Collision Detection and Proximity Search Study of Quadtree Optimization and Applications for AR and Video Games

Muhammad Ra'if Alkautsar - 13523011[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]mraifalkautsar@gmail.com, *13523011@std.stei.itb.ac.id*

***Abstract*— Efficiently determining which objects in a virtual scene intersect or lie within a given distance of one another is fundamental to real-time simulations, ranging from video games to augmented reality. Brute force approaches that compare every pair of objects quickly become infeasible as the scene size grows. In this work, we investigate the use of a 2D quadtree as a divide-and-conquer method. We begin by reviewing the theory of spatial partitioning and quadtree construction, then implement four different methods: brute-force collision, quadtree-accelerated collision, brute-force enighbour search, and quadtrree-accelerated neighbour search over various input. The result confirms that quadtree pruning reduces the per-query cost. In proximity search, where each query region is a fixed-size square, the algorithm yields near-logarithmic scale. In contrast, collision detection queries with variable-sized rectangles can straddle cell boundaries and degrade performance to linear time. Our study demonstrates that with appropriate configuration, quadtrees offer dramatic speedups for localized queries, making them a practical choice for real-time interactive applications.**

***Keywords*—Quadtree, collision, proximity, search, detection, augmented reality, video games**

I. Introduction

When simulating physics and interactions of objects in space, computing collisions are a big part of the work. Collision detection is a computational problem of detecting an intersection of two or more objects in virtual space. It tries to solve the questions of if, when, and where two or more objects intersect. It is a classic problem in computational geometry or computer graphics and has applications in various sections of computing, such as physical simulation, videogames, and robotics. Most collision detection algorithms devised can be grouped into operating on 2D or 3D spatial objects, but there might be niche explorations into algorithms that operate in higher dimensions.

Whether in a real-time game engine, a robotics planner, or various other algorithms that simulate physical contact, collision detection is at the heart of it. A fast and efficient algorithm is necessary so that problems can be solved with minimal computing cost.

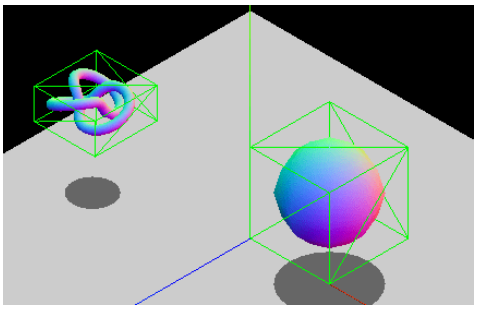

Figure 1.1. Illustration of collision detection
(Source: https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_collision_detection)

In general, collision detection is divided into two phases—broad phase and narrow phase. In the broad phase, data structures like quadtrees operate on bounding rectangles (in 2D) or boxes (in 3D). When a new object is created, it is inserted into the tree. If an object moves, it is reinserted into the tree. During the query process, the tree is traversed down those branches whose region overlaps the given parameters of the query. This hierarchical pruning capability is especially powerful in sparsely populated or unevenly distributed scenes. In the narrow phase, the algorithm then applies precise geometric test to confirm the actual contact points.

Figure 1.2. AR technology on mobile
(Source: https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_collision_detection)

The application domains for collision detection are vast and diverse. In videogames, bottlenecks in performance translate into skipped or dropped frames. In robotics, collision detection ensures safe planning of movements in environments. In

augmented reality or simulations, collision detection allows for realistic results. Yet, despite decades of work, each domain presents unique trade-offs in speed, memory usage, and implementation complexity.

Beyond simply detecting pairwise collisions, many applications also require efficient proximity detection, finding all objects within a given distance or "radius" of each query point.

Proximity detection differs subtly from traditional collision detection, but it remains similar in that it can be optimized using the various same methods used to optimize pairwise collisions. It broadens the question: rather than just identifying whether objects collide, it asks whether objects are within a certain radius or distance fom each other. In practical applications, this is essential for simulations where each entity interacts with others based on their relative positions, even if they do not exactly physically collide. For example, disease propagation range or interactions between molecules.

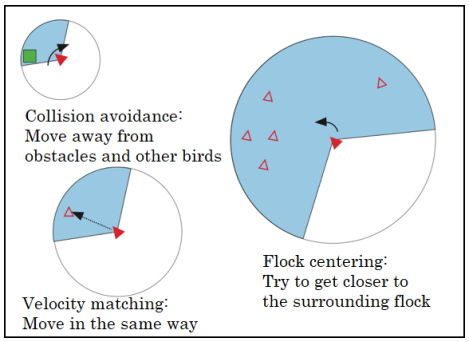

Figure 1.3. Simulating a flock of birds
(Source: https://mas.kke.co.jp/en/model/voide-model)

Consider simulating a flock of birds or a crowd of virtual agents (a quite common problem in various video games), each individual continuously monitors neighbors within a certain radius to smoothly adjust trajectories and avoid collisions. Similarly, for warehouse robots navigating through aisles, safe distances need to be maintained to prevent collisions while optimizing their paths. These systems must, in a very rapid manner, solve queries like the collection of all objects that are near their position efficiently, even as thousands of objects move through space simultaneously.

