Utilizing Divide and Conquer Algorithm for Area of Effect (AoE) Detection in Video Game Maps

Muhammad Adha Ridwan - 13523098
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: adharidwan2115@gmail.com , 13523098@std.stei.itb.ac.id

***Abstract*— This paper presents the utilization of divide and conquer algorithm for determining which areas are affected by Area of Effect (AoE) abilities in video games. Traditional collision detection methods for AoE calculations often result in performance bottlenecks when handling multiple entities simultaneously. By implementing a divide and conquer approach, we partition the game space into hierarchical regions, enabling efficient elimination of entities outside the AoE radius before performing detailed intersection tests. Our proposed algorithm recursively subdivides the affected area into smaller quadrants, reducing computational complexity from O(n) to O(log n) for entity filtering.**

Keywords— divide and conquer; area of effect; optimization; video games.

I. Introduction

Modern video games increasingly rely on Area of Effect (AoE) mechanics for engaging gameplay experiences. However, determining which entities are affected by AoE abilities has become a significant performance bottleneck as game worlds grow more complex with hundreds or thousands of interactive objects.

Traditional AoE collision detection employs brute-force methods that check every entity against AoE boundaries, resulting in O(n) complexity where n represents the total number of game objects. This linear scaling leads to frame rate drops in scenarios with high entity counts, particularly when multiple AoE effects occur simultaneously in games like MMORPGs and real-time strategy titles.

The challenge intensifies with diverse AoE shapes including circular areas, rectangular zones, cone-shaped attacks, and irregular polygons. Each variation requires specialized intersection algorithms while maintaining real-time performance constraints of a few milliseconds per calculation.

This paper proposes a divide and conquers approach to optimize AoE collision detection by recursively partitioning affected areas into smaller regions. This enables efficient elimination of entities outside AoE boundaries before performing detailed intersection tests, reducing computational complexity from O(n) to O(log n). Our solution addresses the need for scalable AoE processing systems that maintain real-time performance while preserving accuracy required for fair gameplay mechanics.

II. Theoretical Foundation

A. Divide and Conquer Algorithm

The Divide and Conquer algorithm are an approach to solving complex problems by breaking them down into smaller sub-problems, solving each sub-problem separately, and combining the results to obtain the final solution.

This algorithmic strategy is particularly effective for problems that exhibit optimal substructure and overlapping sub-problems, making it applicable to a wide range of computational challenges from sorting and searching to mathematical computations and optimization problems.

This approach consists of three main steps:

1) *Divide*

   Break the problem into several sub-problems that are similar to the original problem but smaller in size. The goal is to create sub-problems that are ideally of approximately equal size to ensure balanced workload distribution. This step requires careful analysis to identify the natural breaking points of the problem structure.

2) *Conquer*

   Solve each sub-problem either directly (if it has reached a sufficiently small base case) or recursively by applying the same divide and conquer approach. The base case typically involves problems small enough to be solved using straightforward methods without further subdivision.

3) *Combine*

   Merge the solutions of each sub-problem to form the complete solution to the original problem. This step often requires careful consideration of how partial solutions interact and must be integrated to maintain the correctness and efficiency of the overall algorithm.

The division step systematically breaks the problem into smaller, more tractable sub-problems while preserving the essential characteristics of the original problem. The conquering step addresses each sub-problem through recursive application of the algorithm or direct solution when the problem size becomes manageable. Finally, the combining step merges solutions from sub-problems into the comprehensive final solution, ensuring that the integrity and correctness of the solution are maintained throughout the process.

B. Area of Effect in Video Games

Area of Effect (AoE) is a fundamental game mechanic where abilities, spells, or attacks affect multiple targets within a specified area rather than just a single target. This mechanic has become essential in role-playing games (RPGs), real-time strategy (RTS) games, multiplayer online battle arenas (MOBAs), and first-person shooters (FPS), adding layers of tactical depth through spatial gameplay and strategic positioning.

