Application of Transformation Matrices for Generating 3D Game Maps

Ahmad Wicaksono - 13523121
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
ahmadwicaksono031004@gmail.com, 13523121@std.stei.itb.ac.id

***Abstract*—Transformation matrices play a crucial role in generating 3D maps for video games, providing the mathematical framework for essential operations such as translation, scaling, rotation, and projection within three-dimensional environments. This research investigates their application in the development of 3D game worlds, starting with a historical analysis of early implementations, including *Battlezone* (1980), a pioneering title that introduced fundamental 3D mapping techniques. The study examines the mathematical principles underpinning transformation matrices and explores their impact on rendering performance and visual detail in modern gaming. By integrating historical context with contemporary practices, this work aims to advance the understanding of 3D map generation, highlighting innovative approaches for improving efficiency and enhancing the realism of virtual spaces.**

***Keywords*—Translation, Scaling, Rotation, Projetion, Matrices.**

I. Introduction

The evolution of 3D game design has been marked by significant advancements in computational techniques, many of which stem from mathematical concepts. Among these, transformation matrices have emerged as a fundamental tool in shaping the environments within video games. The use of matrix manipulation in graphics originated in the 1960s, driven by innovators like Ivan Sutherland, who developed graphical systems capable of representing and transforming 3D space. These foundational advancements shaped our current understanding of spatial transformations and cemented the matrix as a fundamental tool in graphical computations..

Transformation matrices are mathematical structures that allow for the transformation of 3D objects within a virtual world. These transformations include operations like rotation, scaling, and translation, all of which enable developers to position and alter the geometry of objects. The power of transformation matrices lies in their ability to combine these operations into a single, efficient process. This capability is essential in real-time applications, where computational resources are limited, and smooth performance is critical. As games evolved, the role of transformation matrices expanded, enabling the creation of larger, more dynamic worlds that respond in real-time to player interactions.

A particularly interesting application of transformation matrices is in procedural generation, a method that uses algorithms to generate vast game worlds dynamically. This technique allows developers to create expansive and diverse environments without manually crafting every detail. Transformation matrices are applied throughout this process, as they define the positioning and manipulation of terrain and objects within these procedurally generated worlds. By combining randomness with mathematical precision, procedural generation provides players with expansive and ever-changing environments that would be impossible to create by hand.

However, the use of transformation matrices in large-scale game maps introduces significant challenges, particularly in the area of optimization. As the complexity of a game world increases, so too does the computational load of applying transformations to every object and structure. Optimization techniques are crucial for ensuring that these transformations are executed efficiently, maintaining the smoothness of gameplay. Developers often rely on methods such as simplifying calculations, reducing unnecessary transformations, and using pre-computed data to improve performance, ensuring that complex worlds can run on a variety of hardware without sacrificing quality.

The final key aspect of 3D game design that relies heavily on transformation matrices is rendering. Once transformations are applied to objects within a game world, the next step is rendering, where 3D objects are projected onto a 2D screen. This process involves multiple stages, such as perspective projection, to create the illusion of depth and distance in the game. The efficiency of transformation matrices plays a crucial role in the quality of this projection, affecting the realism of lighting, shadows, and the overall visual appeal of the game. Effective rendering ensures a smooth and visually engaging conversion of the 3D world into a 2D display.

This research focuses on exploring the role of transformation matrices in generating 3D game maps, emphasizing their application in procedural generation, optimization, and rendering techniques. By investigating the role of these mathematical tools in the development of modern game worlds, the study analyzes to highlight how transformation matrices not only facilitate the creation of interactive environments but also contribute to their optimization and visual presentation. Through this analysis, we are hoped to provide a deeper understanding of the mathematical foundations that underpin the cutting-edge experiences in 3D game design.

II. Theoretical Basis

A. Linear Transformation

In linear transformation is a function where **T:**$\boldsymbol{R^n} \rightarrow \boldsymbol{R^m}$is satisfies the following statement:

1. **T(x+y) = T(x) + T(y)**
2. **T(ax) = aT(x)**

for any vectors **x, y** ∈ $\boldsymbol{R^n}$ and any scalar a ∈ **R**.

Identifying f(**x**) is a linear transformation or not is by looking at each component of f(**x**). if each terms is a number times one of the components of **x**, then f is a linear transformation.

