komputasi yang berat sehingga performa *rendering* menjadi tidak optimal. Performa yang buruk akan sangat mengganggu jika ray casting digunakan dalam sebuah gim video.

Salah satu algoritma yang dapat mengoptimasi komputasi yang berat pada *ray casting* adalah DDA. Makalah ini akan membahas implementasi DDA pada *ray casting* dan membandingkan performa *ray casting* dengan DDA dan tanpa DDA melalui FPS (*Frames Per Second*) yang dihasilkan dan kompleksitas waktu yang diperlukan.

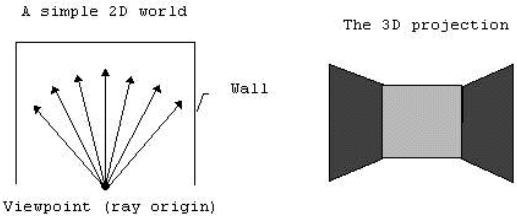
II. LANDASAN TEORI

A. Ray Casting

Ray casting adalah sebuah metode rendering yang memanfaatkan sebuah garis *ray* (cahaya) virtual yang dipancarkan dipancarkan dari sudut pandang sebuah mata atau kamera virtual sampai sinar tersebut menabrak sebuah objek di dalam sebuah ruangan virtual. Sinar tersebut merepresentasikan pancaran cahaya dari sebuah objek ke mata atau kamera penonton [1].

Sudut *ray* (sinar) yang dipancarkan dihitung berdasarkan *field-of-view* (FOV) penonton, yaitu ukuran sudut pandang dari sebuah kamera virtual yang menunjukkan seberapa luas citra yang dapat dilihat oleh penonton.

Gambar berikut menunjukkan hasil transformasi dari gambar 2D yang disimulasikan menjadi ruangan 3D Degnan *ray casting.* Semakin jauh jarak dari penonton ke suatu titik, maka semakin panjang sinar yang dipancarkan. Perbedaan panjang sinar tersebut akan ditransformasikan menjadi perbedaan tinggi dinding yang di-*render* sehingga memberikan efek ruangan 3D karena semakin jauh jarak suatu objek maka akan terlihat semakin kecil.



Gambar 1. Hasil Rendering dengan Ray Casting, diambil dari [2]

B. Digital Differential Analyzer

Digital Differential Analyzer (DDA) merupakan suatu algoritma untuk membentuk suatu garis berdasarkan nilai dx dan dy, dengan rumus dy = m × dx. Garis dibentuk dengan menentukan dua buah titik, yaitu titik awal (x0,y0) dan titik akhir (x1,y1). Koordinat dari kedua titik tersebut kemudian dikonversi menjadi bilangan bulat [3].

Selanjutnya, dihitung dy = y1-y0 dan dx = x1-x0. Kemudian ditentukan sebuah *step* (langkah), jika nilai mutlak dx lebih besar dari nilai mutlak dy, maka *step*=|dx|, jika sebaliknya, maka *step*=|dy|. Lalu, dihitung nilai penambahan untuk setiap iterasi, yaitu x_inc = dx/*step* dan y_inc = dy/*step*. Kemudian diiterasikan operasi penambahan nilai x dengan x_inc dan y dengan y_inc hingga nilai x = x1 dan y = y1.

C. Vektor di Ruang Euclidean

Vektor merupakan suatu besaran yang memerlukan dua

komponen untuk bisa eksis dan direpresentasikan, yaitu panjang dan arah, berbeda dari besaran skalar yang tidak memerlukan komponen arah. Contoh dari besaran skalar adalah temperatur 20 derajat Celsius atau panjang 5cm. Sedangkan contoh besaran vektor adalah sebuah gaya sebesar 100 Newton ke arah bawah [4].

Engineer dan ilmuwan merepresentasikan vektor di 2 dan 3 dimensi sebagai panah dengan kepala panah menunjukkan arah vektor dan panjang panah sebagai magnitudo atau besarnya vektor tersebut. Karena vektor hanya memiliki dua komponen yaitu panjang dan arah, maka dua vektor yang mempunyai panjang dan arah yang sama dikatakan sebagai vektor yang ekuivalen dan sama.

