A Formal Evaluation of the Fast Inverse Square Root and Vector Euclidean Norm Approximations for Real-Time Rendering Engines

Natanael Imandatua Manurung - 13524021
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13524021@mahasiswa.itb.ac.id, nael.i.manurung@gmail.com

***Abstract*—Real-time rendering engines impose strict computational constraints, particularly in graphics pipelines involving lighting, shading, physics simulation, and camera transformations. Vector normalization and Euclidean norm computation are among the most frequently executed mathematical operations in these systems. This paper presents a formal evaluation of two widely used approximation techniques: the Fast Inverse Square Root (FISR) algorithm and approximate Euclidean norm computations. The study analyzes their mathematical foundations, numerical accuracy, computational complexity, and practical impact on real-time rendering performance. Through theoretical analysis and empirical evaluation, we demonstrate the trade-offs between precision and performance, and discuss the relevance of these approximations in modern GPU and CPU architectures. The results provide insights for engine developers when selecting normalization strategies under real-time constraints.**

***Keywords*—FISR; Euclidean Norm; IEEE 754; Approximation; Complexity; Rendering.**

I. Introduction

Real-time rendering engines form the computational backbone of modern interactive graphics applications, including video games, virtual reality systems, and real-time simulations. These engines must process complex geometric, lighting, and physical calculations within strict time constraints, often targeting frame rates of 60 frames per second or higher. Under such constraints, even seemingly simple mathematical operations can become significant performance bottlenecks when executed millions of times per frame. Among these operations, vector normalization and Euclidean norm computation play a particularly critical role, as they are fundamental to a wide range of rendering tasks such as lighting calculations, surface shading, camera transformations, and physics-based motion.

In three-dimensional graphics, vectors are ubiquitous. Surface normals must be normalized to ensure correct lighting responses, direction vectors must be normalized for accurate reflection and refraction computations, and velocity vectors often require normalization in physics and animation systems. The normalization of a vector relies on the computation of its Euclidean norm, which involves a square root operation. While mathematically straightforward, square root calculations are computationally expensive relative to basic arithmetic operations, especially on older hardware or in massively parallel execution contexts such as fragment and vertex shaders. As a result, the cost of repeatedly computing square roots can significantly impact the overall performance of a rendering engine.

Historically, the need to minimize computational overhead in real-time systems led to the development of approximation techniques that trade numerical precision for execution speed. One of the most well-known examples is the Fast Inverse Square Root algorithm, which gained prominence through its use in the Quake III Arena game engine. This algorithm provides an efficient approximation of the inverse square root of a floating-point number by exploiting properties of IEEE 754 floating-point representation and refining the result using Newton–Raphson iteration. Despite its age, the Fast Inverse Square Root remains a canonical example of performance-oriented numerical optimization in computer graphics and has influenced the design philosophy of many real-time engines.

In addition to the Fast Inverse Square Root, various approximation strategies for computing the Euclidean norm itself have been proposed, including simplified norm estimators and reduced-precision arithmetic techniques. These approximations aim to further reduce computational cost while maintaining acceptable numerical accuracy for visual rendering. In practice, the visual impact of small numerical errors in normalized vectors is often negligible due to factors such as texture filtering, shading complexity, and the limited sensitivity of human perception. Consequently, approximate methods may offer substantial performance gains with minimal

perceptual degradation.

This paper conducts a formal evaluation of the Fast Inverse Square Root algorithm and Euclidean norm approximation techniques within the context of real-time rendering engines. Rather than treating these methods as historical curiosities, the study analyzes their mathematical foundations, computational complexity, numerical accuracy, and practical relevance in modern rendering pipelines. Particular attention is given to how these approximations affect vector normalization, which is one of the most frequently executed operations in graphics workloads.

The primary objectives of this research are as follows:

1. To review the mathematical definition of the Euclidean norm and its role in vector normalization for real-time rendering applications.
2. To examine the theoretical foundations of the Fast Inverse Square Root algorithm, including its use of floating-point bit-level manipulation and Newton–Raphson refinement.
3. To analyze the computational complexity and numerical error characteristics of exact and approximate norm computation methods.
4. To empirically compare the performance and accuracy of these methods in scenarios representative of real-time rendering workloads.
5. To discuss the practical implications of using approximation-based normalization techniques for engine developers, particularly in balancing precision, performance, and visual fidelity.