B. Transformation Matrices

linear transformation can be represented by matrices. If T is a linear transformation for $\boldsymbol{R^n} \rightarrow \boldsymbol{R^m}$ and x is a column of n, then

T(x) = A(x)

It is called as transformation matrix of T for some m x n matrix of A.

To find the transformation matrix A for a linear transformation T(x), one can apply T to each vector of the standard basis and use the resulting vectors as the columns of the matrix.

$$A = \begin{bmatrix} T(\mathbf{e}_1) & T(\mathbf{e}_2) & \cdots & T(\mathbf{e}_n) \end{bmatrix}$$

Figure 2.1. Matrix Column A (https://en.wikipedia.org/wiki/Transformation_matrix)

The matrix representation of vectors and operators varies depending on the chosen basis. A different basis will yield a similar matrix, but the process for determining the components remains consistent. For instance, a vector v can be expressed in terms of basis vectors,

$$E = \begin{bmatrix} \mathbf{e}_1 & \mathbf{e}_2 & \cdots & \mathbf{e}_n \end{bmatrix}$$

Figure 2.2. Basis Vectors (https://en.wikipedia.org/wiki/Transformation_matrix)

With vector coordinates:

$$[\mathbf{v}]_E = \begin{bmatrix} v_1 & v_2 & \cdots & v_n \end{bmatrix}^{\mathrm{T}}:$$

Figure 2.3. Vectors v Coordinates (https://en.wikipedia.org/wiki/Transformation_matrix)

$$\mathbf{v} = v_1\mathbf{e}_1 + v_2\mathbf{e}_2 + \cdots + v_n\mathbf{e}_n = \sum_i v_i\mathbf{e}_i = E[\mathbf{v}]_E$$

Figure 2.4. Sum Product of v and Basis Vectors (https://en.wikipedia.org/wiki/Transformation_matrix)

Now, expressing the result of the transformation matrix A upon V, in the given basis:

$$\begin{aligned} A(\mathbf{v}) &= A\left(\sum_i v_i\mathbf{e}_i\right) = \sum_i v_i A(\mathbf{e}_i) \\ &= \begin{bmatrix} A(\mathbf{e}_1) & A(\mathbf{e}_2) & \cdots & A(\mathbf{e}_n) \end{bmatrix}[\mathbf{v}]_E = A\cdot[\mathbf{v}]_E \\ &= \begin{bmatrix} \mathbf{e}_1 & \mathbf{e}_2 & \cdots & \mathbf{e}_n \end{bmatrix} \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \end{aligned}$$

Figure 2.5. Transformation Matrix A(v) (https://en.wikipedia.org/wiki/Transformation_matrix)

The most common geometric transformations that leave the origin unchanged are linear transformations, such as rotation, scaling, shearing, reflection, and orthogonal projection. Any affine transformation that is not solely a translation will have a fixed point, which can be set as the origin to convert the transformation into a linear one. In two dimensions, these linear transformations can be expressed using a 2×2 matrix.

1. Stretching

A stretch in the xy-plane is a linear transformation that magnifies distances in a specific direction by a constant factor while leaving distances in the perpendicular direction unchanged. Here, we focus on stretches along the x-axis and y-axis. A stretch along the x-axis can be described by the equations x′=kx and y′=y, where k is a positive constant. If k>1, the transformation results in a "stretch," while k<1 causes a "compression," though it is still referred to as a stretch. When k=1, the transformation becomes the identity transformation, meaning it has no effect.

The matrix stretchs by a factor *k* on the x-axis is given by:

$$\begin{bmatrix} k & 0 \\ 0 & 1 \end{bmatrix}$$

Figure 2.6. Matrix Stretching on X-Axis (https://en.wikipedia.org/wiki/Transformation_matrix)

Similarly, a stretch by a factor *k* on the y-axis has the form $x' = x$ and $y' = ky$, so the matrix associated with this transformation is

$$\begin{bmatrix} 1 & 0 \\ 0 & k \end{bmatrix}$$

Figure 2.7. Matrix Stretching on Y-Axis

(https://en.wikipedia.org/wiki/Transformation_matrix)