Vektor yang memiliki magnitudo nol disebut sebagai vektor nol. Vektor nol secara alami tidak memiliki arah sehingga bisa diberikan arah sesuai kebutuhan atau persamaan yang ingin diselesaikan.

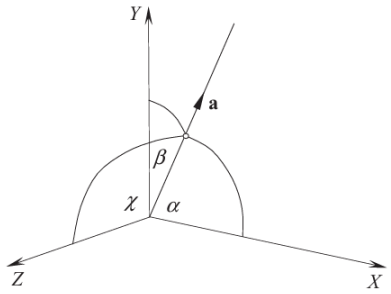
C1. Aljabar Vektor

Vektor 3D yang direpresentasikan dengan **v** [a, b, c] memiliki magnitudo sebagai berikut [4].

$$\|\mathbf{a}\| = \sqrt{x_a^2 + y_a^2 + z_a^2}.$$

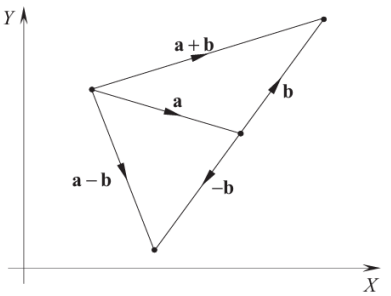
Vektor memiliki arah atau orientasi yang dapat direpresentasikan dengan cosinus dari sudutnya dengan sumbu kartesian.

$$\cos\alpha = \frac{x_a}{\|\mathbf{a}\|} \quad \cos\beta = \frac{y_a}{\|\mathbf{a}\|} \quad \cos\chi = \frac{z_a}{\|\mathbf{a}\|}.$$



Gambar 2. Orientasi Vektor, diambil dari [5]

Penjumlahan dan pengurangan vektor dapat dijelaskan secara singkat dan sederhana dengan ilustrasi berikut.



Gambar 3. Penjumlahan dan Pengurangan Vektor, diambil dari [5]

Penjumlahan vektor juga dapat dilihat sebagai transformasi, lebih spesifiknya translasi sebuah vektor [4]. Jika dua buah vektor **v** dan **w** memiliki pangkal yang berimpitan, maka **v** + **w** dapat dilihat dari dua sudut pandang, yaitu

1. Titik ujung vektor **v**+**w** adalah titik ujung **v** yang ditranslasikan ke arah **w** sejauh magnitudo **w**.
2. Titik ujung vektor **v**+**w** adalah titik ujung **w** yang ditranslasikan ke arah **v** sejauh magnitudo **v**.

Sehingga, **v**+**w** dapat dianggap sebagai **v** ditranslasikan oleh **w** atau sebaliknya.

Jika **v** = (v_1, v_2, …, v_n) dan **w** = (w_1, w_2, …, w_n) adalah vektor di R^n dan k adalah skalar, maka berlaku:

w + **v** = ($w_1 + v_1$, $w_2 + v_2$, …, $w_n + v_n$)
k**w** = (kw_1, kw_2, …, kw_n)
-**v** = ($-v_1$, $-v_2$, …, $-v_n$)
w - **v** = **w** + (-**v**) = (w_1-v_1, w_2-v_2, …, w_n-v_n)

C2. Vektor Satuan dan Proyeksi Ortogonal

Vektor satuan adalah sebuah vektor yang memiliki panjang (magnitudo) satu satuan. Vektor ini berguna ketika kita hanya memerlukan arah tanpa memedulikan magnitudo vektor. Vektor satuan diperoleh dengan membagi sebuah vektor dengan magnitudonya.

