Efficiently Storing and Querying a Star Database

Likitha Madhav M.R

June 02 2024

Introduction

The task of efficiently storing and querying a star database on a device with limited memory and processing power can be approached using various data structures like the octree, k-d tree, or spherical grid with hashing. The octree is suitable for handling uneven distributions of stars, while the k-d tree, although efficient for range searches in k-dimensional space, can be complex to implement and less intuitive for spherical coordinates. Hence, a spherical grid with hashing would be an efficient data structure for this problem. It is simple, easy to understand, provides direct access to grid cells by minimizing memory access overhead, and is efficient for uniformly distributed data.

Organization of the Database

The celestial sphere is made up of latitudes and longitudes ranging from -90° to +90°, and -180° to +180°, respectively. Dividing the range of latitude into 10° intervals gives 18 divisions, i.e., (90 - (-90)) / 10. Similarly, dividing the range of longitudes into 10° intervals gives 36 divisions. By combining the latitude and longitude divisions, we get a total number of cells on the celestial sphere, i.e., 18 x 36 = 648 cells. Each cell represents a specific region on the sphere defined by a range of latitudes and longitudes. The celestial sphere can now be visualized as a spherical grid with cells. To map the cells within the spherical grid, a hash table is used, where each key corresponds to a cell in the spherical grid. Each value in the hash table is a list of stars within that cell. Each star is represented by its 3D coordinates and additional properties like magnitude.

Lookup Procedure

To identify a relevant cell, convert the query vector's 3D coordinates to spherical coordinates (latitude and longitude) to determine the corresponding cell index. To ensure stars within 10 degrees are considered, the neighboring cells adjacent to the identified cell are also checked. For each star in the relevant cell, the angular distance from the query vector is computed using the dot product to

find the cosine of the angle between the vectors. Once this is done, stars are filtered to include only those within the 10° threshold.

Number of Items to Lookup and Computations per Query

For a given query, the relevant cell and its neighboring cells are accessed. Assuming a uniform distribution, each cell will contain approximately 10,000 / 648 ≈15 stars. For each query, the dot product between the query vector and each star's vector is computed to determine the angular distance, and the cosine value is computed to filter stars within 10°.

Performance

The spherical grid with hashing balances memory usage and computational requirements efficiently. By directly mapping stars to grid cells the method ensures fast query response time.

Algorithm

The provided code implements the spherical grid with hashing using C data structures. It includes functions for initializing the grid, inserting stars, computing dot products, checking angular thresholds, and querying stars within a 10-degree range of a given 3D vector.

```
#include <math.h>
#include <stdlib.h>

#define MAX_STARS_PER_CELL 30  // Estimated average number of stars per cell,
                    //handle cases where there is clustering or uneven distribution
#define NUM_LONGITUDE_DIVISIONS 36  // Number of divisions along the longitude
#define NUM_LATITUDE_DIVISIONS 18  // Number of divisions along the latitude
#define TOTAL_CELLS (NUM_LONGITUDE_DIVISIONS * NUM_LATITUDE_DIVISIONS)
                                  // Total number of grid cells

typedef struct {
    float x, y, z;  // 3D coordinates of the star
    float magnitude;  // Additional properties such as magnitude
} Star;

typedef struct Cell {
    Star stars[MAX_STARS_PER_CELL];  // Array to store stars within the cell
    int star_count;  // Number of stars currently in the cell
} Cell;
```

```
// Array representing the spherical grid
Cell grid[TOTAL_CELLS];

// Function to compute the dot product of two vectors
float dot_product(float v1[3], float v2[3]) {
    return v1[0] * v2[0] + v1[1] * v2[1] + v2[2] * v2[2];
}

// Function to check if a star is within the 10-degree threshold of a query vector
int is_within_threshold(Star* star, float query_vec[3], float cos_threshold) {
    float star_vec[3] = {star->x, star->y, star->z};
    return dot_product(star_vec, query_vec) >= cos_threshold;
}

// Function to find the grid cell index for a given star or query vector
int find_cell_index(float x, float y, float z) {
    // Convert Cartesian coordinates to spherical coordinates (latitude and longitude)
    float lat = atan2(y, sqrt(x*x + z*z)) * 180.0 / M_PI;
    float lon = atan2(z, x) * 180.0 / M_PI;
    // Calculate latitude and longitude indices
    int lat_idx = (int)((lat + 90) / 10);
    int lon_idx = (int)((lon + 180) / 10);
    // Return the cell index based on latitude and longitude indices
    return lat_idx * NUM_LONGITUDE_DIVISIONS + lon_idx;
}

// Function to insert a star into the appropriate cell in the grid
void insert_star(Star* star) {
    int cell_idx = find_cell_index(star->x, star->y, star->z);
    if (grid[cell_idx].star_count < MAX_STARS_PER_CELL) {
        grid[cell_idx].stars[grid[cell_idx].star_count++] = *star;
    }
}

// Function to initialize the grid by setting the star count of each cell to 0
void initialize_grid() {
    for (int i = 0; i < TOTAL_CELLS; i++) {
        grid[i].star_count = 0;
    }
}

// Function to query stars within 10 degrees of a given 3D vector
void query_stars_within_10_degrees(float query_vec[3], Star* result[], int* result_count) {
    // Calculate the cosine of the 10-degree threshold
    float cos_threshold = cos(10 * M_PI / 180.0);
```

```
    // Find the cell index of the query vector
    int cell_idx = find_cell_index(query_vec[0], query_vec[1], query_vec[2]);

    *result_count = 0;
    // Iterate through all cells in the grid (including neighboring cells)
    for (int i = 0; i < TOTAL_CELLS; i++) {
        for (int j = 0; j < grid[i].star_count; j++) {
            // Check if each star in the cell is within the 10-degree threshold
            if (is_within_threshold(&grid[i].stars[j], query_vec, cos_threshold)) {
                result[*result_count] = &grid[i].stars[j];
                (*result_count)++;
            }
        }
    }
}

// Main function
int main() {
    // Initialize the grid
    initialize_grid();

    // Example: Insert a star into the grid
    Star star1 = {1.0, 0.0, 0.0, 1.0};
    insert_star(&star1);

    // Array to store the result of the query
    Star* result[100];
    int result_count;

    // Query vector
    float query_vec[3] = {1.0, 0.0, 0.0};
    // Query stars within 10 degrees of the query vector
    query_stars_within_10_degrees(query_vec, result, &result_count);

    // Print the result
    for (int i = 0; i < result_count; i++) {
        printf("Star found: (%f, %f, %f)\n", result[i]->x, result[i]->y, result[i]->z);
    }

    return 0;
}
```