2. Rotation

For rotation by an angle **clockwise** about the origin the functional form is x'= xcosθ + ysinθ and y' = -xsinθ + ysinθ in a form of matrix become:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Figure 2.8. Matrix Clockwise Rotation (https://en.wikipedia.org/wiki/Transformation_matrix)

For a **counterclockwise**, the matrix become:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Figure 2.9. Matrix Counterclockwise Rotation (https://en.wikipedia.org/wiki/Transformation_matrix)

3. Shearing

A shear parallel to x axis has x' = x + ky and y' = y in matrix become:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Figure 2.10. Matrix Shear on X-Axis (https://en.wikipedia.org/wiki/Transformation_matrix)

Otherwise, a shear parallel to y axis has y' = y + kx and x' = x in matrix become:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ k & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Figure 2.11. Matrix Shear on Y-Axis (https://en.wikipedia.org/wiki/Transformation_matrix)

4. Translation

A translation matrix moves an object to one or more three axes. A matrix transformation representing the translation has the matrix form:

$$\mathbf{M} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 2.12. Matrix Translation (https://en.wikipedia.org/wiki/Transformation_matrix)

Applying translation to a point v reveals that matrix M adds the translation of tx, ty, and tz to the component of v and produce the translation and has the matrix form:

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \\ 1 \end{pmatrix} = \begin{pmatrix} v_x + t_x \\ v_y + t_y \\ v_z + t_z \\ 1 \end{pmatrix}$$

Figure 2.13. Product Result of Matrix Translation with the component of v (https://en.wikipedia.org/wiki/Transformation_matrix)

C. Ray Casting

Raycasting is a rendering technique that generate a 3D perspective from a 2D map. In the early days of computing, when hardware was much slower, real-time 3D engines were not feasible. Raycasting gives an efficient solution. It achieves high speed by performing calculations for each vertical line of the screen. The most famous game utilizing this method is Wolfenstein 3D.

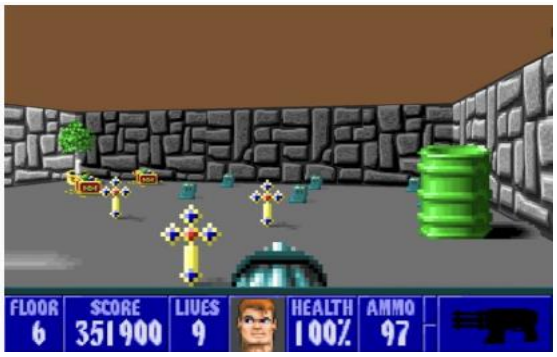


Figure 2.14. Wolfenstein 1990 Gameplay (https://online.oldgames.sk/play/dos/wolfenstein-3d/7739)

The raycasting engine in Wolfenstein 3D was highly simplified, enabling it to run even on a 286 computer. All walls in the game had the same height and were represented as orthogonal squares on a 2D grid, as illustrated in a map for Wolfenstein 3D.

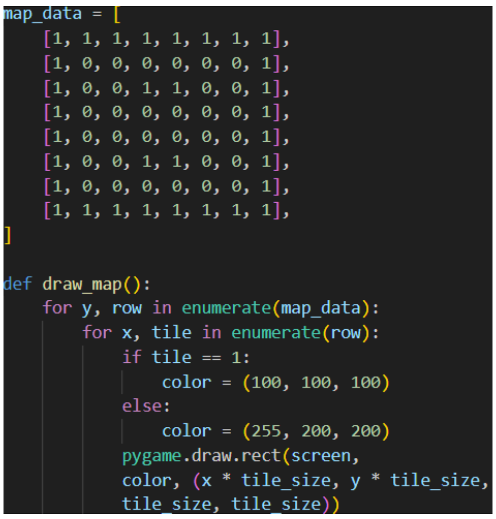
This engine had significant limitations, making features like stairs, jumping, or height variations impossible to implement. However, later games such as Doom and Duke Nukem 3D employed more advanced raycasting engines, introducing capabilities like sloped walls, varying heights, textured floors and ceilings, and transparent walls. In these games, sprites (enemies, items, and other objects) were represented as 2D images, though this tutorial does not delve into sprites.

