Implementation of Bilinear Interpolation in Fast Approximation Anti-Aliasing (FXAA) for Basic Raster Graphics

Nicholas Andhika Lucas, 13523014[1,2]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13523014@std.stei.itb.ac.id, [2]realandhikalucas@gmail.com

***Abstract*—Fast Approximation Anti-Aliasing (FXAA) is a popular graphic setting choice for many people, notably in the gaming industry, as it provides a low cost and moderate quality of anti-aliasing. This study explores the implementation of bilinear interpolation in an FXAA algorithm to reduce aliasing in raster graphics. The algorithm is created using Python libraries, and the experiment tested its effectiveness using sample images ranging from simple shapes to complex spheres. The experiment demonstrated the bilinear interpolation capability in reducing the aliasing effects on the sample images in a particularly quick manner. Other findings include a limitation in contrast detection and blending capabilities compared to other algorithms.**

***Keywords*—bilinear interpolation, fast approximation anti-aliasing (FXAA), raster graphics**

I. Introduction

In this current age of advanced digital technology, many video games, namely triple-A games pursue high-end visuals that rely on precise visual rendering. This is done in hopes of satisfying a person’s need for realistic, immersive, and high-quality graphics. One of the leading factors in achieving this goal is by creating a smooth and unjagged visual, thereby reducing a “blocky and pixelated” graphic. This is done by utilizing anti-aliasing, a method to smoothen edges of diagonal lines or curves from images.

Multiple anti-aliasing techniques exist and are chosen by users depending on the quality and the processing time demanded by the user. Fast Approximate Anti-Aliasing (FXAA) stands out as one of the most popular choices for image rendering, namely in video games, because of the efficiency of the computation while still providing a high-quality end-result. FXAA typically uses a type of bilinear interpolation to process and reduce the aliasing effect in images (blending). Although uncommon, other interpolation methods may be used, such as linear and bicubic interpolation.

This study aims to implement the FXAA technique on a simple raster image using Python. Furthermore, this paper also experiments with the implementation of a weighted interpolation method in the FXAA algorithm. In doing so, this study hopes to analyze and evaluate the impact of the weighted interpolation method integrated into FXAA, providing insights into their suitability for different contexts.

II. Basic Theory

A. Raster Graphics

Raster graphics are images that are made of pixels containing a specific detail of color, commonly displayed in RGB. Pixels – the smallest addressable unit in an image – contain a specific value of color, that when combined creates a complete image, similar to mosaic. Raster graphics are very commonly found every day, since most mediums of graphics are based on pixels. For example, pictures taken with our phone, in photographs, and on websites.

Computer images generally store images in raster graphic format such as JPEG or PNG. Although versatile, raster graphics may appear “blocky” or “pixelated” when they are magnified. This is due to how raster graphics are stored, since they are resolution dependent – since it is based on pixel units – [1].

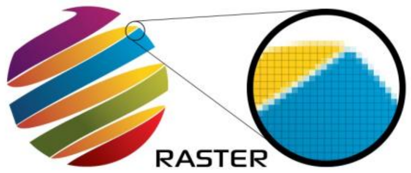


Image 1. Raster graphics. Source: printcnx.com.

B. Fast Approximation Anti-Aliasing (FXAA)

Anti-aliasing in context of digital image is any number of algorithms or techniques developed to reduce the jagged/aliased appearance of pixel-based images on a screen. First introduced in 1972, anti-aliasing has become a staple in determining graphic quality. It is essential in today’s age, where most graphics used digitally are raster graphics, images composed of pixels. The problem arises when defining an image in pixels. An image is bound to a certain resolution, therefore limiting the number of pixels to represent a single point. In lower resolutions, this limitation becomes more evident and may create a “blocky and pixelated” graphic, instead of a smooth appearance. This is where anti-aliasing comes into play and reduces the effect of this jaggedness.

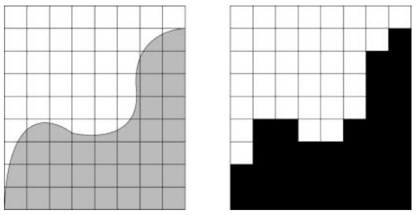


Fig 2. Example of limitations of representing a curved line in pixels. Source: vokigames.com

The principle of anti-aliasing is creating additional shades of pixels to create a smoother appearance on the borders of the image. The shades of pixels along the edges take reference from neighboring pixels to create a gradient that reduces the striking transition of colors which would create a jagged appearance. This is essential in representing curved lines in raster images [2].

