Application of Linear Algebra for Optimizing Collision Detection in Precision Parry-Based Combat Systems for Sekiro-like Games

M. Rayhan Farrukh, 13523035[1,2]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13523035@std.stei.itb.ac.id [2]rayhan.farrukh@gmail.com

***Abstract*—Collision detection is a crucial part of game development, paricularly in games requiring meticulous precision-based mechanics. This paper explores the application of linear algebra in optimizing collision detection algorithms, specifically *Axis-Aligned Bounding Box* (AABB), and *Separating Axis Theorems* (SAT). These algorithms were implemented in a simulation designed to showcase their integration into *Sekiro*-like parry-based combat system. AABB delivers efficient calculations for the broad-phase collision detection, while SAT is useful in narrow-phase collision detection for its precise calculations. These algorithms combined create a balance between performance and accuracy, crucial for implementing precise mechanics, like *Sekiro*'s combat system**

***Keywords*—Collision Detection, Linear Algebra, Sekiro-like, Video Games**

I. Introduction

Sekiro: Shadows Die Twice is an Action-Adventure Video Game developed by FromSoftware and published by Bandai Namco. It was released in 2019 and received critical acclaims and even winning The Game Award's *Game of The Year 2019* for its unique and revolutionary gameplay focusing on its combat system.

Sekiro was developed by FromSoftware under the direction of Hidetaka Miyazaki who is known for his revolutionary and visionary ideas for video games due to his previous work on games like *Bloodborne* and *Dark Souls*. Each game that was directed by Miyazaki has a signature arduous difficulty, focusing on relentless enemy NPC which makes the combat gameplay in these games stand out from an ocean of action-adventure games. However, within Miyazaki's catalog of critically praised video games, *Sekiro* stands out from the rest due to its combat's focus on *precision parrying*.

This combat system offers a truly unique experience, blending intensity and fluidity with a fast-paced, almost songlike rhythm that keeps player engaged. The precision parrying mechanic is the centerpiece of the combat's design, demanding impeccable timing and accuracy. For this system to work, it requires an exceptionally precise collision detection to ensure each parry feels responsive and realistic. This is where the world of *Sekiro* meets linear algebra to deliver an awesome experience.

Sekiro has not only set a high bar for actio-adventure video games but has also inspired a new wave of similar games that expands upon its revolutionary combat system to emulate its rhythm-like precision gameplay. By leveraging concepts of linear algebra, the developers of such games can refine their games, to ensure the precision required to make these games work is implemented optimally.

II. Theoretical Foundation

A. Vector

A *vector* is a mathematical term that refers to objects which has a magnitude and direction. Vectors are tipically represented as arrows in a coordinate space where the length corresponds to magnitude and the axis orientation indicates the direction. A vector is represented as:

$$\text{v} = (x, y)$$

Where x is the component along the horizontal axis, and y is the component along the vertical axis.Vectors are crucial for spatial representation and transformations and commonly used in fields such as physics, computer graphics, navigation, etc.

B. Vector Arithmetic

Vector arithmetic are mathematical operations performed on vectors, such as vector addition, subtraction, and projection. In collision detection, vector arithmetic is used to calculate relationships between objects in a space, such as their position, distances, and orientations. Key operations for collision detection include:

1. Vector Subtraction (difference)
   Used to compute displacement between two points. For example, the vector from one objects's center to another can be calculated as:

   $$d = p_2 - p_1 \quad (1)$$

   where p_1 and p_2 are the vectors of position of the two objects.
2. Dot Product

Measures the alginment of two vectors. Dot product is often used in *Separating Axis Theorem* to project shapes onto an axis to calculate overlaps. For vectors **v** = (x_1,y_1) and **u** = (x_2,y_2), the dot product is calculated as:

$$\text{v} \cdot \text{u} = x_1 x_2 + y_1 y_2 \quad (2)$$

3. Magnitude
The *magnitude* of a vector |**v**| represents its length or size. For a vector **v** = (x,y), its magnitude is defined as:

