Box Occlusion Analysis

Elliot Wyrick

Abstract

This report presents an algorithm for calculating the total occlusion angle of randomly generated rectangles as viewed from an origin point. The solution achieves O(n log n) time complexity through efficient angle calculation, interval merging, and geometric processing techniques. The program itself uses SFML (simple and fast Multimedia library) integration in order to provide visual confirmation of the algorithm's effectiveness.

Problem Statement

This program tackles two main problems. These include:

1. Building a framework which can generate and display n random rectangles. These rectangles need to:
    a. Have a random orientation around the center of the 2D view.
    b. Have randomly generated side lengths.
    c. Each have an individual random rotation.
    d. Not collide or intersect in any way with any other box.
2. Building an efficient algorithm for finding all ranges in which any two or greater boxes occlude the other, and consolidate them in an efficient manner.

Approach and Algorithm

1. Rectangle Generation (box_gen.cpp)

The algorithm begins by generating n random rectangles with the following parameters:

- **Distance from origin**: Random value between min_radius and max_radius.
- **Rectangle dimensions**: Random length and width between min_box_bounds and max_box_bounds.
- **Orientation angle**: Random angle (0-359°) determining the rectangle's rotation
- **Position angle**: Random angle (0-359°) determining the rectangle's position relative to the origin.

Corner Calculation

For each rectangle, four corners are calculated using trigonometric transformations:

corner0 = (distance * cos(position_angle), distance * sin(position_angle))
corner1 = corner0 + (width * cos(orientation_angle), width * sin(orientation_angle))
corner2 = corner0 + (-length * sin(orientation_angle), length * cos(orientation_angle))
corner3 = corner2 + corner1 - corner0

Collision Detection

A bounding box intersection algorithm prevents rectangle overlap:

- Each rectangle's bounding box is computed by finding the minimum and maximum x and y values for all corners.
- New proposed rectangles are validated against existing ones by comparing the new boxes' bounds against all other preexisting boxes' bounds.
- Invalid placements are rejected and regeneration occurs

2. Occlusion Angle Calculation (calculate_occlusion.cpp)

The core algorithm processes rectangles to determine occlusion angles:

Step 1: Angle Range Calculation

For each rectangle, the algorithm:

1. Finds the minimum and maximum angles spanning the rectangle
2. Handles wraparound cases where rectangles span the 0°/360° boundary

Step 2: Wraparound Handling

When a rectangle's angular span exceeds 180°, it indicates wraparound across the 0°/360° boundary. The algorithm splits such rectangles into two intervals:

- **Below interval**: [min_angle_above_180°, 360°]
- **Above interval**: [0°, max_angle_below_180°]

Step 3: Interval Sorting

The algorithm employs merge sort (O(n log n)) to sort angular intervals by their start angles. This allows for a linear sweep in the next step which prevents the need for multiple loops.

```
void mergesort(vector<array<float, 2>>& box_angles, int left, int right) {
```

```
    if (left < right) {
        int mid = left + (right - left) / 2;

        mergesort(box_angles, left, mid);       // Sort first half
        mergesort(box_angles, mid + 1, right); // Sort second half

        merge(box_angles, left, mid, right);   // Merge sorted halves
    }
}
```

Step 4: Occlusion Sweep

After sorting, the algorithm performs a linear sweep through the intervals to identify overlapping regions that represent occlusion angles.

```
// Overlap detection sweep
float current_max = minmax[0][1];

for (int i = 1; i < minmax.size(); i++) {
    float next_min = minmax[i][0];
    float next_max = minmax[i][1];

    if (next_min < current_max) {
        // Intersection region found
        overlap.push_back({next_min, min(current_max, next_max)});
    }

    // Always extend current max
    current_max = max(current_max, next_max);
}
```

Step 5: Merging Occluded Angles

In order to prevent overlapping in the final set of occluded ranges, the overlapping sectors are put through a merging loop in order to consolidate and simplify the final set.

```
// Merge overlapping intervals
for (int i = 1; i < intervals.size(); i++) {
    if (next_min < current_max) {
        // Overlap detected - merge intervals
```

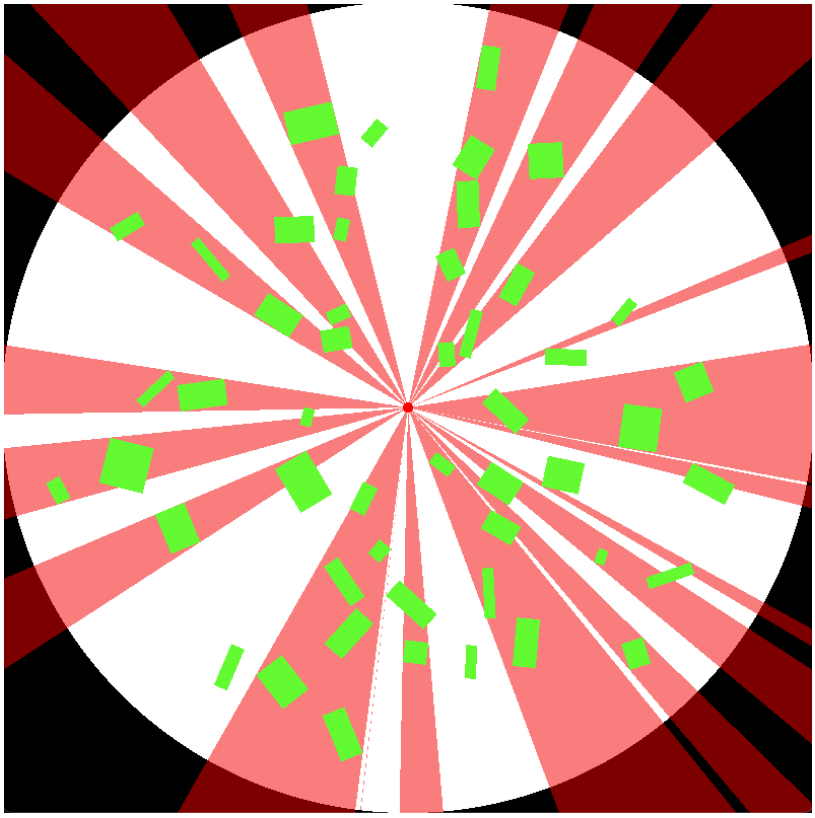
```
            overlap.push_back({next_min, min(current_max, next_max)});
        }
        current_max = max(current_max, next_max);
    }
```

3. Visualization (visuals.cpp)

The SFML-based visualization renders:

- Green rectangles showing generated boxes.
- Red triangular sectors highlighting occlusion angles.
    - These are set to half opacity to confirm no overlapping sectors.
- Circular boundary representing the box generation range.
- Central red dot marking the origin, the point of view from which all occlusion calculations are made.

Example output with 50 generated boxes:



Time Complexity Analysis

The algorithm achieves O(n log n) complexity through the following operations:

Operation	Complexity	Justification
Rectangle Generation	O(n)	Linear iteration with constant-time geometric calculations
Angle Calculation	O(n)	Four corner angles per rectangle, constant per rectangle
Wraparound Processing	O(n)	At most doubles the interval count, maintains linear complexity
Merge Sort	O(n log n)	Standard merge sort complexity
Interval Merging	O(n)	Single linear pass through sorted intervals
Total	**O(n log n)**	Dominated by sorting operation

Conclusion

This implementation successfully addresses the rectangle occlusion problem with highly optimal O(n log n) time complexity. The algorithm efficiently handles geometric edge cases, provides accurate occlusion angle calculations, and includes visualization for validation.