CASE STUDY: COLLISION DETECTION SYSTEM

Amrita School of Computing, Coimbatore

Department of Computer Science and Engineering

2022 -2023 Even Semester

B-Tech/II Year CSE/4th Semester

19CSE212

Data Structures and Algorithm

Roll No	Name
CB.EN.U4CSE21105	ANWESHA BAIDYA
CB.EN.U4CSE21108	ASHWIN RAJESH SHARMA
CB.EN.U4CSE21110	BALAJI K

PROBLEM STATEMENT:

This case study looks at the possible solutions of enhancing a traditional collision detecting system in 2D Games, using an enhance quad tree, which is a hybrid of K-D Tree and Quad Tree. The purpose is to reduce the space and time complexity while also being efficient in detecting collisions.

INTRODUCTION:

In gaming, collision handling refers to the process of detecting and responding to collisions between objects or entities within the game world. When two objects intersect or come into contact with each other, collision handling mechanisms are responsible for determining the appropriate response or behavior.

Primarily employed to stop objects moving through each other and the environment. Collision Detection is everywhere in computer games: between characters and characters, between characters and terrain, etc.

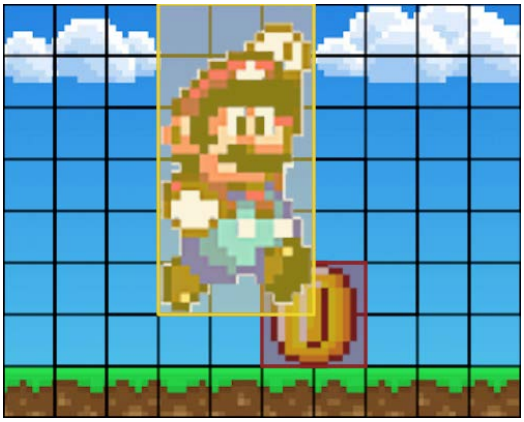
Algorithms to detect collisions in 2D games depend on the type of shapes that can collide (e.g. Rectangle to Rectangle, Rectangle to Circle, Circle to Circle). Generally, you will have a simple generic shape that covers the entity known as a "hitbox" so even though collision may not be pixel-perfect, it will look good enough and be performant across multiple entities.

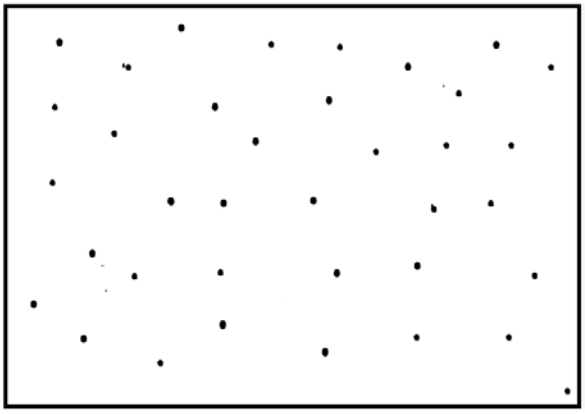
Here are the main aspects of collision handling in 2D games:

- Collision Detection: This step involves detecting when two objects have collided. Various techniques are used for collision detection in 2D games, including:
    1. Bounding Box Collision: Objects are approximated by rectangles, and collision is detected when their bounding boxes overlap.
    2. Pixel-Perfect Collision: Objects are represented by their pixel data, and collisions are detected by checking if any pixels overlap.
    3. Geometric Algorithms: More advanced algorithms such as separating axis theorem (SAT) or swept AABB can be used for more precise collision detection between complex shapes.

- Collision Response: Once a collision has been detected, the collision response mechanism determines how the objects should react or interact with each other. The response can include:
    1. Bouncing or Reflecting: Objects change their direction or velocity upon collision, simulating a bouncing effect.
    2. Destruction or Removal: Objects can be destroyed or removed from the game when they collide with certain conditions.
    3. Triggering Events: Collisions can trigger specific events or actions, such as activating switches, opening doors, or initiating animations.
    4. Physics Simulations: Collisions can be used to simulate realistic physics interactions, such as objects falling, rolling, or colliding with dynamic forces.

