6.172 Project 2: Final Write Up

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

October 2018

1 Introduction

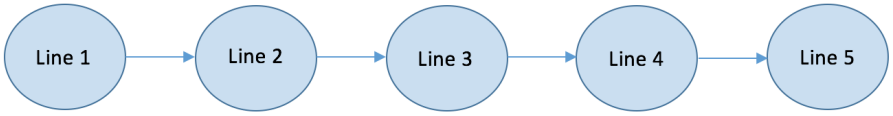
The initial implementation iterates over each line in the frame, checking for collisions with every other line. This algorithm is O(n^2), which becomes incredibly slow for a large amount of lines. A quadtree structure aims to remedy this, by recursively partitioning the grid into quadrants. Each quadrant 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

For our Beta submission, we implemented a serial quadtree. For our final submission, we parallelized the quadtree, optimized our intersection collision detection methods, and made many tweaks to our code to make it faster. We will proceed to describe the design we had for our beta submission and then describe the optimizations we made for our final submission.

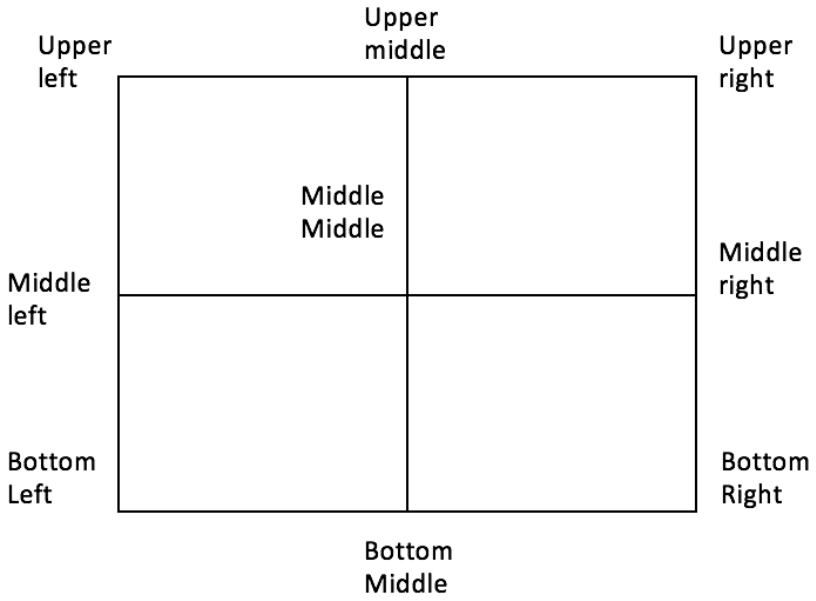
2.1 Data Structures

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 collisionWorld, 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 misfit heads and tails corresponding to misfits in the top half, left half, bottom half or right half of the current frame. 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 or the hemispheres. 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
- Node* misfit head_left
- Node* misfit tail_left
- Node* misfit (head and tail for all hemispheres)
- 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 nine linked lists: one linked list for each child Quadtree containing all the lines that belong in each child, a linked list for each hemisphere's misfits, and a linked list for global misfits. 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 9 disjoint lists. So essentially, at each step of the recursion we are breaking up the global linked list into 9 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 other lines (those in the 4 children and other misfits), and increment a global count. By categorizing the misfits into hemisphere's, we reduce the number of quadtrants we need to check intersections with. 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 Optimizations for Final Submission

3.1 Parallelization of the Quadtree

In order to parallelize our quadtree, we needed to make sure there were no race conditions when appending to the IntersectionEventList. To do this we created a reducer for the intersectionEventList. Each process would then update the intersectionEventList using the reducer view. We also needed a way to count the intersections accurately without race conditions between the processes. This was accomplished by extending the IntersectionEventList struct to contain a size field, which would store the size of the intersectionEventList. Since we already have a reducer for IntersectionEventList, the count could be incremented by all the processes in parallel without race conditions.

Any process where large amount of work was done was pulled into a helper function and attempted to be parallelized. We experimented parallelizing many functions in our code, and many resulted in further speedups.

Performance wise, the parallelization of the quadtree gave us a 3 second speedup on mit.in and a 20 second speedup on koch.in.

3.2 Reducing line intersection Checks

We then further optimized our code by modifying our intersection detection algorithm such that our intersect lines check would be called less often, as that seemed to be a performance bottleneck we found after running perf-report. Before checking if two lines intersect, we first check whether their parallelogram collide. If their parallelograms do not overlap, then no collision is possible, and we short circuit the check call to the intersect lines check.

This algorithmic change to our code resulted in a 6 second speedup on mit.in and a 4 second speedup on koch.in.

3.3 Subdividing misfit lines

We realized that we could speed up our code by subdividing the misfit lines into upper, lower, and right hemisphere misfits. This way we reduce the number of pairwise collision checks for the misfits. For example, now, for upper hemisphere misfits we no longer need to check pairwise intersections with the bottom hemisphere misfits.

This change gave us another performance boost of 1 sec on mit.in and 1 second on koch.in.

3.4 Other changes

Many fields in the line struct were precomputed on creation of the quadtree to avoid redundant calculations. Many simple functions were replaced with macros for additional speedup. Functions were declared static inline to reduce overhead.

4 Addressing MITPOSSE comments

We addressed MITPOSSE comments by fixing our code according to their suggestions and replying to their comments on Github. At a high level, there were a few areas they suggested we modify, listed below.

4.1 Use of Const

One of our mentors reminded us to use const keyword when parameters were not being changed. We have fixed our code so that parameters that are not modified in a function are made const.

4.2 Code Refactor to avoid copy-paste

We have refactored our code so that there is very minimal duplicate code.

4.3 Adding function comments

In our header files, we documented the functionality of all of our methods.

4.4 Using better variable names

One of our mentors had mentioned that some of our variable names were not very elucidating. We have changed some of the variable names so that it is clear what the variable holds.

4.5 General Comments

As a whole, redundant code was put into helper functions and the code was majorly reformatted to be read more easily. Function arguments were modified so that multiply different processes could leverage the same function call. Indentation, spacing, and other stylistic changes were taken into account.

5 State of Completeness

We have made significant performance improvements from our beta submission. We have reduced the runtime of mit.in from 11.01 sec down to 1.56, koch.in from 45.9 to 5.11. This was the best we could do before the deadline. If we had more time, we would make further optimizations to reduce our runtime to below the final-cutoffs. We would replace our linked lisk structures with arrays to leverage this more optimized data type.

We also fixed our memory leak bug we had in our beta submission by adding free methods to free our quadtree and global linked list at the end of every frame. This would prevent out of memory errors if there test input were to be very large.