There are several distinct types of AoE mechanics that serve different tactical purposes. Targeted AoE abilities require players to select specific locations for effects to occur, such as fireball spells or grenade firearms that explode at chosen positions. Point-Blank AoE (PBAoE) effects originate from the character's current position and radiate outward in circular patterns, like explosive shockwaves or aura-based abilities that buff allies or debuff enemies. Cone AoE abilities project forward in fan-shaped areas, commonly seen in breath weapons or shotgun blasts that affect multiple enemies in front of the player. Line AoE effects travel in straight paths, hitting all targets along their trajectory, such as lightning bolts or piercing arrows. Chain AoE abilities begin with a primary target and jump to nearby secondary targets, like chain lightning or spreading plague effects that can devastate clustered enemies.

Figure 1. Point-Blank AoE

Figure 2. Cone-Shaped AoE

Across different game genres, AoE mechanics serve varied but crucial roles. In RPGs, they are essential for managing encounters with multiple enemies, with mage characters typically specializing in devastating area spells that can turn the tide of battle. RTS games heavily utilize AoE through artillery units and siege weapons that can devastate clustered enemy forces, making formation and unit positioning crucial strategic elements. MOBAs feature sophisticated AoE systems where team fight positioning and coordinated ability usage often determine match outcomes, with ultimate abilities frequently featuring powerful area effects. FPS games incorporate AoE through grenades, explosive weapons, and special abilities that add tactical depth beyond traditional point-and-shoot gameplay.

C. Quadtree Partitioning

A quadtree is a tree-based n-ary data structure where each internal node of the quadtree has exactly 4 children. Quadtrees are commonly used to recursively divide 2-dimensional space into 4 regions or quadrants. The height of a quadtree adjusts according to the level of detail of information stored in each node. If information in a particular node can still be detailed further, it will create 4 child nodes that store clearer details than the original parent node.

Quadtrees are commonly implemented in various processes including:

1) *Image Rendering.*
2) *Image Proccesing.*
3) *Connection Clustering.*
4) *2-dimensional region indexing.*
5) *3-dimensional modeling of terrain data.*
6) *Sparse data storage.*
7) *Fractal analysis.*

The quadtree represents a data structure that is extremely useful because computers always use binary systems in performing all operations. Quadtrees represent the simplest form of 2-dimensional representation by computers since the number of children in each node totaling 4 can represent 2 values in 2 directions/dimensions.

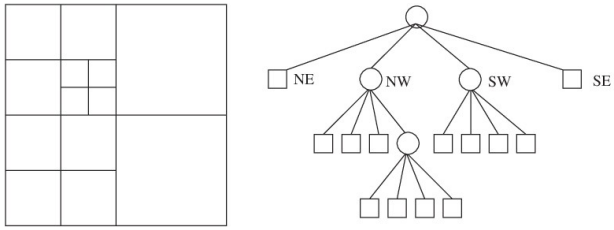


Figure 3. Quadtree Representation

D. Video Games Components

Video games are complex interactive systems composed of various fundamental components that work together to create engaging player experiences. These components form the foundation of game architecture and determine how players interact with the virtual world. Understanding these core elements is essential for game development, design analysis, and technical implementation.

1) Maps

Maps represent the spatial environment and world structure within video games, serving as the foundational framework where all gameplay activities occur. They define the boundaries, terrain, and navigable areas that players can explore and interact with. Maps can range from simple 2D grids in classic arcade games to complex 3D environments in modern open-world titles.

Figure 4. Video Games Map Example

2) Entitiy and Objects

Entities and objects represent the interactive elements within the game world, including characters, items, environmental features, and dynamic elements that players can manipulate or interact with. These components bring life and functionality to the game environment, creating opportunities for player engagement and meaningful gameplay interactions.