- Overlapping Resolution: After a collision occurs, resolving any overlapping or interpenetration between objects is important. This ensures that objects do not visually or physically intersect. Overlapping resolution techniques involve adjusting the position or velocity of colliding objects to separate them properly.



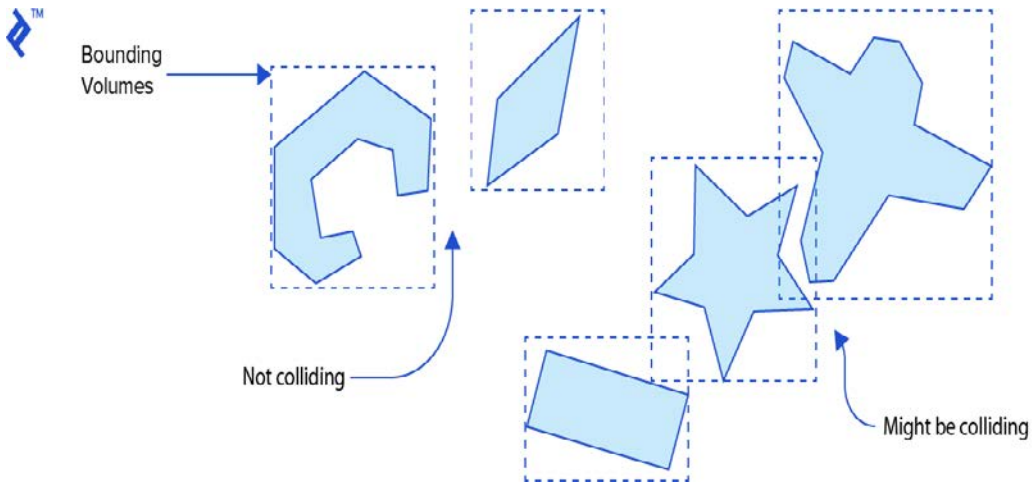


Here are some of the benefits of collision handling in gaming:

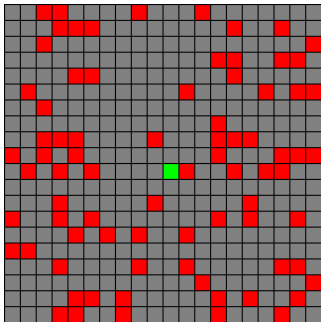
- Realistic physics: Collision handling allows games to simulate realistic physics, which can make the game more immersive and engaging for players.
- Smooth gameplay: Collision handling can help prevent objects from clipping through each other, making the game feel more fluid and responsive.
- Improved gameplay: Collision handling can be used to create a variety of gameplay mechanics, such as knockback, damage, and physics-based puzzles.

Some of the challenges of collision handling in gaming:

- Computational complexity: Collision handling can be a computationally expensive process, especially in games with a large number of objects.
- Accuracy: Collision handling can be difficult to get right, as it is often necessary to balance accuracy with performance.
- Consistency: Collision handling needs to be consistent across the entire game, in order to avoid creating any unfair or unintended gameplay mechanics.



EXISTING DATA STRUCTURE:



