Applications of Vector, Matrix, and Quaternion in Shaders Programming for Game Development Using Godot Engine

Muhammad Kinan Arkansyaddad – 13523152[1,2]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13523152@std.stei.itb.ac.id [2]mkinanarkansyaddad@gmail.com

***Abstract*— Applications of vectors, matrices, and quaternions in shader programming are crucial for enhancing game development within the Godot Engine. Vectors represent points and directions in 3D space, while matrices facilitate transformations such as translation, rotation, and scaling of objects. Quaternions offer a robust solution for handling rotations, effectively avoiding issues like gimbal lock that can arise with traditional methods. By integrating these mathematical constructs, developers can create visually rich and immersive gaming experiences, enabling dynamic interactions and realistic animations in real-time applications.**

***Keywords*—game development, Godot Engine, shaders, transformations.**

I. Introduction

Shaders are specialized programs that run on the GPU (Graphics Processing Unit), designed to control the rendering pipeline by manipulating how surfaces and materials appear in a scene. By defining how vertices and pixels are processed, shaders can achieve complex visual effects, including realistic lighting, shadows, reflections, and post-processing effects like bloom or motion blur. Shaders are integral to modern game development, offering a way to create immersive environments that engage players with visually rich experiences. From dynamic weather systems to detailed textures and particle systems, shaders enable developers to push the boundaries of graphical fidelity and performance optimization.

In game development, shaders are categorized into two main types: vertex shaders and fragment (or pixel) shaders. Vertex shaders handle the manipulation of vertex data, such as transformations and projections, while fragment shaders determine the color and texture of individual pixels. Beyond these, advanced shader types like geometry shaders and compute shaders provide even greater flexibility for developers. The use of shaders has become ubiquitous in creating not only photorealistic graphics but also stylized visuals, allowing developers to craft unique aesthetics tailored to specific game genres.

The Godot Engine, a popular open-source game development platform, offers robust support for shader programming through its Shader Language, which is heavily influenced by GLSL (OpenGL Shading Language). Godot also provides a Visual Shader Editor, which allows developers to create shaders using a node-based interface, making shader programming more accessible to those without prior coding experience. This dual approach ensures that both novice and experienced developers can harness the power of shaders to implement custom effects such as water ripples, global illumination, and procedural texturing. Moreover, Godot integrates shaders seamlessly into its rendering pipeline, enabling developers to apply them to 2D and 3D projects with minimal friction.

This paper explores the practical application of shader programming in game development, focusing on the use of linear algebraic concepts, including vectors, matrices, and quaternions, to create dynamic and visually compelling effects within the Godot Engine. By linking theoretical foundations to real world implementations, this work aims to provide a clear pathway for developers to harness the power of shaders and mathematical algorithms in their game development projects.

II. Theoretical Basis

A. Vector

Vectors are a fundamental mathematical tool used to represent quantities with both magnitude and direction. In 3D graphics, vector v is typically expressed as:

$$v = \begin{bmatrix} x \\ y \\ z \end{bmatrix},$$

where x, y, and z represent its components along the corresponding axes. In shaders, vectors are used to define positions, directions (e.g., light rays), and normals, which are critical for calculating how light interacts with surfaces.

The dot product of two vectors a and b is particularly important for determining angles between them, especially in lighting calculations. It is defined as:

$$a \cdot b = a_x b_x + a_y b_y + a_z b_z = \| a \| \| b \| \cos\theta.$$

In practice, this equation is used in shaders to compute diffuse lighting, where $\cos\theta$ determines the intensity of light hitting a surface based on the angle between the light direction and the normal surface. A higher value indicates that the surface is directly facing the light source, resulting in brighter

illumination.

The cross product, another essential vector operation, generates a vector perpendicular to two given vectors. This operation is critical in calculating tangent and bitangent vectors, which are used in normal mapping to simulate fine surface details such as bumps and grooves.

B. Matrices

Matrix is a rectangular array of numbers, symbols, or expressions, arranged in rows and columns. Matrices are typically denoted by uppercase letters, such as A, B, or C, and their dimensions are expressed as m×n, where m represents the number of rows and n represents the number of columns. Each element of a matrix is identified by its position, denoted as a_{ij}, where i is the row index and j is the column index. This structured format allows for efficient representation and manipulation of linear equations and transformations in various applications, including computer graphics and game development.