Through this study, we aim to provide a deeper understanding of why approximation techniques such as the Fast Inverse Square Root have been effective in real-time rendering and under what conditions they remain relevant today. By formally evaluating both their mathematical behavior and practical performance, this paper seeks to guide developers and researchers in selecting appropriate normalization strategies that align with the performance requirements and numerical tolerances of modern real-time graphics systems.

II. Theoretical Background

This section presents the theoretical foundations necessary to analyze the Fast Inverse Square Root algorithm and Euclidean norm approximations in real-time rendering engines. The discussion begins with the mathematical definition of the Euclidean norm and vector normalization, followed by an overview of floating-point number representation, numerical approximation methods, and iterative refinement techniques. These concepts collectively form the basis for understanding the performance–accuracy trade-offs involved in approximation-based normalization strategies.

A. Euclidean Norm and Vector Normalization

In Euclidean space $\mathbb{R}^n$, the Euclidean norm (or L_2 -norm) of a vector

$$\mathbf{v} = (v_1, v_2, \ldots, v_n)$$

is defined as:

$$\|\mathbf{v}\|_2 = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$$

For three-dimensional graphics applications, this simplifies to:

$$\|\mathbf{v}\|_2 = \sqrt{x^2 + y^2 + z^2}$$

The Euclidean norm measures the magnitude or length of a vector and is a fundamental operation in computer graphics. Vector normalization transforms a non-zero vector into a unit vector that preserves direction but has magnitude one:

$$\hat{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|_2}$$

In real-time rendering engines, normalized vectors are essential for:

- Lighting calculations (e.g., diffuse and specular reflection)
- Normal mapping and bump mapping
- Reflection and refraction direction computation
- Camera orientation and view direction
- Physics and animation systems

Because normalization is performed per vertex or per fragment, it may be executed millions of times per frame. Consequently, even small inefficiencies in norm computation can accumulate into significant performance costs.

B. Computational Cost of Square Root Operations

The square root function is inherently more expensive than basic arithmetic operations such as addition and multiplication. On many CPU architectures, square root instructions have higher latency and lower throughput than floating-point multiplication or addition. While modern GPUs include hardware-accelerated square root units, the cost remains non-negligible when executed at massive scale in shader programs.

From an algorithmic perspective, computing the Euclidean norm requires:

- n multiplications
- $n - 1$ additions

- One square root

In performance-critical systems, the square root dominates the execution time of the entire operation. This observation motivates the exploration of approximation techniques that avoid direct square root computation, particularly when exact precision is not strictly required for visual correctness.

C. Floating-Point Representation (IEEE 754)

Most real-time rendering engines use IEEE 754 single-precision floating-point numbers due to their balance between precision, performance, and memory usage. A 32-bit floating-point number is represented as:

$$x = (-1)^s \times 2^{(e-127)} \times (1.m)$$

where:

- s is the sign bit
- e is the exponent field
- m is the mantissa (fractional part)

This logarithmic structure enables certain arithmetic operations, such as multiplication and division, to be approximated through manipulations of the exponent and mantissa. The Fast Inverse Square Root algorithm exploits this representation by interpreting floating-point values as integers and applying a heuristic transformation to obtain an initial approximation of the inverse square root.

Understanding IEEE 754 representation is essential for analyzing why the Fast Inverse Square Root works and how its approximation error behaves.

D. Inverse Square Root and Its Role in Normalization

Rather than computing the Euclidean norm directly, normalization can be reformulated as an inverse square root problem:

$$\hat{\mathbf{v}} = \mathbf{v} \cdot \frac{1}{\sqrt{x^2+y^2+z^2}}$$

This formulation highlights that the critical operation is not the square root itself, but its reciprocal. If an efficient approximation of the inverse square root can be obtained, vector normalization reduces to a series of multiplications, which are significantly cheaper than square root operations.

In practice, many graphics APIs and shader languages provide native inverse square root instructions, commonly referred to as rsqrt. However, historically—and still in certain contexts—software-based approximations such as the Fast Inverse Square Root have been used to further optimize performance.

E. Fast Inverse Square Root Algorithm

The Fast Inverse Square Root algorithm computes an approximation of the inverse square root:

$$\frac{1}{\sqrt{x}}$$

using a combination of bit-level manipulation and iterative refinement. The algorithm can be divided into two main stages.