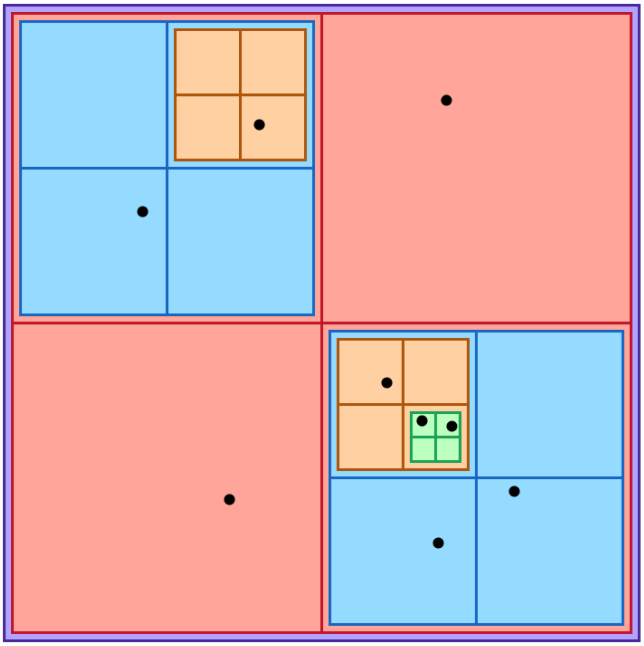
In computer graphics a traditional approach of matrix is very popular, which makes it easier to visualize the space in hand, and track it down in form of coordinates.

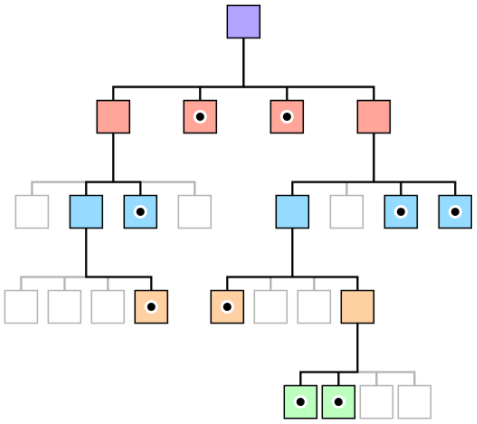
Quad Trees are also used for certain functionalities, but we propose the data structure which is a hybrid of Quad Tree and K-D tree, which makes the Quad tree more efficient than it is already. The point here is to achieve spatial partitioning.

PROBLEMS IN THE ALREADY EXISTING APPROACH:

There is a high space and time complexity which can be observed in the traditional method and with a heavy update activity, it doesn't appear as the best solution and hence we try to make it better.

PROPOSED HYBRID DATA STRUCTURE WITH SUITABLE DIAGRAM:





We will be using quad tree, with a hybrid of K-D Tree.

Quad Tree, which can be used for spatial partitioning can help to eliminate the other part of space, which is not no

DESCRIPTION:

In computer science, a grid file or bucket grid is a method of accessing an area where one or more cells in the grid refer to small content, dividing the area into a non-periodic grid. Grid data, a parallel data structure, provides a good way to store indexes on disk to create hard-to-find data.

This method is the traditional and easy to visualize method hence good for understanding the process.

Collision detection is an essential part of 2D video game development. It involves determining whether two or more objects in the game world interact or overlap. This information is used to control various game mechanics, such as detecting when a player collides with an enemy, a bullet hits a target, or a character interacts with an object.

There are many ways to use collision detection in 2D video games. Here are some popular methods:

1.Bounding Box: The easiest and most common way is to show toys as rectangles (bounding boxes) and check if the rectangles intersect. It involves comparing the controls and sizes of the bounding box between two objects. If the boxes overlap, a collision is detected. Containers are packed quickly but do not provide a smooth collision, especially for poor quality products.

2.Pixel Perfect Collisions: This method includes collision detection at the pixel level. Each object is represented by its pixel data, and the collision is done by comparing the pixels of the two objects to see if they are in the same place. Pixel-perfect scanning provides accurate detection, but is expensive, especially for large devices or large objects.

3. Circular Collision: Use this technique when the object has a circle.

It involves comparing the distance between two circles and their radii. A collision is detected if the distance between the centers is less than the sum of the radii. The grinding panel is easy to use and works for round or spherical objects.

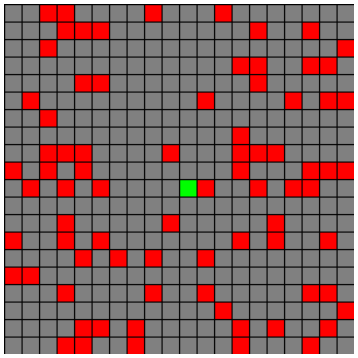
4. Axis of Separation Theorem (SAT): SAT is one of the best methods for checking collisions of irregularly shaped objects, including concave and convex polygons.