Transformation matrices are a specific type of matrix used to perform linear transformations on geometric data in computer graphics[1]. These matrices enable operations such as translation, rotation, and scaling, which are essential for manipulating 3D models within a scene. In shader programming, particularly within the graphics pipeline, transformation matrices are applied to the vertices of 3D objects to determine their final position and orientation in relation to the camera view. For instance, a 4x4 transformation matrix can represent a combination of these transformations, allowing for efficient computation and rendering of complex scenes.

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Fig 2.1 Rotation matrix examples
Source:https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2023-2024/Algeo-18-Ruang-vektor-umum-Bagian4-2023.pdf

C. Quaternion

Quaternions are a mathematical extension of complex numbers and are particularly useful in representing rotations in three-dimensional space. A quaternion is typically expressed in the form

$$q = a + bi + cj + dk$$

where a is the scalar component and b, c, d are vector components. This representation allows for a compact and efficient way to encode rotation, avoiding some of the pitfalls associated with other methods, such as Euler angles, which can suffer from gimbal lock. In the context of game development and shader programming, quaternions provide a robust framework for smoothly interpolating rotations and performing complex transformations on 3D objects.

When applying a quaternion to rotate a vector p in three-dimensional space, the resulting vector p′ can be calculated using the formula

$$p' = qpq^{-1}$$

where q^{-1} is the conjugate of the quaternion q. This operation effectively rotates the vector p around a specified axis defined by the quaternion. For example, if a vector p is rotated, the quaternion can be constructed like this

$$q = \cos\left(\frac{\theta}{2}\right) + \sin\left(\frac{\theta}{2}\right)u,$$

where u is the unit vector of the axis[2]. This formulation allows for efficient computation of the rotated vector, making quaternions a preferred choice in real-time graphics applications, such as those developed using the Godot Engine.

D. Shaders

Shaders are specialized programs that run on the GPU (Graphics Processing Unit) and are essential for rendering graphics in real-time applications, such as video games. In the context of the Godot Engine, shaders are used to control the rendering pipeline, allowing developers to define how objects are drawn on the screen. Shaders can be categorized into several types, including vertex shaders, fragment (or pixel) shaders, and compute shaders, each serving distinct roles in the graphics rendering process. Vertex shaders are responsible for processing vertex data, transforming 3D coordinates into 2D screen coordinates, while fragment shaders calculate the color and other attributes of each pixel, enabling detailed visual effects and textures.

In Godot, shaders are written in a shading language that is similar to GLSL (OpenGL Shading Language), providing developers with the flexibility to create custom visual effects tailored to their specific needs. The shader pipeline in Godot allows for the implementation of various techniques, such as normal mapping, which simulates detailed surface textures without increasing the polygon count, and dynamic lighting, which enhances realism by accurately simulating how light interacts with surfaces. By utilizing shaders, developers can achieve a wide range of visual styles and effects, from simple color adjustments to complex post-processing effects that significantly enhance the overall aesthetic of a game[3].

III. IMPLEMENTATION

A. Wave Effect Shaders

The wave effect shader demonstrates the fundamental application of vector mathematics and trigonometric functions to create dynamic surface deformation. At its core, the shader manipulates vertex positions using a sinusoidal wave equation

$$y(x,t) = A\sin(\omega x + \varphi t),$$

where A represents amplitude, ω defines spatial frequency, and φ determines temporal frequency. This equation drives the primary mesh deformation, creating smooth, wave-like motion across the surface. The implementation extends beyond simple vertex displacement by incorporating normal vector recalculation, essential for maintaining realistic lighting interactions.