It's important to note that raycasting is different with raytracing. Raycasting is a fast, semi-3D technique capable of running in real-time, even on devices with 4MHz graphical calculators. On the contrary, raytracing is a true 3D rendering technique that produces realistic visuals with reflections and shadows. Raytracing has only recently become viable for real-time use in high-resolution, complex scenes thanks to advancements in computing power.

III. Implementation Method

1. 2D Maps

2D map represents as a matrix where each element encodes attributes of corresponding spatial units. Formally, it is defined as **M[i][j]** where **i** and **j** represent row and column indices and the values state walls (1) and spaces (0). Rendering a 2D map involves iterating through the grid and visualizing each cell with distinct colors or patterns to reflect its state for player's position to enhance interactivity. The straightforward structure of 2D maps make it to be a critical tool for simulating environments, enabling spatial reasoning and analysis in both research and practical applications.

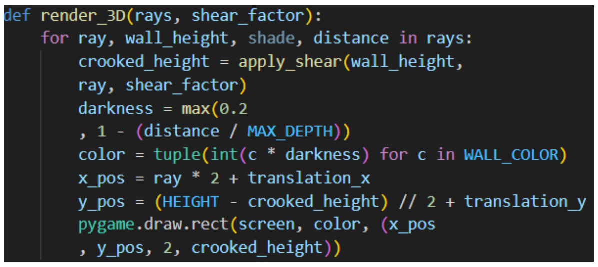


```
map_data = [
    [1, 1, 1, 1, 1, 1, 1, 1],
    [1, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 1, 1, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 1, 1, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 1],
    [1, 1, 1, 1, 1, 1, 1, 1],
]


def draw_map():
    for y, row in enumerate(map_data):
        for x, tile in enumerate(row):
            if tile == 1:
                color = (100, 100, 100)
            else:
                color = (255, 200, 200)
            pygame.draw.rect(screen,
            color, (x * tile_size, y * tile_size,
            tile_size, tile_size))
```

Figure 3.1. Example of 2D Maps
(Writer's Archive)

2. Rendering 3D Maps with Raycasting

The method of raycasting, as implemented in the provided render_3D function, enables the rendering of a three-dimensional environment from a two-dimensional map by projecting rays and calculating their interactions with walls. Each ray in the function represents a column of the screen, with its intersection determining the height and shading of a wall segment. The wall height is adjusted using the apply_shear function, which incorporates a shear factor to modify the visual perspective, simulating distortions or dynamic perspectives in the scene. Shading is calculated by reducing brightness proportionally to the distance of the ray's intersection, using a darkness factor to simulate lighting and depth perception. The computed wall segment is then rendered as a rectangle, positioned based on the ray's index and centered vertically using offsets. This approach efficiently simulates three-dimensional perspectives by processing only visible rays, aligning with the computational efficiency of raycasting while demonstrating practical integration into rendering.

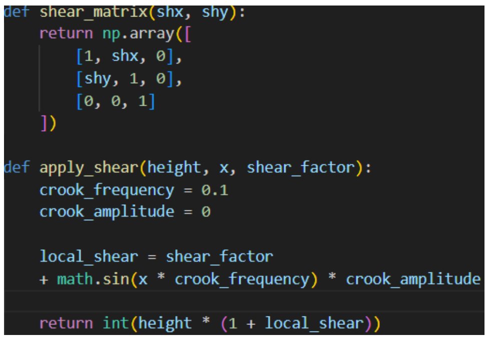


```
def render_3D(rays, shear_factor):
    for ray, wall_height, shade, distance in rays:
        crooked_height = apply_shear(wall_height,
        ray, shear_factor)
        darkness = max(0.2
        , 1 - (distance / MAX_DEPTH))
        color = tuple(int(c * darkness) for c in WALL_COLOR)
        x_pos = ray * 2 + translation_x
        y_pos = (HEIGHT - crooked_height) // 2 + translation_y
        pygame.draw.rect(screen, color, (x_pos
        , y_pos, 2, crooked_height))
```

Figure 3.2. Rendering 3D Maps
(Writer's Archive)

3. Shear Transformation

The shear transformation is applied using the shear matrix function. This matrix modifies the x-coordinates by a factor of shx × y and the y-coordinates by a factor of shy × x.