$$|\text{v}| = \sqrt{x^2 + y^2} \quad (3)$$

This measures straight-line distance from the vector's origin to its endpoint in a coordinate space. The magnitude of a vector can be used to *normalize* a vector. A normalized or unit vector is a vector with a magnitude of 1. For a vector **v** it is calculated as such:

$$\text{v} = \frac{\text{v}}{|\text{v}|} \quad (4)$$

4. Distance of two vectors
The *distance* between two points represented as vectors, is calculated using the magnitude of their difference:

$$\text{distance} = |p_2 - p_1| \quad (5)$$

The distance is particularly useful in broad-phase collision detection for determining whether two objects are close enough to detect potential overlap. For example, in center-based *Axis-Aligned Bounding Box,* the distance along each axis is compared to the sum of half their widths and heights. This allows for quick way to detect potential collision.

C. Projection

A *projection* is a mapping of a vector onto another vector, by calculating only the part of the vector that aligns with the direction of another. This operation helps simplify calculations by focusing only on the parts shared directionally between two vectors. The projection between two vectors is calculated as:

$$proj = \frac{\text{v}\cdot\text{u}}{|\text{u}|^2}\text{u} \quad (6)$$

where **v** is the vector being projected onto **u**. This formula measures how much of **v** lies in the direction of **u**. If u is a unit vector, then the equation is simplified to:

$$proj = (\text{v} \cdot \text{u}) \cdot \text{u} \quad (7)$$

where $(\text{v} \cdot \text{u})$ reperesent the scalar projection (magnitude) of **v** onto **u**. Which when multiplied again with **u**, it gives the directional projection, aligning the magnitude with the direction of **u**.

D. Collision Detection

Collision detection is the proccess of determining whether two or more object instersect or touch. In the context of game environment this is crucial for creating realistic interactions, like keeping characters from walking through walls or clipping through the map. In practice, collision detection is divided into two phases: broad-phase and narrow-phase.

In broad-phase collision detection, the goal is to quickly identify potention collisions by checking whether two objects are close to each other. This phase usually utilize simplified shape forms like bounding boxes for efficiency. Once potential collisions are identified the narrow-phase system is called, performing detailed calculations to confirm the actual collision of the two objects. An example of a broad-phase collision detection algorithm is *Axis-Aligned Bounding Box* (AABB), whereas an example of algorithm for narrow-phase is *Separating Axis Theorem* (SAT). The combination of these two phases ensures that collision detection remains efficient while maintaining precision.

E. Axis-Aligned Bounding Box (AABB)

A bounding box is a rectangular shaped boundary that encloses an object in a dimension. It is useful for simplifying complex shapes into manageable rectangles so that it can be used efficiently for tasks like collision detection.

An *Axis-Aligned Bounding Box* is a bounding box where the sides are aligned to the coordinate axes. In other words, it is a bounding box that does not rotate. An AABB is typically defined by two points, the minimum corner (x_{min}, y_{min}), and the maximum corner (x_{max}, y_{max}). Alternatively, it can be represented as a single point for its center coordinate along with width and height. Collision checks using AABB involve determining whether the boxes overlap on both the *x*-axis and the *y*-axis, which is done with comparisons of the edges position. This makes it advantageous to utilize in broad-phase collision detection due to its low computational cost.

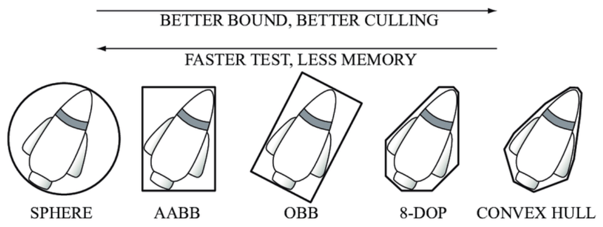


Figure 1. Illustration of AABB Compared to Other Bounding Shapes
Source: https://www.researchgate.net/bounding-shapes.png

F. SAT

Separating Axis Theorem (SAT) is a popular algorithm used for the narrow-phase of collision detection. It works by projecting the shapes onto a series of axes and checking if their projections overlap, if all axes overlap, then collision occurs. Conversely if there is even one axis where the projection do not overlap, then no collision occurs. The axes used are the *normals* of each edge of the object. SAT is used specifically for convex shapes. A convex shape is a shape whereby for any line drawn through the shape, only two points will ever intersect with that line. Although SAT only work for convex shapes, it *can* be used for non-convex shapes also with a little modification. Non-convex shapes can be decomposed into a combination of convex shapes.