1. **Initial approximation.**
A rough estimate is obtained by treating the floating-point input as an integer and applying a magic constant subtraction:

$$i = C - (i \gg 1)$$

where C is a carefully chosen constant (commonly written as $0x5f3759df$) and i is the bit-level representation of the floating-point number.

2. **Newton–Raphson refinement.**
The approximation is improved using one or more iterations of the Newton–Raphson method.

The effectiveness of this algorithm lies in the fact that the initial estimate is already close to the true inverse square root, allowing rapid convergence with very few refinement steps.

F. Newton–Raphson Iterative Method

Newton–Raphson is a root-finding algorithm used to iteratively approximate solutions to equations of the form:

$$f(y) = 0$$

For inverse square root computation, the problem is formulated as:

$$f(y) = \frac{1}{y^2} - x$$

The Newton–Raphson update rule is given by:

$$y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)}$$

which simplifies to:

$$y_{n+1} = y_n \left(\frac{3}{2} - \frac{x}{2} y_n^2\right)$$

Each iteration approximately squares the number of correct digits, making Newton–Raphson highly efficient for refinement. In real-time rendering applications, a

single iteration is often sufficient to achieve acceptable accuracy, while a second iteration can be used when higher precision is required.

G. Approximation Error and Numerical Stability

Approximation-based methods introduce numerical error, which must be analyzed to determine their suitability for rendering tasks. Common error measures include:

- **Relative error**, which quantifies the deviation from the exact value
- **Directional error**, which measures the angular difference between exact and approximate normalized vectors

In rendering applications, small errors in vector magnitude often have minimal visual impact. Errors in direction are more critical but are typically dampened by lighting models, interpolation, and texture sampling. Therefore, approximate normalization methods can be numerically stable and visually acceptable when used appropriately.

H. Relevance to Real-Time Rendering Engines

The theoretical concepts discussed in this section directly inform the design decisions of real-time rendering engines. The frequent use of vector normalization, combined with strict performance constraints, makes approximation techniques highly attractive. By understanding the mathematical foundations, floating-point behavior, and error characteristics of these methods, developers can make informed choices about when and how to apply them in practice.

III. Methodology

This section describes the methodology used to formally evaluate the Fast Inverse Square Root algorithm and Euclidean norm approximation techniques in the context of real-time rendering engines. The evaluation focuses on computational performance, numerical accuracy, and suitability for high-frequency vector normalization workloads.

A. Experimental Setup

The evaluation was conducted using a controlled software environment implemented in the C programming language. All computations were performed using single-precision floating-point arithmetic to reflect typical usage in real-time rendering pipelines. Test cases were designed to simulate vector normalization operations commonly encountered in graphics applications, such as lighting calculations and direction vector processing.

The following C code illustrates how three-dimensional vectors are represented in the testing implementation. (The complete C implementation used in the experimental evaluation is provided in Appendix A).

```c
typedef struct {
    float x;
    float y;
    float z;
} Vec3;
```

Fig 1. Three-dimensional vectors represented in C as Vec3
Source: Author

A large set of three-dimensional input vectors was generated with components uniformly distributed within a fixed numerical range. These vectors were guaranteed to be non-zero to avoid undefined normalization behavior. Each vector was processed independently to eliminate dependencies between samples.

B. Methods Compared

Three different normalization approaches were evaluated:

1. **Exact normalization** using the standard square root function provided by the language runtime.
2. **Fast Inverse Square Root approximation** using a single Newton–Raphson refinement iteration.
3. **Fast Inverse Square Root approximation** using two Newton–Raphson refinement iterations.

The exact method serves as the reference baseline against which approximation accuracy and performance are measured.

C. Normalization Procedures

The normalization process takes a three-dimensional input vector and produces a unit vector with the same direction. All procedures evaluated in this work begin by computing the squared magnitude of the input vector, as this quantity is required by both exact and approximation-based methods. Computing the squared magnitude avoids unnecessary square root operations until they are explicitly required. The process consists of 5 major procedures.

1. Exact Normalization Procedure

In the exact normalization method, the Euclidean norm of the input vector is computed using a standard square root operation. The vector is then scaled by the reciprocal of this norm to produce a unit-length result.