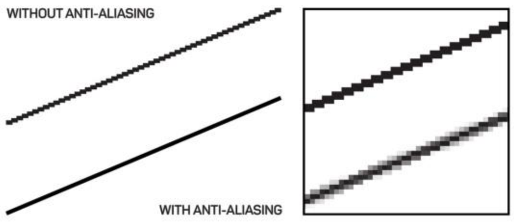


Fig 3. Anti-Aliasing. Source: vokigames.com

An anti-aliasing technique often used is Fast Approximation Anti-Aliasing (FXAA). This technique uses a high contrast filter to find points of images that are likely to have jagged edges, based on its contrast with neighboring pixels. Then, it blends the edges by sampling colors along the edges of a curve and averaging them. The benefits from FXAA come from its performance advantage, compared to other anti-aliasing methods, as it essentially blurs the edges of an image instead of calculating the colors and geometry directly. Although strong in performance, its downsides come in the form of lower to moderate image quality result [2].

According to Flick (2020), the algorithm of FXAA is as follows: First, the selected image is inputted. Then, the luminance data is acquired to find the contrast in color. This is done by approximating it using a green channel from the image. However, this is not always the best option since it is an approximation. Another option which is better is to directly calculate the luminance from the RGB value of the image or retrieve the data from the alpha channel. Next, the algorithm uses a high pass filter to compare the luminance of a pixel and its neighboring pixels as a threshold to determine if a pixel is contrasting enough. Lower contrasting pixels will be excluded from the FXAA algorithm. The threshold chosen can be adjusted accordingly to get the result desired. Then, the contrast between adjacent pixels is used to find the edge of a shape or image. The blending direction of a pixel will follow, perpendicular to the edge direction detected. Following this, the blending factor of a high contrasting pixel is analyzed from the luminance data of a pixel and its neighboring pixels, essentially in a grid. This is where different methods of interpolation come into play to determine the shade of the pixel to be added for blending. Finally, the algorithm detects the length and direction of an edge when it ends to determine its actual direction. The result produced will be a blending or smoothening effect along the edge of contrasting pixels [3].

C. Bilinear Interpolation

Bilinear interpolation is a type of interpolation that is extended from linear interpolation into 2D space. The difference lies in the approximation of a value by sampling four other coordinates or points of value. In simple terms, it can approximate a value that is located within the rectangular shape created by the 4 points of reference. It is also able to be done in one direction, then done in the other direction (e.g. x then y) [4].

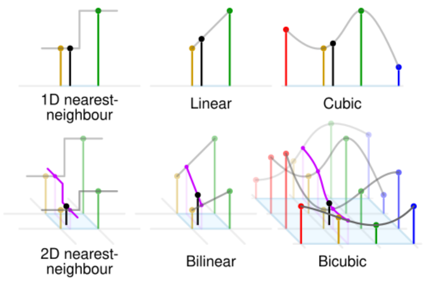


Fig 4. 1D and 2D Interpolation. Source: wikimedia.org

The equation is as follows. Suppose you have a rectangular grid made from 4 points in a 2D space, which includes (x_1, y_1), (x_2, y_1), (x_1, y_2), (x_2, y_2). To calculate the value at point (x, y) that lies within $x_1 \leq x \leq x_2$ and $y_1 \leq y \leq y_2$, the following formula is given:

$$f(x,y) = \frac{(x2-x)(y2-y)}{(x2-x1)(y2-y1)}Q11 + \frac{(x-x1)(y2-y)}{(x2-x1)(y2-y1)}Q21 + \frac{(x2-x)(y-y1)}{(x2-x1)(y2-y1)}Q12 + \frac{(x-x1)(y-y1)}{(x2-x1)(y2-y1)}Q22 \qquad (1)$$

where:
$f(x1, y1) = Q11$
$f(x2, y1) = Q21$
$f(x1, y2) = Q12$
$f(x2, y2) = Q22$

The figure below illustrates how bilinear interpolation approximates a value. In digital image processing, bilinear interpolation first considers the closest 2x2 grid of known pixel values that neighbors a pixel. Then, it takes the weighted average of the 4 pixels to calculate its final interpolated value. In the case where all known pixel distances are equal, the interpolation equation can be reduced to simply dividing their sum by the amount of known pixel points [5]. This is essential in the application of anti-aliasing.