Jika **u** dan **v** adalah vektor tidak nol di dua atau tiga dimensi dan θ adalah sudut di antara keduanya, maka *dot product* vektor **u** dan **v** adalah

$$\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\|\|\mathbf{v}\|\cos\theta$$

Rumus tersebut juga dapat dituliskan sebagai

$$\mathbf{u} \cdot \mathbf{v} = u_1v_1 + u_2v_2 + u_3v_3$$

Rumus *dot product* vektor dan rumus vektor satuan dapat diturunkan menjadi rumus proyeksi orthogonal. Magnitudo proyeksi ortogonal sebuah vektor **u** ke vektor **a** adalah k, dapat diperoleh dari

$$k = \frac{\mathbf{u} \cdot \mathbf{a}}{\|\mathbf{a}\|^2}$$

Kemudian, vektor proyeksi ortogonalnya dapat diperoleh dengan mengalikan k dengan vektor satuan dari vektor **a.**

III. PEMBAHASAN

A. Program Ray Casting

Program untuk simulasi *ray casting* menggunakan *library* pygame untuk bisa menggambar objek-objek yang diperlukan. Berikut merupakan beberapa konstanta untuk simulasi *ray casting* yang digunakan untuk makalah ini.

```
SCREEN_WIDTH = 1200
SCREEN_HEIGHT = 600
FOV = math.pi / 3
HALF_FOV = FOV / 2
RAY_COUNT = 600
MAX_RAY_LENGTH = 600
CELL_SIZE = 64
```

Kemudian berikut merupakan *map* atau peta 2D yang digunakan. Angka 0 melambangkan ruang kosong tanpa *wall* atau dinding sedangkan angka 1-4 merepresentasikan dinding warna-warna berbeda.

```
[[3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 1],
 [3, 0, 0, 0, 3, 0, 4, 1, 0, 0, 0, 1],
 [3, 0, 3, 0, 0, 0, 0, 0, 0, 4, 0, 1],
 [3, 0, 0, 0, 0, 0, 0, 0, 1, 0, 3, 1],
 [3, 2, 0, 0, 0, 2, 0, 0, 0, 0, 0, 1],
 [3, 2, 0, 0, 0, 0, 0, 0, 3, 1, 0, 1],
 [3, 0, 0, 0, 0, 4, 0, 0, 3, 0, 0, 1],
 [3, 0, 3, 0, 0, 0, 0, 1, 0, 0, 0, 1],
 [3, 0, 4, 4, 0, 0, 0, 0, 0, 4, 0, 1],
 [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1]]
```

Berikut merupakan posisi awal pemain atau kamera virtual atau istilah lain yang merepresentasikan penonton dari hasil *ray cast.* Meski tidak secara eksplisit dituliskan sebagai vektor, tetapi keduanya merupakan komponen sumbu-x dan komponen sumbu-y dari vektor posisi pemain. CELL_SIZE adalah 64 sehingga, vektor Lokasi awal pemain adalah vektor (160, 272)

```
player_x = CELL_SIZE * 2.5
player_y = CELL_SIZE * 4.25
```

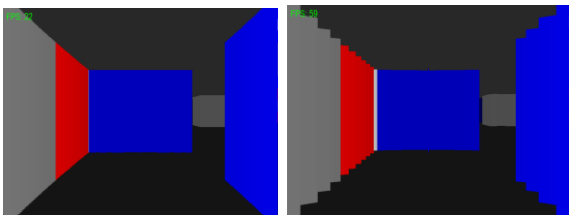
Ray pada *ray casting* sejatinya adalah vektor yang mempunyai dua komponen, yaitu magnitudo atau besar adalah *distance* (jarak pemain ke dinding) dan arah yaitu sudut *ray.* Berikut merupakan algoritma untuk menembakkan *ray* dari pemain hingga mengenai dinding tanpa DDA.

```
def cast_ray(angle):
    ray_x = player_x
    ray_y = player_y
    ray_cos = math.cos(angle)
    ray_sin = math.sin(angle)
    step_size = 1
    distance = 0

    while distance < MAX_RAY_LENGTH:
        ray_x += ray_cos * step_size
        ray_y += ray_sin * step_size
        distance += step_size

        map_x = int(ray_x / CELL_SIZE)
        map_y = int(ray_y / CELL_SIZE)

        if map_x < 0 or map_x >= MAP_WIDTH or
map_y < 0 or map_y >= MAP_HEIGHT:
            break

        if MAP[map_y][map_x] > 0:
            return distance, ray_x, ray_y,
MAP[map_y][map_x]

    return MAX_RAY_LENGTH, ray_x, ray_y, 0
```

Terdapat dua vektor di sini, yang pertama vektor posisi *ray*, yang kedua adalah vektor *ray* itu sendiri. Komponen sumbu x dan sumbu y vektor *ray* direpresentasikan oleh variable ray_x dan ray_y. Keduanya merupakan proyeksi vektor ke sumbu x dan sumbu y.