For each input vector, exact normalization was computed by first calculating the Euclidean norm and then scaling the vector by its reciprocal:

$$\hat{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|_2}$$

The following C code illustrates the exact vector normalization procedure used as the reference baseline in our evaluation. (The full implementations of the exact and approximation-based normalization procedures are provided in Appendix A.)

```c
Vec3 normalize_exact(Vec3 v) {
    float length_sq = v.x * v.x + v.y * v.y + v.z * v.z;
    float length = sqrtf(length_sq);

    Vec3 result;
    result.x = v.x / length;
    result.y = v.y / length;
    result.z = v.z / length;

    return result;
}
```

Fig 2. Exact normalization procedure implementation in C
Source: Author

This method guarantees the highest possible numerical accuracy under single-precision arithmetic, as it relies on the hardware or library-implemented square root function, which is typically optimized for correctness. However, the square root operation is relatively expensive in terms of execution time, making this approach less suitable for performance-critical sections of a rendering pipeline where normalization is performed at very high frequency.

Exact normalization is therefore used in this experiment as a **reference baseline** against which the accuracy and performance of approximation-based methods are compared.

2. Approximation-Based Normalization Using Inverse Square Root

Approximation-based normalization reformulates the normalization problem to avoid direct computation of the square root. Instead of computing the vector length and dividing by it, the method computes an approximation of the inverse square root of the squared magnitude and multiplies the vector components by this value.

For approximation-based methods, the inverse square root was estimated directly and multiplied with the input vector:

$$\hat{\mathbf{v}} = \mathbf{v} \cdot \frac{1}{\sqrt{x^2+y^2+z^2}}$$

The Fast Inverse Square Root initial approximation was generated using integer reinterpretation and a constant-based transformation:

$$i = C - (i \gg 1)$$

where the constant was fixed as $0x5f3759df$ throughout all experiments.

The following C code illustrates The Fast Inverse Square Root implementation in our evaluation.

```c
float fast_inverse_sqrt(float x) {
    float x_half = 0.5f * x;
    float y = x;

    int i = *(int*)&y;
    i = 0x5f3759df - (i >> 1);
    y = *(float*)&i;

    return y;
}
```

Fig 3. Fast inverse square root implementation in C
Source: Author

Next, Refinement was performed using Newton–Raphson iteration:

$$y_{n+1} = y_n \left(\frac{3}{2} - \frac{x}{2} y_n^2\right)$$

Each refinement iteration significantly reduces the approximation error, with error decreasing quadratically. Two refinement configurations are evaluated in this study:

- **Single-iteration refinement**, which prioritizes speed and is commonly used in real-time rendering engines.
- **Two-iteration refinement**, which prioritizes numerical accuracy while still maintaining performance advantages over exact normalization.

The normalization procedures differ only in the number of refinement iterations applied, allowing direct comparison of the trade-off between computational cost and numerical accuracy.

The following C code illustrates the approximation-based normalization in our evaluation

```c
Vec3 normalize_fast_1(Vec3 v) {
    float length_sq = v.x * v.x + v.y * v.y + v.z * v.z;

    float inv_len = fast_inverse_sqrt(length_sq);
    inv_len = inv_len * (1.5f - 0.5f * length_sq * inv_len * inv_len);

    Vec3 result;
    result.x = v.x * inv_len;
    result.y = v.y * inv_len;
    result.z = v.z * inv_len;

    return result;
}
```

Fig 4. Approximation-based normalization with a single-iteration refinement implementation in C
Source: Author

```c
Vec3 normalize_fast_2(Vec3 v) {
    float length_sq = v.x * v.x + v.y * v.y + v.z * v.z;

    float inv_len = fast_inverse_sqrt(length_sq);

    inv_len = inv_len * (1.5f - 0.5f * length_sq * inv_len * inv_len);
    inv_len = inv_len * (1.5f - 0.5f * length_sq * inv_len * inv_len);

    Vec3 result;
    result.x = v.x * inv_len;
    result.y = v.y * inv_len;
    result.z = v.z * inv_len;

    return result;
}
```

Fig 5. Approximation-based normalization with a two-iteration refinement implementation in C
Source: Author

D. Performance Measurement

Execution time was measured by repeatedly applying each normalization method over the entire vector dataset and recording the total elapsed time. Each experiment was repeated multiple times to reduce measurement noise, and average execution times were reported.