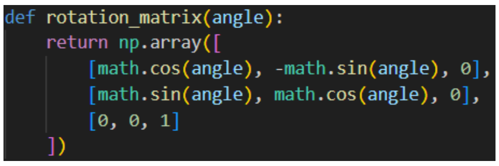


```
def shear_matrix(shx, shy):
    return np.array([
        [1, shx, 0],
        [shy, 1, 0],
        [0, 0, 1]
    ])

def apply_shear(height, x, shear_factor):
    crook_frequency = 0.1
    crook_amplitude = 0

    local_shear = shear_factor
    + math.sin(x * crook_frequency) * crook_amplitude

    return int(height * (1 + local_shear))
```

Figure 3.3. Shear Transformation Function
(Writer's Archive)

4. Rotation Transformation

The rotation transformation is handled by the rotation matrixfunction. It rotates points around the origin using trigonometric equations based on the given angle.

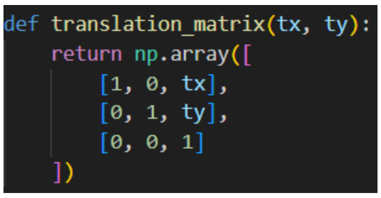


```
def rotation_matrix(angle):
    return np.array([
        [math.cos(angle), -math.sin(angle), 0],
        [math.sin(angle), math.cos(angle), 0],
        [0, 0, 1]
    ])
```

Figure 3.4. Rotation Matrix for Transformation
(Writer's Archive)

5. Translation Transformation

The translation is achieved with the translation matrix function. It shifts points in the 2D space by adding tx to the x-coordinate and ty to the y-coordinate to get a moving objects or the player's position better.

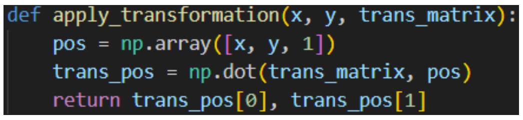


```
def translation_matrix(tx, ty):
    return np.array([
        [1, 0, tx],
        [0, 1, ty],
        [0, 0, 1]
    ])
```

Figure 3.5. Translation Matrix for Transformation
(Writer's Archive)

6. Application of Transformation to 3D Maps

In game, transformations are applied sequentially using the apply_transformation function. First, shear is applied to distort the space, followed by rotation to adjust the player's perspective, and finally translation to position the player correctly in the transformed environment. This order ensures a consistent and logical transformation.



```
def apply_transformation(x, y, trans_matrix):
    pos = np.array([x, y, 1])
    trans_pos = np.dot(trans_matrix, pos)
    return trans_pos[0], trans_pos[1]
```

Figure 3.6. Application of Transformation Matrices (Writer's Archive)

IV. Implementation Testing

1. 2D and 3D Maps Generator

The implementation testing of the 2D and 3D map generation focuses on evaluating the process of dynamically constructing and rendering both map types within the game environment. The 2D map generation involves creating a grid-based layout where tiles represent different types of terrain (e.g., walls, floors), providing a simple but effective way to organize the game world. The 3D map generation, on the other hand, uses raycasting techniques to simulate depth perception and produce a first-person perspective, where rays are cast from the player's position to detect collisions with walls and calculate the appropriate visual rendering based on distance. This testing ensures that both map generation methods function seamlessly, allowing for smooth transitions between the 2D grid and the immersive 3D view while maintaining accuracy in spatial representation.

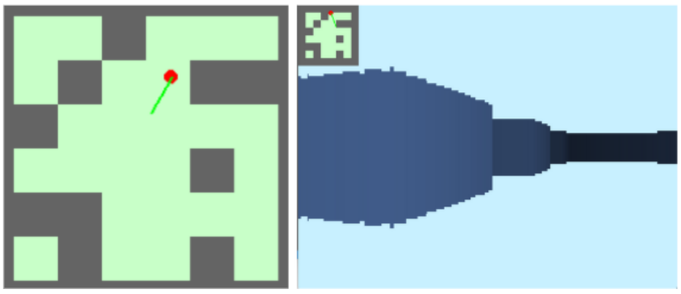


Figure 4.1. Generating 2D and 3D Maps (Writer's Archive)

2. Shear Effect

The walls should be rendered with a static color that is affected by distance but remains consistent across frames. The shear effect should subtly distort the height of the walls, making them appear skewed in y directions.