Algoritma ini sangat sederhana tetapi memerlukan komputasi yang berat karena harus melakukan transformasi dengan jumlah relatif banyak. Misal **r** adalah vektor *ray* yang baru, **r0** vektor *ray* lama, dan **d**=(cos(angle),sin(angle)) (dengan angle = sudut **r0**) maka transformasi yang dilakukan adalah **r** = **r0** + step_size × n × **d**, dengan n jumlah step (belum diketahui). Nilai sin dan cos dari sudut **r0** tidak akan lebih dari 1 dan nilai step adalah 1, sehingga jika pada kemungkinan terburuk, yaitu *ray* tidak menabrak dinding, maka akan dilakukan transformasi terus menerus sampai panjang vektor **r** menjadi MAX_RAY_LENGTH yaitu 600. Kemudian harus dilakukan komputasi yang sama untuk semua *ray* yang ada sejumlah RAY_COUNT yaitu 600. Maka pada kasus terburuk akan dilakukan sejumlah MAX_RAY_LENGTH × RAY_COUNT transformasi.

Masalah ini bisa diatasi dengan menambah nilai variable step_size sehingga akan lebih sedikit transformasi yang diperlukan hingga *ray* menabrak dinding, akan tetapi kualitas visual yang dihasilkan akan menjadi sangat buruk. Berikut perbandingan antara simulasi dengan step_size = 1 dan step_size = 10.



Gambar 4. Ray Casting dengan step_size = 1 (kiri) dan step_size = 10 (kanan)

Maka dari itu, mengubah nilai step_size bukanlah solusi, karena visual yang baik juga merupakan hal yang penting. Diperlukan solusi lain untuk optimasi.

B. Ray Casting dengan DDA

Algoritma *Digital Differential Analyzer* dapat mengoptimasi *ray casting* karena pada algoritma *ray casting* yang dibuat berdasarkan DDA, vektor *ray* tidak perlu maju sedikit demi sedikit. Meski begitu, algoritma ini tetap menggunakan *step*, hanya saja ukuran *step*-nya sama dengan panjang sel sehingga langkah yang diperlukan hingga menabrak dinding akan jauh lebih sedikit.

Algoritma *ray casting* dengan DDA dimulai dengan mencari jarak dari koordinat tempat muncul atau dipancarkannya *ray*, yaitu koordinat pemain, menuju sisi sel tempat ia berada. Setelah itu, *ray* akan maju satu sel demi satu sel hingga mencapai sel dinding. Berikut kode baru yang sudah mengimplementasikan DDA berdasarkan referensi di internet [6].

```
def cast_ray(angle):
    ray_x = math.cos(angle)
    ray_y = math.sin(angle)

    map_x = int(player_x / CELL_SIZE)
    map_y = int(player_y / CELL_SIZE)

    wall = 0

    dx = abs(1 / ray_x) if ray_x != 0 else
float('inf')
```

```
    dy = abs(1 / ray_y) if ray_y != 0 else
float('inf')

    if ray_x < 0:
        step_x = -1
        near_x = ((player_x / CELL_SIZE) -
map_x) * dx
    else:
        step_x = 1
        near_x = (map_x + 1.0 - (player_x /
CELL_SIZE)) * dx

    if ray_y < 0:
        step_y = -1
        near_y = ((player_y / CELL_SIZE) -
map_y) * dy
    else:
        step_y = 1
        near_y = (map_y + 1.0 - (player_y /
CELL_SIZE)) * dy

    hit = False
    side = 0  # 0 -> x, 1 -> y
    max_steps = int(MAX_RAY_LENGTH / CELL_SIZE)
    steps = 0

    while not hit and steps < max_steps:
        if near_x < near_y:
            near_x += dx
            map_x += step_x
            side = 0
        else:
            near_y += dy
            map_y += step_y
            side = 1

        steps += 1

        if map_x < 0 or map_x >= MAP_WIDTH or
map_y < 0 or map_y >= MAP_HEIGHT:
            hit = True
        elif MAP[map_y][map_x] > 0:
            hit = True
            wall = MAP[map_y][map_x]

    if hit and wall > 0:
        if side == 0:
            distance = (near_x - dx) * CELL_SIZE
        else:
            distance = (near_y - dy) * CELL_SIZE
    else:
        distance = MAX_RAY_LENGTH

    ray_x = player_x + (distance * ray_x)
    ray_y = player_y + (distance * ray_y)

    return distance, ray_x, ray_y, wall
```