The vertex transformation pipeline begins with the base vertex position and applies the wave displacement along the

normal vector direction. This approach ensures that the deformation follows the natural surface curvature of the mesh. The shader calculates the new vertex position using

$$\text{VERTEX}' = \text{VERTEX} + (\text{NORMAL} \times \text{wave_height}),$$

where wave height varies according to the sine function. Simultaneously, the shader updates normal vectors through

$$N = normalize(-\partial y/\partial x, 1, 0),$$

maintaining accurate light reflection across the deformed surface.

```
shader_type spatial;

uniform float amplitude : hint_range(0.0, 2.0) = 0.5;
uniform float frequency : hint_range(0.0, 5.0) = 1.0;
uniform float speed : hint_range(0.0, 5.0) = 2.0;
uniform vec4 wave_color : source_color = vec4(0.7, 0.8, 1.0, 1.0);

void vertex() {
    vec3 vertexPosition = VERTEX;
    float wave = amplitude * sin(frequency * vertexPosition.x + TIME * speed);
    VERTEX += NORMAL * wave;

    NORMAL = normalize(vec3(
        -amplitude * frequency * cos(frequency * vertexPosition.x + TIME * speed),
        1.0,
        0.0
    ));
}

void fragment() {
    float wave = amplitude * sin(frequency * VERTEX.x + TIME * speed);
    float wave_factor = (wave + amplitude) / (2.0 * amplitude);
    ALBEDO = wave_color.rgb * (0.8 + 0.2 * wave_factor);
    METALLIC = 0.0;
    ROUGHNESS = 0.2;
}
```

Fig 3.1 Wave effect shader implementation using Godot shader language

B. Rotation Shader Using Matrix Transformation

The rotation shader exemplifies efficient geometric transformation through matrix operations. The shader constructs a rotation matrix R(θ) for y-axis rotation using trigonometric functions:

$$R(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

This matrix preserves distances and angles while rotating points around the specified axis, maintaining the mesh's structural integrity during animation. The orthogonal nature of rotation matrices ensures that $RR^T = R^TR = I$ and det(R) = 1, properties that guarantee proper transformation behavior.

Implementation efficiency comes from the shader's use of matrix-vector multiplication for both vertex and normal transformation. Each vertex position undergoes transformation through v' = Rv, while normal vectors transform similarly but without translation components. The shader optimizes performance by computing the rotation matrix once per frame and reusing it across all vertices, reducing per-vertex computation overhead.

The shader extends basic rotation functionality by incorporating time-based animation. By linking the rotation angle to TIME * rotation_speed, it creates continuous, smooth rotation while maintaining precise control through the rotation_speed uniform.

```
shader_type spatial;

uniform float rotation_speed : hint_range(0.0, 5.0) = 1.0;
uniform vec4 base_color : source_color = vec4(1.0, 1.0, 1.0, 1.0);

void vertex() {
    float angle = TIME * rotation_speed;

    // Create rotation matrix
    mat3 rotationMatrix = mat3(
        vec3(cos(angle), 0.0, sin(angle)),
        vec3(0.0, 1.0, 0.0),
        vec3(-sin(angle), 0.0, cos(angle))
    );

    VERTEX = rotationMatrix * VERTEX;
    NORMAL = rotationMatrix * NORMAL;
}

void fragment() {
    ALBEDO = base_color.rgb;
    METALLIC = 0.4;
    ROUGHNESS = 0.3;
}
```

Fig 3.2 Rotation shader using matrix transformation implementation using Godot shader language

C. Water Surface Shader

The water surface shader represents the most complex implementation, combining multiple mathematical concepts to create realistic water movement. The shader employs wave superposition through the equation

$$W(x,z,t) = \Sigma_i A_i sin(k_i \cdot p + \omega_i t),$$

where multiple wave components combine to create complex surface patterns. Each wave component contributes unique amplitude (A_i), direction (k_i), and frequency (ω_i) characteristics, resulting in natural-looking water movement.

