# Spatial Hash Grids

### Introduction

A fundamental problem in many fields is efficiently determining which objects are near a given point in space. This problem appears in computer graphics, physics simulations, video games, robotics, and more. Spatial hash grids offer a simple yet powerful solution for managing large numbers of objects and performing rapid proximity queries.

### The Problem

In systems that involve a vast number of objects distributed in space, operations often require knowledge of nearby entities. Examples include:

* Robotics: Allowing robots to sense and react to nearby obstacles or agents. (Space Infinity, 2021)
* Particle Simulations: Computing interactions in particle systems, such as fluids or gases. (Ten Minute Physics, 2022)
* Video Games: Determining which units are affected by an explosion or enabling flocking behavior among nearby entities. (SimonDev, 2020)

The naive approach to proximity queries involves checking each object against every other object, resulting in a time complexity of O(n2) which quickly becomes computationally infeasible as the number of objects increases.

### Spatial hash grids

Spatial hash grids provide a simple yet effective alternative. They divide a space into discrete cells (buckets) and map objects into these cells using a hash function. This data structure is simple to implement, offers significant performance improvements over the naive approach and works well in both sparse and dense environments.

The approach balances computational efficiency and implementation simplicity, making it a preferred choice for real-time applications such as physics simulations, pathfinding, and game mechanics.

### Python Implementation Overview

**Classes**

* Client: Represents an object with position, dimensions, and a unique identifier.
* SpatialHashGrid: Manages the grid, including insertion, deletion, updating, and querying of clients.

**Core Methods**

* get\_cell\_range(position, dimensions): Calculates the range of cells an object occupies.
* get\_cell\_key(x, y): Generates a unique key for a cell based on its coordinates.
* insert\_client(client): Inserts a client into the appropriate cells.
* delete\_client(client): Removes a client from its cells.
* find\_nearby(position, dimensions): Finds all clients near a given position.

**How It Works**

1. Initialization: The grid is initialized with specified bounds and cell dimensions.
2. Object Management:

* Insertion: Objects are placed into cells based on their positions.
* Updating: When an object moves, it's reinserted to reflect its new position.
* Deletion: Objects can be removed from the grid.

1. Proximity Queries: To find nearby objects, only the cells overlapping the query region are checked, significantly reducing the number of comparisons.

### Performance

**Naive Approach**

Space complexity: The naive method involves maintaining a list of all objects in the system, which requires **O(n)** space.

Time complexity: **O(n2)** (since there are n objects and each performs a proximity query, each taking O(n) time).

**Spatial Hash Grid Approach**

Space complexity: The spatial hash grid requires space to store the objects and the cell mapping (n) for storing the objects and O(m) for the grid cells. So the space complexity is O(n+m) however this simplifies to **O(n)**.

Time complexity: Henning (2023) wrote a detailed analysis of the time complexity of 2d spatial hash grids. They find it to be O(n+r2\*d) where n refers to the objects, r is a constant radius and d is the average density of a query, however the author acknowledges that this simplifies to **O(n)** since n >> r2\*d.

### Implementation

**The complete code is attached below:**

import tkinter as tk

from tkinter import ttk

import random

class Client:

"""

Represents an object (client) in the spatial hash grid.

Attributes:

position (tuple): (x, y) coordinates of the object.

dimensions (tuple): Width and height of the object.

name (str): Identifier for the object.

"""

def \_\_init\_\_(self, position, dimensions, name):

self.position = position

self.dimensions = dimensions

self.name = name

self.indices = None # Stores the range of grid cells the object occupies

def \_\_eq\_\_(self, other):

return isinstance(other, Client) and self.position == other.position and self.dimensions == other.dimensions

def \_\_hash\_\_(self):

return hash((self.position, self.dimensions, self.name))

class SpatialHashGrid:

"""

Spatial Hash Grid to manage and query objects in a 2D space.

"""

def \_\_init\_\_(self, bounds, dimensions):

"""

Initializes the grid.

Args:

bounds (list of tuples): [(min\_x, max\_x), (min\_y, max\_y)] defines the grid's boundaries.

dimensions (tuple): Number of cells in (x, y).

"""

self.bounds = bounds # Bounds of the grid

self.dimensions = dimensions # Number of cells along each axis

self.cell\_map = {} # Dictionary to map cell keys to objects

def new\_client(self, position, dimensions, name):

"""

Adds a new object to the grid.

Args:

position (tuple): (x, y) coordinates of the object.

dimensions (tuple): Width and height of the object.

name (str): Identifier for the object.

Returns:

Client: The created object.