By changing the value of the shear factor, we can observe that the height of the walls appears subtly distorted along the **y**-axis. This means the walls will look like they're being "tilted" or "skewed" in the vertical direction, which can create a more dynamic or even a slanted perspective of the 3D environment.

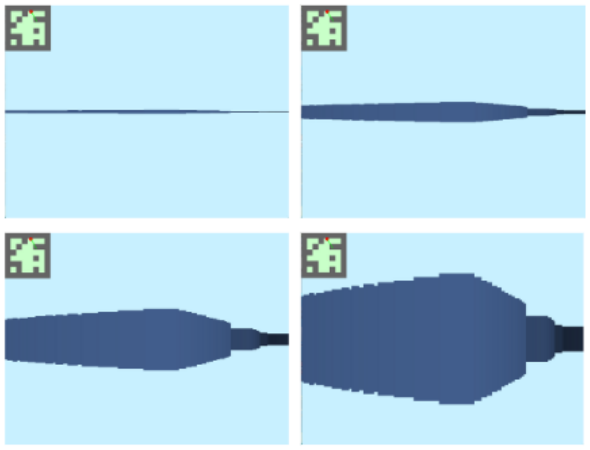


Figure 4.3. Shear Effect (Writer's Archive)

The shear effect has its value, which significantly influences the distortion applied to the 3D wall rendering. The shear factor controls the degree to which the walls are "stretched" or "compressed" horizontally based on the player's position or other dynamic factors in the environment. To test the shear factor, we use four distinct values of shear factor, each representing a different level of shear distortion:

1. **Shear Factor = 0**. This value represents no shear effect, meaning the walls will be rendered without any horizontal stretching or compression, maintaining their original proportions.
2. **Shear Factor = 0.1**. This value represents little shear effect.
3. **Shear Factor = 0.5**. A moderate shear value that introduces a slight horizontal distortion. This value will result in walls being stretched outward, creating a subtle perspective shift that mimics slight warping effects.
4. **Shear Factor = 1**. This value represents a more pronounced shear effect, with walls experiencing a more noticeable horizontal distortion. The walls will appear significantly stretched, simulating exaggerated perspective or fisheye-like effects.

3. Rotation Effect

The rotation effect is achieved by altering the angle at which rays are cast, based on the player's current view direction. As the player turns left or right, the starting angle of each ray changes, adjusting the perceived orientation of the walls. This creates the illusion that the player is rotating within a 3D environment. By calculating the angle for each ray relative to the player's current angle, the entire scene appears to rotate, with walls shifting positions accordingly. The rotation effect ensures that as the player moves their viewpoint, the surrounding environment adjusts in a natural, immersive manner, giving the feeling of turning in place.

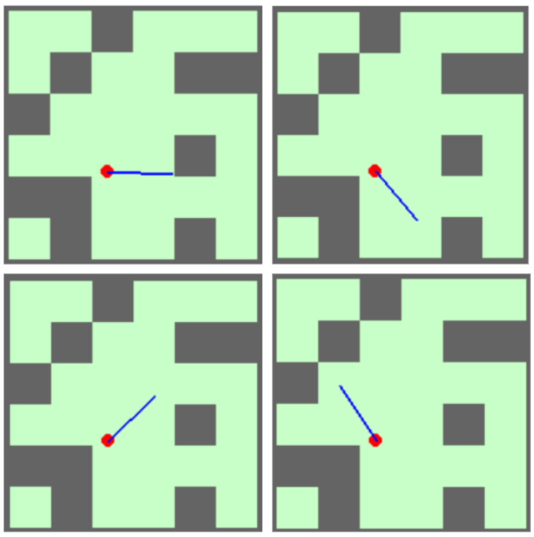
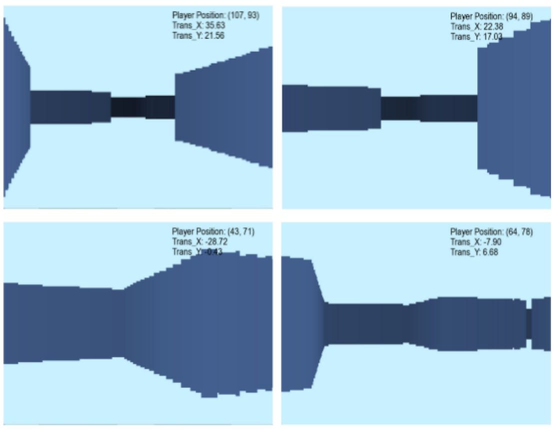


Figure 4.2. Rotation Effect
(Writer’s Archive)

Rotation testing was implemented at four angle values: **1.19°**, **49.89°**, **236.1°**, and **316.32°.** These angles were specifically chosen to evaluate the system’s ability to handle various levels of rotational movement in a 3D environment.