Pada algoritma ini, dihitung dx dan dy yang merepresentasikan seberapa jauh komponen sumbu x dan komponen sumbu y dari vektor *ray* (sebut saja vektor **r**) perlu bergerak untuk untuk menempuh jarak 1 sel. Penjelasan lainnya adalah dx dan dy merupakan faktor pengali ray_x dan ray_y agar masing-masing panjangnya menjadi sama dengan panjang 1 sel.

Kemudian ada percabangan yang akan menentukan nilai step_x dan step_y. Jika ray_x < 0 berarti vektor **r** memiliki arah ke "kiri" peta jika dilihat dari *top-down-view*. Maka dari itu step_x bernilai -1 agar pancaran *ray* seakan-akan bergerak ke sel sebelah kiri yang berarti bergerak pada array 2D peta permainan dengan cara mengurangi indeks elemen yang diakses. Begitu pula dengan percabangan yang lain, jika ray_x > 0 berarti bergerak ke kanan, sehingga indeks pada peta yang diakses akan bertambah (nilai step_x = 1). Jika ray_y < 0 berarti bergerak ke atas sehingga indeks pada peta yang diakses berkurang (step_y = -1). Sedangkan jika ray_y > 0 artinya bergerak ke bawah dan indeks yang diakses pada peta beretambah (step_y = 1).

Pada percabangan, dideklarasikan variable near_x dan near_y, near_x merupakan vektor pada sumbu x dan near_y adalah vektor pada sumbu y. Vektor near_x memiliki panjang yang setara dengan jarak dari sumber *ray* ke sisi kanan atau kiri sel tempat ia berada. Vektor near_y memiliki panjang yang setara dengan jarak dari sumber *ray* ke sisi atas atau bawah sel tempat ia berada.

Selanjutnya dijalankan sebuah *while loop* yang menambah magnitudo vektor near_x dengan dx atau near_y dengan dy secara bergantian karena kita tidak bisa memprediksi vektor mana yang akan berpotongan dengan dinding lebih dahulu. Cara menentukan gilirannya adalah dengan membandingkan panjang vektor near_x dengan near_y agar *progress* keduanya bisa seimbang. Inilah yang menjadi alasan mengapa algoritma ini tetap bisa berjalan jika nilai salah satu komponen vektor *ray* (komponen sumbu x atau sumbu y) bernilai nol. Sebagai contoh jika komponen sumbu x bernilai nol (ray_x = 0) maka dx bernilai tak hingga (*infinite*) sehingga dalam *while loop* ini hanya akan dilakukan penambahan pada near_y. Namun, algoritma ini memiliki kelemahan yang akan dibahas juga.

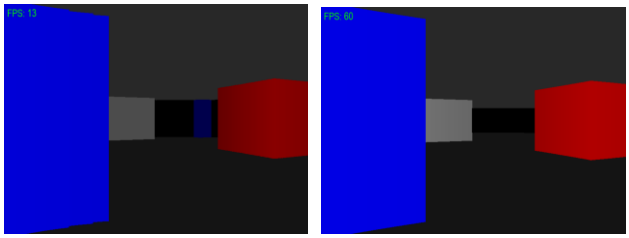
Jika setelah jumlah step maksimum tercapai, program tidak menemukan elemen pada array 2D peta yang nilainya lebih besar dari nol, maka posisi akhir jatuhnya *ray* akan berjarak MAX_RAY_LENGTH dari sumber *ray*. Hal ini berarti magnitudo dari vektor *ray* adalah MAX_RAY_LENGTH.