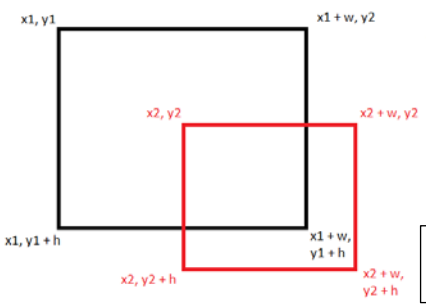
Checks if there is an axis of separation between two objects, that is, a line where the projections of the objects do not overlap. If the split axis is not found, a collision is detected. SAT is more expensive than other methods, but can provide accurate collisions for complex images.

The choice of grinding process depends on the requirements of the game, the complexity of the product and the equipment included. In practice, combinations can be used to increase efficiency and accuracy.

DIRECTION/IDEA FOR PROPOSING DATA STRUCTURES:

Checking two objects for collision which are not close enough is waste of time and resource, hence making loading the processor unnecessarily. Hence quad tree helps to filter out the objects which must be compared and not every other object, reducing the load on the CPU and making the process more efficient.

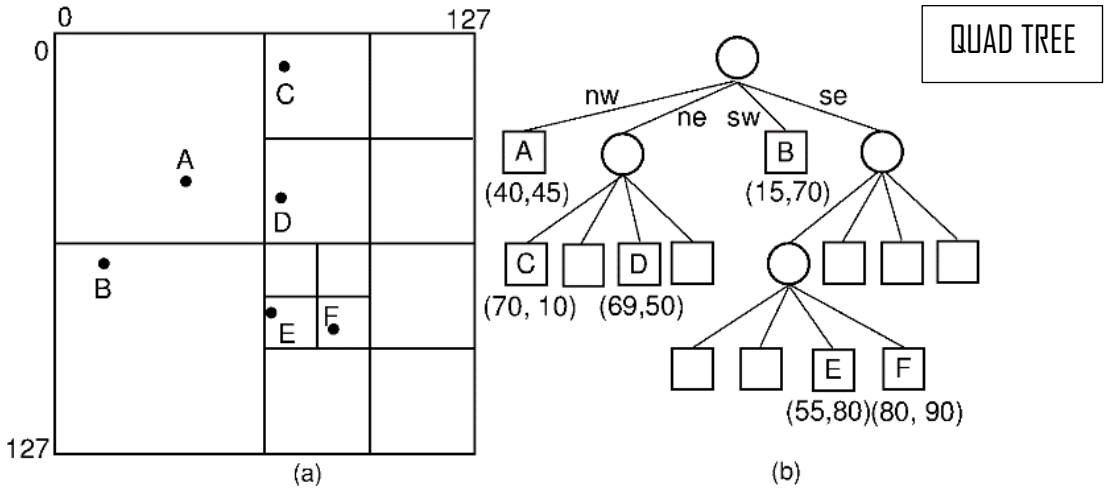




Collision occurs if all of the following are true..