Figure 5. Video Games Entity and Object Examples

III. Problem Analysis

A. Problem Definition and Scope

1) Core Problem Statement

Area of Effect (AoE) detection in video games represents a fundamental computational challenge in real-time interactive systems. The problem can be formally defined as follows:

Given:

- A set of entities $E = \{e_1, e_2, ..., e_n\}$ where each entity e_i has position $p_i \in \mathbb{R}^3$ and radius $r_i \in \mathbb{R}^+$.
- An AoE effect A with center position $c \in \mathbb{R}^3$ and effect radius $R \in \mathbb{R}^+$
- A time constraint T_{max} representing maximum allowable computation time per frame.

Find:

- All entities $e_i \in E$ such that the distance $d(p_i, c) \leq R + r_i$, computed within time constraint T_{max}

2) Problem Scope and Context

Modern video games demand increasingly sophisticated AoE systems with hundreds to thousands of entities interacting simultaneously. Common scenarios include:

1. **Massively Multiplayer Online Games (MMOs)**: 100+ players with multiple AoE abilities active.
2. **Real-Time Strategy (RTS) Games**: Hundreds of units affected by area bombardments.
3. Action RPGs: Complex spell interactions affecting multiple enemy groups.
4. **Battle Royale Games**: Large-scale environmental effects affecting many players.

AoE detection typically occurs multiple times per frame, making it one of the most performance-critical systems in game engines. Poor performance directly impacts:

- Frame rate stability.
- Input responsiveness.
- Overall gameplay experience.
- Battery life on mobile platforms.

3) Mathematical Problem Formulation

The AoE detection problem can be modeled as a range query problem in computational geometry:

- **Spatial Query Definition:** For a circular range query with center c and radius R, find all points $p \in P$ such that: $||p - c||_2 \leq R$.
- **Extended Entity Model:** Considering entities with non-zero radius, the condition becomes: $||p_i - c||_2 \leq R + r_i$
- **Multi-dimensional Considerations:** While many games operate in 2D or 2.5D space, full 3D AoE detection requires:
  - 3D distance calculations: $\sqrt{[(x_1-x_2)^2 + (y_1-y_2)^2 + (z_1-z_2)^2]}$.
  - Spherical intersection tests rather than circular.
  - Increased computational complexity.

B. Approaches

1) Brute Force Approach

The most naive and straightforward approach is to just iterate all blocks or units of the maps to find which entities or objects are inside AoE scope.

Although simple, this approach has weakness likes:

- **Linear Scaling**: Performance degrades linearly with entity count.
- **Cache Inefficiency**: Random memory access patterns.
- **Redundant Calculations**: No spatial locality exploitation.
- **No Early Termination**: Must check every entity regardless of spatial distribution.

Analyzing the algorithm to find the complexity of this approach:

- **Time Complexity**: O(n) per query, where n is the number of entities.
- **Space Complexity**: O(1) additional space.
- **Total Complexity**: O(m×n) for m simultaneous AoE effects.

2) Grid-Based Approach

Another approach is to divide the game world into a regular grid where each cell contains information about objects and entities within its bounds.

While keeping the approach relatively simple, this approach still has weakness like:

- **Fixed Resolution**: Grid size affects both memory usage and query performance.
- **Boundary Issues**: AoE effects spanning multiple cells require checking adjacent cells.
- **Non-uniform Distribution**: Poor performance when entities cluster in few cells.
- **Memory Overhead**: Sparse worlds waste significant memory.

Analyzing the algorithm to find the complexity of this approach:

For a world of size W×H divided into cells of size c×c:

- Grid cells required: (W/c) × (H/c).
- Memory usage: $O((W \times H)/c^2)$.
- Average entities per cell: $n \times c^2/(W \times H)$.
- **Time Complexity**: $O(k + \Sigma_i \text{ entities}_i)$ where k is number of intersecting cells.

3) Quadtree Approach

Another approach leveraging divide and conquer paradigm is to use spatial quadtree partitioning, where its recursively subdivides the space into quadrants for a 2D map based on the entities and objects distribution.

