Aysylu Biktimirova
Nathan Rittenhouse

Project 2.2 Write-up

4.1
nathan_@ipratropium:~/6.172/project2.2$ cqsub ./Line 4000
job id is 4445. output file name is nathan_-30.
Fri Oct 21 11:56:34 2011 active
--- nathan_-30.stdout ---
Number of frames = 4000
1347 Line-Wall Collisions
20276 Line-Line Collisions
88.861 seconds in total

4.2
Is this a problem for the quadtree? What can you do about it?
It is a problem in the sense that one would have to perform a pairwise test of all lines that did not fit into a quadtree with all of the other lines. However, a more efficient way would be to do a pairwise intersection check with the non-quadtree line and all of the lines in the quadtrees that the non-quadtree line crosses.

Can you still use quadtrees to effectively speed up collision detection?
Yes. It highly likely that most of the lines will in fact fit into one of the quadtrees. So, while the quadtree optimization is still O(n^2), in most cases it will not reach n^2 comparisons.

4.3 *Measure and report how long it takes for the program to execute with the more efficient collision detection. Compare with the original runtime. Was the speedup what you expected?*

Number of frames = 4000
1091 Line-Wall Collisions
20321 Quadtree Line-Line Collisions
17.922 seconds in total

Assuming that the entire screen is filled with lines, and we are dividing the screen into four parts, we are doing ¼ of number of comparisons for each line. Therefore, we were expecting a 4x speedup. However, the screen wasn't entirely filled with lines, and lines were concentrated in certain parts of the screen, which gave us even bigger speedup -- observed speedup was 5x.

4.4 *Describe any design decisions you made while rewriting collision detection to use a quadtree. Once you built a quadtree, how did you use it to extract collisions? How did you store line segments in each quadtree node?*

A Quadtree object recursively divides itself into four Quadtree child objects if the number of lines contained in the Quadtree is bigger than the threshold or the depth of the Quadtree is less than the threshold depth. These Quadtrees correspond to the four different quadrants of the larger square that the parent Quadtree owns. The lines of the parent Quadtree are distributed among its children based on whether the line's points both fit into the area defined for the child. However, lines that span multiple child Quadtrees are put into a spanningLines vector and not propagated down to the Quadtree's descendents. If the Quadtree has one or more spanning lines, we detect collisions between the spanning lines and the lines that are located in the children of the Quadtree. If the Quadtree has exactly or fewer than the threshold number of lines, or its at the maximum depth, then we perform pair-wise detection of collisions on the

Aysylu Biktimirova
Nathan Rittenhouse

Quadtree's lines.

Line segments of each Quadtree are stored in a vector named lines. Each Quadtree has three vectors of lines: lines that are located in the Quadtree, spanning lines that span across multiple children of the Quadtree, and the line of intersected pairs of lines that are to be resolved by solveCollision() function.

An instance of the Quadtree object is created by the CollisionWorld, and its dimensions are initialized to that of the BOX. The lines of the CollisionWorld are also the lines of the root Quadtree. The Quadtree calls descend() and returns the number of line-line collisions it detected. In addition, Quadtree updates the lines with the new velocities that resolve the collisions, so that the lines get "unstuck".

Design ideas and justifications:
Idea 1: each line has info about which Quadtree it is in
Idea 2: each Quadtree has info about the lines in its defined area

We picked idea 2, since to detect collisions within a Quadtree, we need to narrow down our search to the lines contained by the Quadtree. Idea 1 would require unnecessary number of extra computations.

Idea 3: A spanning line is present in all the Quadtrees it crosses . This can be determined based on whether the spanning line intersects the boundaries of a Quadtree.
Idea 4: Treat spanning lines individually; perform (n-1)^2 operations to detect collisions between it and all the other n-1 lines contained in a Quadtree.

We picked idea 4, since this prevents us from double-counting the lines and their collisions as we recurse down the Quadtree.

Idea 5: Quadtree dimensions are represented by doubles: do floating point comparisons 4 times.
Idea 6: Quadtree dimensions are represented by integers: do floating point multiplication 1 time when converting from box to window dimensions.
We picked idea 5, since it doesn't require rewriting the intersection functions in IntersectionDetection files. However, in the future we might write a version of the functions that operate on integers as one of our optimizations.