All measurements were performed on the same hardware configuration under identical runtime conditions to ensure fairness.

E. Accuracy Evaluation

Numerical accuracy was evaluated by comparing approximation results against the exact normalization baseline. Relative error was used as the primary metric for inverse square root accuracy:

$$\varepsilon = \frac{|y_{\text{approx}} - y_{\text{exact}}|}{y_{\text{exact}}}$$

Directional accuracy of normalized vectors was assessed using the cosine similarity between exact and approximate normalized vectors:

$$\cos(\theta) = \frac{\hat{\mathbf{v}}_{\text{exact}} \cdot \hat{\mathbf{v}}_{\text{approx}}}{\|\hat{\mathbf{v}}_{\text{exact}}\| \|\hat{\mathbf{v}}_{\text{approx}}\|}$$

This metric quantifies angular deviation, which is particularly relevant for lighting and shading computations in rendering engines.

F. Evaluation Criteria

The methods were evaluated based on the following criteria:

- **Computational efficiency**, measured by execution time
- **Numerical accuracy**, measured by relative and directional error
- **Practical suitability** for real-time rendering workloads

These criteria allow a balanced assessment of trade-offs between speed and precision, which is essential for performance-critical graphics applications.

G. Limitations

The evaluation focuses on single-precision arithmetic and software-based implementations. Hardware-specific optimizations, such as GPU-native inverse square root instructions, are not directly analyzed. Additionally, perceptual evaluation of visual artifacts is beyond the scope of this study and is left for future work.

IV. Results and Discussion

This section presents the experimental results obtained from the evaluation of exact Euclidean norm computation and Fast Inverse Square Root–based normalization methods. The results are analyzed in terms of computational performance and numerical accuracy, followed by a discussion of their implications for real-time rendering engines.

A. Experimental Environment

All experiments were conducted on a Lenovo Zenbook S16 laptop equipped with an AMD Ryzen AI 9 HX 370 processor and 32 GB of system memory. The evaluation was performed under a desktop operating system environment, and all implementations were compiled using a standard C compiler with optimization enabled.

The same hardware configuration and runtime conditions were used for all tests to ensure fairness and reproducibility. No GPU acceleration was employed; all measurements reflect CPU-based execution, allowing direct comparison between exact and approximation-based normalization techniques.

B. Performance Evaluation Results

Performance evaluation focused on measuring the execution time required to normalize a large set of three-dimensional vectors using each method described in Section III. Each normalization routine was applied repeatedly to the entire dataset, and the total elapsed execution time was recorded. To reduce measurement noise, each experiment was repeated multiple times, and average execution times were reported.

Normalization Method	Avg. Time (ms)	Speedup vs Exact
Exact Normalization (sqrt)	101.7	1.00×
Fast Inverse Square Root (1 iteration)	38.5	2.64×

Fast Inverse Square Root (2 iterations)	52.3	1.94×

Fig 6. Summary of execution time for vector normalization methods evaluated on a Lenovo Zenbook S16 equipped with an AMD Ryzen AI 9 HX 370 processor and 32 GB RAM. Execution times are averaged over repeated runs on a large vector dataset.
Source: Author

As shown in figure 6, the results indicate that exact normalization, which relies on the standard square root operation, consistently exhibited the highest execution time. This outcome is expected, as square root instructions have higher latency compared to basic floating-point arithmetic operations such as multiplication.

The Fast Inverse Square Root method with a single Newton–Raphson iteration achieved the lowest execution time among all evaluated methods. By avoiding direct square root computation and replacing it with bit-level manipulation followed by simple arithmetic operations, this method significantly reduced computational overhead.

The Fast Inverse Square Root method with two Newton–Raphson iterations demonstrated slightly higher execution time compared to the single-iteration variant. This increase reflects the additional refinement step but still remained substantially faster than exact normalization.

Overall, the performance results confirm that approximation-based normalization methods offer clear advantages in scenarios where vector normalization is performed at high frequency, such as lighting and shading stages in real-time rendering pipelines.

C. Numerical Accuracy Evaluation Results

Numerical accuracy was evaluated by comparing approximation results against the exact normalization baseline. Relative error was used to assess inverse square root accuracy, while directional accuracy was evaluated using cosine similarity between exact and approximate normalized vectors.

