Analisis Pengaruh Perbedaan Kompleksitas Waktu Algoritma *Fast Inverse Square Root* dengan *Integer Square Root* dalam Simulasi Cahaya

Nathanael Santoso - 13520129[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13520129@std.stei.itb.ac.id

***Abstract*—Kompleksitas Algoritma adalah cara untuk mengkategorisasi waktu atau tempat yang diperlukan suatu algoritma dengan jumlah masukan N untuk mendapatkan hasil tertentu. Algoritma dengan kompleksitas yang besar cenderung lebih sedikit dipakai dibandingkan algoritma dengan kompleksitas kecil karena tidak efisien dalam memakai sumber daya komputasi. Salah satu algoritma yang mahal dari segi komputasi pada akhir abad ke-20 adalah algoritma normalisasi vektor. Normalisasi vektor memiliki aplikasi yang sangat penting dalam grafika komputer terutama dalam simulasi cahaya menggunakan formula pencahayaan Phong karena dalam formula pencahayaan Phong, komputer perlu menormalisasi setiap vektor yang diperlukan. Oleh karena itu, pengurangan kompleksitas waktu dalam algoritma normalisasi akan secara drastis menurunkan waktu yang diperlukan untuk memproses citra. Pada awal abad ke-21, muncul algoritma yang dianggap sangat canggih pada zamannya karena bisa menghitung norma vektor dengan jauh lebih cepat dibandingkan formula lainnya yang dinamakan algoritma *Fast Inverse Square Root* (FISR). Makalah ini membandingkan kompleksitas algoritma FISR dengan algoritma *Integer Square Root* dan menganalis pengaruhnya pada simulasi cahaya menggunakan objek paraboloid dalam bahasa Python.**

***Keywords*—*Fast Inverse Square Root*, *Integer Square Root*, Kompleksitas Algoritma, Simulasi Cahaya**

I. PENDAHULUAN

A. Latar Belakang

Fast Inverse Square Root (FISR) adalah algoritma yang dapat menghitung kebalikan dari akar kuadrat suatu angka dalam tipe data *floating point*. Algoritma ini mulai terkenal setelah rilisnya *source code* untuk game multipemain Quake III Arena yang keluar pada 19 Agustus, 2005. Pada zaman itu, perhitungan vektor ternormalisasi dianggap prosedur yang mahal karena dua hal. Pertama, karena belum ditemukan cara untuk mengaproksimasi akar kuadrat yang cepat. Kedua, karena belum ada proses primitif yang efisien untuk membagi sehingga pembagian ditangani oleh *compiler* yang membuat proses pembagian lebih lambat lagi. Oleh karena itu, saat *source code* Quake III Arena yang mengandung implementasi algoritma FISR pertama kali dibuka untuk akses secara umum, banyak orang terkesan dengan implementasi FISR yang bisa mendapatkan kebalikan akar kuadrat dalam waktu konstan dan tidak menggunakan pembagian. Orang kemudian mulai mencari penulis algoritma ini hingga pada tahun 2006, penulis algoritma asli dari FISR, Greg Walsh, seorang ahli komputer dan salah satu penemu Ardent Computer, mengaku pertama menuliskannya untuk produk Titan graphics minicomputer untuk mencapai target kecepatan pemrosesan yang dijanjikan [1]. Algoritma itu kemudian diadaptasi oleh Gary Tarolli, seorang programmer di NVIDIA, dengan menambahkan aproksimasi pertama FISR yang terkenal hingga menjadi algoritma yang dikenal hingga hari ini [2].

```
float InvSqrt (float x){
    float xhalf = 0.5f*x;
    int i = *(int*)&x;
    i = 0x5f3759df - (i>>1);
    x = *(float*)&i;
    x = x*(1.5f - xhalf*x*x);
    return x;
}
```

Gambar 1. Implementasi Fast Inverse Square Root dalam Bahasa Pemrograman C
(Sumber: https://www.beyond3d.com/content/articles/15, diakses pada tanggal 13 Desember 2021)

Ditemukannya algoritma FISR mempercepat berbagai teknologi pemrosesan grafik, khususnya pada grafika tiga dimensi yang banyak melakukan simulasi cahaya. Hal ini disebabkan formula yang banyak dipakai dalam pemrosesan grafik memerlukan norma vektor untuk berfungsi. Mencari norma vektor sangat penting karena perkalian titik dua unit vektor akan menghasilkan kosinus sudut di antara kedua vektor tersebut. Dengan mengetahui kosinus sudut di antara dua vektor, bisa mendapatkan banyak hal seperti jumlah cahaya yang terpencar ke suatu vektor, perpantulan cahaya, dan lain-lain. Khususnya untuk pemrosesan grafik, formula pencahayaan Phong, formula pencahayaan untuk mendapatkan pencahayaan relatif suatu titik yang merupakan formula pencahayaan populer dan banyak dipakai, menggunakan perkalian titik secara ekstensif. Hal ini menyebabkan simulasi pencahayaan pada permukaan dan bidang mengalami pertumbuhan yang sangat pesat sehingga banyak muncul aplikasi dan permainan yang memiliki resolusi yang tinggi namun tetap bisa dijalankan

dengan sangat cepat.