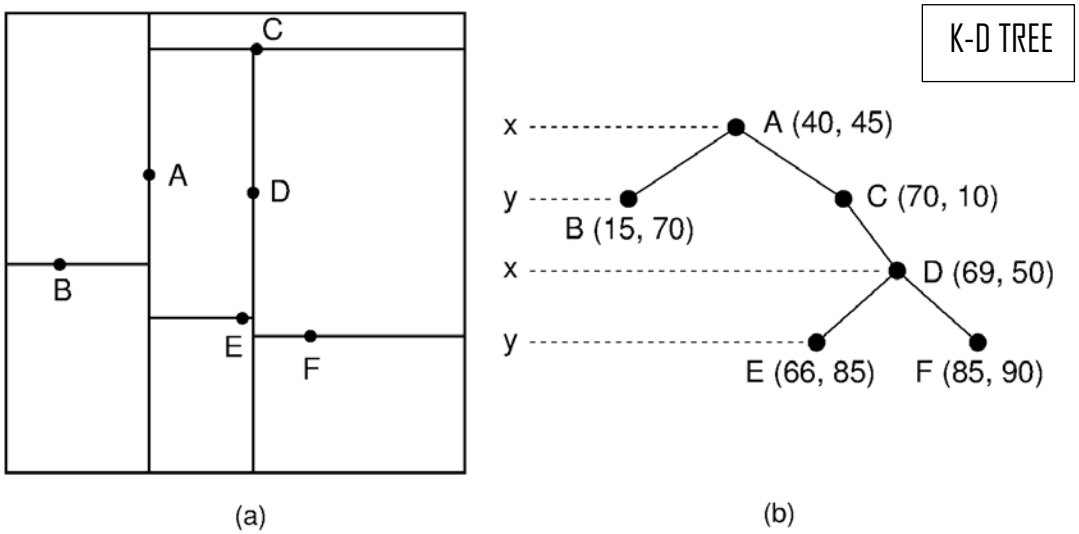
x1 + w > x2,
x1 < x2 + w,
y1 + h > y2,
y1 < y2 + h

Axis Aligned Bounding Box



Every leaf node of a Quad Tree is a point, or array of points allowed in a particular subsection. But the complexity of finding the nearest point to a given point would be O(n) with the arrays. Using a K-D tree for the same could reduce the complexity to O(klogn). So merging the advantages of Quad Tree and a K-D Tree gives us our preferable data structure.

For better visualization of quad tree: https://jimkang.com/quadtreevis/



TIME COMPLEXITY:

The complexity of finding a collision in quadtree is log n for a given region, but for given n elements, the complexity for checking collisions for each one of them is O(nlogn).

Operations	Complexity
Insertion	O(nlogn)
Searching	O(logn)
Finding Nearby Points (K-D tree)	O(2logn)

COMPARISON:

For a gird/matrix:

Operations	Complexity
Insertion	O(1)
Searching	$O(n^2)$

For an array of points (to find the nearest neighbor):

Operations	Complexity
Finding Nearby Points	O(n)

For collision detection, searching is an important aspect and hence can be done efficiently and quickly with a quad tree. And the generic method of storing points(in leaf node) which is through array can be replaced by a K-D tree which has less time complexity.

WHY ITS BETTER THAN THE EARLIER ONE:

A quadtree is a tree data structure in which each internal node has exactly four children. Quadtrees are often used to partition 2D space for collision detection and other spatial operations.

A grid is a regular 2D array of cells. Grids are often used for collision detection, but they can be inefficient for large numbers of objects.

There are several reasons why quadtrees are better than grids for collision handling in 2D games:

Spatial Partitioning: Quadtrees provide dynamic spatial partitioning, allowing for efficient organization and retrieval of objects based on their positions. They recursively divide the 2D space into quadrants, adapting to the density and distribution of objects. This hierarchical structure is particularly beneficial when dealing with non-uniformly distributed objects, as it can reduce the number of comparisons needed during collision detection.

Efficient Collision Detection: Quadtrees excel in scenarios where objects are concentrated in certain areas of the game world. They can quickly eliminate large groups of objects from collision checks by skipping entire quadrants that don't intersect with the target object's boundary. This leads to significant performance improvements compared to a grid, which typically requires iterating over all cells for collision checks.

Dynamic Objects and Movement: Quadtrees handle dynamic objects and their movements more efficiently than grids. When objects move or change their positions, quadtrees can easily update their positions within the tree structure, ensuring accurate collision detection without the need for extensive reorganization. On the other hand, a grid would require frequent updates and potentially extensive cell rearrangements, leading to increased overhead.

Variable Density: In games where object density varies across the game world, quadtrees adapt dynamically to the changing density, resulting in more efficient collision handling. Grids, on the other hand, have fixed cell sizes and may either waste memory by using excessively small cells in low-density areas or suffer from inaccurate collision detection due to large cells in high-density areas.