The simplest way to perform proximity detection is the brute-force approach. While straightforward, checking the distance between every possible pair of objects becomes computationally infeasible as the number of objects grows.

One particularly effective spatial partitioning structure is the quadtree. A quadtree divides the simulation area into progressively smaller quadrants, allowing objects to be grouped by their positions. When in the process of searching for neighbors, the quadtree helps limit the search area to relevant regions, reducing computational cost.

In this paper, we benchmark the quadtree-based approaches for efficient collision detection and proximity detection. We compare their performance to the naïve brute force method.

II. THEORETICAL BASIS

A. Divide and Conquer

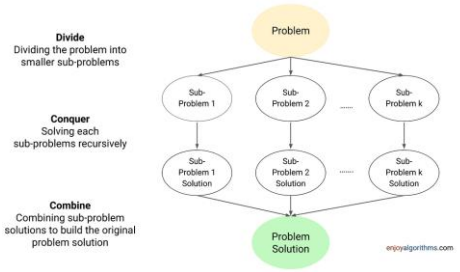

Figure 2.1. Steps of the divide and conquer algorithm
(Source:
https://www.enjoyalgorithms.com/blog/divide-and-conquer)

The divide and conquer approach is a fundamental algorithmic paradigm that solves problems by recursively breaking them into smaller subproblems. Each subproblem is solved independently and the solutions are then combined to solve the original problem. The method has three key steps:

1. Divide, which consists of breaking the problem into smaller subproblems.
2. Conquer, which solves each subproblem recursively until they are simple enough to solve directly.
3. Combine, which merges individual solutions to form a solution to the original problem.

Divide and conquer is known for its efficiency and clarity in solving complex computational problems. Examples of algorithms that utilize this paradigm include quicksort, mergesort, binary search, and spatial indexing methods such as quadtrees and other various trees.

B. Collision and Proximity Detection

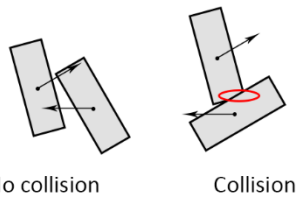

Figure 2.2. Colliding objects
(Source:
https://stackoverflow.com/questions/46268139/2d-collision-detection-without-axis-alignment)

Collision detection is the computational problem of determining whether two or more objects intersect or collide in virtual space. It is a critical component of interactive simulations such as videogames, robotics path planning, virtual reality, and computer-aided design systems. Generally, collision detection is

approached in two phases:

1. Broad-phase Detection: Quickly identifies potential collision pairs by eliminating pairs that are clearly far apart. This stage prioritizes speed over accuracy.
2. Narrow-phase Detection: Performs precise geometric calculations on the narrowed set of candidate pairs identified during the broad phase to accurately detect collisions.

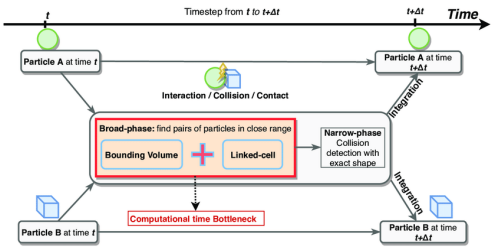

Figure 2.3. Broad phase and narrow phase of proximity detection
(Source: https://www.researchgate.net/figure/Collision-detection-broad-phase-and-narrow-phase-process-workload-in-XDEM-The_fig2_363128648)

Proximity detection, also called neighbor detection, is closely related but distinct from collision detection. It seeks to identify objects within a specific distance or radius from a query point, regardless of whether they physically intersect. This task is crucial in systems such as flocking simulations, crowd simulations, and particle systems where agents or objects interact based on spatial closeness rather than actual collision.

C. Algorithmic Complexity

Algorithmic complexity describes the computational resources required by an algorithm, typically in terms of time or space. Complexity is expressed using Big-O notation. Some example of complexity classes: $O(n^2)$, $O(n \log n)$, and $O(\log n)$.