Sebelum FISR ditemukan, terdapat banyak implementasi algoritma yang kurang efisien untuk menemukan norma dari vektor. Salah satu algoritma tersebut adalah algoritma *Integer Square Root* (ISR) yang bisa mengaproksimasikan akar kuadrat suatu angka dengan cara mencari penghampiran paling besar yang ketika dikuadratkan menghasilkan angka yang lebih kecil dari angka asli. Meskipun kedua algoritma memiliki waktu jalan yang relatif konstan, algoritma ini dikatakan lebih lambat dari FISR. Namun belum ada analisis mendalam yang mendukung kesimpulan tersebut. Oleh karena itu, untuk mengetahui dengan baik pengaruh ditemukannya algoritma FISR terhadap perkembangan grafika komputer, diperlukan analisis kompleksitas waktu yang membandingkan kedua algoritma tersebut dalam lingkungan yang terkendali seperti simulasi cahaya agar mengetahui perbedaan kecepatan dari kedua algoritma tersebut.

B. Rumusan Masalah

Dari latar belakang, didapatkan rumusan masalah sebagai berikut:

1. Apakah kompleksitas waktu masing-masing algoritma FISR dan ISR?
2. Bagaimanakah pengaruh kompleksitas waktu kedua algoritma tersebut terhadap simulasi cahaya?
3. Bagaimanakah pengaruh FISR terhadap pemrosesan grafik dibandingkan ISR yang dapat dicermati dari simulasi cahaya tersebut?

C. Tujuan

Dari rumusan masalah, didapatkan tujuan penulisan makalah sebagai berikut:

1. Mencari kompleksitas waktu algoritma FISR dan ISR dalam notasi standar.
2. Mengetahui pengaruh kompleksitas waktu dari algoritma pencarian kebalikan akar kuadrat terhadap simulasi cahaya.
3. Menyimpulkan pengaruh FISR pada pemrosesan grafik pada zamannnya jika dibandingkan algoritma yang populer pada zaman tersebut seperti ISR.

II. TEORI DASAR

A. Kompleksitas Algoritma

Kompleksitas Algoritma adalah cara untuk mengkategorisasikan kecepatan algoritma berdasarkan langkah karakteristik yang dijalankan dibandingkan dengan jumlah masukan N. Langkah karakteristik tipikal berupa langkah yang memuat aritmatika seperti perkalian, pembagian, pertambahan, dan pengurangan namun juga dapat merupakan langkah logik yang berupa perbandingan atau operasi bit. Kompleksitas algoritma biasanya dinyatakan dalam tiga bentuk umum yaitu big-Oh, big-Omega, dan big-Theta dengan big-Oh sebagai notasi yang paling terkenal.

Secara informal, big-Oh didefinisikan sebagai batas atas suatu algoritma, big-Omega didefinisikan sebagai batas bawah suatu algoritma, dan big-Theta didefinisikan sebagai kompleksitas untuk algoritma yang batas atas dan batas bawahnya sama [3]. Untuk masukan N pada algoritma A, O(N) adalah notasi big-Oh ketika kasus terburuk A adalah linear dalam waktu atau bisa diekspresikan sebagai $T_{max}(A) = CN + \ldots$ dengan $T_{max}(A) \leq KN$ untuk sebuah konstanta K. Sedangkan $\Omega(N)$ adalah notasi big-Omega ketika kasus terbaik A adalah linear atau bisa diekspresikan sebagai $T_{min}(A) = CN + \ldots$ dengan $T_{min}(A) \geq KN$ untuk sebuah konstanta K. Untuk big-Theta, $\theta(N)$ adalah notasi big-Theta ketika kasus terbaik dan terburuk A bisa dinotasikan dengan derajat N yang sama dan bisa direpresentasikan sebagai $K_{min}N \leq T_{min}(A) \leq T_{max}(A) \leq K_{max}N$.