SAT is particularly great for narrow-phase collision detection because it provides precise testing, and is usable on more than a bounding box, as SAT is able to detect collision on irregular convex polygons and other shapes that cannot be enclosed by simple bounding box like AABB. Additionally, SAT can calculate the *Minimum Translation Vector* (MTV), which is the smallest adjustment required to separate overlapping objects, this makes it useful for realistic collision response in video games.

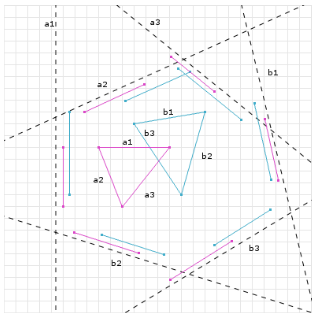


Figure 2. Illustration of Collision in SAT
Source: https://dyn4j.org/assets/posts/2010-01-01-sat-separating-axis-theorem/sat-ex-3.png

III. Methodology

The experiment in this paper will be focused on developing and analyzing the results of a program to simulate and demonstrate the proccess of collision detection algorithms. The simulation will be limited to interactions of objects within 2 dimensional space as they are computationally less complex and more accessible for visualization. This allows for the focus to stay on simulating the algorithms and less on video game development

As explained in the previous chapter, the two algorithms that will be used for the collision detection simulation are *Axis-Aligned Bounding Box* (AABB) and *Separating Axis Theorem* (SAT). The two algorithms are used because they are among the most popular collision detection algorithms with AABB's simplicity complementing SAT's precision to create a balanced collision detection system.

To emphasize each algorithms and also for modularity of the program, it will be separated into three parts:

1. AABB Simulation, to illustrate the workings of AABB algorithm and the simplicity of broad-phase collision detection.
2. SAT Simulation, to demonstrate how SAT algorithm works and how it is used in the narro-phase of collision detection.
3. Main Game, to demonstrate how AABB and SAT can be combined to create a robust parry-based combat system.

Each of these simulations will involve interactive elements where users (or players) can manipulate 2D objects within the simulation to observe how the algorithm detect collisions in real-time. Additionally, each simulation will also display visualization of vectors calculations, providing insight into how mathematical computations are performed when collision occurs.

For the testing of the simulation program, both algorithms will be analyzed in detail, highlighting their difference in terms of efficiency and accuracy with more insights and how they can be used optimally to make a better parry-based combat system in games and ultimately how these algorithms can make for a better gameplay experience for games in the *Sekiro*-like genre.

IV. Implementation

For the implementation of the simulation, the programming language that will be used is Python, with the Pygame library for game logic and visualization, along with NumPy for efficient mathematical calculations. As mentioned in section III, the simulation will be structured into three main modules, each focusing on different aspects of of collision detection: AABB Simulation, SAT Simulation, and the Main Game. These modules are designed to work both independently and cohesively, providing clear demonstration fo the algorithms principles. Besides the three main modules, there are supplementary modules to cover the game logic shared between the main modules. Here is how each module is to be implemented:

A. *Game Logic*

This module handles the behaviour and representation of objects used in the simulation. The object that will be used for the simulation is a pentagon as the core shape paired with a bounding box for AABB representation. Collisions are calcualted using the bounding box for AABB, while the SAT collisions will be calculated using the pentagon.