"""

client = Client(position, dimensions, name)

self.insert\_client(client)

return client

def insert\_client(self, client):

"""

Inserts a client into the appropriate cells based on its position and dimensions.

"""

min\_cell, max\_cell = self.get\_cell\_range(client.position, client.dimensions)

client.indices = (min\_cell, max\_cell) # Track which cells the object occupies

for x in range(min\_cell[0], max\_cell[0] + 1):

for y in range(min\_cell[1], max\_cell[1] + 1):

key = self.get\_cell\_key(x, y)

if key not in self.cell\_map:

self.cell\_map[key] = set()

self.cell\_map[key].add(client)

def find\_nearby(self, position, dimensions):

"""

Finds all objects near a given position within specified dimensions.

Args:

position (tuple): (x, y) coordinates of the query point.

dimensions (tuple): Search area width and height.

Returns:

list: A list of nearby objects.

"""

nearby = set()

min\_cell, max\_cell = self.get\_cell\_range(position, dimensions)

for x in range(min\_cell[0], max\_cell[0] + 1):

for y in range(min\_cell[1], max\_cell[1] + 1):

key = self.get\_cell\_key(x, y)

if key in self.cell\_map:

nearby.update(self.cell\_map[key])

return list(nearby)

def update\_client(self, client, new\_position):

"""

Updates the position of an existing client.

Args:

client (Client): The client to update.

new\_position (tuple): The new (x, y) coordinates for the client.

"""

self.delete\_client(client) # Remove the client from its old cells

client.position = new\_position # Update the client's position

self.insert\_client(client) # Reinsert the client into the grid

def delete\_client(self, client):

"""

Removes a client from the grid.

Args:

client (Client): The client to delete.

"""

if not client.indices:

return # If the client has no assigned cells, do nothing

min\_cell, max\_cell = client.indices

for x in range(min\_cell[0], max\_cell[0] + 1):

for y in range(min\_cell[1], max\_cell[1] + 1):

key = self.get\_cell\_key(x, y)

if key in self.cell\_map:

self.cell\_map[key].discard(client) # Remove the client from the cell

if not self.cell\_map[key]: # If the cell is empty, delete it

del self.cell\_map[key]

client.indices = None # Clear the client's cell indices

def get\_cell\_range(self, position, dimensions):

"""

Calculates the range of grid cells an object occupies.

"""

min\_x = int((position[0] - dimensions[0] / 2 - self.bounds[0][0]) // self.cell\_width())

max\_x = int((position[0] + dimensions[0] / 2 - self.bounds[0][0]) // self.cell\_width())

min\_y = int((position[1] - dimensions[1] / 2 - self.bounds[1][0]) // self.cell\_height())

max\_y = int((position[1] + dimensions[1] / 2 - self.bounds[1][0]) // self.cell\_height())

return (min\_x, min\_y), (max\_x, max\_y)

def get\_cell\_key(self, x, y):

"""

Generates a unique key for each cell based on its (x, y) index.

"""

return f"{x}.{y}"

def cell\_width(self):

"""

Calculates the width of each grid cell.

"""

return (self.bounds[0][1] - self.bounds[0][0]) / self.dimensions[0]

def cell\_height(self):

"""

Calculates the height of each grid cell.

"""

return (self.bounds[1][1] - self.bounds[1][0]) / self.dimensions[1]

class SpatialHashGridVisualizer:

"""

Visualizes the spatial hash grid using tkinter.

Includes object reporting and state visualization.

"""

def \_\_init\_\_(self, grid, width, height):

self.grid = grid

self.width = width

self.height = height

self.root = tk.Tk()

self.root.title("Spatial Hash Grid Visualization")

# Top explanation frame

self.top\_frame = tk.Frame(self.root)

self.top\_frame.pack(side=tk.TOP, fill=tk.X)

self.heading = tk.Label(self.top\_frame, text="Spatial Hash Grid", font=("Arial", 18, "bold"))

self.heading.pack(pady=5)

self.explanation = tk.Label(

self.top\_frame,

text=("This visualization demonstrates the use of a spatial hash grid for efficiently managing moving objects. "

"Objects are tracked in cells, allowing fast lookup for nearby items."),

wraplength=600,

justify="center",

)

self.explanation.pack(pady=5)

# Left canvas for visualization

self.canvas = tk.Canvas(self.root, width=width, height=height, bg="white")

self.canvas.pack(side=tk.LEFT)