4. Translation Effect

The translation effect shifts the entire 3D environment horizontally or vertically, creating the sensation of moving through the scene. By modifying the translation_x and translation_y values, the world moves along with the player, making it feel as though the player is walking or flying through the space. This effect alters the position of the walls and other elements in the environment, pushing them along the x or y-axis, thereby providing a dynamic movement through the world. The translation effect is essential for creating an interactive exploration experience, allowing the player to navigate the scene while maintaining a consistent perspective.



```
Player Position: (107, 93)
Trans_X: 35.63
Trans_Y: 21.56
```

```
Player Position: (94, 89)
Trans_X: 22.38
Trans_Y: 17.03
```

```
Player Position: (43, 71)
Trans_X: -28.72
Trans_Y: -0.43
```

```
Player Position: (64, 78)
Trans_X: -7.90
Trans_Y: 6.68
```

Figure 4.4. Translation Effect
(Writer’s Archive)

V. Conclusion

The research and implementation of rotation, translation, and shear effects in 3D raycasting are particularly relevant for first-person perspective (FPS) games and immersive virtual environments. These types of games often require dynamic camera movements and environmental interactions that respond smoothly to player input. The rotation effect enhances the player’s ability to look around and explore the world, creating a sense of agency. The translation effect enables realistic movement through the game world, allowing the player to navigate environments in a fluid and intuitive way. The shear effect could be used in artistic or stylized games where the environment intentionally distorts to create unique visual experiences, or in procedural generation for creating dynamic, ever-changing landscapes. As these techniques improve, they could be incorporated into a wide range of games, including action-adventure games, simulation games, strategy games, and virtual reality (VR) experiences. The continuous advancement of these visual and gameplay techniques will be essential for the development of next-generation games that offer deep, interactive worlds..

VII. Acknowledgment

The writer would like to express heartfelt gratitude to God for His continuous blessings, wisdom, and guidance throughout this academic journey. Without His divine assistance, the completion of this research paper would not have been possible.

The writer would also like to appreciate the Linear and Geometric Algebra lecturers, Ir. Rila Mandala, M.Eng., Ph.D., and Dr. Ir. Rinaldi Munir, M.T., for dedication to teaching. Their expertise, patient guidance, and willingness to share knowledge have helped in shaping the writer's understanding of the subject.

The writer would like to express gratitude to family and friends who have been constantly supporting. Their love and support have made the completion of this research paper possible.

References

[1] Pygame 2.6.0 Release. [Online]. Available: https://www.pygame.org/news. [Accessed: December 28, 2024].

[2] "Matrix Transformation Hierarchy." [Online]. Available: https://groups.csail.mit.edu/graphics/classes/6.837/F03/lectures/05_transformation_hierarchy.pdf. [Accessed: December 30, 2025].

[3] "Linear Transformation Definition (Euclidean)." [Online]. Available: https://mathinsight.org/linear_transformation_definition_euclidean. [Accessed: December 31, 2024].

[4] "Spatial Transformation Matrices." [Online]. Available: https://www.brainvoyager.com/bv/doc/UsersGuide/CoordsAndTransforms/SpatialTransformationMatrices.html. [Accessed: January 1, 2025].

[5] Gentle, James E. "Matrix Transformations and Factorizations." In *Matrix Algebra: Theory, Computations, and Applications in Statistics*. Springer, 2007. ISBN 9780387708737.

[6] Jules, C. "2D Transformation Matrices Baking," Feb. 25, 2015.

[7] "Wolfenstein 3D." [Online]. Available: https://online.oldgames.sk/play/dos/wolfenstein-3d/7739. [Accessed: January 1, 2025].

PERNYATAAN

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

Bandung, 2 Januari 2025

[illegible]

Ahmad Wicaksono/13523121