B. Fast Inverse Square Root)

Fast Inverse Square Root (FISR) adalah algoritma yang menghitung dengan cepat kebalikan akar kuadrat dari suatu angka. FISR merupakan implementasi yang lebih eifisien dari fungi Newton-Raphson. Langkah umum untuk algoritma FISR adalah sebagai berikut:

1. Ubah representasi masukan dari *floating point* menjadi *integer*.
2. Aproksimasi harga hasil dengan cara mengurangi angka heksadesimal 0x5f3759df dengan masukan yang sudah digeser bitnya ke kiri satu kali.
3. Lakukan Newtons's method terhadap aproksimasi sesuai ketepatan yang diinginkan. (umumnya sekali)

Sejak penemuannya, sudah terdapat banyak analisis terhadap algoritma FISR khususnya pada angka aproksimasinya. Biasa disebut angka ajaib, angka ini sudah banyak diteliti dan dioptimalisasi. Beberapa optimalisasi yang tercatat adalah optimalisasi angka ajaib dan Newton's Method oleh Jan Kadleck yang mengatakan bahwa 0x5F1FFFF9 adalah angka heksadesimal yang optimal jika pada Newton's Method diganti 1.5 dengan 2.38924456 dan hasil kalinya dikalikan dengan 0.703952253 [4] dan juga optimalisasi untuk sistem operasi 64-bit oleh Charles McEniry [5].

C. Integer Square Root

Integer Square Root adalah algoritma yang menghitung akar kuadrat dari suatu angka dengan mengaproksimasi setiap digit mulai dari digit terbesar hingga terkecil. Algoritma ini dalam basis 2 diimplementasikan dengan pergeseran bit untuk mempermudah jalannya operasi. Langkah umum untuk algoritma ISR yang diadaptasikan untuk tipe data *floating point* adalah sebagai berikut:

1. Ubah representasi masukan dari *floating point* menjadi *integer*.
2. Aproksimasikan harga pertama sebagai *floating point* dengan representasi ilmiah $1 \times 2^{e/2}$ dengan nilai e eksponen dari harga awal dibagi dua.
3. Ambil increment sebagai bit bernilai 1 pada posisi 23 ($1 << 22$).
4. Periksa apakah kuadrat dari aproksimasi tambah increment lebih kecil atau sama dengan masukan. Jika iya, tambahkan increment pada aproksimasi.
5. Ulangi langkah untuk posisi 22 sampai 1.

Terdapat ISR dengan teknik pencarian biner yang lebih cepat daripada ISR dengan satu-per-satu, namun tidak bisa dipakai dalam implementasi *floating point.*

D. Formula Pencahayaan Phong

Phong's Lighting Formula atau Formula Pencahayaan Phong adalah formula yang menghitung pencahayaan pada suatu titik relatif terhadap sumber cahaya dan perspektif pengamat. Formula Pencahayaan Phong untuk cahaya dinyatakan dalam gambar berikut.

$$S_p = C_p[\cos(i)(1-d)+d] + W(i)[\cos(s)]^n,$$

Gambar 2. Formula Pencahayaan Phong
(Sumber: https://users.cs.northwestern.edu/~ago820/cs395/Papers/Phong_1975.pdf diakses pada tanggal 13 Desember 2021)

Formula pada Gambar 2. bisa diturunkan menjadi formula pada gambar berikut.

$$\mathbf{I_{amb}} = K_{amb}\mathbf{C_{diff}}\mathbf{C_{light}}$$

$$\mathbf{I_{dir}} = K_{diff}(\mathbf{N}\cdot\mathbf{L})\,\mathbf{C_{diff}}\mathbf{C_{light}} + K_{spec}\,(\mathbf{R}\cdot\mathbf{V})^{Nphong}\,\mathbf{C_{spec}}$$

$$\mathbf{I_{point}} = [K_{diff}(\mathbf{N}\cdot\mathbf{L})\,\mathbf{C_{diff}}\mathbf{C_{light}} + K_{spec}\,(\mathbf{R}\cdot\mathbf{V})^{Nphong}\,\mathbf{C_{spec}}] / (d_0 + d)^{n_1}$$

Gambar 3. Penurunan Formula Pencahayaan Phong
(Sumber: https://people.eecs.berkeley.edu/~ug/slide/pipeline/assignments/as8/SLIDEHOME/docs/slide/spec/spec_frame_nongeonode.shtml#lighting diakses pada tanggal 13 Desember 2021)

I_{SC} merepresentasikan tinkat iluminasi suatu titik dengan sumber cahaya SC (ambang, terarah, titik), CJ merepresentasikan warna jenis cahaya J (sumber, ambang, spekular, difusi) yang dikalikan dalam formula (e.x. RGB(127, 127, 127)), KT merepresentasikan konstanta ril cahaya tipe T (ambang, difusi, specular) dengan jarak dari 0 sampai 1, d merepresentasikan jarak cahaya dari benda, d0 merepresentasikan jarak dari suatu kerangka acuan, Nphong merepresentasikan eksponen Phong yang menyatakan kekuatan sorotan pada materi, dan N, L, R, V yang masing-masing merepresentasikan vektor normal bidang, negasi vektor cahaya, vektor pantulan cahaya, dan negasi vektor pengamat dalam vektor satuan/*unit vector*.