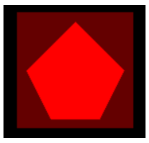


Figure 3.The Shape Used in the Simulation
Source: Author's Document

The bounding box is calculated as a rectangle that encloses the pentagon. Although AABB tipically tightly encloses the object, the boudning box used in the simulation has a tiny amount of offset as padding to account for game logic (which will be explained in *Main Game* section later). Although in the AABB collision calculation the center point representation is used, for the visualization around the pentagon, the bounding box is defined by its minimum and maximum x and y coordinates:

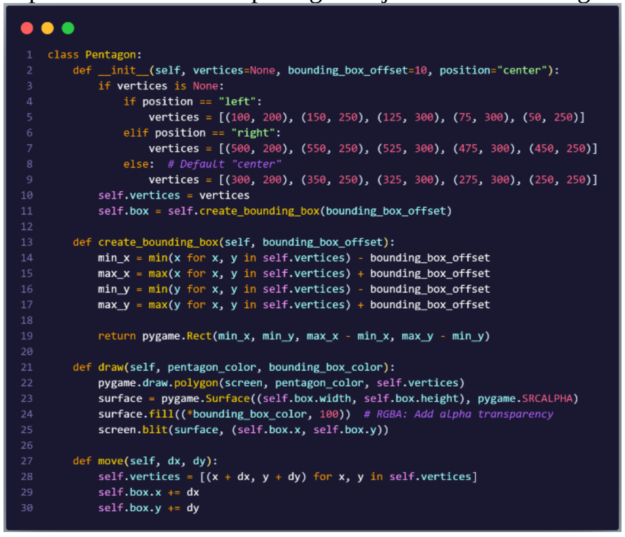
$$min_x = min_{x_i} - \text{offset}, \quad max_x = max_{x_i} + \text{offset}$$

$$min_y = min_{y_i} - \text{offset}, \quad max_y = max_{y_i} + \text{offset}$$

where (x_i,y_i) are the vertices of the pentagon. The bounding box is then represented as:

$$box = Rect(min_x, min_y, max_x - min_x, max_y - min_y)$$

where *Rect* refers to a rectangle defined by its top-left corner (*min_x*,*min_y*), and its width and height. The code implementation for the pentagon object is shown in Fig. 3.



```python
class Pentagon:
    def __init__(self, vertices=None, bounding_box_offset=10, position="center"):
        if vertices is None:
            if position == "left":
                vertices = [(100, 200), (150, 250), (125, 300), (75, 300), (50, 250)]
            elif position == "right":
                vertices = [(500, 200), (550, 250), (525, 300), (475, 300), (450, 250)]
            else:  # Default "center"
                vertices = [(300, 200), (350, 250), (325, 300), (275, 300), (250, 250)]
        self.vertices = vertices
        self.box = self.create_bounding_box(bounding_box_offset)

    def create_bounding_box(self, bounding_box_offset):
        min_x = min(x for x, y in self.vertices) - bounding_box_offset
        max_x = max(x for x, y in self.vertices) + bounding_box_offset
        min_y = min(y for x, y in self.vertices) - bounding_box_offset
        max_y = max(y for x, y in self.vertices) + bounding_box_offset

        return pygame.Rect(min_x, min_y, max_x - min_x, max_y - min_y)

    def draw(self, pentagon_color, bounding_box_color):
        pygame.draw.polygon(screen, pentagon_color, self.vertices)
        surface = pygame.Surface((self.box.width, self.box.height), pygame.SRCALPHA)
        surface.fill((*bounding_box_color, 100))  # RGBA: Add alpha transparency
        screen.blit(surface, (self.box.x, self.box.y))

    def move(self, dx, dy):
        self.vertices = [(x + dx, y + dy) for x, y in self.vertices]
        self.box.x += dx
        self.box.y += dy
```

Figure 4. Implementation of Pentagon Object
Source: Author's Document

Moving forward, for the sake of brevity, implementation details related to the game logic (except for the *Main Game)* will not be discussed in this paper, as they are not directly relevant to the main focus. The full source code can be found in the appendix for further reference.

B. *AABB Simulation*

The *Axis-Aligned Bounding Box* (AABB) algorithm is used for efficient broad-phase collision detection in the simulation. Collision is detected by comparing the positions and dimensions between bounding boxes of different objects. The detection is calculated using the distance between the center point of different objects, and also the total sum of their half sizes.
To calculate the relative positions, the center of each bounding box is defined by:

$$\text{center}_x = x + \frac{width}{2}, \quad \text{center}_y = y + \frac{height}{2}$$