Difficulties during design: We attempted to implement our own line-wall collision detection algorithm before update of the positions of the lines. However, the CollisionWorld detects line-wall collisions after the positions of the lines have been updated according to their velocity. In the end, we decided to use the CollisionWorld mechanism for detecting lines intersecting with the wall. This is because the step that is responsible for detecting line-wall collisions is run after the position update step. Seeing as line-wall intersections require a constant number of comparisons each, we decided that it would require too much overhead and complexity to warrent performing line-wall detection inside the Quadtree class.

4.5

Figure 1 shows how execution time of our program changes based on the maximum number

of lines (between 1 and 40) in a quadtree. We observed that with no more than 18 lines in any quadtree, we get best performance results. We get comparatively worse results with fewer than 18 lines in a quadtree, since we end up dividing the window of the screensaver into larger number of quadtrees. With more than 18 lines in a quadtree, we spend more time performing pairwise comparison, and spanning lines, lines that span across more than one quadtree, are being compared to a larger number of other lines in a quadtree.

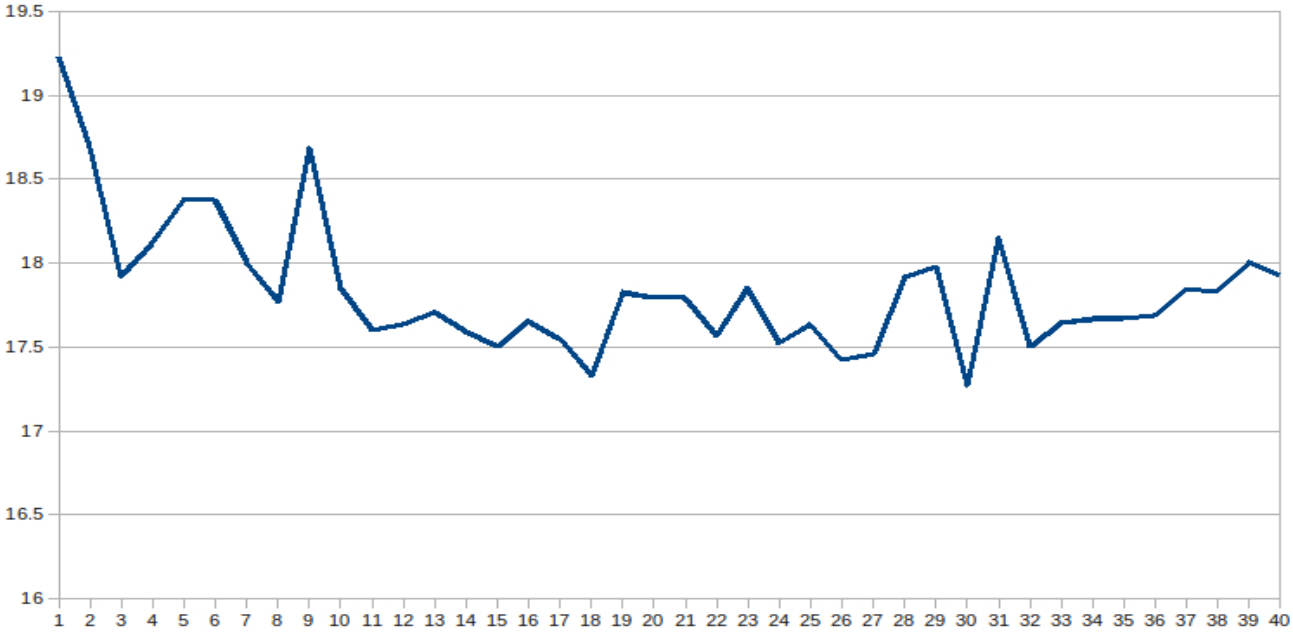

Figure 1: Plot of the execution time of the screensaver as a function of maximum number of lines in the quadtree

4.6

With division threshold = 20:
Format is (maxDepth, runtime in seconds):
1: 32.385
2: 19.819
3: 17.762
4: 17.762
5: 17.77
6: 17.768
7: 17.787
8: 17.774
9: 17.779
10: 17.773
11: 17.775
12: 17.781
13: 17.786
14: 17.78
15: 17.781
16: 17.774
17: 17.781
18: 17.77
19: 17.789