Seperti terlihat pada Gambar 2., formula asli mengambil kosinus sudut antara cahaya datang dan norma bidang. Oleh karena itu, diperlukan menormalisasi setiap vektor. Untuk mengurangi jumlah vektor yang perlu dinormalisasi, menurut Power, T. J. (1997) turunan formula Phong dapat disimplifikasi lagi dengan cara melakukan substitusi perkalian titik R dan V menjadi perkalian titik N dan L-ditambah-V/2 sehingga hanya perlu menormalisasi 3 vektor saja [6].

E. Python

Python adalah Bahasa Pemrograman Tingkat Tinggi yang dicetuskan dan dikembangkan oleh Guido Van Rossum. Python mengutamakan keterbacaan kode dan modularitas sehingga menjadi salah satu bahasa pemrograman yang paling populer. Python memiliki banyak modul untuk keperluan pemodelan matematis, pembelajaran mesin, dan pemrosesan gambar.

Khususnya pada makalah ini, python dipakai sebagai alat implementasi dari simulasi cahaya dan dua algoritma pembanding. Dengan memanfaatkan modul Numpy dan Python Imaging Library (PIL), dan time, dapat diimplementasikan kode python yang bisa diwaktu dan mendekati implementasi asli algoritma dalam bahasa C dengan pemroses grafik sederhana. Simulasi cahaya dilakukan dalam bahasa python dilakukan dengan cara membuat matriks ketinggian yang merupakan fungsi dari formula paraboloid, kemudian membuat vektor normal dengan mencari normal dari turunan lokal setiap koordinat dalam matriks. Vektor kemudian dinormalisasi dan dimasukkan ke dalam implementasi Formula Pencahayaan Phong dan keluarannya dimasukkan ke dalam gambar yang diproses memakai PIL.

III. METODOLOGI DAN PENGUMPULAN DATA

A. Metodologi

1. Hipotesis

Fast Inverse Square Root lebih kecil kompleksitas waktunya dibandingkan dengan Integer Square Root dan lebih cepat dalam simulasi cahaya.

2. Teknik Pengumpulan Data

Pengumpulan data dilakukan dengan pertama mengimplementasikan kode algoritma dalam Bahasa Pemrogaman Python dengan mensimulasikan type casting pada Bahasa Pemrograman C menggunakan modul Numpy, kemudian mengimplementasikan suatu permukaan dengan cara membuat objek paraboloid dan vektor menggunakan kelas dalam Python, membuat metode masing-masing kelas, dan juga membuat pemrosesan gambar menggunakan PIL.

Kemudian Dimasukkan tes data dengan variasi size citra dan juga sudut cahaya yang kemudian disimpan dalam file dalam format GIF. Percobaan diulang tiga kali untuk mendapatkan akurasi yang baik. Semua data dan hasil kode kemudian diupload ke repository di GitHub agar bisa dilacak dengan mudah [7]. Dilakukan juga studi kepustakaan dari sumber luar untuk mendapatkan opini profesional untuk menunjang hasil dari data tersebut.

B. Data

1. Implementasi FISR dalam Python

FISR diimplementasikan dalam python menggunakan modul Numpy untuk mensimulasikan type casting bahasa C, selebihnya implementasi sesuai dengan referensi [2].

```
def FISR(input: float, accurate: bool = False) -> float:
    """ Converts an input float to the inverse square root using the Fast Inverse Square Root algorithm
    """
    # Approximate inverse square root using raw type casting and magic number 0x5f3759df
    approx_float = float32(input)
    cast_int = approx_float.view(int32)
    cast_int = int32(0x5f3759df) - int32(cast_int >> 1)
    approx_float = cast_int.view(float32)

    # Newtonian Iteration
    half = input * 0.5
    approx_float = approx_float * (1.5 - (half * approx_float * approx_float))
    if accurate:
        approx_float = approx_float * (1.5 - (half * approx_float * approx_float))

    return float(approx_float)
```

Gambar 4. Implementasi FISR dalam Python
(Sumber: Dokumen penulis)

