6.172 Project 2
Adriano Hernandez & Grant Miller

Design Overview

As suggested, we improved render to only render the pixels corresponding to a sphere instead of checking every sphere with every pixel. In this algorithm, the eight corners of a bounding box of the sphere were projected onto the canvas and then the maximum and minimum x and y coordinates (on the canvas) were found. For each pixel within those bounds a ray was drawn to the sphere, and if it intersected, the lambert diffuse and color were used. This whole process was mostly sequential, though across pixels there was some parallelization in the form of a cilk_for.

We followed the suggestion to parallelize the forces algorithm and only check each pair once using force symmetry. This gave us a couple tiers and a parallelism of around 9-10 on large inputs. For the final, we would like to continue to tune our coarseness to achieve maximum speedup. We used the algorithm drawn out (with triangles and rectangles) in the assignment sheet. We struggled to deal with floating point woes, but eventually introduced a double precision vector to deal with it.

We used primarily regular collision detection, though we did attempt to prune by (sort of) sorting along an axis. For each sphere, we approximately only checked the spheres after it such that their trajectory would take them on an intersection course with this sphere's trajectory. This pruning struggled with floating point precision, but usually worked. What we did more precisely was to follow the heuristic that if two spheres are on a collision course, then the vectors from the eye to their faces (i.e. from eye to center minus radius) are roughly on the same axis. If two spheres are on the same axis, then they collide only if their movement on that axis makes them intersect in all three axes. So if we check if they will intersect in that axis, we can discard most spheres. To approximate the velocity in that rough axis we used the absolute speed in all directions, which is a high upper bound. This speed took into account the latest accelerations. We then scaled the calculated displacement (i.e. of their trajectories) by multiplying the speed by the distance in the rough eye to sphere axis and an arbitrarily large number, representing a large bounding box. That number, called the heuristic stop ratio (it tells our heuristic at what point in space, relative to the moving sphere, to stop) for us was 4. This pruning gave us around 20 tiers.

We also attempted to use a parallel and/or sequential quicksort but neither improved performance even when taking the median of the first, last, and middle elements of each sub-array as the pivot. Parallel quicksort had hard-to-debug race conditions even though we tested it rigorously with various assertions and it passed tests. We ditched it for that reason and kept insertion sort.

We figured out how to use the number of frames global variable from main to do malloc calls at most two times per run, which saved us some overhead. It is at most twice since we initialize the variables we want malloc-ed to null (zero) and then malloc them if they're null;

however, because staff tier testing does not use the global number of frames, but instead a hardcoded three, we had to free after three and then re-allocate again if the test was not a tier test (which was no stored in any globally accessible location).

Multisort: Bonus Design

We also tried a complicated collision pruning algorithm we called “multisort” in which we ran three sorts to find the minimum set of potential colliding balls for any given ball. Because we had, at that point, not yet figured out how to do one-time mallocs, we had added three indices (integer) to each sphere such that the index (integer) represented the location (index) in the array of the data that belonged at that sphere’s index (location). This way we could have only one copy of every sphere while at the same time having four arrays, total, each representing some ordering of the spheres, sorted by a different key (i.e. distance from eye or x, y, or z position offset by radius, the latter of which represents the minimum face of the bounding box of that sphere in that axis). To get the data for axis 1 (i.e. x) at index 0, for example, you would use the expression spheres[spheres[0].multisort_indices[1]]. This double indirection introduces some overhead but lowers memory consumption and keeps the array in cache. Note that these sorts should be done by incoming locations (i.e. trajectories) and not current locations.

Multisort was too complicated, so we dropped it. However, note that the algorithm is potent: you can run three sorts in parallel to sort along the three axes. Then, for each axis, for any ball you can use binary search to find the ball furthest along that axis such that its minimum face intersects the maximum face (on that axis) of that initial ball. If there is no such ball for any of the three axes then the ball cannot collide with any other ball. Otherwise, one may iterate along the axis from that ball to the aforementioned one after, such that this is the axis with the fewest number of balls (in that sub-array). If any of those balls is not in the bounds found by the other axes (i.e. of that axis’ maximum ball’s minimum face and the original ball’s maximum face) then it cannot intersect the ball (effectively, we are looking for the intersect of these sets of balls along each axis). Note that because collisions are symmetric, we only need two nested loops such that the inner one starts at an index after the outer one. (Collisions are symmetric insofar as if A collides with B, then B collides with A).