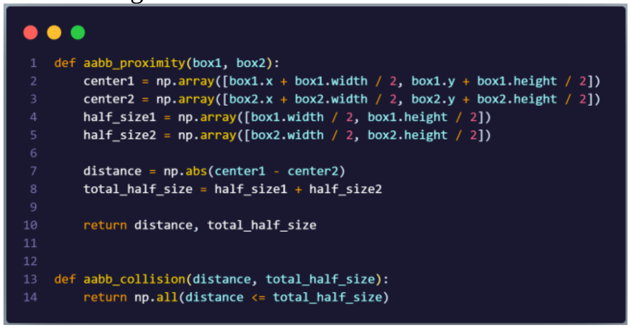
where (*x*,*y*) represents the point of the top-left corner of the box. As for the half-sizes, it is computed by:

$$\text{halfsize}_x = \frac{width}{2}, \quad \text{halfsize}_y = \frac{height}{2}$$

The distance will then be calculated according to (5), with the distances of the center along the x-axis and the y-axis being calculated separately. The collision is then calculated by comparing these distances to the sum of the half sizes of the objects for which collision is checked for. Collusion occurs if this condition is satisfied:

$$\text{distance} \leq \text{halfsize}_1 + \text{halfsize}_2 \quad (8)$$

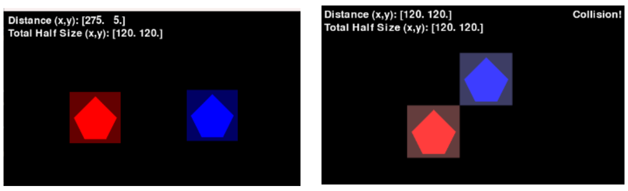
where the conditions should be fulfilled by both the x-axis and the y-axis. Implementation of AABB collision is shown in Fig. 4.



```python
def aabb_proximity(box1, box2):
    center1 = np.array([box1.x + box1.width / 2, box1.y + box1.height / 2])
    center2 = np.array([box2.x + box2.width / 2, box2.y + box2.height / 2])
    half_size1 = np.array([box1.width / 2, box1.height / 2])
    half_size2 = np.array([box2.width / 2, box2.height / 2])

    distance = np.abs(center1 - center2)
    total_half_size = half_size1 + half_size2

    return distance, total_half_size


def aabb_collision(distance, total_half_size):
    return np.all(distance <= total_half_size)
```

Figure 5. Implementation of AABB Collision
Source: Author's Document

To better demonstrate the proccess of AABB algorithm, a visualization program was created using Pythons's Pygame library. The program features two pentagons, colored red and blue, which can be interactively moved around by the user to simulate collisions. The program also displays the distance between the center of the two pentagons along with their total half size, for both the x and y axes. On top of that, a collision alert is shown, with the text "Collision!" appearing top right while the color of the bounding boxes lightup if collision occurs.



```
Distance (x,y): [275.  5.]
Total Half Size (x,y): [120. 120.]
```

```
Distance (x,y): [120. 120.]                Collision!
Total Half Size (x,y): [120. 120.]
```

(a) (b)

Figure 6. (a) The Program's Display Under Normal Conditions. (b) The Program's Display When Collision Occurs.
Source: Author's Document

C. *SAT Simulation*

The *Separating Axis Theorem* (SAT) is used for precise detection for collisions between the convex shapes (in this case the pentagons) as opposed to the bounding boxes. SAT uses projections of the shapes onto *separating axes*, to test if they all overlap. This algorithm simplify collision checking by reducing the shapes into 1D lines to compare along the axes.

In this simulation, SAT is implemented by first calculating the *normals* of each edge of the pentagon. Here, the normal is a vector perpendicular to the edge, which is used as the separating axes to ensure the projection are performed along critical directions for detecting overlaps. The pentagon will then be projected onto each normals to check for gaps between the projections. If no gaps are