# Right panel for reporting

self.right\_frame = ttk.Frame(self.root, width=300, height=height)

self.right\_frame.pack(side=tk.RIGHT, fill=tk.Y)

self.right\_frame.pack\_propagate(False)

self.report\_label = tk.Label(self.right\_frame, text="Object States:", font=("Arial", 14, "bold"))

self.report\_label.pack(anchor="w", padx=10, pady=5)

self.report = tk.Text(self.right\_frame, height=25, wrap=tk.WORD)

self.report.pack(fill=tk.BOTH, expand=True, padx=10, pady=5)

self.objects = []

self.previous\_states = {} # Track previous color states by object name

def add\_random\_objects(self, count):

for i in range(count):

x = random.randint(0, self.width)

y = random.randint(0, self.height)

size = random.randint(10, 30)

obj = self.grid.new\_client((x, y), (size, size), f"Obj-{i}")

color = "blue"

self.objects.append(

(

obj,

self.canvas.create\_oval(

x - size / 2,

y - size / 2,

x + size / 2,

y + size / 2,

fill=color,

),

self.canvas.create\_text(x, y, text=obj.name, fill="black"),

)

)

self.previous\_states[obj.name] = "blue" # Initialize state by name

def update\_positions(self):

for obj, oval, label in self.objects:

dx = random.randint(-5, 5)

dy = random.randint(-5, 5)

new\_x = min(max(obj.position[0] + dx, 0), self.width)

new\_y = min(max(obj.position[1] + dy, 0), self.height)

self.grid.update\_client(obj, (new\_x, new\_y))

self.canvas.move(oval, dx, dy)

self.canvas.move(label, dx, dy)

def find\_nearby(self, position, radius):

nearby\_objects = self.grid.find\_nearby(position, (radius \* 2, radius \* 2))

updated\_states = {}

for obj, oval, label in self.objects:

if obj in nearby\_objects:

self.previous\_states[obj.name] = "green"

self.canvas.itemconfig(oval, fill="green")

else:

self.previous\_states[obj.name] = "blue"

self.canvas.itemconfig(oval, fill="blue")

updated\_states[obj.name] = self.previous\_states[obj.name]

self.update\_report(updated\_states)

def update\_report(self, states):

self.report.delete(1.0, tk.END)

for name, state in states.items():

self.report.insert(tk.END, f"{name}: {state}\n")

def run(self):

def update\_loop():

self.update\_positions()

self.find\_nearby((self.width // 2, self.height // 2), 100)

self.root.after(100, update\_loop)

update\_loop()

self.root.mainloop()

if \_\_name\_\_ == "\_\_main\_\_":

# Define grid bounds and dimensions

bounds = [(0, 800), (0, 600)]