Final Results

We tried to use a form of pruning for collision detection by sorting the balls on an axis as described above. This took us roughly to tier 73. Our regular algorithm using parallelized pairwise forces with symmetry and the suggested improvement to render reached around tier 51. We ran into some minor correctness issues for very large inputs due to floating point woes, but managed to rectify the majority.

Final Release Plans

We intend to create a small library to run our program faster. We want to remove repeated mallocs as much as possible and will use our aforementioned conditional initialization of global variables trick more often. We use the number of frames to know when to free. On tier tests we call free after three frames. We also would like to improve sorting, because it will be important to our pruned collision detection. We may implement a parallel merge sort to that end. Alternatively, we may simply find a simpler way to prune (i.e. using some form of grid).

We also want to improve render. We will explore whether we are able to use projections and change of bases to reduce computation. We may be forced to make heavy use of doubles to avoid floating point errors. We also will want to parallelize render better. To that end we intend to render by chunks (or by spheres). We may have to store multiple copies of the canvas, or subsets of the canvas, to avoid race conditions.

We also unfortunately discovered on Friday morning that our code had correctness errors likely related to floating point errors. We will need to build out a better and more importantly faster testing infrastructure in order to make sure we are still in line with the original implementation. Due to the complexity of our algorithms, it will be hard to guarantee that we execute the floating point calculations in the same order as the staff so we will have to spend quite a bit of time ensuring that our methods can still be used.

There are also more exotic changes that can be made if we have time. For example, displacement can be projected onto the canvas. We may be able in that way to move only the edges of sphere projections to create even further speed-ups. This will be challenging due to, at least, the simple fact that the colors may change. Pre-computing colors will be important going forward.

Help Acknowledgement

We would like to help Jay Lang for giving hints for implementing quicksort, even though it was never used in our submission. We also want to thank Isabel, Sualeh, and their other TAs/LAs for answering our questions on Piazza and giving us feedback on our optimization ideas in OH.

Log

Adriano (excluding initial 30-minute meeting for contract, etcetera).

Sunday, October 3rd, 2021	10-11AM	Thought about and discussed algorithmic ideas with a friend.
Monday, October 4th, 2021	9-11AM	Pairwise Forces
Tuesday, October 5th, 2021	9-11AM, 5-7PM, 10-11:30PM	Parallelized Pairwise forces, embarked on

		collision detection optimization (that was too hard) involving multiple sorted lists.
Wednesday, October 6th, 2021	9-11AM, 3-5PM	Debugging, Perf, Failed quicksort
Thursday, October 7th, 2021	5-8PM, 9-11:30PM	Debugging, Perf, Quicksort, Parallel Quicksort, New Insertion Sort, Collision Pruning

Adriano Total: Around 17.

Grant

Time Start	Duration	Description
10/1/21, 1:30PM	.5	Met to discuss team contract and initial optimization ideas
10/4/21, 6PM	1	Reread project doc and starting initial implementation of optimized ray tracing
10/4/21, 8PM	4	Still working on optimized ray tracing, bounding cubes are being projected, now need to figure out coloring
10/5/21	2	Got coloring to work
10/5/21, 8PM	3	Optimized coloring so it doesn't try to color a pixel multiple times
10/6/21, 1PM	2	Worked on render and profiled
10/6/21, 3:15PM	1.5	Met to discuss our optimizations and merge our code
10/7/21, 10AM	2.5	Tried to optimize/vectorize simulate, relatively unsuccessfully
10/7/21, 4PM	3	Met and tried to implement some last minute optimizations for beta
10/7/21, 9PM	3	Rewrote the update accel code, made a few more aesthetic inputs, got to 73 with Adriano's merge code
Total	22.5	