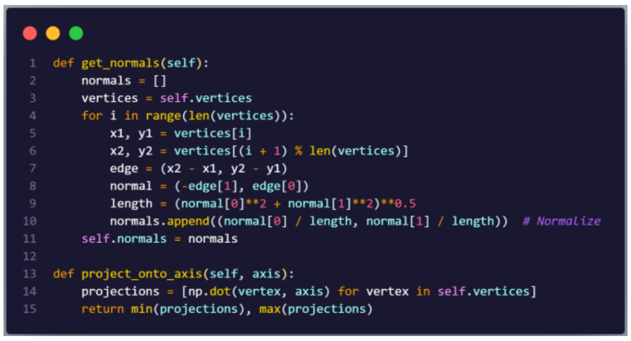
found on any of the axes, then a collision is determined to have occurred.
The edge of the pentagons are defined as the difference between consecutive vertices:

$$\text{edge} = (x_2 - x_1, y_2 - y_1)$$

where *x₁*, *y₁* are the coordinates of the starting vertex of the edge, *x₂*, *y₂* are the coordinates of the next vertex directly connected to the starting vertex. From the calculated edge of the pentagon, the normal for each edges are then determined as:

$$\text{normal} = (-\text{edge}_y, \text{edge}_x)$$

after calculated, the normals are then normalized to ensure they have a magnitude of 1, using (4). These normals are then used as then used as the separating axes, on which the pentagon will be projected. Each vertex of the pentagon is projected onto the normals using (7). For each of the axis, the minimum and maximum projection values are calculated, representing the range of the pentagon's projection. The implementation for these calculations is added as methods into the *Pentagon* class shown previously.



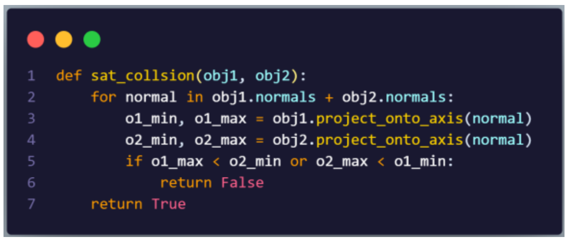
```python
def get_normals(self):
    normals = []
    vertices = self.vertices
    for i in range(len(vertices)):
        x1, y1 = vertices[i]
        x2, y2 = vertices[(i + 1) % len(vertices)]
        edge = (x2 - x1, y2 - y1)
        normal = (-edge[1], edge[0])
        length = (normal[0]**2 + normal[1]**2)**0.5
        normals.append((normal[0] / length, normal[1] / length))  # Normalize
    self.normals = normals

def project_onto_axis(self, axis):
    projections = [np.dot(vertex, axis) for vertex in self.vertices]
    return min(projections), max(projections)
```

Figure 7. Implementation of Normals and Projections Calculations
Source: Author's Document

The projection values are then compared with the projections of the other pentagon, to look for gaps. A gap between two projections occurs when the maximum of one projection is completely to the left or to the right than the minumim of the other shape. A gap is detected if either of the following conditions is true:

$$\max_1 < \min_2 \ \textbf{or} \ \max_2 < \min_1$$

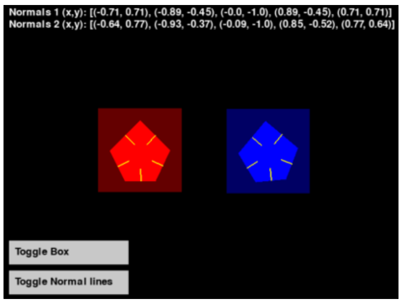
where max and min represents the maximum and minimum values of the projections, and the subscript indicating the pentagon to which they belong. These gap are then used to determine if a collision occurs, by iterating over each projections. If gap is found on any axis, the iteratiion stops, and it is determined that no collision is happening. The implementation for the SAT collision detection is shown in Fig. 8.



```python
def sat_collsion(obj1, obj2):
    for normal in obj1.normals + obj2.normals:
        o1_min, o1_max = obj1.project_onto_axis(normal)
        o2_min, o2_max = obj2.project_onto_axis(normal)
        if o1_max < o2_min or o2_max < o1_min:
            return False
    return True
```