2. Implementasi ISR dalam Python

ISR diimplementasikan dalam python menggunakan Numpy seperti algoritma FISR untuk mensimulasikan type casting bahasa C, selebihnya implementasi sesuai dengan dasar teori

pada bab II.A.

```python
def ISR(input: float, inverse: bool = False) -> float:
    """ Converts an input float to the inverse square root using Integer Square Root Algorithm
    """
    # Approximate square root to 1 x 2^e, e = (E - 127) // 2 + 127, E is the biased exponent
    src_float = float32(input)
    cast_int = src_float.view(int32)
    approx_float = (((((0x7f800000 & cast_int) >> 23) - 127) // 2) + 127) << 23

    # Approximate other integers using integer square root
    for i in range(22, -1, -1):                     # For i in range of digits in mantissa - 1 to 0
        increment = 1 << i                          # Increment is 1 at the position i + 1
        est = (approx_float + increment).view(float32)  # set est = approximation + increment in float form
        if (est * est <= src_float):                # If the original input is more than est ^ 2:
            approx_float += increment               #     > add increment to approximation

    if inverse:
        return 1 / float(approx_float.view(float32))
    return float(approx_float.view(float32))
```

Gambar 5. Implementasi ISR dalam Python
(Sumber: Dokumen penulis)

3. Implementasi Vektor dalam Python

Vektor diimplementasikan menggunakan kelas yang menyimpan absis, ordinat, dan aplikat sebuah vektor 3 dimensi. Kemudian diimplementasikan fungsi normalize, mul_by_factor, add_vector, length, dan dot untuk mempermudah jalannya program.

```python
class Vector:
    def __init__(self, x: float, y: float, z: float):
        self.abscissa = float(x)
        self.ordinate = float(y)
        self.applicate = float(z)

    def __str__(self) -> str:
        return "(%.2f, %.2f, %.2f)" % (self.abscissa, self.ordinate, self.applicate)

    def copy(self):
        copy = Vector(self.abscissa, self.ordinate, self.applicate)
        return copy

    def normalize(self, default: bool = True): ···

    def mul_by_factor(self, x: float): ···

    def add_vector(self, v): ···

    def length(self) -> float: ···

    def dot(self, v): ···
```

Gambar 6. Implementasi Vektor dalam Python
(Source: Dokumen penulis)

4. Implementasi Paraboloid dalam Python

Paraboloid diimplementasikan menggunakan kelas yang menyimpan tabel ketinggian yang dipetakan dengan fungsi:

$$f(x,y) = d_z - (m_x x - d_x)^2 - (m_y y - d_y)^2, \quad (1)$$

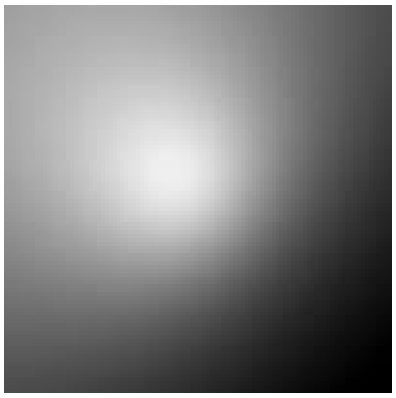
dengan x dan y berupa indeks pada matriks, m_x dan m_y merupakan koefisien x dan y, dan d_x, d_y, dan d_z merupakan perpindahan posisi graf. Fungsi paraboloid diberikan parameter dasar sebagai berikut:

$$f(x,y) = 250 - (0.025x - 12.5)^2 - (0.025y - 12.5)^2. \quad (2)$$

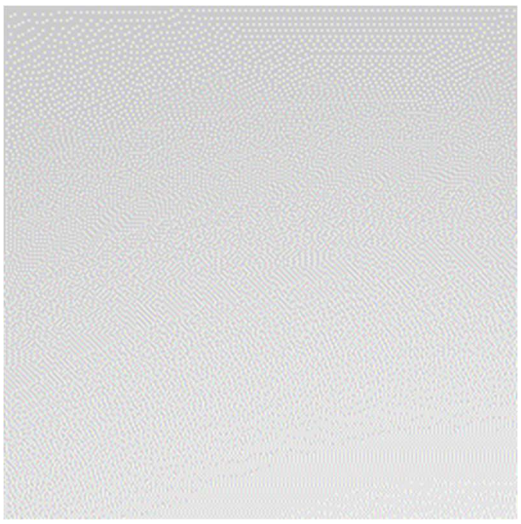
Kemudian dibuat matriks norma bidang dengan mengambil turunan lokal dari setiap posisi x, y dan menormalisasikannya. Diimplementasikan juga metode changelight untuk mengubah vektor cahaya dan shademap sebagai pemrosesan citra menggunakan modul PIL. Pada instansiasi class dan changelight diberikan pilihan apakah menggunakan normalisasi vektor FISR atau ISR dengan parameter default.

```python
class Paraboloid:
    def __init__(self, size: int = 100, light_direction: Vector = Vector(-1, -1, 3), mx: float = 0.025,
    my: float = 0.025, dx: float = 12.5, dy: float = 12.5, dz: float = 250, default: bool = True):
        self.shape = size
        self.dir = light_direction.normalize(default)
        self.heights = [[dz - (mx * x - dx) ** 2 - (my * y - dy)
                         ** 2 for y in range(size + 2)] for x in range(size + 2)]
        print("height map initialized...")
        self.norms = [[Vector(self.heights[x - 1][y] - self.heights[x + 1][y], self.heights[x][y - 1] -
                              self.heights[x][y + 1], 2).normalize(default) for y in range(1, size + 1)] for x in range(1, size + 1)]
        print("norm map initialized...\n")

    def changelight(self, light_direction: Vector = Vector(-1, -1, 3), default: bool = True): ···

    def shade_map(self, num: str = '', specular: bool = True) -> Image: ···
```