D. Spatial Partitioning Methods

Spatial partitioning methods organize data into structured formats to efficiently perform queries, such as collision checks or proximity searches. The goal of spatial partitioning os to reduce the number of object comparisons by limiting search to relevant region of space. Common spatial partitioning methods:

- Uniform grid: space divided into equal-sized cells
- Binary space partitioning: recursively divide space with hyperplanes into two partitions
- Octrees (3D) / Quadtrees (2D): recursively subdivide space into eight (octree) or four (quadtree) sub-regions.
- k-d Trees: binary space-partitioning structures dividing space recursively along alternating axes.

E. Quadtree

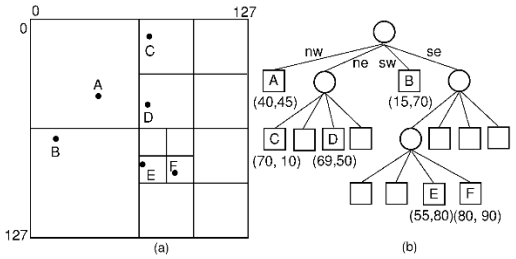

Figure 2.4. Illustration of a quadtree
(Source: https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/PRquadtree.html)

Quadtree is a hierarchical data structure that recursively subdivides a two-dimensional spatial region into four quadrants. Each node of the quadtree represents a rectangular region of space, which may be subvided further. The quadtree structure consists of:

a. Root node: entire region covered by the quadtree.
b. Internal node: nodes subdivided further into four child nodes.
c. Leaf nodes: nodes not subdivided, storing object references directly.

The basic operations of a quadtree are:

- Insertion: When inserting an object, the quadtree recursively places it into the smallest possible node whose boundaries fully contain the object. If a node exceeds a certain object count threshold (the bucket size), it subdivides itself into four child quadrants to better distribute objects.
- Query (Retrieval): Given a search region or point, the quadtree efficiently retrieves all candidate objects within nodes that intersect or contain the query region. The search process efficiently traverses only relevant branches, significantly reducing the total number of object comparisons needed.

III. COMMON PROBLEMS IN AR AND GAMES

A. Object Placement

Figure 3.1. Object placement in The Sims 4
(Source: https://screenrant.com/sims-4-move-objects-up-guide/)

In augmented reality, virtual objects must appear to be firmly anchored in real environment. Detecting collisions between a new virtual object and existing

real-world surfaces often fails when the plane detection or depth sensing is incomplete. For example, in an AR furniture placement simulation, a couch that is placed "through" or overlapping a real-world table breaks immersion. A robust collision detection is required to prevent such from happening.

In another case, program designers often provide "snap" functionality, where virtual items automatically align with detected planes. However, proximity searches that simply find the nearest plan can snap objects to unintended surfaces if the detected normals are ambiguous or incorrect. To distinguish horizontal from vertical planes and impose context-aware constraints, the program demands combining proximity radius checks with orientation priors and heuristics.

B. Physics Simulation

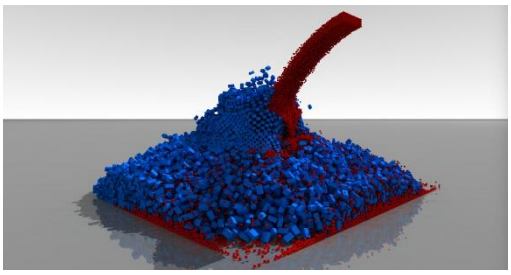

Figure 3.2. Simulation involving many physical objects
(Source: https://medevel.com/os-physics-engine/)

Physics simulation in virtual environments—whether for games, engineering, or scientific modeling—relies heavily on collision detection and proximity search to enforce realistic object interactions.

As object counts grow (hundreds to thousands), the efficiency of the broad-phase spatial index becomes the dominant factor in overall simulation performance.

C. Crowd Simulation

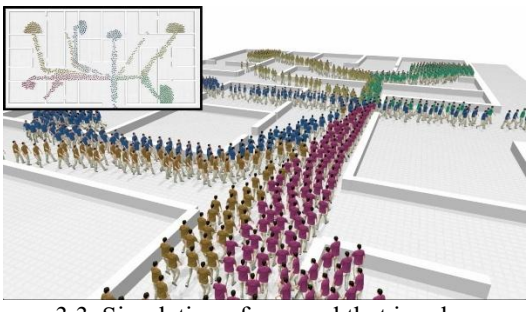

Figure 3.3. Simulation of a crowd that involves many individual moving objects
(Source: https://www.youtube.com/watch?v=9SVC7XBhBpk)

Crowd simulation models the movement and interactions of large numbers of autonomous agents, each reacting to neighbors and the environment. Crowds frequently cluster, disperse, or flow along corridors, causing hot spots in the spatial index. Uniform grids can suffer from overloaded cells; trees can become unbalanced if too many agents occupy a node. Variants of spatial partitioning such as quadtrees where the size can dynamically change help with such problems.