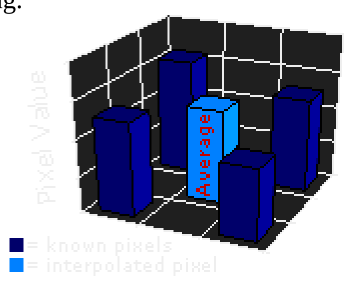


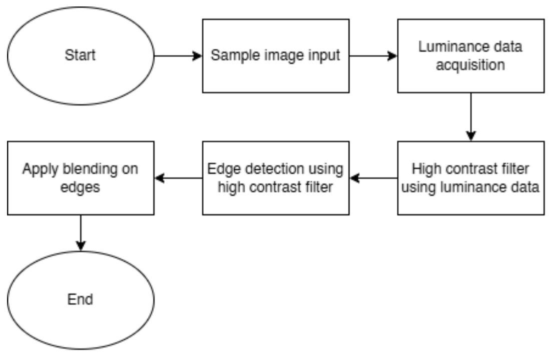
Fig 5. Bilinear Interpolation in Digital Image Processing. Source: cambridgeincolour.com

III. EXPERIMENT AND ANALYSIS

This experiment will utilize Python to model a simple Fast Approximation Anti-Aliasing (FXAA) algorithm to demonstrate the impact of bilinear interpolation in creating a smoother anti-aliasing effect. Several Python modules will be imported to assist the experiment, including *cv2*, *numpy*, and *matplotlib*. The algorithm used in the experiment will be adapted from Flick's guide – see reference [3] – but will be interpreted as his guide is made for implementation in OpenGL, whilst this experiment aims to only show the basic implementation using Python.

In reference to Flick, the main steps for recreating the FXAA algorithm is as follows:

1. Input a sample raster image.
2. Acquire the luminance data using RGB values.
3. Create a high contrast filter using the luminance data detected.
4. Detect parts that are categorized as edges based on its contrast value. Ignore lower contrasting pixels.
5. Apply the bilinear interpolation for blending the pixels on the edges.



Start
Sample image input
Luminance data acquisition
High contrast filter using luminance data
Edge detection using high contrast filter
Apply blending on edges
End

Fig 6. Experiment Flow. Source: Author (made in draw.io)

A. Sample Images

The samples used in the experiment utilizes Python's *numpy* library to generate a variety of images, ranging from a basic diagonal line to spirals and gradients. The images used are placed in a separate file called *sample.py*, while the main program, *fxaa.py* calls the generated images from the sample file. The samples are used to test the ability of the bilinear interpolation in reducing the effects of aliasing or jaggedness. Samples used have a resolution between 100x100 to 500x500 pixels.

B. Luminance Data Acquisition

```
def get_luminance(image):
    return 0.2126 * image[:, :, 2] + 0.7152 *
image[:, :, 1] + 0.0722 * image[:, :, 0]
```

In this function, the RGB image is converted to grayscale so the image can be processed based on its luminance value.

C. FXAA Algorithm

Ensuring the image processed is in float32 format

```
image = image.astype(np.float32) / 255.0
```

Contrast calculation

```
    contrast = np.zeros_like(luminance,
dtype=np.float32)
    for y in range(1, luminance.shape[0] - 1):
        for x in range(1, luminance.shape[1] - 1):
            m = luminance[y, x]  # Middle pixel
            n = luminance[y - 1, x]  # North neighbor
            s = luminance[y + 1, x]  # South neighbor
            e = luminance[y, x + 1]  # East neighbor
            w = luminance[y, x - 1]  # West neighbor

            highest = max(m, n, s, e, w)
            lowest = min(m, n, s, e, w)
            contrast[y, x] = highest – lowest
```

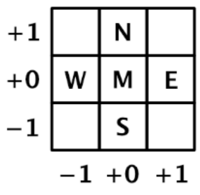


Fig 7. Contrast Calculation Illustration. Source: catlikecoding.com

In this function, FXAA uses the middle pixel and its direct horizontal and vertical neighbors to calculate the contrast. Compass directions are used to easily refer to the neighboring data. These 5 pixels are sampled to then be processed. The local contrast calculated between these pixels are the difference between the highest and the lowest luminance value. This local contrast value becomes the threshold for low and high contrast pixels. Therefore, this becomes a simple high pass filter.