Aysylu Biktimirova
Nathan Rittenhouse

20: 17.77
21: 17.778
22: 17.765
23: 17.778
24: 17.783
25: 17.774

We saw best performance with depths 3 and 4. We don't really see much speedup as we increase the depth, since as we divide recursively into more quadtrees, the number of lines in them will decrease, and we will have a lot of quadtrees with very few lines. This causes small gains from having few lines in a quadtree, and big penalties for dividing the quadtrees further. We observe that performance is worst at depth 1, since at that depth we need to perform pairwise comparisons on ¼ of lines, and (n-k)^2 + k^2 comparisons on k spanning lines.

4.7

We precalculate lengths of all lines as they are created at the beginning, and store them in a mass field of a Line.

We decided not to update the information about which Quadtrees store what lines because we handle spanning lines separately from the lines that are contained within a Quadtree. The overhead of updating the lines information would be bigger than the performance gain.

One of the useful optimizations is to improve on calculations of lines. Our idea is to perform all airthmetic operations on integers instead of doubles. We haven't implemented this optimization for the beta, because it would require us to rewrite the current implementation of intersect() in IntersectionDetection.

5.1

Our top six functions as detected by the gprof.

time	seconds	seconds	calls	s/call	s/call	name
23.85	8.54	8.54	1949202908	0.00	0.00	crossProduct(double, double, double, double)
22.47	16.58	8.04	1949202908	0.00	0.00	direction(Vec, Vec, Vec)
18.41	23.17	6.59	389840038	0.00	0.00	intersectLines(Vec, Vec, Vec, Vec)
7.40	25.82	2.65	97460038	0.00	0.00	intersect(Line*, Line*, double)
4.64	27.48	1.66	97460689	0.00	0.00	pointInParallelogram(Vec, Vec, Vec, Vec, Vec)
3.65	28.78	1.31	292383181	0.00	0.00	Vec::operator-(Vec)

5.4 *Describe the changes you have made.*

We parallelized the outer for-loop in detectLineLineCollisions() and detectLineLineCollisionsTwoLines() using cilk_for. We saw 2.3x speedup compared to the previous implementation. We also experimented with pragma grainsizes, but determined that the default one produces best results. We changed the intersectedPairs<IntersectionInfo> list to be a Cilk's reducer_list_append.

Aysylu Biktimirova
Nathan Rittenhouse

In addition, we parallelized our descend() function. We parallelized it to do collision detection on each of the four Quadtrees in parallel. We created a Cilk's reducer_oppadd<int> that accumulates the number of line-line collisions.

5.5 *Determine the span and work of your parallel code by executing it through CilkView. How much parallelism do you see?*

Work :	113,676,216,391 instructions
Span :	3,008,361,764 instructions
Burdened span :	10,942,886,000 instructions
Parallelism :	37.79
Burdened parallelism :	10.39

We got 5.6x speedup with 12 workers vs 1 worker (18.861 vs 3.372 sec). See Figure 2 for the output produced by Cilkview.