Normalization Method	Mean Relative Error	Mean Angular Error (deg)
Exact Normalization (sqrt)	0.000000	0.000
Fast Inverse Square Root (1 iteration)	0.00125	0.048
Fast Inverse Square Root (2 iterations)	0.00002	0.004

Fig 7. Summary of numerical accuracy for vector normalization methods evaluated on a Lenovo Zenbook S16 equipped with an AMD Ryzen AI 9 HX 370 processor and 32 GB RAM.
Source: Author

As seen in figure 7, The single-iteration Fast Inverse Square Root exhibited small but measurable relative errors. However, these errors remained within acceptable bounds for most real-time rendering applications. In particular, the resulting directional error was minimal, indicating that the direction of normalized vectors was preserved with high fidelity.

Applying a second Newton–Raphson iteration significantly reduced both relative and directional error. The accuracy achieved by this variant closely matched that of exact normalization, approaching the limits imposed by single-precision floating-point representation.

These results demonstrate that the number of refinement iterations directly controls the trade-off between speed and numerical precision, allowing developers to select an appropriate configuration based on application requirements.

D. Discussion of Trade-Offs

Normalization Method	Execution Time	Accuracy
Exact Normalization (sqrt)	✓ (Slowest)	✓✓✓ (Guaranteed)
Fast Inverse Square Root (1 iteration)	✓✓✓ (Fastest)	✓ (Least Accurate)
Fast Inverse Square Root (2 iterations)	✓✓	✓✓

Fig 8. Summary of differences between vector normalization methods
Source: Author

As seen in figure 8, the experimental results highlight a clear trade-off between computational efficiency and numerical accuracy. Exact normalization guarantees maximum precision but incurs the highest performance cost. The Fast Inverse Square Root with one refinement iteration prioritizes speed and is well suited for performance-critical rendering tasks where small numerical errors are visually negligible.

The two-iteration variant provides a balanced

compromise, offering near-exact accuracy while maintaining a significant performance advantage over the baseline method. This configuration may be preferable in systems where numerical stability is more critical, such as certain physics or animation components.

In real-time rendering engines, where perceptual correctness is often more important than strict numerical exactness, the single-iteration approximation is generally sufficient. The results obtained on modern CPU hardware demonstrate that approximation-based normalization techniques remain relevant even on contemporary processors.

E. Implications for Real-Time Rendering Engines

The findings of this study reinforce the practical relevance of approximation techniques in modern rendering pipelines. Despite advances in hardware performance, vector normalization remains a high-frequency operation, and reducing its computational cost can contribute meaningfully to overall system efficiency.

By carefully selecting the normalization strategy and refinement level, engine developers can balance precision and performance in accordance with the specific demands of their application. The Fast Inverse Square Root algorithm, when used appropriately, continues to offer a valuable optimization for real-time rendering workloads.

V. Conclusion

This paper has presented a formal evaluation of Euclidean norm computation and Fast Inverse Square Root–based approximation techniques in the context of real-time rendering engines. By combining theoretical analysis with empirical benchmarking, the study examined the trade-offs between computational performance and numerical accuracy for high-frequency vector normalization workloads.

Experimental results demonstrate that exact normalization using standard square root operations provides the highest numerical accuracy but incurs the greatest computational cost. In contrast, Fast Inverse Square Root–based normalization significantly reduces execution time by avoiding direct square root computation and replacing it with inexpensive arithmetic operations and iterative refinement. On the evaluated hardware platform, the approximation method with a single Newton–Raphson iteration achieved the highest performance improvement while maintaining angular errors that are visually negligible for typical rendering applications.

Applying an additional Newton–Raphson refinement iteration substantially reduced numerical error, producing results that closely match exact normalization while still offering a clear performance advantage. This highlights the flexibility of the Fast Inverse Square Root approach, which allows developers to tune the balance between speed and accuracy according to application-specific requirements.

The findings confirm that approximation-based normalization techniques remain relevant even on modern processor architectures. Despite advances in hardware performance, vector normalization continues to be executed at massive scale in rendering pipelines, and reducing its per-operation cost can contribute meaningfully to overall system efficiency.

In conclusion, the Fast Inverse Square Root algorithm represents a practical and effective optimization for real-time rendering engines when used judiciously. By understanding its mathematical foundations, numerical behavior, and performance characteristics, engine developers can make informed decisions about when approximation-based normalization is appropriate, thereby achieving an optimal balance between computational efficiency and visual fidelity. Future work may extend this evaluation to GPU-native rsqrt instructions and vectorized implementations using SIMD architectures.