Real-time safety-critical applications−airport terminal simulations, traffic flow analysis, or VR crowd experience may involve tens of thousands of agents. GPU ‐ accelerated neighbor search (e.g., via compute shaders on uniform grids) or multi ‐ threaded tree traversal can be necessary to achieve interactive rates. The overhead of data transfers, synchronization, and index rebuilds must be balanced against raw query speed.

IV. COLLISION DETECTION

A. Query Rectangle

Let the test rectangle be

$$T = [t_x, t_y, t_w, t_h] \quad (1)$$

where (t_x, t_y) and (t_w, t_h) its width/height.

We define the query rectangle as

$$Q = [t_x, t_x + t_w] \times [t_y, t_y + t_h] \quad (2)$$

Any stored rectangle colliding with T must intersect Q.

B. Quadtree Retrieval

Start at the root node v. If the node's bounding box $B(v)$ does not overlap Q, skip that entire subtree because no collisions are possible there.
Otherwise, collect all rectangles stored in v, then recurse only to child nodes c whose boxes satisfy

$$B(c) \cap Q \neq \emptyset \quad (3)$$

By doing this, empty regions get quickly pruned.

Let Sq be the multiset of all rectangles gathered in this way. It guarantees

$$\{R_i : R_i \cap T \neq \emptyset\} \subseteq S_q \quad (4)$$

In other words, every true collision candidate lies in Sq.

C. Exact AABB Overlap Test

For each candidate rectangle $R = [x, y, w, h] \in S_q$, perform the axis-aligned bounding-box test:

$$\neg\big(x > t_x + t_w \vee x + w < t_x \vee y > t_y + t_h \vee y + h < t_y\big) \quad (5)$$

If this holds, R truly overlaps T. (i.e. collision)

D. Complexity Analysis

Let N be the total number of stored rectangles and $h \approx log\ N$ the quadtree height. After retrieval, we have $|S_q| = k$ candidates.
For the retrieval cost, we visit $O(h)$ nodes (pruning subtrees whose boxes miss Q) and collect their objects. For the filter cost, we perform k simple AABB tests. In total

$$O(h + k) = O(\log N + k) \quad (6)$$

which for modest k is far faster than the O(N) brute-force alternative.

V. PROXIMITY SEARCH

A. Query Square

Mathematically, the problem of proximity search can be defined as to be described. Given a set of points

$$P = \{ p_i = (x_i, y_i) \}_{i=1}^{N} \quad (7)$$

within the plane and a query point

$$q = (q_x, q_y) \quad (8)$$

we want to find all points within a radius R, as follows:

$$N(q, R) = \{ p_i \in P : |(p_i - q)| \leq R \} \quad (9)$$

Computationally, we form the axis-aligned bounding square around the query circle.

$$Q = [q_x - R, q_x + R] \times [q_y - R, q_y + R] \quad (10)$$

Which forms:

$$N(q, R) \subseteq \{ p_i : p_i \in Q \} \quad (11)$$

B. Quadtree Retrieval

Starting at the root node v, we do:

- If the node's bounding box B(v) does not overlap Q, stop (no points in this subtree can lie in Q).
- Otherwise, add this node's stored points to our candidate list, then recurse only into those child quadrants whose boxes intersect Q.

Mathematically, if v has children c, we visit just those c with

$$B(c) \cap Q \neq \emptyset$$

After this step, we have a candidate set

$$S_Q = \{ p \in P : p \text{ was collected from any visited node} \} \quad (12)$$

Which satisfies

$$P \cap Q \subseteq S_Q \quad (13)$$

C. Pruning False-Positives

We then prune out false positives by checking each candidate against the true circle.

$$(p_x - q_x)^2 + (p_y - q_y)^2 \leq R^2$$

Points that pass this are exactly the true neighbours within radius R.

D. Complexity Analysis

Let the tree height be

$$h \approx \log N$$

and let

$$k = |S_q|$$

be the number of candidates returned. Then, the quadtree retrieval touches O(h) nodes (pruning entire subtrees when possible) and inspects k points in the last stage. So, the total work required is roughly

$$O(h + k) = O(log\ N + k)$$

which for moderate k is far less than the O(N) of checking every point.