This approach still has weakness like:

- **Dynamic Balancing**: Maintaining optimal tree structure as entities move.
- **Update Overhead**: Frequent rebuilding for highly dynamic scenarios.
- **Depth Management**: Preventing excessive subdivision.
- **Memory Fragmentation**: Dynamic node allocation/deallocation.

Analyzing the algorithm to find the complexity of this approach:

- Recursively subdivides space into quadrants.
- Uses spatial bounds to quickly eliminate entire regions
- **Time Complexity**: O(log n + entities in intersecting nodes)

IV. Implementation

We will keep the implementation simple by using 2D Map and 2D Entities.

Programming languages chosen for implementation is Python with libraries such as pygames making the development phases of the system to be fast and also allowing to design the system as modular.

A. System Architecture

The application will be based on two layers, application layer and logic layers, as shown below:

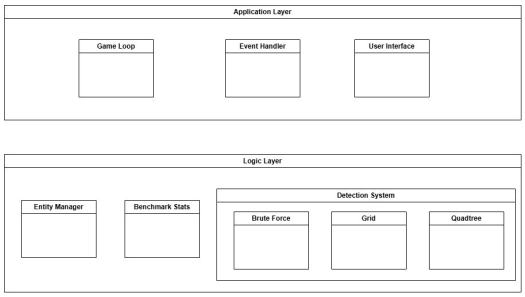


Figure 6. System Architecture

B. Spatial Data Structure

1) Grid

```
  def    grid_distribution(num_entities,    width,    height,
margin=10):
        entities = []

        aspect_ratio = width / height
        cols = int(math.sqrt(num_entities * aspect_ratio))
        rows = int(num_entities / cols)

        while cols * rows < num_entities:
            if cols <= rows:
                cols += 1
            else:
                rows += 1

        x_spacing = (width - 2 * margin) / cols
        y_spacing = (height - 2 * margin) / rows

        entity_count = 0
        for row in range(rows):
            for col in range(cols):
                if entity_count >= num_entities:
                    break
```

```
                base_x = margin + col * x_spacing + x_spacing / 2
                base_y = margin + row * y_spacing + y_spacing / 2

                offset_x = random.uniform(-x_spacing * 0.3, x_spacing * 0.3)
                offset_y = random.uniform(-y_spacing * 0.3, y_spacing * 0.3)

                x = base_x + offset_x
                y = base_y + offset_y

                x = max(margin, min(width - margin, x))
                y = max(margin, min(height - margin, y))

                entities.append(Entity(x, y))
                entity_count += 1

            if entity_count >= num_entities:
                break

        return entities
```