Quadtrees are more efficient for large numbers of objects. As the number of objects in a grid increases, the number of potential collisions also increases. This can lead to performance problems, especially in real-time games. Quadtrees, on the other hand, can be used to efficiently partition 2D space, which can significantly reduce the number of potential collisions that need to be checked.

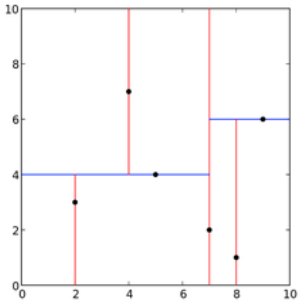
Quadtrees can handle objects of varying sizes. Grids are typically used for objects of a uniform size. However, in many games, objects can have different sizes. This can make it difficult to use grids for collision detection, as it can be difficult to determine which objects are colliding with each other. Quadtrees, on the other hand, can handle objects of varying sizes, making them a more versatile option for collision detection.

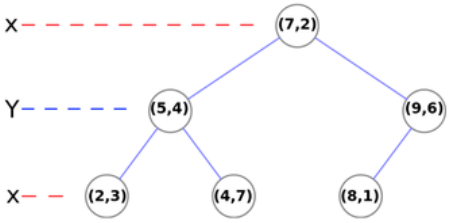
Quadtrees can be used for other spatial operations. In addition to collision detection, quadtrees can also be used for other spatial operations, such as finding the nearest object to a given point or finding all objects within a given region. This makes them a more versatile data structure than grids, which can only be used for collision detection.

Overall, quadtrees are a more efficient and versatile data structure than grids for collision handling in 2D games. If you are working on a game with a large number of objects or objects of varying sizes, then quadtrees are a good option to consider.

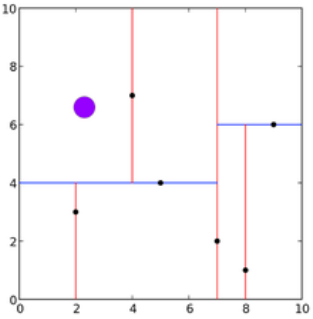
WHAT IS K-D TREE:

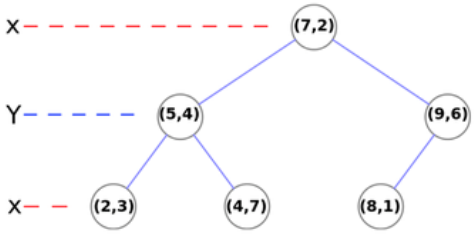
A K-D Tree is a binary tree in which each node represents a k-dimensional point. The insertion in this is similar to a BST, but alternating between comparisons with X and Y every level.



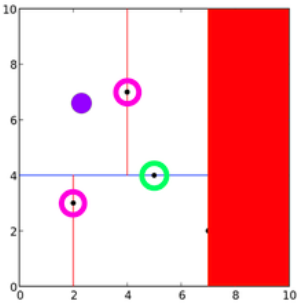


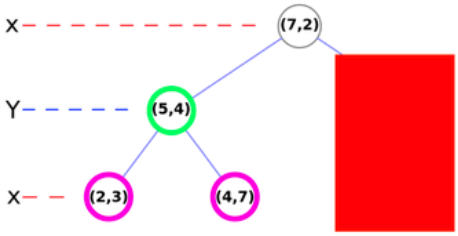
Euclidian Distance is then calculated between 2 points of the same sub tree, eliminating the other half. Hence with logn complexity we can calculate the nearest neighbor.



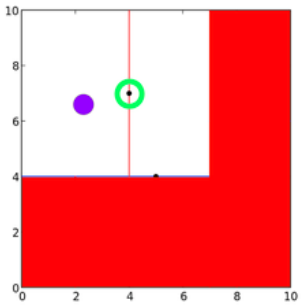


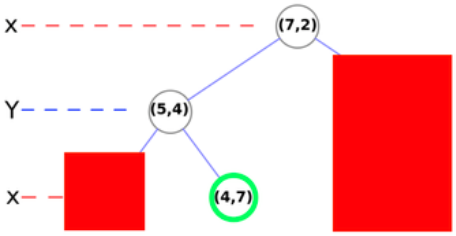
The patch implies the part of tree which can be straight away ignored, keeping the complexity to logN.