Applying contrast threshold to the filter

```
    edge_threshold = 0.1
    edges = contrast > edge_threshold
```

A minimum threshold is set to differentiate which pixels should be processed using FXAA. In this experiment, the value 0,1 is set as the default threshold.

Applying bilinear interpolation

```
for y in range(1, image.shape[0] - 1):
        for x in range(1, image.shape[1] - 1):
            if edges[y, x]:
                # Sample neighboring pixels
                north = image[y - 1, x]
                south = image[y + 1, x]
                west = image[y, x - 1]
                east = image[y, x + 1]
                northwest = image[y - 1, x - 1]
                northeast = image[y - 1, x + 1]
                southwest = image[y + 1, x - 1]
                southeast = image[y + 1, x + 1]

                # Compute total weighted average
                local_avg = (
                    2 * north + 2 * south + 2 * west
+ 2 * east +
                    northwest + northeast + southwest
+ southeast
                ) / 12.0

                # Blend original and local average
                output[y, x] = np.clip(local_avg,
0.0, 1.0)
    return (output * 255).astype(np.uint8)
```

When a pixel passes the high contrast filter, it is then processed in the FXAA algorithm which implements the bilinear interpolation method. It is similar to the previous calculation to determine the local contrast, using the NEWS cross (neighboring pixels on horizontal and vertical ends). But, to sufficiently represent the neighborhood and achieve a more accurate result, the other four diagonal neighboring pixels are also counted in the calculation, described in the figure below.

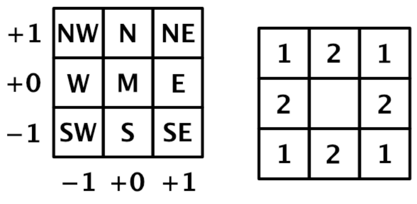


Fig 8 and 9. Blending Calculation Illustration. Source: catlikecoding.com

Pixels located at the northwest, northeast, southwest, and southeast of the middle pixel are accounted. But, diagonal neighbors are spatially further away from the middle pixel than that of the horizontal and vertical neighbors. This is why in Flick's variation of FXAA algorithm; weights are applied to show the important of the corresponding neighboring pixels. NESW neighbors have twice the amount of weight over the diagonal neighbors. The equation then becomes as such:

$$P = \frac{2 \cdot N + 2 \cdot S + 2 \cdot W + 2 \cdot E + NW + NE + SW + SE}{12} \qquad (2)$$

Weights reflect the importance of neighbors that are closer to the center pixel. The equation is simplified as the distance between the known points and the middle pixel is the same, which is 1 pixel.

D. Result

Simple Shapes

The samples that are tested using this FXAA algorithm that fall in the simple shapes category are diagonal lines, triangles, and stars.

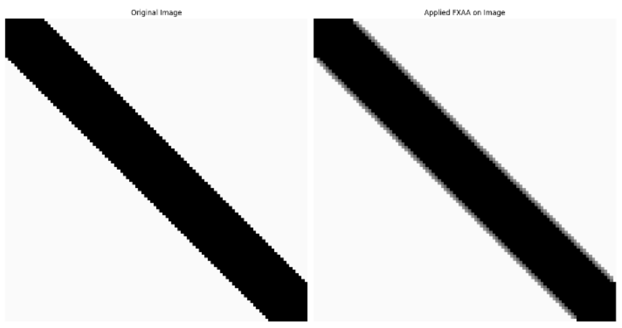


Fig 10. Diagonal Line 100x100 Sample. Source: Author (made with *matplotlib*)

Time required for processing is 0,01 seconds.

In this test, the anti-aliasing effects is shown to have properly blended the edges of the jagged shape. An observation to be made is that the blurriness from the anti-aliasing effect is much more present in lower resolution, but less visible in higher resolution. In this case, Figure 10 has a resolution of 100x100 pixels, while Figure 11 below has a resolution of 500x500 pixels.