Normal vector calculation becomes particularly crucial for water surfaces, as it directly impacts light reflection and refraction effects. The shader computes normals using the gradient of the height field:

$$N = normalize(-\partial W/\partial x, 1, -\partial W/\partial z).$$

This calculation considers both primary and secondary wave contributions, ensuring accurate specular highlights. The implementation optimizes these calculations by using vector dot products for wave direction computation and careful normal vector renormalization.

```
shader_type spatial;

uniform float wave_speed : hint_range(0.0, 5.0) = 1.0;
uniform float wave_height : hint_range(0.0, 1.0) = 0.2;
uniform vec2 wave_direction = vec2(1.0, 1.0);
uniform vec4 water_color : source_color = vec4(0.0, 0.3, 0.7, 0.8);
uniform vec4 foam_color : source_color = vec4(1.0, 1.0, 1.0, 1.0);

void vertex() {
    vec3 vertex = VERTEX;
    float time = TIME * wave_speed;

    // Primary wave
    float wave1 = sin(dot(wave_direction, vertex.xz) + time) * wave_height;

    // Secondary wave
    vec2 perpDirection = vec2(-wave_direction.y, wave_direction.x);
    float wave2 = sin(dot(perpDirection, vertex.xz) + time * 1.2) * wave_height * 0.5;

    vertex.y += wave1 + wave2;

    // Calculate normal
    vec3 normal = normalize(vec3(
        -wave_height * wave_direction.x * cos(dot(wave_direction, vertex.xz) + time),
        1.0,
        -wave_height * wave_direction.y * cos(dot(wave_direction, vertex.xz) + time)
    ));

    VERTEX = vertex;
    NORMAL = normal;
}

void fragment() {
    float foam = clamp(pow(UV.y, 3.0), 0.0, 1.0);
    ALBEDO = mix(foam_color.rgb, water_color.rgb, foam);
    METALLIC = 0.1;
    ROUGHNESS = 0.2;
    ALPHA = water_color.a;
}
```

Fig 3.3 Water Surface Shader implementation using Godot shader language

D. Quaternion Deformation Shader

The quaternion deformation shader uses the mathematical concepts of quaternions to achieve dynamic and rotational mesh deformation. This technique introduces spatially and temporally dependent rotations to vertex positions and normal vectors, maintaining smooth and continuous deformation across the surface of the mesh. The approach involves combining quaternion operations such as multiplication and axis-angle conversion to create the desired transformations.

The deformation process begins by defining a deformation axis and computing rotation angles based on both spatial and temporal factors. Each vertex's position determines a spatial rotation angle proportional to its distance from the origin, while a temporal rotation angle introduces time-based animation. These angles are converted into quaternions using the axis-angle formula:

$$q = (axis \cdot sin(angle/2), cos(angle/2)).$$

A spatial quaternion and a temporal quaternion are calculated independently, then combined through quaternion multiplication to create a unified transformation.

Vertex positions are updated using the combined quaternion to ensure deformation follows the desired axis. The quaternion rotation formula is applied where the original vertex vector is rotated, resulting in smooth and continuous transformations. Normal vectors, critical for accurate lighting and shading, are recalculated using the same quaternion operation to maintain realistic reflections across the deformed surface.

The fragment shader enhances the visual output by computing a deformation intensity, measured as the distance between the original and deformed vertex positions. This deformation intensity is used to modulate the surface color, creating a gradient that highlights areas of significant transformation. Lighting parameters such as metallic and roughness are also adjusted to ensure consistency with the material properties.

```
shader_type spatial;

uniform float time_scale : hint_range(0.0, 5.0) = 1.0;
uniform float deform_strength : hint_range(0.0, 2.0) = 0.5;
uniform vec3 deform_axis = vec3(1.0, 1.0, 1.0);

// Quaternion multiplication
vec4 quat_mult(vec4 q1, vec4 q2) {
    return vec4(
        q1.w * q2.xyz + q2.w * q1.xyz + cross(q1.xyz, q2.xyz),
        q1.w * q2.w - dot(q1.xyz, q2.xyz)
    );
}

vec4 axis_angle_to_quat(vec3 axis, float angle) {
    float half_angle = angle * 0.5;
    return vec4(axis * sin(half_angle), cos(half_angle));
}

vec3 rotate_vector_by_quaternion(vec3 v, vec4 q) {
    vec4 q_conjugate = vec4(-q.xyz, q.w);
    vec4 v_quat = vec4(v, 0.0);
    vec4 result = quat_mult(quat_mult(q, v_quat), q_conjugate);
    return result.xyz;
}

void vertex() {
    vec3 normalized_axis = normalize(deform_axis);

    // Create position-dependent deformation
    float vertex_angle = length(VERTEX.xyz) * deform_strength;
    float time_angle = TIME * time_scale;

    // Combine spatial and temporal rotations
    vec4 spatial_quat = axis_angle_to_quat(normalized_axis, vertex_angle);
    vec4 time_quat = axis_angle_to_quat(normalized_axis, time_angle);
    vec4 combined_quat = quat_mult(spatial_quat, time_quat);

    // Apply deformation
    vec3 deformed_position = rotate_vector_by_quaternion(VERTEX, combined_quat);
    VERTEX = mix(VERTEX, deformed_position, deform_strength);

    // Update normals
    NORMAL = rotate_vector_by_quaternion(NORMAL, combined_quat);
}

void fragment() {
    float deform_amount = length(VERTEX - (MODEL_MATRIX * vec4(VERTEX, 1.0)).xyz);
    ALBEDO = vec3(0.5) + 0.5 * normalize(vec3(deform_amount));
    METALLIC = 0.4;
    ROUGHNESS = 0.6;
}
```