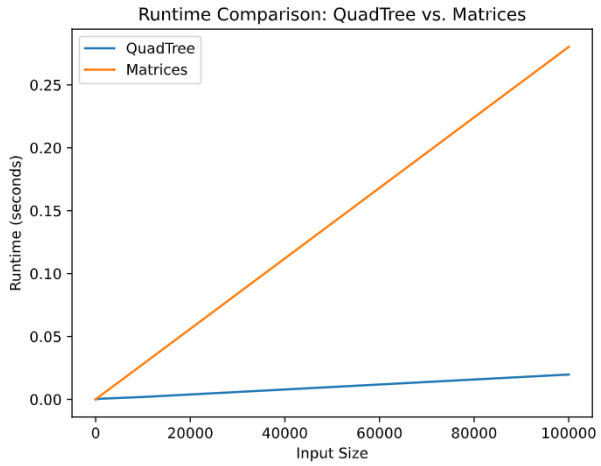


We recursively perform the operations to find the nearest neighbor





So, incorporating this data structure along with quad tree makes it beneficial for us to identify nearest object to a given point which then can be used to



This comparison with matrices is when the quadrants are already divided by quad tree and the points are retrieved from the given range boundary, hence providing a complexity of n.

CODE:

```
import numpy as np
####
class Node:
    def __init__(self, point, axis):
        self.point = point  # Coordinates of the point stored in the node
        self.axis = axis  # Axis (0 for x-axis, 1 for y-axis)
        self.left = None  # Left child node
        self.right = None  # Right child node

class KDTree:
    def __init__(self):
        self.root = None

    def insert(self, point):
        if self.root is None:
            self.root = Node(point, axis=0)
        else:
            self._insert_recursive(point, self.root)

    def _insert_recursive(self, point, node):
        axis = node.axis
        if point[axis] < node.point[axis]:
            if node.left is None:
                node.left = Node(point, axis=(axis + 1) % 2)
            else:
                self._insert_recursive(point, node.left)
        else:
            if node.right is None:
```

```
                node.right = Node(point, axis=(axis + 1) % 2)
            else:
                self._insert_recursive(point, node.right)

    def construct_kdtree(self, points):
        for point in points:
            self.insert(point)

    def find_nearest_neighbor(self, target):
        best_node = None
        best_distance = float('inf')

        def traverse(node, point, depth):
            nonlocal best_node, best_distance

            if node is None:
                return

            axis = node.axis

            if node.point != point:
                distance = np.linalg.norm(np.array(node.point) -
np.array(point))
                if distance < best_distance:
                    best_node = node
                    best_distance = distance

            if point[axis] < node.point[axis]:
                traverse(node.left, point, depth + 1)
            else:
                traverse(node.right, point, depth + 1)

            if abs(node.point[axis] - point[axis]) < best_distance:
                if point[axis] < node.point[axis]:
                    traverse(node.right, point, depth + 1)
                else:
                    traverse(node.left, point, depth + 1)

        traverse(self.root, target, 0)
        return best_node
#########################
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y


class Rectangle:
    def __init__(self, x, y, width, height):
```

```
        self.x = x
        self.y = y
        self.width = width
        self.height = height

    def contains(self, point):
        return (self.x - self.width/2 <= point.x <= self.x + self.width/2
and
                self.y - self.height/2 <= point.y <= self.y + self.height/2)

    def intersects(self, range_boundary):
        return not (self.x - self.width/2 > range_boundary.x +
range_boundary.width/2 or
                    self.x + self.width/2 < range_boundary.x -
range_boundary.width/2 or
                    self.y - self.height/2 > range_boundary.y +
range_boundary.height/2 or
                    self.y + self.height/2 < range_boundary.y -
range_boundary.height/2)
class QuadTree:
    def __init__(self,boundary,capacity):
        self.boundary = boundary
        self.capacity = capacity
        self.points = []
        self.kdtree = KDTree()
        self.divided = False      # Flag indicating if the node is divided

        self.northwest = None
        self.northeast = None
        self.southwest = None
        self.southeast = None