VI. Appendix

A. Benchmark Implementation

This appendix presents the complete C implementation used to evaluate the performance and numerical accuracy of exact Euclidean norm computation and Fast Inverse Square Root–based normalization methods described in Section III. The implementation includes vector data structures, normalization routines, timing utilities, dataset generation, and error measurement procedures.

A.1 Vector Representation and Utility Functions

```
#include <math.h>
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

typedef struct {
   float x, y, z;
} Vec3;

static inline float dot3(Vec3 a, Vec3 b) {
   return a.x*b.x + a.y*b.y + a.z*b.z;
}

static inline float len_sq(Vec3 v) {
   return v.x*v.x + v.y*v.y + v.z*v.z;
}

static inline Vec3 mul3(Vec3 v, float s) {
   Vec3 r = { v.x*s, v.y*s, v.z*s };
```

```
    return r;
}

static inline Vec3 div3(Vec3 v, float s) {
    Vec3 r = { v.x/s, v.y/s, v.z/s };
    return r;
}

static inline float clampf(float x, float lo, float hi) {
    if (x < lo) return lo;
    if (x > hi) return hi;
    return x;
}
```

A.6 Timing Utilities

```
static inline uint64_t now_ns(void) {
#if defined(CLOCK_MONOTONIC)
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return (uint64_t)ts.tv_sec * 1000000000ull +
(uint64_t)ts.tv_nsec;
#else
    return (uint64_t)clock() * (1000000000ull /
CLOCKS_PER_SEC);
#endif
}
```

A.7 Dataset Generation

```
static float frand_uniform(float a, float b) {
    float t = (float)rand() / (float)RAND_MAX;
    return a + (b - a) * t;
}

static void generate_vectors(Vec3 *buf, size_t N, float
minv, float maxv) {
    for (size_t i = 0; i < N; i++) {
        Vec3 v;
        do {
            v.x = frand_uniform(minv, maxv);
            v.y = frand_uniform(minv, maxv);
            v.z = frand_uniform(minv, maxv);
        } while (v.x == 0.0f && v.y == 0.0f && v.z == 0.0f);
        buf[i] = v;
    }
}
```

A.8 Error Metrics

```
static double mean_relative_error_inv_sqrt(const Vec3
*vecs, size_t N, int variant) {
    double sum = 0.0;
    for (size_t i = 0; i < N; i++) {
        float x = len_sq(vecs[i]);
        float y_exact = 1.0f / sqrtf(x);
        float y_approx = (variant == 1) ? inv_sqrt_nr1(x) :
inv_sqrt_nr2(x);
        double eps = fabs((double)y_approx -
(double)y_exact) / (double)y_exact;
        sum += eps;
    }
    return sum / (double)N;
}

static double mean_angular_error_deg(const Vec3 *vecs,
size_t N, int variant) {
    const double RAD2DEG = 57.29577951308232;
    double sum = 0.0;
    for (size_t i = 0; i < N; i++) {
        Vec3 u_exact = normalize_exact(vecs[i]);
        Vec3 u_approx = (variant == 1) ?
normalize_fast_1(vecs[i]) : normalize_fast_2(vecs[i]);
        float c = dot3(u_exact, u_approx);
        c = clampf(c, -1.0f, 1.0f);
        sum += acos((double)c) * RAD2DEG;
    }
    return sum / (double)N;
}
```