Jika sebelum max_step dicapai, program berhasil menemukan elemen pada array 2D peta yang nilainya lebih dari nol, maka ini menandakan bahwa sebuah dinding telah ditabrak oleh *ray*, yang berarti vektor *ray* berpotongan dengan dinding. Kemudian ditentukan jarak (*distance*) yaitu dari (near_x – dx) * CELL_SIZE atau (near_y - dy) * CELL_SIZE. Alasan dari pengurangan dengan dx atau dy adalah jika *ray* berhasil menabrak dinding berarti ia sudah melewati sisi dinding sehingga pasti ada suatu kelebihan pada operasi *while loop* sebelumnya sebesar dx atau dy. Sedangkan perkalian dengan CELL_SIZE dilakukan karena nilai near_x atau near_y pada awalnya merupakan nilai koordinat player dibagi dengan CELL_SIZE, sehingga perkalian dilakukan agar nilainya kembali "normal".

Pada algoritma ini, kasus terburuk akan membuat

program harus melakukan transformasi sebanyak max_steps yaitu MAX_RAY_LENGTH / CELL_SIZE untuk setiap *ray*, ini jauh lebih sedikit dibandingkan tanpa DDA yaitu sebanyak MAX_RAY_LENGTH untuk setiap *ray*.

Algoritma *ray casting* dengan DDA ini memang memiliki kelebihan di bagian performa dan kecepatan, tetapi juga memiliki kekurangan di beberapa bagian. Salah satunya adalah dalam me-*render* dinding akan ada kemungkinan seluruh bagian dinding tidak ditabrak oleh *ray*. Ini mungkin terjadi karena nilai dx atau dy yang besar karena nilai ray_x atau ray_y yang kecil disebabkan oleh *ray* pada sudut-sudut tertentu. Kemungkinan selanjutnya adalah pada *while* loop yang ada, ada suatu penjumlahan dengan dx atau dy secara terus menerus sehingga ada dinding yang terlewat karena perjalanan *ray* seakan-akan seperti belok menghindari sel yang mengandung tembok. Pada akhirnya masalah tersebut tetap terkait dengan nilai dx dan dy. Masalah ini dapat dilihat pada gambar berikut. Pada *ray casting* tanpa DDA, terlihat dinding biru di ujung yang ter-*render* secara parsial sehingga memberikan efek transisi dari hitam (tidak ada dinding terlihat) ke dinding biru sedikit demi sedikit. Efek ini tidak ada pada *ray casting* dengan DDA. Terlihat bahwa dinding biru di ujung tidak terlihat sama sekali. Dinding tersebut baru akan di-*render* sepenuhnya setelah player maju tanpa ada efek transisi *render* parsial.



Gambar 5. Ray Castin Tanpa DDA (kiri) dan dengan DDA (kanan)

Pada gambar 5 juga bisa dilihat bahwa efek perbedaan tingkat pencahayaan lebih baik pada *ray casting* tanpa DDA dibandingkan dengan DDA.

C. Proses Menggambar Dinding

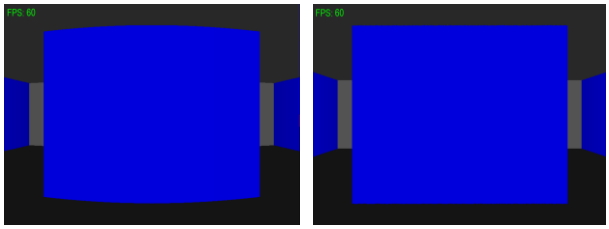
Penggunaan *ray casting* sejatinya hanya merupakan cara untuk menentukan tinggi dinding yang akan digambar pada layer. Berikut merupakan potongan kode untuk menggambar dinding dari output hasil *ray casting*.

```
    ray_angle = player_angle - HALF_FOV
    ray_increment = FOV / RAY_COUNT
    wall_width = SCREEN_WIDTH / RAY_COUNT

    for ray in range(RAY_COUNT):
        distance, ray_end_x, ray_end_y,
wall_type = cast_ray(ray_angle)
        distance *= math.cos(ray_angle -
player_angle)
        wall_height = min((CELL_SIZE *
SCREEN_HEIGHT) / (distance if distance != 0 else
0.00001), SCREEN_HEIGHT)
        wall_top = (SCREEN_HEIGHT - wall_height)
/ 2
        color = get_wall_color(wall_type,
distance)
        pygame.draw.rect(view_surface, color,
(ray * wall_width, wall_top, wall_width + 1,
wall_height))
        ray_angle += ray_increment
```