    def insert(self, point):
        # If the point is not within the boundary, ignore it
        if not self.boundary.contains(point):
            return

        # If the capacity is not reached, add the point to the current node
        if len(self.points) < self.capacity:
            self.points.append(point)
            temp=(point.x, point.y)
            self.kdtree.insert(temp)
        else:
            # If the node is not divided, create sub-quadrants and
redistribute the points
            if not self.divided:
                self.subdivide()
```

```
            self.northwest.insert(point)
            self.northeast.insert(point)
            self.southwest.insert(point)
            self.southeast.insert(point)


    def subdivide(self):
        # Create sub-quadrants with half the width and height of the current
boundary
        x = self.boundary.x
        y = self.boundary.y
        w = self.boundary.width / 2
        h = self.boundary.height / 2

        # Create sub-quadrants
        nw_boundary = Rectangle(x - w/2, y + h/2, w, h)
        ne_boundary = Rectangle(x + w/2, y + h/2, w, h)
        sw_boundary = Rectangle(x - w/2, y - h/2, w, h)
        se_boundary = Rectangle(x + w/2, y - h/2, w, h)

        self.northwest = QuadTree(nw_boundary, self.capacity)
        self.northeast = QuadTree(ne_boundary, self.capacity)
        self.southwest = QuadTree(sw_boundary, self.capacity)
        self.southeast = QuadTree(se_boundary, self.capacity)

        self.divided = True
    def query_range(self, range_boundary):
        # Create a list to store the points within the range
        points_in_range = []

        # If the query range does not intersect with the boundary, return the
empty list
        if not self.boundary.intersects(range_boundary):
            return points_in_range

        # Check each point in the current node and add it to the result list
if it is within the range
        for point in self.points:
            if range_boundary.contains(point):
                points_in_range.append(point)

        # If the node is divided, recursively query each sub-quadrant and add
the points to the result list
        if self.divided:
            points_in_range += self.northwest.query_range(range_boundary)
            points_in_range += self.northeast.query_range(range_boundary)
            points_in_range += self.southwest.query_range(range_boundary)
            points_in_range += self.southeast.query_range(range_boundary)
```

```
        return points_in_range
    def nearest(self,point):
        return self.kdtree.find_nearest_neighbor(point)

############################

boundary=Rectangle(0, 0, 100, 100)
qt=QuadTree(boundary,4)

qt.insert(Point(10, 10))
qt.insert(Point(20, 20))
qt.insert(Point(30, 30))
qt.insert(Point(35, 35))
qt.insert(Point(50, 50))
qt.insert(Point(60, 60))
qt.insert(Point(70, 70))
qt.insert(Point(80, 80))
qt.insert(Point(90, 90))

# Query points within a range
range_boundary = Rectangle(25, 25, 10,10)
points_in_range = qt.query_range(range_boundary)

# Print the points within the range
print("Points which are colliding are:")
for point in qt.query_range(range_boundary):
    print("(",point.x,",",point.y,")")
a=qt.nearest((50,50))
print("Nearest point is:",a.point)
```

CONCLUSION:

Hence for the purpose of detecting collision in 2D graphics game, A quad tree is a better choice with a hybrid of K-D tree with better time complexity and space complexity than a traditional Grid system.

REFERENCES:

https://www.google.com/url?sa=i&url=https%3A%2F%2Flevelup.gitconnected.com%2F2d-collision-detection-8e50b6b8b5c0&psig=AOvVaw0OAFlVAnFqYZcXzWnRWof_&ust=1687191111690000&source=images&cd=vfe&ved=0CBEQjRxqFwoTCLig4Omazf8CFQAAAAAdAAAAABAE

https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/PRquadtree.html