2) Quadtree

```
class QuadTreeNode:
    def __init__(self, x, y, width, height, max_entities=10, max_depth=5, depth=0):
        self.x = x
        self.y = y
        self.width = width
        self.height = height
        self.max_entities = max_entities
        self.max_depth = max_depth
        self.depth = depth
        self.entities = []
        self.children = []
        self.divided = False

    def contains(self, entity):
        return (self.x <= entity.x < self.x + self.width and
                self.y <= entity.y < self.y + self.height)

    def intersects_circle(self, center_x, center_y, radius):
        # Find closest point on rectangle to circle center
        closest_x = max(self.x, min(center_x, self.x + self.width))
        closest_y = max(self.y, min(center_y, self.y + self.height))

        # Calculate distance from circle center to closest point
        distance = math.sqrt((center_x - closest_x)**2 + (center_y - closest_y)**2)
        return distance <= radius

    def subdivide(self):
        half_width = self.width / 2
        half_height = self.height / 2

        self.children = [
            QuadTreeNode(self.x, self.y, half_width, half_height, self.max_entities, self.max_depth, self.depth + 1),
            QuadTreeNode(self.x + half_width, self.y, half_width, half_height, self.max_entities, self.max_depth, self.depth + 1),
            QuadTreeNode(self.x, self.y + half_height, half_width, half_height, self.max_entities, self.max_depth, self.depth + 1),
            QuadTreeNode(self.x + half_width, self.y + half_height, half_width, half_height, self.max_entities, self.max_depth, self.depth + 1)
        ]
        self.divided = True

    def insert(self, entity):
        if not self.contains(entity):
            return False

        if len(self.entities) < self.max_entities or self.depth >= self.max_depth:
            self.entities.append(entity)
            return True

        if not self.divided:
            self.subdivide()

        for child in self.children:
            if child.insert(entity):
                return True
        return False

    def query_range(self, center_x, center_y, radius, results, comparisons_counter):
        if not self.intersects_circle(center_x, center_y, radius):
            return

        for entity in self.entities:
            comparisons_counter[0] += 1
            distance = math.sqrt((entity.x - center_x)**2 + (entity.y - center_y)**2)
            if distance <= radius:
                results.append(entity)

        # Recursively check children
        if self.divided:
            for child in self.children:
                child.query_range(center_x, center_y, radius, results, comparisons_counter)

    def draw(self, screen, color=GRAY):
        pygame.draw.rect(screen, color, (self.x, self.y, self.width, self.height), 1)

        if self.divided:
            for child in self.children:
                child.draw(screen, color)
```

C. Detection Algorithm

1) Brute Force

```
class BruteForceDetector:
    def __init__(self, entities):
        self.entities = entities
        self.name = "Brute Force"
        self.color = RED

    def detect_targets(self, center_x, center_y, radius):
        targets = []
        comparisons = 0

        for entity in self.entities:
            comparisons += 1
            distance = math.sqrt((entity.x - center_x)**2 + (entity.y - center_y)**2)
            if distance <= radius:
                targets.append(entity)

        return targets, comparisons

    def update(self):
        pass

    def draw_structure(self, screen):
        pass
```

2) Grid

```
class GridDetector:
    def __init__(self, entities, world_width, world_height, cell_size=50):
        self.entities = entities
        self.world_width = world_width
        self.world_height = world_height
        self.cell_size = cell_size
        self.cols = math.ceil(world_width / cell_size)
        self.rows = math.ceil(world_height / cell_size)
        self.name = "Grid-Based"
        self.color = BLUE

        self.grid = [[[] for _ in range(self.cols)] for _ in
```

```
range(self.rows)]
        self.update()

    def get_cell(self, x, y):
        """Get grid cell coordinates for a point"""
        col = int(x // self.cell_size)
        row = int(y // self.cell_size)
        col = max(0, min(self.cols - 1, col))
        row = max(0, min(self.rows - 1, row))
        return row, col

    def update(self):
        # Clear grid
        for row in self.grid:
            for cell in row:
                cell.clear()

        # Place entities in grid cells
        for entity in self.entities:
            row, col = self.get_cell(entity.x, entity.y)
            self.grid[row][col].append(entity)

    def detect_targets(self, center_x, center_y, radius):
        targets = []
        comparisons = 0

        min_col = max(0, int((center_x - radius) //
self.cell_size))
        max_col = min(self.cols - 1, int((center_x + radius)
// self.cell_size))
        min_row = max(0, int((center_y - radius) //
self.cell_size))
        max_row = min(self.rows - 1, int((center_y + radius)
// self.cell_size))

        # Check entities in relevant cells
        for row in range(min_row, max_row + 1):
            for col in range(min_col, max_col + 1):
                for entity in self.grid[row][col]:
                    comparisons += 1
                    distance = math.sqrt((entity.x -
center_x)**2 + (entity.y - center_y)**2)
                    if distance <= radius:
                        targets.append(entity)

        return targets, comparisons

    def draw_structure(self, screen):
        """Draw grid lines"""
        for i in range(self.cols + 1):
            x = i * self.cell_size
            pygame.draw.line(screen, LIGHT_GRAY, (x, 0), (x,
self.world_height))

        for i in range(self.rows + 1):
            y = i * self.cell_size
            pygame.draw.line(screen, LIGHT_GRAY, (0, y),
(self.world_width, y))
```