Gambar 7. Implementasi Paraboloid dan Pemrosesan Grafik dalam Python
(Source: Dokumen penulis)



Gambar 8. Contoh Simulasi Cahaya Paraboloid Sederhana
(Source: Dokumen penulis)



Gambar 9. Contoh GIF Simulasi Cahaya Resolusi 256 x 256
(Source: Dokumen penulis)

5. Data Hasil Pengolahan Grafik

Dengan memasukkan parameter default dan juga argument size dan angle cahaya 15, maka dilakukan instansiasi bidang, kemudian diolah citra berdasarkan 30 arah cahaya yang berbeda yang ditentukan dengan rumus:

$$V(angles) = Tuple(x, \sqrt{angles^2 - x^2}, 3\ x\ angles). \quad (3)$$

Kemudian semua citra disimpan dalam bentuk GIF dan dicatat waktu pelaksanaan algoritmanya. Size dimasukkan dalam jarak 64 sampai 1024 dengan setiap size berikut merupakan dua kali size sebelum. Setiap size diulang tiga kali dan dimuat pada tabel berikut.

Type	Size	T1 (s)	T2 (s)	T3 (s)
FISR	64	0.90512	0.93542	0.942155123
	128	3.58805	3.77252	3.714707375
	256	14.4278	14.9582	15.70045257
	512	66.5487	69.8238	67.80553651
	1024	300.636	299.561	304.3693345
ISR	64	1.07092	1.09028	1.058126211
	128	4.15863	4.30121	4.059834957
	256	25.003	25.9728	24.00114799
	512	88.6432	89.8179	86.07606387
	1024	336.957	337.823	348.7634969

Table 1. Table of Test Data for Sizes
(Source: Dokumen penulis)

IV. Pengolahan Data dan Analisis

A. Pengolahan

1. Perhitungan Kompleksitas Waktu Algoritma
   a. Fast Inverse Square Root
      Dengan mengacu pada Gambar 4., menghitung jumlah operasi aritmatik dan logika pada algoritma didapatkan kompleksitas waktu $T_{min}(A) = 7$ jika hanya melakukan iterasi Newton Method satu kali dan $T_{max}(A) = 11$ jika melakukan iterasi Newton Method dua kali. Untuk masukan N apapun, kompleksitas algoritma bisa dinyatakan dalam notasi:
      - $O(1)$,
      - $\Omega(1)$,
      - $\theta(1)$.
   b. Integer Square Root
      Dengan mengacu pada Gambar 5., menghitung jumlah operasi aritmatika dan logika pada algoritma didapatkan kompleksitas waktu $T_{min}(A) = 92N/32 + 7$ dan $T_{max}(A) = 115N/32 + 7$ untuk masukan dengan jumlah bit N apapun yang bisa dinyatakan dalam notasi:
      - $O(N)$,
      - $\Omega(N)$,
      - $\theta(N)$.