VI. IMPLEMENTATION

```python
class Quadtree:
    def __init__(self, bounds, level=0, max_objects=8, max_levels=6):
        self.bounds = bounds
        self.level = level
        self.max_objects = max_objects
        self.max_levels = max_levels
        self.objects = []
        self.nodes = []

    def split(self):
        x, y, w, h = self.bounds['x'], self.bounds['y'], self.bounds['width'], self.bounds['height']
        hw, hh = w / 2, h / 2
        nl = self.level + 1
        self.nodes = [
            Quadtree({'x': x + hw, 'y': y,      'width': hw, 'height': hh}, nl, self.max_objects, self.max_levels),
            Quadtree({'x': x,      'y': y,      'width': hw, 'height': hh}, nl, self.max_objects, self.max_levels),
            Quadtree({'x': x,      'y': y + hh,'width': hw, 'height': hh}, nl, self.max_objects, self.max_levels),
            Quadtree({'x': x + hw, 'y': y + hh,'width': hw, 'height': hh}, nl, self.max_objects, self.max_levels)
        ]

    def insert(self, point):
        if self.nodes:
            idx = self.get_index(point)
            if idx != -1:
                self.nodes[idx].insert(point)
                return
        self.objects.append(point)
        if len(self.objects) > self.max_objects and self.level < self.max_levels:
            if not self.nodes:
                self.split()
            i = 0
            while i < len(self.objects):
                obj = self.objects[i]
                idx = self.get_index(obj)
                if idx != -1:
                    self.nodes[idx].insert(self.objects.pop(i))
                else:
                    i += 1

    def retrieve(self, range_rect, results):
        idx = self.get_index(range_rect)
        if idx != -1 and self.nodes:
            self.nodes[idx].retrieve(range_rect, results)
        results.extend(self.objects)
        return results
```

Figure 6.1. Quadtree class

Generally, the base quadtree implementation for collision detection and proximity search is the same, with the main difference at getting the index of a point or a rectangle. quadtree class is created with bounds, level, maximum objects, and maximum levels as its parameter.

Regarding the insertion of a new object to the quadtree, when there are no child nodes, everything is stored in self.objects. Once len(self.objects) exceeds max_objects (the variable initialized in the constructor) and the depth allows, it calls split() to create four children.

After splitting, it re-evaluates each stored object: if it now fits wholly in one child (via get_index), move it down. Else (it straddle multiple children), it remain in the parent's self.objects. This ensures that each leaf holds only a small number of objects that can't fit neatly into its descendants.

In the retrieve() function, range_rect is either a test-collider's bounding box or the search-radius square. If get_index(range_rect) returns a child index (i.e. the query fits entirely in one quadrant), recurse into that child. Always add self.objects, because they include any items that straddle boundaries and might collide or lie within the search area.

A. Collision Detection

```python
def get_index(self, rect):
    x, y, w, h = self.bounds['x'], self.bounds['y'], self.bounds['width'], self.bounds['height']
    mid_x, mid_y = x + w/2, y + h/2

    top = rect['y'] + rect['height'] <= mid_y
    bottom = rect['y'] >= mid_y
    left = rect['x'] + rect['width'] <= mid_x
    right = rect['x'] >= mid_x

    if top and right:
        return 0
    if top and left:
        return 1
    if bottom and left:
        return 2
    if bottom and right:
        return 3
    return -1
```

Figure 6.2. Get index function of collision detection

For collision detection, the get_index() function, 'top' tests whether the rectangle's bottom edge (y + height) is no lower than the horizontal midline, 'bottom' tests whether its top edge (y) is no higher than that same midline, 'left' tests whether rectangle's right edge (x + width) is entirely to the left of the vertical midline, and 'right' tests whether its left edge (x) is entirely to the right of that midline.

In short, the function returns a quadrant index only if the entire rectangle fits fully inside that child. Otherwise, it returns -1, so that overlapping or straddling rectangles stay in the parent

```python
WIDTH, HEIGHT = 800, 600
N = 2000            # number of random rectangles
REPS = 2000         # number of test rectangles
MIN_SIZE = 10
MAX_SIZE = 50

# Generate N random axis-aligned rectangles
rects = []
for _ in range(N):
    w = random.uniform(MIN_SIZE, MAX_SIZE)
    h = random.uniform(MIN_SIZE, MAX_SIZE)
    x = random.uniform(0, WIDTH - w)
    y = random.uniform(0, HEIGHT - h)
    rects.append({'x': x, 'y': y, 'width': w, 'height': h})

# Build quadtree
qt = Quadtree({'x': 0, 'y': 0, 'width': WIDTH, 'height': HEIGHT})
for r in rects:
    qt.insert(r)
```

Figure 6.3. Generation of N random rectangles and construction of the quadtree

For the purpose of this implementation, various number of random rectangles of various size and positions are created.

```python
plt.figure(figsize=(8,6))
for r in rects:
    plt.gca().add_patch(plt.Rectangle((r['x'], r['y']), r['width'], r['height'],
                                      edgecolor='blue', facecolor='none', alpha=0.5))
plt.xlim(0, WIDTH)
plt.ylim(0, HEIGHT)
plt.gca().invert_yaxis()
plt.title('Random Rectangles')
plt.show()
```