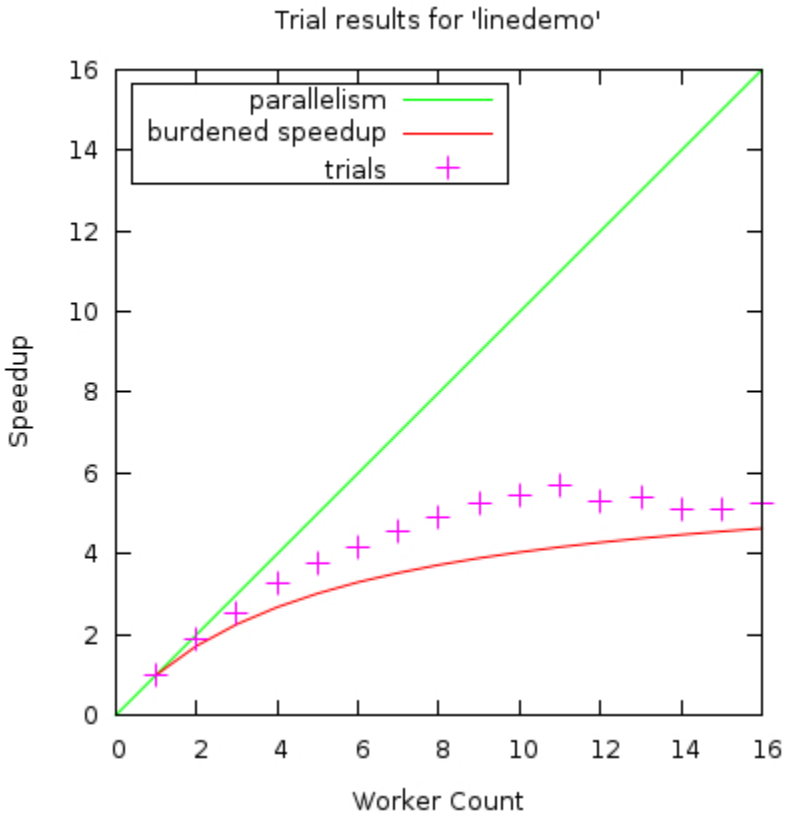

Figure 2. Cilkview output

Vary the parameters of your quadtree (such as the max depth and max number of nodes per quadrant), as well as any other spawn cut offs you have used in your code. What is the

Aysylu Biktimirova
Nathan Rittenhouse

maximum amount of parallelism you can achieve?

With division threshold set at 7, and max depth set at 3, we got comparable parallelilsm. Since runs of Cilkview take approximately 20 mins, we weren't able to perform more experiments as we were trying to meet the beta deadline. We'll perform more experiments and determine our optimal max depth and division threshold by the final submission.

Parallelism Profile

Work :	113,684,628,518 instructions
Span :	3,012,510,036 instructions
Burdened span :	10,946,864,473 instructions
Parallelism :	37.74
Burdened parallelism :	10.39

5.6 *Tune your code so that it executes as fast as possible when running with 12 threads. Describe any decisions, trade offs, and further optimizations that you made.*

Provide a clear and concise description of your parallelization strategy and implementation.
Our parallelization strategy mainly involved performing the checks for collisions done inside of different Quadtrees on different cores. This was achieved by calling cilk_spawn inside of our descend() function, when the four child Quadtrees are told to check for collisions or recurse.

The other parallelism results were in parallelizing the pairwise check for collisions between lines that span multiple Quadtrees and the lines inside the four child Quadtrees. This was done by using cilk_for to break up the outer loop, which iterated over the spanning lines, across cores. We knew from empirical evidence that there were more spanning lines than there were lines that fit into Quadtrees. This would reduce the number of total lines accessed on any given core.

On the base case of our recurrence in the descend() method, the pairwise collision check was parallelized in a similar manner to that described above.

Discuss any experimentation and optimizations performed.

We experimented with grain size of the parallelism by defining pragma in the code. However, we determined that the default value of grain size was most optimal.

During the implementation of the line-length caching system, we first tried using a map<Line*,double> structure to hold the values. We found that this performed worse than the provided code. So, we changed map to an unordered_map. This did actually result in a performance boost, but also caused icc to emit errors when building the gtest code and when

incorporating the cilk annotations. To correct for this, we removed unordered_map, the -std=c++0x flag it necessitated, and simply added a member to the Line structure that contained the line's length. This resulted in similar performance.

Justify the choices you made.
The main decision in terms of code parallelization we made concerned whether to parallelize distributeLines, the function that distributes the lines inside of a Quadtree to the child Quadtrees. We implemented a parallelized version of this, which required us to use a reducer_list_append<Line*> for the structure that holds the lines. However, we found this to be a source of slowdown, likely because the elements inside a reducer_list_append are not iterable. This required us to call the get_value() method often, which would be be a decrease in performance over accessing an element in a list or vector directly.

Discuss what you decided to abandon and why.

See our attempts to implement our own line-wall collision detector in part 4.4 under "Difficulties during design".

Also, see the section "Justify the choices you made" for additional information on one of the parallelization strategies we decided to abandon.

Provide a breakdown of who did what work.

We can discussed every design decision. During the implementation one of us was driving, the other navigating, and we took turns doing both roles, thus spending equal amount of time on both roles.