3) Quadtree

```
class QuadTreeDetector:
    def __init__(self, entities, world_width, world_height,
max_entities=10):
        self.entities = entities
        self.world_width = world_width
        self.world_height = world_height
        self.max_entities = max_entities
        self.name = "Quadtree"
        self.color = GREEN
        self.root = None
        self.update()

    def update(self):
        self.root = QuadTreeNode(0, 0, self.world_width,
self.world_height, self.max_entities)

        for entity in self.entities:
            self.root.insert(entity)

    def detect_targets(self, center_x, center_y, radius):
        targets = []
        comparisons = [0]  # Use list to make it mutable for
nested function

        self.root.query_range(center_x, center_y, radius,
targets, comparisons)

        return targets, comparisons[0]

    def draw_structure(self, screen):
        if self.root:
            self.root.draw(screen, LIGHT_GRAY)
```

D. Performance Benchmark

```
class PerformanceTracker:
    def __init__(self):
        self.reset()

    def reset(self):
        self.measurements = {
            'Brute Force': [],
            'Grid-Based': [],
            'Quadtree': []
        }

    def add_measurement(self, method, time_taken,
comparisons):
        if len(self.measurements[method]) >= 60:  # Keep last
60 measurements
            self.measurements[method].pop(0)
        self.measurements[method].append((time_taken,
comparisons))

    def get_average_stats(self, method):
        if not self.measurements[method]:
            return 0, 0

        times = [m[0] for m in self.measurements[method]]
        comparisons = [m[1] for m in
self.measurements[method]]
        return sum(times) / len(times), sum(comparisons) /
len(comparisons)
```

V. Testing and Analysis

The detection approach will be tested against five different entities distribution, uniform, clustered, sparse, and ring.

This different approach is used for finding which is perfect for each distribution while also finding each approach weaknesses.

Each detection approach all has the same radius of 200 and 2000 entities scattered around the map.

