6.172 Project2 Beta Write-up

Gregory Hui, Hyun Sub Hwang

Write-up 1. Run the unmodified screensaver using *cqrun* and report its runtime for 1000 frames. Report the number of collisions detected during the screensavers execution.

Here is the output of the program.

```
Elapsed execution time: 15.908931s
170 Line-Wall Collisions
2097 Line-Line Collisions
```

The number of line-wall collisions is 170 and the number of line-line collision is 2097.

Write-up 2. Figure 2 illustrates that quadtrees can readily handle points, and Figure 3 suggests that line segments can at least sometimes be handled by quadtrees. However, in Figure 4 one of the line segments cannot fit into any of the partitions. Is this a problem for the quadtree? What can you do about it? Can you still use quadtrees to effectively speed up collection detection? Hint: Do all line segments need to be stored in leaf nodes?

For line segments that cannot fit into any of the partitions, we think of them as being stored at the parent node, and are compared to all of the other line segments in this parent nodes subtrees (including other line segments that are stored in this root node). This still speeds up collision detection because we only need to check for collisions with line segments that are descendents of the node that stores this not-fitting-into-partition line segment, so at every subsequent quadtree level, we do fewer and fewer such collision detection checks.

Write-up 3. Report the number of collisions detected during the screensavers execution.

After implementing the quadtree structure, we got the following output.

```
Elapsed execution time: 4.703436s
170 Line-Wall Collisions
2097 Line-Line Collisions
```

It reported the same number of collisions.

Write-up 4. Measure and report how long it takes for the program to execute with the more efficient collision detection. Make sure to run the program without the graphics flag to get accurate performance results. Compare with the original runtime. Was the speedup what you expected?

It took 4.703436s with the quadtree, 15.908931s originally. This is more than a 3x speedup, and was around what we expected with a basic quadtree as using quadtree effectively reduces the number of calls to *intersect*.

Write-up 5. Describe any design decisions you made while rewriting collision detection to use a quadtree. For example, once you built a quadtree, how did you use it to extract collisions? How did you store line segments in each quadtree node?

Our first implementation of quadtree nodes do not store any lines; *quadtree.c* is more like a recursive function than a data structure. We originally would store line segments that did not fit in any partitions in each node, but then realized that we could compare then in the quadtree function, but then we dont need to store them anymore. So it became a recursive function that compares the not-fitting-in-partition lines with all other lines, then recursively calling the function on the 4 partitions with the lines that belong in those partitions. The base case was when there was less than or equal to a threshold # of elements called on the recursive function, in which case we just checked all pairs of lines for collisions and did not recurse.

Write-up 6. Vary the maximum number of elements N a quadtree node can store before it needs to be subdivided. Measure the performance impact that this has on the runtime. Briefly comment on why this did or did not have a performance impact.

With different maximum number of elements (N) in one quadtree, we got the following result with "*cqrun ./screensaver 1000*".

N	3	4	5	10	20	35	50
Elapsed Time(s)	4.703436	4.702253	4.697282	4.695603	4.679690	4.711269	4.715992

Having N = 20 seems to be around the optimal number. This has a performance impact because recursive calls are relatively expensive (need to jump, push variables to stack, pop variables from stack, etc.), but the cost of doing the naive all-pairs checking grows quadratically with N. So finding the balance between reducing number of recursive calls and reducing size of all-pairs checking impacts performance.

Write-up 7. Describe the optimizations that you tried and how well they work. This should be the same as the number you recorded for the unmodified code.

(Together) We tried representing the array of lines as a linked list, so that we would not have to allocate memory to the indices of lines and pass them down the recursive calls (only the first and last pointers of the linked list). This has a very slight runtime improvement (around 4.65s fastest).

(Together) We tried storing the length of each line as an instance variable. Then the collision world can reference this value rather than recalculate the lines length each time. However, this resulted in a code slowdown (to 4.70s). Perhaps the extra space needed to store the lines length has a greater effect than the savings from re-calculating every time.

(Gregory) An optimization that results in a significant performance improvement is computing the specific quadrants that each not-fitting-in-partition lines are present in, and then comparing each not-fitting-in-partition line only with the specific quadrants that it exists in. This results in a runtime of 8.486181s with 4000 timesteps (cqrun ./screensaver 4000) and around 3.2s with 1000 timesteps. We will use the 4000 timesteps benchmark from here on out. An important detail is that it is possible for a line's parallelogram representation of its movement over a timestep to have all 4 points in 2 quadrants, but for the parallelogram to intersect 3 or all 4 quadrants. This occurs when the line contains points in opposite quadrants (quadrants 1 and 3 or 2 and 4). In these cases, we say that the line is present in all 4 partitions, and compare the line with all other lines that don't fit in partitions as well as all the lines in the 4 subtrees. In the other case, the line will belong to only the two adjacent quadrants that it has points in, and we compare this line with other not-fitting lines and only the lines in the 2 subtrees that the line is present in.

(Hyun Sub) Another optimization we tried was updating the quadtree each timestep rather than rebuilding it. It reduces the number of calls to *malloc* and *free*. This resulted in a small speedup, which is about 0.02s 4000 timesteps.

(Gregory) Another optimization we tried is having bit operations for deciding which quadrant each line belonged to. With these bit operations, the runtime with 4000 timesteps improved slightly to 8.275778s.

(Gregory) We discovered a bug in determining which quadrant lines should be put in. The quadrants lines were being mistakenly assigned to the wrong quadrants, causing more comparisons between lines that should not be compared. After fixing this bug by assigning quadrant lines to the correct quadrant, we shaved another .5s off the runtime with down to 4.0s 4000

(Hyun Sub) Another major performance improvement was achieved by improving performance of functions in *intersection_detection.c*. Although implementation of methods to test collisions of lines is efficient, all of *intersect*, *pointInParallelogram*, and *intersectLines* uses the expensive function *direction*. We could achieve the big performance achievement by adding a shortcut that returns the result early with inexpensive operation (e.g) comparison). We check whether points can be vertical line or horizontal line using comparison of coordinates, which is cheap operation, and return *false* early if this is the case.

(Hyun Sub) Also, for some functions, the order of calculation was crucial. For example, we do not have to create vector structures *v1* and *v2* in the function *intersect* before we calculate *angle*. Also, in *quadtree.c*, as it it is more likely that line falls into one of child nodes, we check whether line falls into one of child nodes first to improve the performance. With this changes, we could achieve 2.5s with *"cqrun ./screensaver 4000"*.

(Hyun Sub) We also tuned hyperparameters: *THRESHOLD*, which is the maximum number of lines in one node, and *MAXIMUM_DEPTH*, which is the maximum depth of tree. The optimal value is *(THRESHOLD, MAXIMUM_DEPTH) = (5,3)* and we achieved 2.49s with *"cqrun ./screensaver 4000"*.

Summary

We have used a quadtree implementation to speed up the performance of the collision detector. Major performance improvements were also gained by optimizing the intersection detection methods. For the final, we will try to use parallelization to further improve the performance of our program, and perhaps look into ways of updating the quadtree in a more efficient manner; instead of reinserting all lines, only shift lines from quadtrees if their movement shifts them to another quadrant.