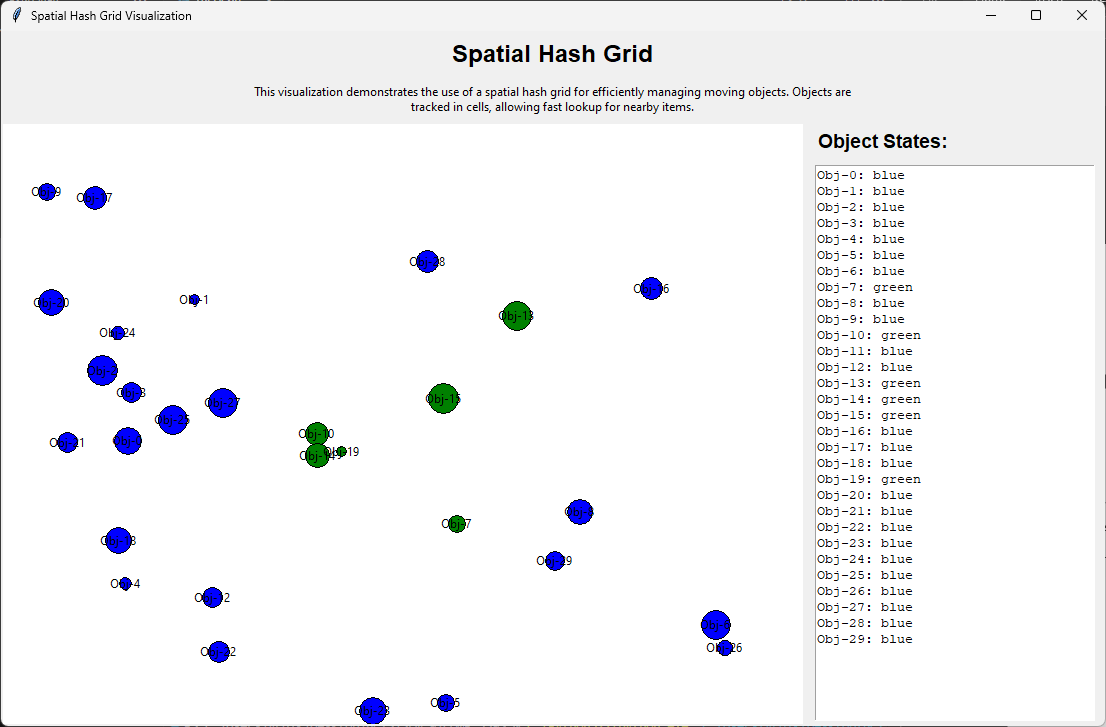
dimensions = (10, 10) # 10x10 grid

grid = SpatialHashGrid(bounds, dimensions)

visualizer = SpatialHashGridVisualizer(grid, 800, 600)

visualizer.add\_random\_objects(30)

visualizer.run()



### Testing

The spatial hash grid was thoroughly tested with various scenarios to ensure it works correctly. Tests checked key features like adding clients, updating their positions, finding nearby clients, and removing clients.

Here is the full implementation of the testing code:

from spatial\_hash\_grid import SpatialHashGrid, Client

import unittest

class TestSpatialHashGrid(unittest.TestCase):

def setUp(self):

bounds = [(0, 100), (0, 100)]

dimensions = (10, 10)

self.grid = SpatialHashGrid(bounds, dimensions)

def test\_insert\_and\_find\_client(self):

client = self.grid.new\_client((50, 50), (10, 10), 'TestClient')

nearby = self.grid.find\_nearby((50, 50), (10, 10))

self.assertIn(client, nearby, "Client should be found near its own position.")

def test\_update\_client\_position(self):

client = self.grid.new\_client((50, 50), (5, 5), 'TestClient')

self.grid.update\_client(client, (80, 80))

nearby\_old = self.grid.find\_nearby((50, 50), (5, 5))

nearby\_new = self.grid.find\_nearby((80, 80), (5, 5))

self.assertNotIn(client, nearby\_old, "Client should not be found at old position after update.")

self.assertIn(client, nearby\_new, "Client should be found at new position after update.")

def test\_delete\_client(self):

client = self.grid.new\_client((50, 50), (10, 10), 'TestClient')

self.grid.delete\_client(client)

nearby = self.grid.find\_nearby((50, 50), (10, 10))

self.assertNotIn(client, nearby, "Client should not be found after deletion.")

def test\_multiple\_clients\_in\_same\_cell(self):

client1 = self.grid.new\_client((20, 20), (10, 10), 'Client1')

client2 = self.grid.new\_client((25, 25), (10, 10), 'Client2')

nearby = self.grid.find\_nearby((22, 22), (10, 10))

self.assertIn(client1, nearby, "Client1 should be found near position.")

self.assertIn(client2, nearby, "Client2 should be found near position.")

def test\_clients\_in\_different\_cells(self):

client1 = self.grid.new\_client((10, 10), (5, 5), 'Client1')

client2 = self.grid.new\_client((90, 90), (5, 5), 'Client2')

nearby = self.grid.find\_nearby((10, 10), (10, 10))

self.assertIn(client1, nearby, "Client1 should be found near its position.")

self.assertNotIn(client2, nearby, "Client2 should not be found near Client1.")

def test\_client\_equality(self):

client1 = Client((50, 50), (10, 10), 'Client1')

client2 = Client((50, 50), (10, 10), 'Client2')

self.assertEqual(client1, client2, "Clients with same position and dimensions should be equal.")

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

### Conclusion

The spatial hash grid provides a simple yet powerful solution for managing and querying large numbers of moving objects in dynamic environments. Its efficiency lies in reducing the computational complexity of spatial queries while maintaining ease of implementation. By combining this technique with real-time visualization and reporting, our implementation showcases a practical and educational example of applying spatial hashing in real-world scenarios.

### References

Space Infinity. (2021, April 13). *Inside Amazon’s Highly Automated Robotic Warehouse*. YouTube. https://www.youtube.com/watch?v=bKRS0A-fPiE

Ten Minute Physics. (2022, March 1). *11 - Finding collisions among thousands of objects blazing fast*. YouTube. https://www.youtube.com/watch?v=D2M8jTtKi44

SimonDev. (2020, November 30). *Spatial Hash Grids & Tales from Game Development*. Youtube. https://www.youtube.com/watch?v=sx4IIQL0x7c

Henning. (2023, March 24). Spatial Hashing vs ECS. https://leetless.de/posts/spatial-hashing-vs-ecs/