Fig 3.4 Quaternion deformation shader implementation using Godot shader language

IV. RESULT AND ANALYSIS

A. Wave Effect Shaders

Fig 4.1 Wave effect shader demonstration going up

Fig 4.2 Wave effect shader demonstration going down

The wave effect shader is a powerful tool that brings dynamic wave-like motion to mesh surfaces by utilizing sine waves to create visually captivating deformations. This shader is particularly beneficial for game developers looking to simulate the natural movement of various materials, such as flowing flags, billowing fabrics, or even gentle terrain undulations. By manipulating parameters like wave height (amplitude), frequency of the waves, and the speed of animation, developers can achieve a wide range of effects that enhance the realism and aesthetic appeal of their games.

For instance, when applied to cloth physics, the wave effect shader can simulate the way fabric reacts to wind or movement, creating a more immersive experience for players. Animated banners that flutter in the breeze can be brought to life, adding a layer of dynamism to the game environment. Additionally, this shader can be used to create simple water surfaces that ripple and flow, providing a visually pleasing alternative to more complex fluid dynamics simulations.

The versatility of the wave effect shader makes it an invaluable asset in a game developer's toolkit. Whether it's for enhancing the realism of a character's clothing, adding life to environmental elements, or creating engaging visual effects, this shader allows for creative expression and technical innovation, ultimately contributing to a richer gaming experience. By harnessing the power of wave motion, developers can transform static scenes into vibrant, interactive worlds that captivate players and draw them deeper into the game.

B. Rotation Shader Using Matrix Transformation

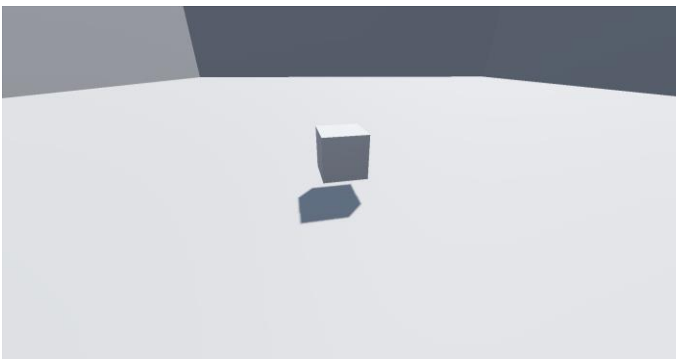

Fig 4.3 Rotation shader demonstration

The rotation shader is an essential tool in game development that enables the continuous rotation of objects through the use of matrix transformations. This shader is particularly effective for enhancing the visual appeal of various in-game elements, such as pickups, power-ups, and interactive objects, by drawing the player's attention to them. By implementing smooth, seamless rotation around an object's axis, the shader not only adds dynamism to the scene but also ensures that the lighting remains consistent and realistic through the proper rotation of normal vectors.

One of the most common applications of the rotation shader is in the creation of rotating collectibles. Imagine a shiny coin or a glowing orb that spins gently in place, enticing players to approach and interact with it. This captivating motion can significantly enhance the player's experience, making the game world feel more alive and engaging. Additionally, the shader is ideal for floating items, such as magical artifacts or power-ups, that need to maintain a constant rotation to convey a sense of energy and allure.