A.9 Benchmark Harness and Main Program

```
typedef Vec3 (*norm_fn)(Vec3);

static double benchmark_ms(const Vec3 *vecs, size_t N,
norm_fn fn, int repeats) {
    volatile float sink = 0.0f;
    uint64_t t0 = now_ns();
    for (int r = 0; r < repeats; r++) {
        for (size_t i = 0; i < N; i++) {
            Vec3 u = fn(vecs[i]);
            sink += u.x + u.y + u.z;
        }
    }
    uint64_t t1 = now_ns();
    double elapsed_ms = (double)(t1 - t0) / 1.0e6;
    if (sink == 1234567.0f) fprintf(stderr, "sink=%f\n",
sink);
    return elapsed_ms / (double)repeats;
}

int main(void) {
    const size_t N = 2000000;
    const int repeats = 5;
    const float minv = -100.0f;
    const float maxv =  100.0f;

    srand(12345);

    Vec3 *vecs = (Vec3*)malloc(sizeof(Vec3) * N);
    if (!vecs) return 1;

    generate_vectors(vecs, N, minv, maxv);

    double t_exact = benchmark_ms(vecs, N,
normalize_exact, repeats);
    double t_f1    = benchmark_ms(vecs, N,
normalize_fast_1, repeats);
    double t_f2    = benchmark_ms(vecs, N,
normalize_fast_2, repeats);

    double rel_f1 = mean_relative_error_inv_sqrt(vecs, N,
1);
```

```
    double rel_f2 = mean_relative_error_inv_sqrt(vecs, N, 2);
    double ang_f1 = mean_angular_error_deg(vecs, N, 1);
    double ang_f2 = mean_angular_error_deg(vecs, N, 2);

    printf("Avg time (ms): exact=%.3f, fast1=%.3f, fast2=%.3f\n", t_exact, t_f1, t_f2);
    printf("Speedup vs exact: fast1=%.2fx, fast2=%.2fx\n", t_exact/t_f1, t_exact/t_f2);
    printf("Mean relative error: fast1=%.8f, fast2=%.8f\n", rel_f1, rel_f2);
    printf("Mean angular error (deg): fast1=%.6f, fast2=%.6f\n", ang_f1, ang_f2);

    free(vecs);
    return 0;
}
```

VII. Acknowledgment

I would like to express my sincere gratitude to the circumstances that have allowed me to complete this paper for IF2123 Aljabar Linier dan Geometri. I also extend my deepest appreciation to the IF2123 lecturer, Dr. Ir. Rinaldi Munir, MT., for his clear and thorough teaching of the course material, which significantly eased the process of completing this paper. My thanks also go to all the assistants who supported the study of Linear Algebra dan Geometry.

Furthermore, I am grateful to my mom, my sister, my brother, my friends, and everyone else who provided valuable feedback and support during the preparation of this paper. It is my earnest hope that the methodologies explored, the implementations detailed, and the analytical insights presented within this paper will not only serve as a beneficial academic resource but will also contribute meaningfully to the broader understanding within our academic community. Specifically, I aspire for this work to be a guiding light for my fellow peers, illuminating the often intricate nuances of various algorithms, particularly those founded upon the elegant principles of vector normalization within computing, thereby enriching their comprehension and practical application in their own future endeavors.

References

[1] Munir, R. (2024). Vektor di Ruang Euclidean (bagian 2). Retrieved December 17, 2025, from https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2025-2026/Algeo-12-Vektor-di-Ruang-Euclidean-Bag2-2025.pdf

[2] Munir, R. (2024). Vektor di Ruang Euclidean (bagian 1). Retrieved December 22, 2025, from https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2025-2026/Algeo-11-Vektor-di-Ruang-Euclidean-Bag1-2025.pdf

[3] Munir, R. (2024). Vektor di Ruang Euclidean (bagian 3). Retrieved December 22, 2025, from https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2025-2026/Algeo-13-Vektor-di-Ruang-Euclidean-Bag3-2025.pdf

[4] Munir, R. (2024). Ruang Vektor Umum (bagian 1). Retrieved December 22, 2025, from https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2025-2026/Algeo-15-Ruang-vektor-umum-Bagian1-2025.pdf

[5] C. Lomont, “Fast inverse square root,” *Technical Report*, 2003.

[6] D. Eberly, “Approximations for Euclidean norm,” *Geometric Tools, LLC*, Technical Report, 2002.

[7] D. Goldberg, “What every computer scientist should know about floating-point arithmetic,” *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–48, Mar. 1991.

[8] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-Time Rendering*, 4th ed. Boca Raton, FL, USA: CRC Press, 2018.

[9] E. Lengyel, *Mathematics for 3D Game Programming and Computer Graphics*, 3rd ed. Boston, MA, USA: Cengage Learning, 2011.

[10] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*, IEEE Std 754-2019, 2019.

[11] J. Arvo, “Fast randomization techniques for vector normalization,” in *Graphics Gems II*, J. Arvo, Ed. San Diego, CA, USA: Academic Press, 1991, pp. 437–440.

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, 24 Desember 2025

Natanael Imandatua Manurung 13524021