Figure 8. Implementation of SAT Collision Detection
Source: Author's Document

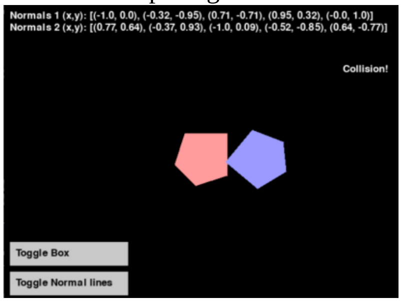
The SAT simulation follows AABB simulation, with a few tweaks and adjustments to account for unique aspects of the SAT algorithm. Unlike AABB, SAT considers the orientation of the pentagons, requiring the ability to rotate the pentagon for better visualization. The user can not only move the pentagons around the screen, as in the AABB simulation, but also rotate them to see how SAT handles collisions.



```
Normals 1 (x,y): [(-0.71, 0.71), (-0.89, -0.45), (-0.0, -1.0), (0.89, -0.45), (0.71, 0.71)]
Normals 2 (x,y): [(-0.64, 0.77), (-0.93, -0.37), (-0.09, -1.0), (0.85, -0.52), (0.77, 0.64)]
```

Figure 9. Interface of the SAT Simulation Program
Source: Author's Document

As shown in Fig. 9. The simulation displays the normal vectors of the pentagons, allowing users to observe how they change as the pentagons are rotated Additionally the program includes buttons to show or hide the bounding boxes and the normal lines for better visualization. Fig 10. Shows how the program looks when a collision happens, with the pentagons in a rotated position, and the bounding boxes hidden to show the pentagons better.



```
Normals 1 (x,y): [(-1.0, 0.0), (-0.32, -0.95), (0.71, -0.71), (0.95, 0.32), (-0.0, 1.0)]
Normals 2 (x,y): [(0.77, 0.64), (-0.37, 0.93), (-1.0, 0.09), (-0.52, -0.85), (0.64, -0.77)]
Collision!
```

Figure 10. Program's Appereance with Collisions Between Rotated Pentagons
Source: Author's Document

D. *Main Game*

The *Main Game* serves as a culmination for the collisions detections algorithms, combining AABB and SAT to create a functional demonstration of a parry-based combat system. This interactive simulation shows how the algorithms seamlessly integrate to craft complex gameplay scenarios, enabling real-time collision detection, enabling precision for an engaging parrying mechanics.

The game features a single boss fight, with both the player and the boss represented as pentagons with a certain amount of hitpoints. The player controls the red pentagon using the keyboard, evading the boss pentagon as it chases the player around the arena. If the boss touches the player, the player takes a hit, reducing their hitpoints. On the other hand, the player has an opportunity to *parry* the boss just before it touches the player by pressing the *spacebar*. Successfully parrying the boss's attacks will subtract the boss's hitpoint. The game ends when either the player's or the boss's hit point reach zero.

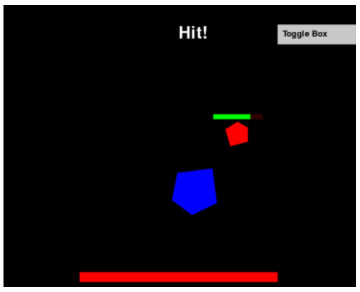
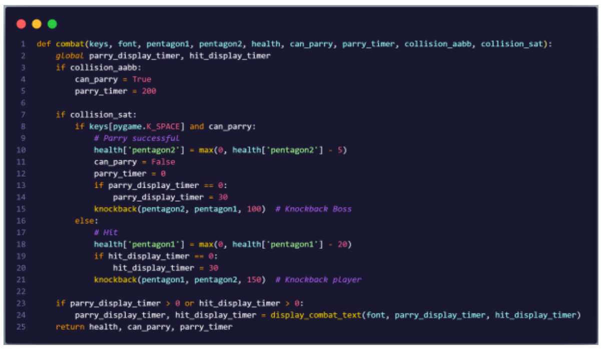