Dideklarasikan variable ray_angle untuk menyimpan nilai sudut dari *ray* pertama, yaitu *ray* paling kiri dari perspektif penonton (sumber *ray*). Kemudian ray_increment berisi selisih sudut antara sesama *ray*. Variabel wall_width menyimpan lebar setiap kolom dari tembok yang akan digambar karena metode *rendering* dengan *ray casting* ini menggambar tembok dengan cara menggambar sejumlah kolom yang tinggi kolomnya (wall_height) ditentukan dari hasil perhitungan dengan *ray casting*.

Kemudian dilakukan iterasi untuk setiap *ray* yang ada. Untuk setiap ray, digambar suatu kolom dinding. Tinggi kolom dinding, wall_height, diperoleh dari CELL_SIZE dikali dengan SCREEN_HEIGHT dibagi dengan jarak *distance* dari penonton ke dinding yang diperoleh dari *ray casting*. Semakin besar jaraknya maka kolom tembok yang digambar akan semakin kecil sehingga memunculkan efek visual tembok yang berada jauh. Variabel wall_top digunakan agar program tahu dari mana harus mulai menggambar kolom dinding.

Alasan terdapat operasi pengalian *distance* dengan cos dari selisih sudut *ray* dengan sudut pandang penonton adalah untuk menghilangkan efek *fisheye*, yaitu ukuran tembok yang sedikit terdistorsi seperti melihat dari lensa mata ikan.



Gambar 6. Efek Fisheye (kiri) dan Tanpa Efek Fisheye (kanan)

Penyebab efek fisheye terjadi adalah karena jarak dari penonton ke bagian dinding dihitung dari panjang *ray* yang menabrak dinding tersebut sehingga semakin ke samping, jaraknya akan semakin jauh sehingga nampak semakin kecil, proses ini memang sebenarnya mirip dengan bagaimana mata manusia bekerja. Namun, efek tersebut pada *ray casting* terlalu dramatis karena mata manusia tidak akan mendistorsi bentuk benda sesignifikan itu. Maka dari itu, perlu diatasi dengan memproyeksikan semua *ray* terhadap garis lurus arah pandang penonton. Misal vektor yang merepresentasikan arah pandang penonton adalah **p** dan vektor *ray* **r**, maka rumus untuk mencari proyeksi **r** ke **p** dapat diubah bentuknya sehingga menjadi **r** × cos(ray_angle – player_angle). Ini pada dasarnya sama dengan jarak tegak lurus dari penonton ke

kolom dinding yang di-*render*.

D. Perbandingan FPS

Ray casting tanpa DDA akan memerlukan banyak sekali komputasi sehingga seharusnya performa dan FPS yang didapatkan akan jauh lebih rendah dibandingkan dengan *ray casting* dengan DDA.

Berikut merupakan perbandingan keduanya yang telah dibulatkan dari simulasi 60 detik.

Tabel 1. Perbandingan FPS Ray Casting dengan DDA dan Tanpa DDA

	Minimum	Rata-Rata	Maks
Tanpa DDA	5	5	21
DDA	59	59	60

Dilakukan pengamatan dengan software Task Manager, simulasi *ray casting* dengan DDA menggunakan hingga 15% CPU sedangkan jika tanpa DDA bisa menggunakan hingga 34%. Ini menunjukkan bahwa *ray casting* dengan DDA bisa mengurangi beban komputasi hingga setengahnya.

Ray casting dengan atau tanpa DDA memiliki kelebihan dan kekurangan masing-masing. Jika mementingkan visual maka tanpa DDA lebih baik jika perangkat yang digunakan mampu menjalankan komputasi dengan kencang. Jika mengutamakan performa maka *ray casting* dengan DDA jauh lebih baik. Jika ingin hasil yang paling optimal dari visual dan performa, maka diperlukan algoritma lain yang memiliki efisiensi DDA tetapi tetap mempertahankan detail simulasi seperti *ray casting* tanpa DDA.