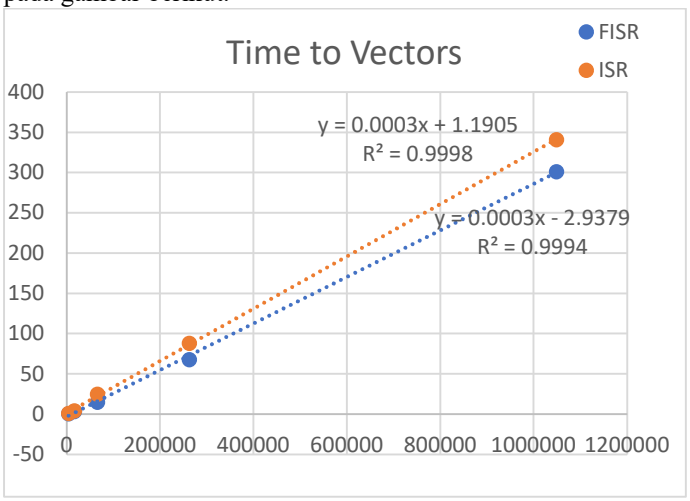
2. Pemrosesan Data Simulasi Cahaya

Data yang diperoleh pada Table 1. kemudian diambil rata-ratanya untuk setiap size, kemudian dihitung jumlah vektor yang dinormalisasi oleh algoritma dan waktu yang diperlukan untuk menormalisasi setiap vektor dihitung dengan waktu rata-rata dibagi jumlah vektor. Kemudian dihitung perbedaan dari size yang sama pada kategori FISR dan ISR dalam persentase. Semua pengolahan tersebut dimuat pada tabel berikut.

Vectors	Average (s)	Time Per Vector (ms)	Difference (%)
4127	0.927566608	0.22475566	13.56276098
16415	3.691758315	0.224901512	11.53701094
65567	15.02884086	0.229213489	39.86613223
262175	68.05937179	0.259595201	22.81685729
1048607	301.5221146	0.287545396	11.62403687
4127	1.073109945	0.260021794	15.69087715
16415	4.173223575	0.254232323	13.04162459
65567	24.99230703	0.381172038	66.29563957
262175	88.17906268	0.336336656	29.56196974
1048607	341.1811356	0.325366067	13.15293939

Table 2. Table of Test Data
(Source: Dokumen penulis)

Data pada Table 2. Vectors dan Average kemudian dimasukkan ke dalam *scatter plot* dengan aksis x sebagai jumlah vektor, aksis y sebagai average dan diberikan *trendline* regresi linear lengkap dengan rumus dan R kuadrat. *Scatter Plot* dimuat pada gambar berikut.



Gambar 10. Scatter Plot dan Trendline Waktu Algoritma
(Source: Dokumen penulis)

B. Analisis

Dapat dilihat pada Tabel 2. Bahwa algoritma pada bagian FISR semua lebih cepat dari algoritma ISR lebih dari 10 persen. Jika dilihat *average time per vector* juga tidak terdapat perubahan signifikan antara iterasi kecuali pada ISR dengan size 256. Dari data ini bisa disimpulkan bahwa algoritma FISR lebih cepat dari algoritma ISR. Namun, jika dilihat dari kompleksitas waktu FISR dibandingkan dengan ISR maka timbul pertanyaan karena FISR dan ISR beda banyak kompleksitas algoritmanya, namun perbedaan waktu tidak lebih banyak dari 40 persen. Hal ini dikarenakan jumlah bit floating point yang konstan untuk sebuah sistem operasi. Jika N dianggap kontanta dengan nilai 32, maka kompleksitas $T_{max}(A)$ dari algoritma ISR adalah 122 sehingga dan dapat dinotasikan sebagai $O(1)$. Oleh karena itu, dapat dilihat pada graf yang terkandung dalam Gambar 10. bahwa kedua metode tidak berbeda secara signifikan dari segi waktu.

Dari kesimpulan paragraph sebelum, muncul sebuah pertanyaan mengenai alasan FISR bisa tercatat begitu berpengaruh pada zamannya. Meskipun FISR tidak menang terlalu banyak melawan integer square root dalam segi kompleksitas algoritma, namun jika memperhitungkan juga kompleksitas waktu dari setiap operasi aritmatika, maka hasil akan berbeda. Kompleksitas algoritma terbaik dari setiap operasi matematika dan logika pada akhir abad ke-20 adalah logaritmik atau O(log(N)) untuk penambahan dan pengurangan, linear logaritmik atau O(Nlog(N)) untuk perkalian, kuadratik atau $O(N^2)$ untuk pembagian dan konstan untuk operasi bit.

Kompleksitas waktu baru dengan memperhitungkan kompleksitas setiap operasinya adalah:

1. $T_{max}(A) = 7N\log(N) + 3\log(N) + 1$, untuk algoritma FISR yang direpresentasikan O(Nlog(N)) dan
2. $T_{max}(A) = 2N^2 + 23N\log(N) + 25\log(N) + 49$, untuk algoritma ISR yang direpresentasikan $O(N^2)$.

Jika dilihat dari segi kompleksitas waktu yang baru tentu saja pada zamannya, FISR sangat inovatif karena tidak hanya orde yang lebih kecil dari ISR, jika dibandingkan instruksi pada orde yang sama pun memerlukan langkah yang lebih sedikit.