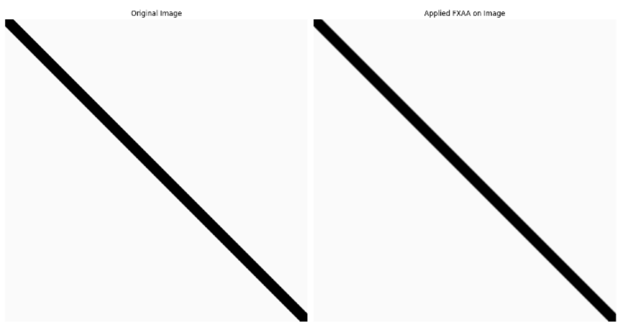


Fig 11. Diagonal Line 500x500 Sample. Source: Author (made with *matplotlib*)

Time required for processing: 0,29 seconds.

In the following samples tested, which are the triangle and star shape, the anti-aliasing effect is also able to blend the rough edges of the corresponding shape. This proves that the interpolation implemented works.

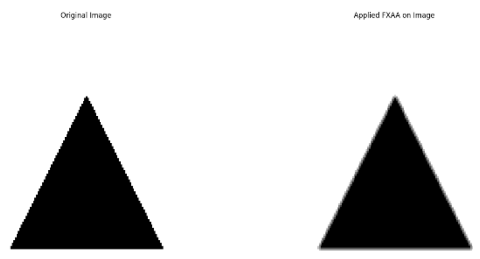


Fig 12. Triangle 200x200 Sample. Source: Author (made with *matplotlib*)

Time required for processing: 0,05 seconds.

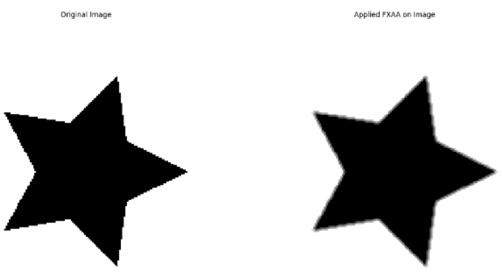


Fig 13. Star 200x200 Sample. Source: Author (made with *matplotlib*)

Time required for processing: 0,05 seconds.

Colored Graphics

In this type of sample, the FXAA algorithm's capabilites in detecting the low and high contrast pixels are tested. It is also used to test the capabilities of interpolation blending for pixels with a variety of RGB and luminance value.

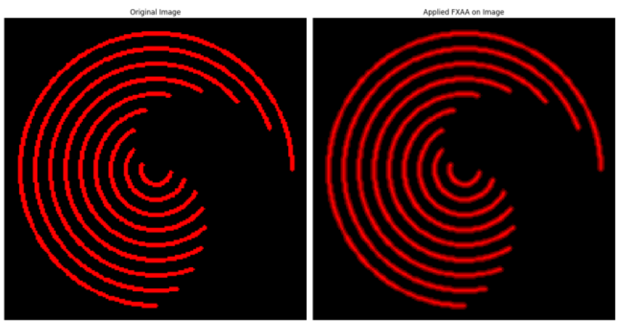


Fig 14. Colored Spiral 200x200 Sample. Source: Author (made with *matplotlib*)

Time required for processing: 0,14 seconds.

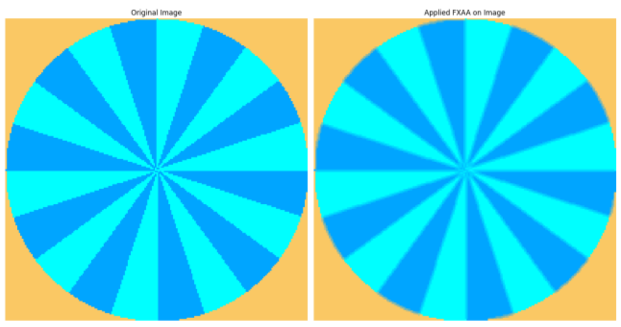


Fig 15. Colored Radial 200x200 Sample. Source: Author (made with *matplotlib*)

Time required for processing: 0,14 seconds.

The experiment shows that the FXAA algorithm is capable of applying its anti-aliasing effects, even on curves, odd shapes, and a mix of colors. However, noticable downsides include a lower quality of anti-aliasing, especially on round shapes, and lower quality of contrast detection. An experiment regarding the preferred threshold used for the algorithm might be a beneficial addition in the research.