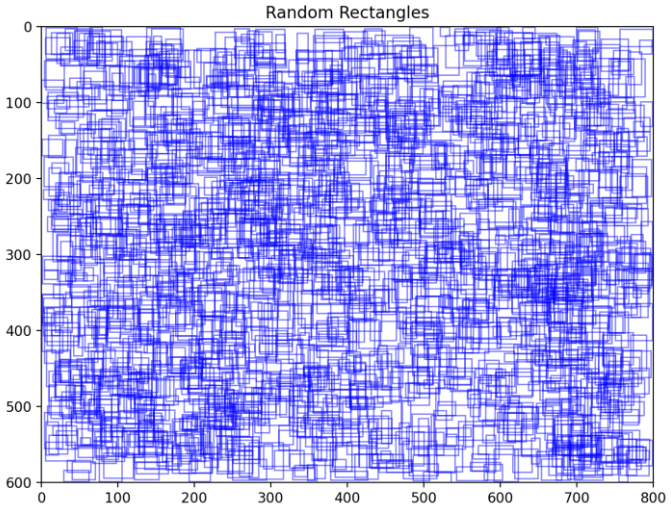

Figure 6.4. Rectangle visualization

The Matplotlib library is used to visualize all the rectangles that have been created.

```python
def brute_collision_test(test_rect):
    tx, ty, tw, th = test_rect['x'], test_rect['y'], test_rect['width'], test_rect['height']
    for r in rects:
        # AABB collision: overlap on both x and y axes
        if not (r['x'] > tx + tw or
                r['x'] + r['width'] < tx or
                r['y'] > ty + th or
                r['y'] + r['height'] < ty):
            pass  # collision

def quad_collision_test(test_rect):
    # retrieve candidates via quadtree using test_rect as range
    cands = qt.retrieve(test_rect, [])
    tx, ty, tw, th = test_rect['x'], test_rect['y'], test_rect['width'], test_rect['height']
    for r in cands:
        if not (r['x'] > tx + tw or
                r['x'] + r['width'] < tx or
                r['y'] > ty + th or
                r['y'] + r['height'] < ty):
            pass  # collision
```

Figure 6.5. The collision test for brute force and quadtree

The tests are done in two methods: brute force and quadtree-accelerated.

In the brute force testing, for each test rectangle, it scans every stored rectangle and uses four comparisons to detect axis-aligned overlap. It performs O(N) comparisons, where N is the total number of rectangles.

In the quadtree-accelerated testing, it only retrieves those rectangles whose quadtree nodes overlap test rectangle. It then runs the same four-comparison AABB check on that (much smaller) candidate set. The cost per test is O(log N + k), where k is the number of candidates returned.

```python
def benchmark(fn):
    tests = []
    for _ in range(REPS):
        w = random.uniform(MIN_SIZE, MAX_SIZE)
        h = random.uniform(MIN_SIZE, MAX_SIZE)
        x = random.uniform(0, WIDTH - w)
        y = random.uniform(0, HEIGHT - h)
        tests.append({'x': x, 'y': y, 'width': w, 'height': h})

    t0 = time.perf_counter()
    for test in tests:
        fn(test)
    t1 = time.perf_counter()
    return (t1 - t0) * 1e3  # milliseconds

bf_time = benchmark(brute_collision_test)
qt_time = benchmark(quad_collision_test)

print(f"Brute-force: {bf_time:,.0f} ns for {REPS} tests on {N} rectangles")
print(f"Quadtree  : {qt_time:,.0f} ns for {REPS} tests on {N} rectangles")
```

Figure 6.6. Collision detection test benchmark

The benchmarking process is done by building a list of REPS or test rectangles, randomly positioned rectangles each sized between MIN_SIZE and MAX_SIZE. Timing is done by recording a high-resolution start time ‘t0’, calling the provided test function fn(test) once per random rectangle, and recording end time with t1. Result is then returned as the total elapsed time in milliseconds.

B. Proximity Search

```python
def get_index(self, point):
    x, y, w, h = self.bounds['x'], self.bounds['y'], self.bounds['width'], self.bounds['height']
    mid_x, mid_y = x + w / 2, y + h / 2
    left = point['x'] < mid_x
    right = point['x'] >= mid_x
    top = point['y'] < mid_y
    bottom = point['y'] >= mid_y

    if top and right:
        return 0
    elif top and left:
        return 1
    elif bottom and left:
        return 2
    elif bottom and right:
        return 3
    else:
        return -1
```