Moreover, the rotation shader is also well-suited for mechanical objects, such as gears, turbines, or robotic components, that require a continuous spinning motion to reflect their functionality. By incorporating this shader, developers can create visually striking animations that not only serve a practical purpose but also contribute to the overall aesthetic of the game.

In essence, the rotation shader is a versatile and powerful tool that allows developers to breathe life into their game environments. By utilizing this shader, they can create a sense of movement and interaction that captivates players, encouraging exploration and engagement with the game world. Whether it's a simple collectible or a complex mechanical device, the rotation shader plays a crucial role in enhancing the visual storytelling and immersive experience of modern games.

C. Water Surface Shader

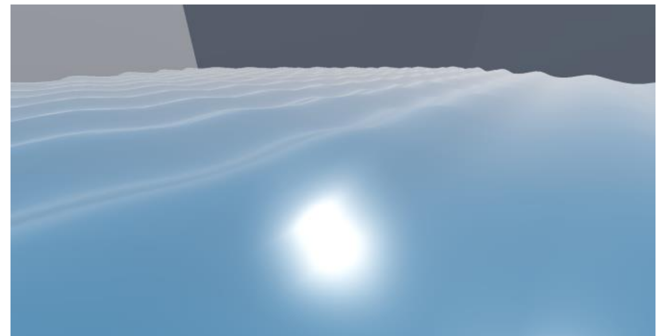

Fig 4.4 Water surface shader demonstration

The water surface shader is a tool designed to simulate realistic water surfaces, bringing a sense of authenticity and immersion to game environments. By combining multiple wave patterns and accurately calculating how light interacts with the water, this shader creates visually stunning representations of lakes, oceans, and other bodies of water that require dynamic wave movement.

In the realm of game development, the water surface shader is invaluable for crafting environments that feel alive and responsive. Whether it’s a serene lake reflecting the sky or a tumultuous ocean with crashing waves, this shader can adapt to various scenarios, enhancing the overall aesthetic of the game. One of its standout features is the ability to incorporate foam effects at the peaks of waves, adding an extra layer of realism that mimics how water behaves in nature. This detail is

particularly important for creating immersive experiences, as it helps players feel as though they are truly interacting with a living body of water.

Moreover, the shader offers developers the flexibility to customize wave directions, allowing for the simulation of everything from gentle ripples on a calm lake to the chaotic swells of a stormy sea. This adaptability makes it suitable for a wide range of environments and gameplay scenarios. Developers can easily adjust parameters such as wave height, speed, and even the color of the water to align with their game's unique art style and environmental conditions.

For instance, a tranquil, sunlit lake might feature soft, rolling waves with a clear blue hue, while a dark, stormy ocean could showcase towering waves with deep green and gray tones, reflecting the turbulent atmosphere. This level of customization not only enhances the visual fidelity of the game but also allows developers to create a more engaging and believable world for players to explore.

In summary, the water surface shader is a powerful asset in the toolkit of game developers, enabling them to create realistic and captivating water effects that significantly enhance the player's experience. By simulating the intricate behaviors of water and providing extensive customization options, this shader plays a crucial role in bringing game environments to life, making them more immersive and visually appealing.

D. Quaternion Deformation Shader

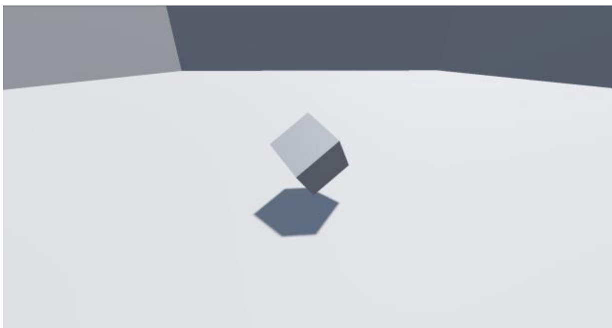

Fig 4.5 Quaternion deformation shader demonstration with 0.5 deformation strength and deformation axis of 1 for each axis

Quaternion deformation shader is a tool that enables the creation of organic, flowing deformations in 3D meshes, utilizing quaternion mathematics to intricately twist and warp vertices based on their distance from the center and the passage of time. Unlike traditional rotation shaders, which often produce straightforward and predictable movements, this shader generates complex and smooth deformations that can ripple through objects in a continuous and mesmerizing manner.