Complex Shapes

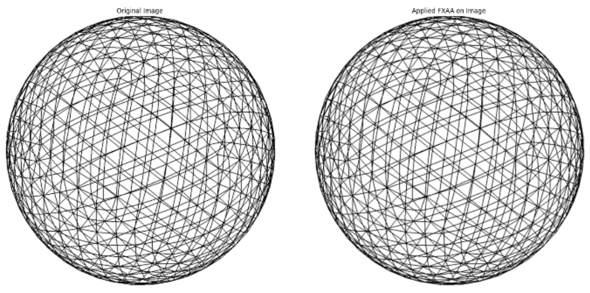


Fig 16. Icosphere. Source: Author (made with *pyvista*)

Time required for processing: 11,70 seconds.

Different from the previous experiments, this sample tested is aimed to test the capabilities of the FXAA Algorithm for very complex shapes with a higher resolution, in this case 1645x819 pixels. A noticable difference is the time required for processing, significantly higher than previous tests. This might be due to the amount of anti-aliasing effect applied on the image, based on the high contrast filter, topped with the higher resolution size. In hindsight, the anti-aliasing is able to create a smoother appearance on the complex sphere. A more noticeable appearance in the lower-quality of blending might be noticed if the picture is zoomed, compared to other interpolation methods, but still is sufficient in creating a better appearance than the original image.

Another observation found from this experiment is the obstacle found from applying anti-aliasing to images. Some interpolation applied minimize the sharpness of edges, but in cases where these sharpness are actually desired. The current FXAA method is unable to differentiate the important of these points and applies the anti-aliasing indiscriminately.

D. Conclusion

This experiment was conducted to implement the bilinear interpolation technique in a simple Fast Approximate Anti-Aliasing Algorithm. The objective of the experiment was to apply the interpolation method properly and evaluate the result, finding insights about its impact. The simple experiment produced an anti-aliasing effect that is able to smoothen jagged or aliased effects on pixels. This is done by applying the bilinear interpolation on high-contrast pixels to smoothen the transition in pixel shades. Insights found from the experiment are the FXAA algorithm's quick processing time and its moderate-quality of anti-aliasing effect, most significantly along diagonal and curved edges. Limitations or downsides that may come from this algorithm are fixed threshold for edge detection in some sample images and lower performance rate for higher complexity images. While the bilinear implementation succeeded, many improvements can be made to the experiment, including: testing a variety of contrast threshold, variation in edge detection, and experimenting a higher complexity interpolation method for blending.

VII. Acknowledgment

The author thanks all parties that have directly contributed to the creation of this paper. The author especially thanks Mr. Rinaldi Munir as the teacher of the 2024/2025 Linear Algebra and Geometry course, and Mr. Veriskt Mega Jaya whose suggestions in his research paper have inspired in the creation of this paper.

References

[1] GeeksforGeeks, "What is Raster Graphics?," geeksforgeeks.com. https://www.geeksforgeeks.org/raster-graphics/ (Accessed Jan. 1, 2025).

[2] L. Anna, "Anti-Aliasing in Gaming: The Battle for Perfect Graphics • VOKI Games," vokigames.com. https://vokigames.com/anti-aliasing-in-gaming-the-battle-for-perfect-graphics/ (Accessed Jan. 1, 2025).

[3] J. Flick, "FXAA," catlikecoding.com. https://catlikecoding.com/unity/tutorials/advanced-rendering/fxaa/ (Accessed Jan. 1, 2025).

[4] GeeksforGeeks, "Bilinear Interpolation." What is Bilinear Interpolation? - GeeksforGeeks (Accessed Jan. 1, 2025).

[5] Cambridge in Colour, "Understanding digital image interpolation," geeksforgeeks.com. cambridgeincolour.com. https://www.cambridgeincolour.com/tutorials/image-interpolation.htm (Accessed Jan. 1, 2025).

[6] V. Jaya, "Aplikasi Interpolasi Biliner pada Pengolahan Citra Digital," *Institut Teknologi Bandung*, 2014 (Accessed Jan. 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, 27 Desember 2024

[illegible]

Nicholas Andhika Lucas (13523014)