A. Uniform Distribution

1) Brute Force

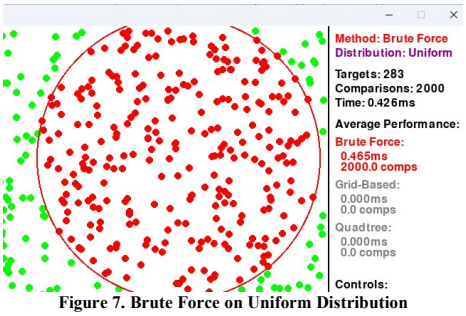


Figure 7. Brute Force on Uniform Distribution

2) Grid

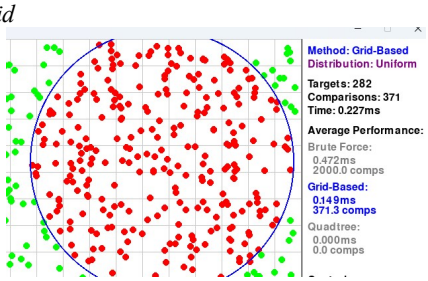


Figure 8. Grid on Uniform Distribution

3) Quadtree

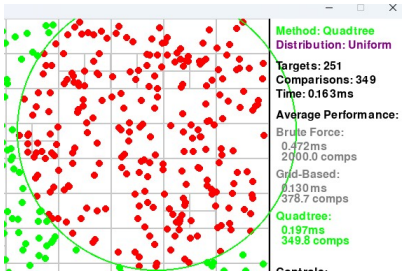


Figure 9. Quadtree on Uniform Distribution

B. Clustered Distribution

1) Brute Force

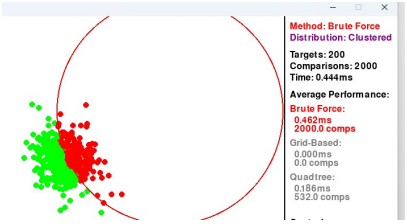


Figure 10. Brute Force on Clustered Distribution

2) Grid

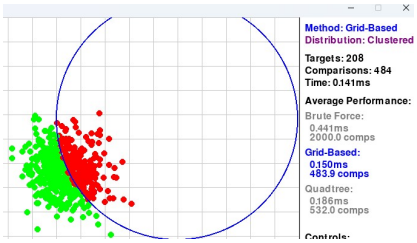


Figure 11. Grid on Clustered Distribution

3) Quadtree

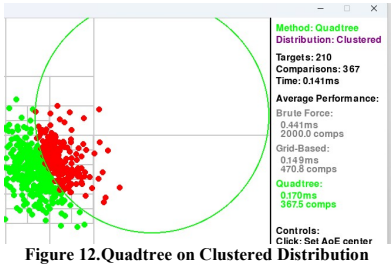


Figure 12.Quadtree on Clustered Distribution

C. Sparse Distribution

1) Brute Force

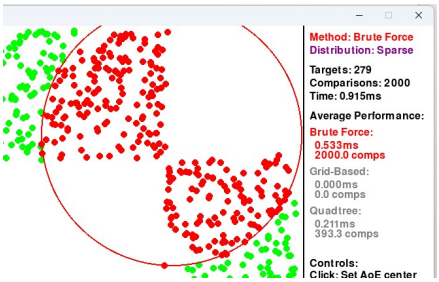


Figure 13. Brute Force on Sparse Distribution

2) Grid

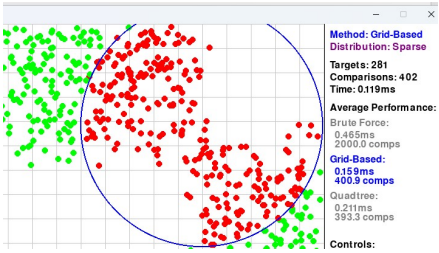


Figure 14. Grid on Sparse Distribution

3) Quadtree

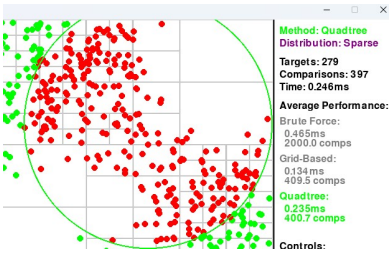


Figure 15. Quadtree on Sparse Distribution

D. Ring Distribution

1) Brute Force

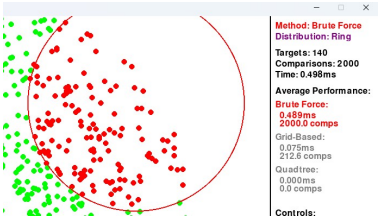


Figure 16. Brute Force on Ring Distribution

2) Grid

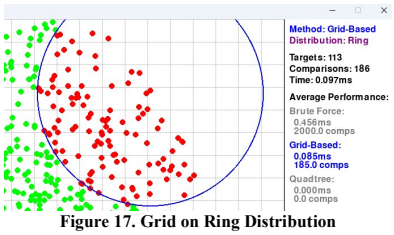


Figure 17. Grid on Ring Distribution

3) Quadtree

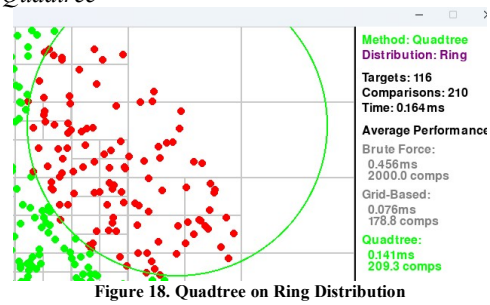


Figure 18. Quadtree on Ring Distribution

Through the testing, we find that for each distribution, it has one approach that suited for the best.