Game developers can use this shader to craft otherworldly effects that captivate players and enhance the immersive quality of their games. For instance, it can be used to design alien structures that pulse and breathe, giving them a life-like quality that draws players into the experience. Additionally, it is perfect for creating tentacles or appendages that writhe and move organically, adding a sense of realism to creatures that might otherwise feel static or lifeless.

The shader is particularly effective in genres that thrive on visual intrigue and surrealism, such as horror games, where objects need to transform unnaturally to evoke fear and tension. Imagine a scene where a seemingly ordinary object begins to twist and contort, creating an unsettling atmosphere that keeps players on edge. In fantasy games, it can be employed to depict shape-shifting creatures that fluidly change form, enhancing the magical and unpredictable nature of the game world. Similarly, in sci-fi settings, the shader can bring to life biomechanical or alien technology, creating visual effects that challenge the boundaries of reality.

For example, developers might use this shader to create a formidable boss monster that constantly morphs its shape, making it a dynamic and unpredictable adversary. Alternatively, it could be applied to an ancient magical artifact that twists and writhes with an otherworldly power, captivating players with its mysterious allure. Another exciting application could be a spatial anomaly that warps the geometry of everything nearby, creating a visually stunning and disorienting effect that enhances the sci-fi narrative.

One of the standout features of this shader is its ability to maintain smooth transitions, effectively avoiding issues like vertex tearing or sharp discontinuities that can detract from the visual experience. This quality makes it ideal for high-quality visual effects that need to appear fluid and organic, rather than mechanical or rigid. By incorporating this shader into their projects, developers can elevate the visual storytelling of their games, creating environments and characters that feel alive and responsive, ultimately enriching the player's journey through fantastical worlds.

V. CONCLUSION

The use of vector, matrix, and quaternion operations in shader programming plays a crucial role in game development, particularly within the Godot Engine. These mathematical concepts are essential for creating dynamic and visually compelling effects, optimizing rendering processes, and achieving smooth animations. Understanding their application allows developers to unlock new possibilities in game design, enhancing both performance and aesthetics. By integrating these concepts into real-time game environments, developers can create more immersive and interactive experiences, pushing the boundaries of what is achievable in modern game development.

VI. APPENDIX

Shader demonstration video:
https://drive.google.com/drive/folders/1MRQaNRgrC0LZ9E5Kq43McidSc_NqGcNp?usp=sharing
Project source code:
https://github.com/kin-ark/Learning-Shaders

VII. ACKNOWLEDGMENT

The author expresses gratitude to all parties who have assisted in the making of this paper, especially to:

1. Allah Swt.
2. Both parents, for providing moral and material support.

3. Friends who have encouraged and aided in the completion of this paper.
4. Dr. Judhi Santoso, M.Sc. and Arrival Dwi Sentosa, S.Kom, M.T. as the lecturers for the IF2123 Linear Algebra and Geometry course, for their invaluable guidance and support throughout the semester.

The author deeply appreciates all the assistance, encouragement, and kindness received from these individuals and groups, without which the completion of this paper would not have been possible.

REFERENCES

[1] Munir, Rinaldi. 2023. “Ruang Vektor Umum (bagian 4) dan Transformasi Linier”. https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2023-2024/Algeo-18-Ruang-vektor-umum-Bagian4-2023.pdf (accessed on 1 January 2025).
[2] Munir, Rinaldi. 2023. “Aljabar Quaternion (bagian 2)”. https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2023-2024/Algeo-26-Aljabar-Quaternion-Bagian2-2023.pdf (accessed on 1 January 2025).
[3] Godot Documentation. (2023). *Godot Engine Documentation*. https://docs.godotengine.org/ (accessed on 1 January 2025)

STATEMENT

I hereby declare that this paper I have written is my own work, not an adaptation or translation of someone else's paper, and not plagiarism.

Bandung, 1 January 2025

Muhammad Kinan Arkansyaddad
13523152