Figure 11. Screenshot of the Game During Combat
Source: Author's Document

Figure 12. Screenshot of the Game's Victory Screen
Source: Author's Document

AABB and SAT are both used in the combat mechanisms to detect collisions between the boss and the player. First, AABB utilizes bounding boxes to detect for overlaps. If the bounding boxes overlap, a parry window is initialized, which remains for 60 frames (1 second), giving the player an opportunity to parry the boss. Following this, the program then checks for a more precise collision between the pentagons using SAT, for which the collision is treated as a hit to the player. Both kinds of collisions (hit and parry) will result in a knockback to either the player or the boss, following the subtraction of their hitpoints. The implementation of the combat system is shown in Fig. 13.



```python
def combat(keys, font, pentagon1, pentagon2, health, can_parry, parry_timer, collision_aabb, collision_sat):
    global parry_display_timer, hit_display_timer
    if collision_aabb:
        can_parry = True
        parry_timer = 200

    if collision_sat:
        if keys[pygame.K_SPACE] and can_parry:
            # Parry successful
            health['pentagon2'] = max(0, health['pentagon2'] - 5)
            can_parry = False
            parry_timer = 0
            if parry_display_timer == 0:
                parry_display_timer = 30
            knockback(pentagon2, pentagon1, 100)  # Knockback Boss
        else:
            # Hit
            health['pentagon1'] = max(0, health['pentagon1'] - 20)
            if hit_display_timer == 0:
                hit_display_timer = 30
            knockback(pentagon1, pentagon2, 150)  # Knockback player

    if parry_display_timer > 0 or hit_display_timer > 0:
        parry_display_timer, hit_display_timer = display_combat_text(font, parry_display_timer, hit_display_timer)
    return health, can_parry, parry_timer
```

Figure 13. Implementation of the Game's Combat System
Source: Author's Document

Please note that the game loop and most utility functions related to routine game logic are not shown in this paper. This is done to keep the paper concise, as this is not a paper for general game development, but only for the collision detection aspect of it. For a complete overview, refer to the appendix to access the full source code.

V. CONCLUSION

Linear algebra is extremely useful for optimizing collision detection systems in video games, by leveraging algorithms like AABB and SAT. These methods simplify complex geometric interactions into manageable and compact computations to create an efficient and precise collision game mechanics. The simulation developed for this paper demonstrates how AABB and SAT can work together as complementary tools to optimize collision detection for real-time gameplay. These techniques, along with other, can be employed to enhance game mechanics in video games, particularly those relying on precise timing and accuracy. These games require responsive and accurate collision detection to ensure game mechanics are implemented properly. *Sekiro*-likes are a prime example of such games, where parrying, timing, and accuracy are central to the gameplay experience. Having precise collision detection system in these games ensures that combat mechanics function seamlessly, creating an engaging and rewarding experience.

VI. APPENDIX

a. Github repository for this project: https://github.com/grwna/SplitSecond-collision-detection-game

VII. ACKNOWLEDGMENT

The author would like to begin by expressing heartfelt gratitude to God for His endless blessings and guidance, as without it, this paper would not have been written succesfully. The deepest thanks also extended to my lecturer for Linear and Geometric Algebra, Ir. Rila Mandala, M.Eng., Ph.D. along with Dr. Ir. Rinaldi Munir, M.T. for their dedication to share knowledge and guide students with patience and expertise.

The author would also like to thank famuly and friends for their constant and unwavering support and encouragement, which have been very important throughout the writing of this paper, and this academic journey. Finally, the author hopes that this paper provides readers with valuable insights into the world of game development and linear algebra.

REFERENCES

[1] Dyn4j, "SAT (Separating Axis Theorem)," Dyn4j, Jan. 2010. [Online]. Available: https://dyn4j.org/2010/01/sat/. [Accessed: Dec. 29, 2024]

[2] Mozilla Developer Network, "3D Collision Detection," Mozilla, [Online]. Available: https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_collision_detection. [Accessed: Dec. 28, 2024].

[3] Pygame Community, *Pygame Documentation*. [Online]. Available: https://www.pygame.org/docs/. [Accessed: 30-Dec-2024].

[4] R. Munir, *Vektor di Ruang Euclidean (Bagian 1)*, Institut Teknologi Bandung. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2024-2025/Algeo-11-Vektor-di-Ruang-Euclidean-Bag1-2024.pdf. [Accessed: 30-Dec-2024].

PERNYATAAN

I hereby declare that this paper is an original work, written entirely on my own, and does not involve adaptation, translation, or plagiarism of any other individual's work.

Bandung, 30 Desember 2024

[illegible]

M. Rayhan Farrukh, 13523035