Uniform distribution has Grid approach perform the best. Because grid accessed each grid with O(1) and each grid is uniform, the comparison would be fast.

Clustered distribution has Quadtree approach perform the best. Because there are many empty cells, quadtree didn't split it making the comparison fewer thus making it faster.

Sparse distribution has Grid approach perform the best, because some of the cells are not fully filled but also not empty, making quadtree approach to split to few inefficient nodes.

Ring distribution has Grid approach perform the best with the same reason as above.

Each method handle detection differently, making each has their own pros and cons. Based on the testing and data, Grid based approach perform the best between three of them, making grid approach flexible to multiple distribution pattern.

VI. Conclusion

The testing results demonstrate that **Grid-based partitioning emerges as the most versatile and consistently performing algorithm** for AoE detection across diverse spatial distributions. While each algorithm showed optimal performance under specific conditions—with Quadtree excelling in clustered distributions due to its ability to avoid subdividing empty regions—the Grid approach maintained superior or competitive performance across all tested scenarios.

The key findings reveal that:

- **Uniform distributions** favor Grid partitioning due to O(1) access time and uniform cell utilization
- **Clustered distributions** benefit from Quadtree's adaptive subdivision, reducing unnecessary comparisons in sparse areas
- **Sparse and ring distributions** perform best with Grid partitioning, as partial cell occupancy renders Quadtree's subdivision strategy inefficient

The Grid-based approach's consistent performance stems from its predictable access patterns and stable computational complexity, making it particularly suitable for real-time game environments where consistent frame rates are crucial. While Quadtree offers theoretical advantages for highly clustered data, its performance degrades significantly when dealing with partially filled regions, limiting its practical applicability in dynamic game scenarios.

Therefore, for implementing AoE detection systems in video games where object distributions may vary dynamically, the Grid-based divide-and-conquer approach provides the optimal balance of performance, predictability, and flexibility across different spatial distribution patterns, making it the recommended solution for practical game development applications.

Video Link at Youtube

https://youtu.be/-yubB5M-2oQ

Repository Link

https://github.com/adharidwan/Makalah-IF2211.git

Acknowledgment

The completion of this paper could not have been possible without the aid of all IF2211 lecturers, especially Dr. Ir. Rinaldi Munir who has taught the K02 for the Algorithm Strategy. The writer has learned a hefty amount of information in the process of developing this paper.

References

[1] Adams, E., & Dormans, J. (2012). "Game Mechanics: Advanced Game Design". New Riders https://www.peachpit.com/store/game-mechanics-advanced-game-design-9780321820273

[2] J. Juul, "The game of video game objects: A minimal theory of when we see pixels as objects rather than pictures," in *Extended Abstracts of the 2021 Annual Symposium on Computer-Human Interaction in Play (CHI PLAY '21)*, New York, NY, USA: Association for Computing Machinery, 2021, pp. 376–381, https://dl.acm.org/doi/abs/10.1145/3450337.3483449

[3] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/07-Algoritma-Divide-and-Conquer-(2025)-Bagian1.pdf, Accesed 22 June 2025

[4] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/08-Algoritma-Divide-and-Conquer-(2025)-Bagian2.pdf, Accesed 22 June 2025

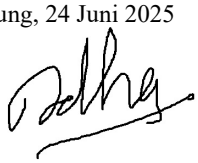
[5] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/09-Algoritma-Divide-and-Conquer-(2025)-Bagian3.pdf, Accesed 22 June 2025

[6] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/10-Algoritma-Divide-and-Conquer-(2025)-Bagian4.pdf, Accesed 22 June 2025

STATEMENT

Hereby, I declare this paper I have written with my own work, not a reproduction or translation of someone else's paper, and not plagiarized.

Bandung, 24 Juni 2025



Muhammad Adha Ridwan 13523098