IV. KESIMPULAN

Algoritma *ray casting* tanpa DDA pada kasus terburuk akan melakukan transformasi vektor *ray* 2D sebanyak MAX_RAY_LENGTH (panjang maksimum garis *ray*) untuk setiap *ray*. Sedangkan *ray casting* dengan menggunakan DDA pada kasus terburuk akan melakukan transformasi vektor *ray* sebanyak MAX_RAY_LENGTH / CELL_SIZE (panjang maksimum garis *ray* dibagi panjang sebuah sel) untuk setiap *ray*.

Ray casting dengan DDA menghasilkan visual yang cenderun lebih buruk dibandingkan tanpa DDA. *Ray casting* tanpa DDA lebih mulus dalam menampilkan dinding parsial yang jaraknya di ujung panjang *ray* maksimum, sedangkan jika menggunakan DDA maka hanya ada dua kemungkinan, yaitu dinding tersebut di-*render* atau tidak, tidak ada transisi tampilan dinding parsial seperti tanpa DDA. *Ray casting* tanpa DDA menghasilkan efek perbedaan pencahayaan yang lebih baik dan bervariasi dibandingkan dengan DDA.

Hasil *rendering* dengan *ray casting* dapat menimbulkan efek *fisheye* yang dapat diatasi dengan memproyeksikan vektor *ray* terhadapt vektor arah pandang penonton atau kamera.

Penggunaan CPU pada *ray casting* dengan DDA jauh lebih kecil dibandingkan tanpa DDA, bahkan mencapai 50% lebih rendah. FPS yang dihasilkan pada *ray casting* dengan DDA bisa mencapai lebih dari 10 kali lipat dibandingkan tanpa DDA.

V. UCAPAN TERIMA KASIH

Penulis mengucapkan puji syukur kepada Tuhan Yang Maha Esa atas rahmat dan karunia-Nya sehingga makalah berjudul "Analisis Perbandingan Performa dan Visual *Ray Casting* dengan Operasi Vektor Menggunakan DDA dan Tanpa DDA" ini dapat selesai tepat waktu. Penulis juga ingin mengucapkan terima kasih kepada orang tua dan teman-teman yang selalu mendukung dan memberikan semangat terutama secara mental sehingga penulis mampu terus menyusun makalah ini. Penulis juga ingin berterima kasih kepada Dr. Rila Mandala selaku dosen pengajar Aljabar Linier dan Geometri K1 2024/2025 yang telah membagikan ilmunya dan membimbing penulis dalam KBM. Penulis juga mengucapkan terima kasih kepada Dr. Rinaldi Munir sebagai salah satu dosen pengampu mata kuliah Aljabar Linier dan Geometri yang telah memberikan banyak materi dan referensi baik dalam KBM maupun dalam penyusunan makalah ini. Terakhir, penulis ingin mengucapkan terima kasih kepada pihak-pihak lain yang telah membantu dalam penyusunan makalah ini.

REFERENSI

[1] https://www.giantbomb.com/ray-casting/3015-1517/. Diakses pada 1 Januari 2025.

[2] https://permadi.com/1996/05/ray-casting-tutorial-1/. Diakses pada 1 Januari 2025.

[3] Monica Troka, "Visualizing The Digital Differential Analyer (DDA) Algorithm Using Adobe Animate", [Daring]. Tersedia pada: https://topazart.info/e-journals/index.php/ijetai/article/view/59/69. Diakses pada 1 Januari 2025.

[4] Howard Anton, Elementary Linear Algebra, 10th edition, John Wiley amnd Sons, 2010.

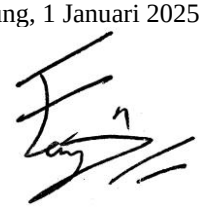
[5] John Vince, Geometric Algebra for Computer Graphics. Springer. 2007

[6] https://lodev.org/cgtutor/raycasting.html. Diakses pada 1 Januari 2025.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 1 Januari 2025



Fajar Kurniawan - 13523027