Figure 6.7. Get index function for proximity search

For proximity search, the left and right tests by comparing the point’s x to mid_x. The top and bottom tests by comparing the point’s y to mid_y. If the point sits strictly in one of the four quadrants, it will return that index. Otherwise (if it lies exactly on a midline), it returns -1 so that it stays in the parent.

```python
WIDTH, HEIGHT = 800, 600
N = 5000
REPS = 2000
R = 20

# Generate N random points
points = [{'x': random.uniform(0, WIDTH), 'y': random.uniform(0, HEIGHT)} for _ in range(N)]

# Build quadtree
qt = Quadtree({'x': 0, 'y': 0, 'width': WIDTH, 'height': HEIGHT})
for p in points:
    qt.insert({'x': p['x'], 'y': p['y']})
```

Figure 6.8. Points generation

Generally, the implementation of the proximity search is similar to that of collision detection. Various number of random points of various positions are created.

```python
xs = [p['x'] for p in points]
ys = [p['y'] for p in points]
plt.figure(figsize=(8, 6))
plt.scatter(xs, ys, s=5, color='blue', alpha=0.6)
plt.title('Random Points Distribution')
plt.xlim(0, WIDTH)
plt.ylim(0, HEIGHT)
plt.gca().invert_yaxis()  # optional, to match screen coordinates
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```

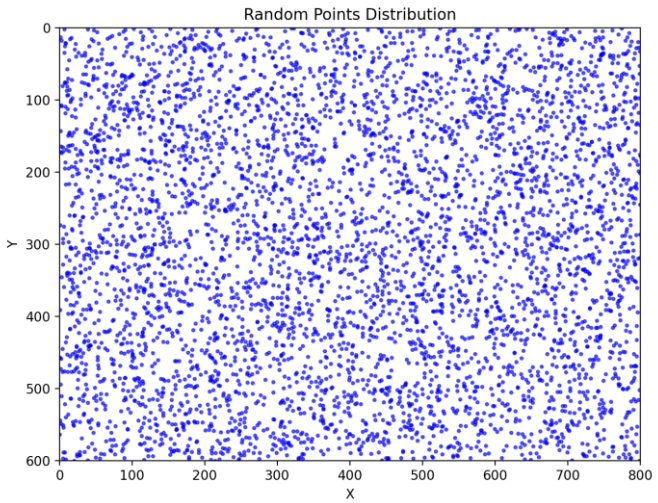

Figure 6.9. Visualization of the points

```python
def brute_neighbor_search(p):
    rx, ry = p['x'], p['y']
    rr = R * R
    for q in points:
        dx = rx - q['x']
        dy = ry - q['y']
        if dx*dx + dy*dy <= rr:
            pass

def quad_neighbor_search(p):
    range_rect = {'x': p['x']-R, 'y': p['y']-R, 'width': 2*R, 'height': 2*R}
    cands = qt.retrieve(range_rect, [])
    rx, ry = p['x'], p['y']
    rr = R * R
    for q in cands:
        dx = rx - q['x']
        dy = ry - q['y']
        if dx*dx + dy*dy <= rr:
            pass

```

Figure 6.9. Brute force and quadtree proximity search

In the brute force searching, the algorithm checks every point in points to see if it lies within the circle of radius R. The cost per query is O(N) distance computations, where N is the total number of points.

Whereas in the quadtree neighbour search, the pruning step first build the axis-aligned square and call qt.retrieve(Q, []), which returns only the points in quadrants overlapping Q. It runs the same circle test as brute force searching, but now only on the much smaller candidate set. The cost per query is roughly O(log N + k), where k is the number of candidates returned.

```python
def benchmark(fn):
    qs = [{'x': random.uniform(0, WIDTH), 'y': random.uniform(0, HEIGHT)} for _ in range(REPS)]
    t0 = time.perf_counter()
    for q in qs:
        fn(q)
    t1 = time.perf_counter()
    return (t1 - t0) * 1e3

bf_time_ns = benchmark(brute_neighbor_search)
qt_time_ns = benchmark(quad_neighbor_search)