V. KESIMPULAN

Dari pembahasan dapat disimpulkan beberapa hal sebagai berikut:

1. Kompleksitas waktu untuk algoritma FISR adalah konstan dengan notasi big-Oh O(1), sedangkan kompleksitas waktu untuk algoritma ISR adalah linear dengan notasi big-Oh O(N) untuk N jumlah bit yang dimasukkan. Jika N dianggap konstan maka ISR juga dianggap konstan sehingga kompleksitas algoritma berubah menjadi O(1). Didapatkan bahwa kompleksitas kedua algoritma terdapat pada tingkatan yang sama yaitu konstan untuk jumlah bit yang konstan dan masukan N apapun.
2. Kompleksitas waktu algoritma FISR dan ISR tidak berpengaruh banyak pada kecepatan simulasi cahaya karena pertama, kode diimplementasikan pada bahasa tingkat tinggi dan banyak bagian pemrograman sudah dioptimasi seperti pembagian sehingga perbedaan waktu antara kedua metode tersebut tidak akan berbeda jauh. Kedua, kompleksitas waktu kedua algoritma adalah konstan pada sistem operasi yang sama sehingga tidak akan berbeda jauh waktu jalan simulasi cahaya antara kedua implementasi.
3. Tidak dapat dipungkiri bahwa FISR membawa pengaruh yang sangat besar pada pemrosesan grafis pada zamannya karena pada zaman tersebut pembagian merupakan proses yang sangat mahal dan belum terdapat primitive pemrosesan untuk pembagian. ISR juga memerlukan langkah 11 kali lipat (122 dibandingkan 11) langkah yang diperlukan FISR untuk mendapatkan hasil yang tidak jauh berbeda sehingga untuk program yang lebih optimal akan terasa jauh lebih besar perbedaan kompleksitas algoritma. Sehingga dapat disimpulkan bahwa FISR membawa perubahan yang cukup signifikan pada zamannya.

VI. UCAPAN TERIMA KASIH

Penulis ingin mengucapkan terima kasih kepada Tuhan Yang Maha Esa karena dengan berkat dan rahmat-Nya yang maha kuasa, penulis dapat menyelesaikan makalah berjudul "Analisis Pengaruh Perbedaan Kompleksitas Waktu Algoritma Fast Inverse Square Root dengan Integer Square Root dalam Simulasi Cahaya" dengan baik. Penulis juga ingin menyampaikan ucapan terima kasih kepada dosen mata kuliah IF2120 Matematika Diskrit Kelas K03, Ibu Dr. Nur Ulfa Maulidevi, S.T, M.Sc., dan juga Bapak Dr. Ir. Rinaldi Munir, M.T. yang menyediakan bahan ajar dan template makalah sehingga penulis dapat menyusun makalah dengan baik. Penulis juga mengucapkan terima kasih kepada kedua orang tua dan keluarga penulis yang senantiasa mendoakan dan mendukung penulis dalam pembuatan makalah ini.

REFERENCES

[1] Mathisen, T. (2006, December 19). *Origin of quake3's fast invsqrt() - part Two. Beyond3D.* Retrieved December 13, 2021, from https://www.beyond3d.com/content/articles/15

[2] Mathisen, T. (2006, November 29). Origin of quake3's fast invsqrt(). Beyond3D. Retrieved December 13, 2021, from https://www.beyond3d.com/content/articles/8/

[3] Nasar, Audrey A. (2016) "The history of Algorithmic complexity," The Mathematics Enthusiast: Vol. 13: No. 3, Article 4. Retrieved December 13, 2021, from https://scholarworks.umt.edu/tme/vol13/iss3/4B.

[4] Kadlec, J. Ř. (2010). Řrřlog::improving the fast inverse square root. Retrieved December 13, 2021, from http://rrrola.wz.cz/inv_sqrt.html.

[5] McEniry, C. (2007, August). *The Mathematics Behind The Fast Inverse Square Root Function Code.* Retrieved December 13, 2021, from https://web.archive.org/web/20150511044204/http://www.daxia.com/bibis/upload/406Fast_Inverse_Square_Root.pdf

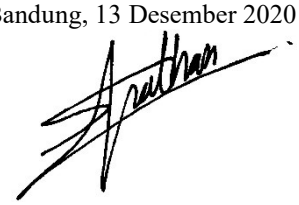
[6] Power, T. J. (1997, February 16). Phong lighting and specular highlights. Retrieved December 13, 2021, from https://www.robots.ox.ac.uk/~att/index.html

[7] Santoso, N. (2021, December 14). *Kode Simulasi Cahaya Dengan Fast Inverse Square Root dan Integer Square Root dalam Python*. GitHub. Retrieved December 14, 2021, from https://github.com/nart4hire/Makalah_Matdis

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, 13 Desember 2020



Nathanael Santoso 13520129