6.172 Project 2: Beta Write Up

Sam Ubellacker, Rui Li
subella@mit.edu, ruili@mit.edu

October 2018

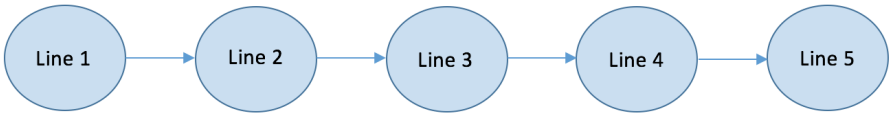
1 Introduction

The initial implementation iterates over each line in the frame, checking for collisions with every other line. This algorithm is O(n^2), which becomes incredibly slow for a large amount of lines. A quadtree structure aims to remedy this, by recursively partitioning the grid into quadrants. Each quadrant contains lines that can only collide with other lines in the same quadrant. This algorithm greatly reduces the amount of collision checks needed to be done, as each line is only compared against other lines in the same quadrant. Lines that span across several quadrants are handled separately and are compared against every other line in the parent quadrant.

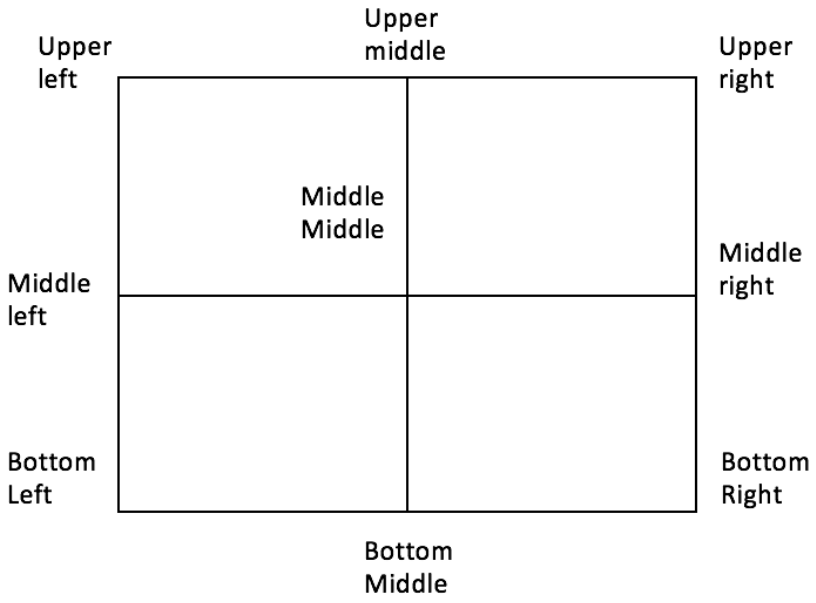
2 Design Overview

2.1 Data Structures

For the beta, we implemented a recursive Quadtree structure as suggested. For efficiency, we malloc one global linked list, where each node in this list contains a pointer to one original line in collisionWorld, and a pointer to a next node. In this way, each node represents one line in collision World, so the global linked list will store all the lines in the world. Here is what the linked list looks like:



Then for our Quadtree structure, we store the bounds that that quadtree corresponds to using a Rectangle structure. Each Rectangle structure contains the vector representation of the upper left, upper middle, upper right, middle left, center, middle right, bottom left, bottom middle, and bottom right coordinates for the bounds. Here is an illustration of what the Rectangle struct represents:



We also store for a quadtree a pointer to the head and the tail of the list of lines that fit in the quadtree. Similarly, we store a misfit head and a misfit tail pointer to point to the list of lines that cannot fit within the bounds of the quadtree's 4 children quadrants. Additionally, we store 4 pointers to the children Quadtrees. Here is what our Quadtree struct looks like:

Quadtree:
- Node* head
- Node* tail
- Node* misfit head
- Node* misfit tail
- Quadtree* children[4]
- Rectangle* bounds

2.2 Procedure

We first initialize a root Quadtree that contains all lines. At each level of the recursion, we test each point in the line's spanning parallelogram to see if the line belongs in one of the four children, or if it is a misfit. After iterating through all lines, we end up with five linked lists: one linked list for each child Quadtree containing all the lines that belong in each child, and a 5th linked list for the misfits in the current quadrant. Since we are not mallocing extra copies of the global linked list we started with, we reassign the links so that all misfits are linked together, and all nodes in the same quadrant are linked together. The

links between these lists will be broken so we end up with 5 disjoint lists. So essentially, at each step of the recursion we are breaking up the global linked list into 5 disjoint lists.

Once all the lines have been assigned as belonging to either one of the four children, or a misfit, we count the total number of times the misfits have collided with all other lines (those in the 4 children and other misfits), and increment a global count. We then recurse on the four children of this Quadtree.

Finally, when we get to a leaf Quadtree, where the number of lines in it is less than some threshold(a tunable parameter we set), we count the number of pairwise line intersections in it, and increment our global count.

3 Completeness, Performance, Bugs

Our beta submission consists of this serial Quadtree implementation described above. Our implementation was tested with every available input test, and collision count was verified using the original implementation for every input. The quadtree design greatly decreased execution time over the original version. For mit.in at 4000 frames, executation was around 6 times faster than the original.

We are aware of a memory leak bug that we have yet to fix. When mallocing the global linked list, we never destroy and free it at the end of a frame. As a result, memory usage compounds each frame. For a very large frame input, the program may run out of memory. This issue will be addressed in the final version.

4 Future Plans

With our serial quadtree working, the next major step is to parallelize it. Our initial plan is to run children quadtrees in parallel. Because each quadtree is working on a disjoint section of the global linked list, there should be no determinacy races. We plan to use a reducer to handle children aggregation. We will explore other various improvements mentioned in the prompt, such as updating the quadtree's state each frame instead of reconstructing it.