print(f"Brute-force: {bf_time_ns:,.0f} ns for {REPS} reps on {N} points")
print(f"Quadtree  : {qt_time_ns:,.0f} ns for {REPS} reps on {N} points")
```

Figure 6.10. Proximity search test benchmark

The benchmark randomly creates two thousands points of varying positions and time how long it takes to call fn(q) for each query, using time.perf_counter() for high-resolutionn timing. It then converts elapsed seconds to milliseconds.

VII. TEST RESULTS AND COMPARISON

A. Collision Detection

No.	N (rectangles)	Brute Force (ms)	Quadtree (ms)
1	500	63	17
2	1000	121	28
3	1500	202	43
4	2000	251	52
5	4000	497	103
6	8000	1010	218
7	16000	1983	551
Average		590	145

Table 7.1. Collision detection tests and comparison

Tests are done for the collision detection implementation with various number of rectangles for input. For N > 500 rectangles, the time elapsed for quadtree starts out lower but both increases linearly with increasing number of rectangles.

B. Proximity Search

No.	N (points)	Brute Force (ms)	Quadtree (ms)
1	500	107	6
2	1000	202	6
3	1500	327	7
4	2000	418	7
5	4000	831	8
6	8000	1719	10
7	16000	3362	11
Average		995	8

Table 7.2. Proximity search tests and comparison

In the proximity search case, time elapsed for the quadtree grows far slower. For N > 500 rectangles, the increase of elapsed time for the quadtree case is minimal even for exponential increase of input. Whereas for the brute force case, time increases linearly with the number of inputs.

C. Discussion

As expected, the time taken for the quadtree method is lower than brute force. For proximity search, the quadtree method has far lower time increase, even when increasing the number of points exponentially. Whereas for the collision detection case, the time taken using the quadtree method is still lower, but for N > 500 when N is increased, the time taken increases linearly. This happens because the shape and size of the query makes all the difference.

For fixed-radius (proximity) queries, query region is always a 2R × 2R square (then a little circle test is created inside). As N is grown, the area of the square stays the same, so on average it touches roughly the same constant number of leaf cells and returns a small k independent of N.

The complexity per query is

$$O(\log N + k) = O(\log N + const)$$

As N is doubled, the algorithm only pay log N extra steps, which is why the quadtree times hover around 6-10 ms, even as N goes from 500 to 16000.

This is different with rectangle collision queries. Each test rectangle has a random width/height up to the maximum (10—50 px). Its query region is exactly itself. Because the rectangles are not tiny points, many straddle the mid-lines or are large enough to overlap multiple child nodes. They end up stored in parent nodes instead of deep leaves.

On retrieval, the quadtree must traverse more branches (maybe the whole tree) and returns a much larger candidate set k. In the worst case, k can grow in proportion to N, so the pruned scan becomes almost as expensive as brute-force.

The time complexity is

$$O(\log N + k(N)),$$

But k(N) grows with N, so the overall time gets closer to O(N).

Some of the solutions for better collision performance are tuning the maximum objects and maximum levels in the quadtree, using “loose” quadtrees which slightly expand child bounds to let objects get deeper, and switching to a grid or sweep-and-prune broad-phase.

VII. CONCLUSIONS

Collision detection and proximity search are some of the common problems encountered on the domain of AR and videogames. The tests done by creating a simple implementation of solving collision detection and proximity search by brute force and quad-tree accelerated method proves that spatial partitioning (in this case, quadtree-based) can accelerate the queries.

Quadtree excel when query regions are small relative to the entire domain and when objects are well-distributed so that most objects settle into deep leaves.

When queries or stored objects are large and cross many cell boundaries, pruning suffers and the candidate count k can approach N, reducing the quadtree’s advantage.

In video games and AR/VR, proximity search queries often involved or smal radii, which is suitable for quadtree acceleration. But, for complex collision detection between many

variable-sized bounding boxes, it may require additional structure or hybrid methods to maintain real-time performance. Quadtrees remain a simple and effective tool for spatial queries and, with the right tuning, can be a choice to meet the real-time demands of graphics or simulation.

VIII. APPENDIX

Video Link: https://youtu.be/YttwWJyYUP0

IX. ACKNOWLEDGEMENT

This paper would not have been completed without the help of others. Therefore, the author would like to thank:

1. Allah SWT,
2. The author's parents and family,
3. The lecturers in charge of Discrete Mathematics, and
4. The author's friends

who have always supported the author during the making of this paper.

REFERENCES

1. https://www.zachmakesgames.com/node/22
2. https://pvigier.github.io/2019/08/04/quadtree-collision-detection.html
3. https://gameprogrammingpatterns.com/spatial-partition.html
4. https://medium.com/%40ahmetturkgenc10/efficient-collision-detection-using-quadtrees-in-game-development-c4e94370bda3
5. https://algorithmic-robotics.org/papers/55_Quad_tree_Based_Collision_D.pdf
6. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/07-Algoritma-Divide-and-Conquer-(2025)-Bagian1.pdf
7. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/08-Algoritma-Divide-and-Conquer-(2025)-Bagian2.pdf
8. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/09-Algoritma-Divide-and-Conquer-(2025)-Bagian3.pdf
9. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/10-Algoritma-Divide-and-Conquer-(2025)-Bagian4.pdf
10. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/11-Algoritma-Decrease-and-Conquer-2025-Bagian1.pdf
11. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/12-Algoritma-Decrease-and-Conquer-2025-Bagian2.pdf

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, 24 Juni 2025

Muhammad Ra'if Alkautsar (13523011)