Report of Solution

Eren Erisken
`erenerisken@gmail.com`

March 25, 2022

1 Introduction

This is the report of my solution for the challenge. Firstly, I want to say that I really enjoyed the challenge and it was a great refresher for algebraic topics and also some parts of computer graphics. I wasn't expecting this kind of challenge including these concepts and I really appreciate it. Also, since I decided to code it on C++ for better efficiency, I realised how I missed C++ since it has a special meaning for me given that it was the first programming language I learned when I was at high school. I couldn't have the chance to use it for almost a year, so it was nice to get back :)

I think that's enough for emotional part, so let's begin with how I approached the problem.

2 First Approach

When I first read the challenge, I was surprised since I wasn't expecting a challenge with such mathematical background. At first, I thought that it was about driving me to think complex solutions and it has a trick which simplifies the problem. Since thinking on 2D space is way more easier, I started to imagine different scenarios and suddenly thought that I found the trick. I got really excited since there was a great solution for 2D. Since all spheres would rotate in the same plane, if any two sphere had different angular velocities, they were meant to coincide in a position where the distance is equal to difference between radius of their orbits. Therefore, a solution in that case would be straightforward. I was just gonna order them with respect to their radius and then iterate over the vector of spheres. If any two spheres had the same angular velocity, then the distance between them would be equal to initial distance at any time and if they had different velocities, then closest distance will be the difference of orbital radius. This solution has a good time complexity $O(n * log(n))$ where n is the number of spheres. So, I thought that I could somehow generalize this idea to 3D space.

Things were not that simple though, as I didn't need much time to realize that this idea would not work in a 3D space as the spheres would rotate in different orbits and these orbits only have 2 respective positions where orbits get as close as radius difference. However, the spheres were not supposed to be at those points at the same time. So I could easily refute this hypothesis.

3 Optimization Model Approach

After returning the point I started, I started to construct a mathematical model. My aim was formulating the distance between any two sphere in terms of time t. This was meant to be possible theoretically, as only unknown would be t. So I started on searching the Web about how I could find a formula for position of an object rotating on a circular orbit in 3D space where its route and angular velocity is known. I expected that it would be straightforward since we are already working on spherical coordinates and there should exist an easy formula for this situation, right? Well... We say "Evdeki hesap çarşıya uymadı" in Turkish for this situation. Its translation is "The account at home did not fit the market" which perfectly fits to this situation. I realised that it was gonna be much more difficult than I thought it would.

Problem was that the equation I tried to derive from my researches quickly began to accumulate a lot of trigonometric expressions, square roots and many other magical terms :P

At that point, I found out that writing an optimization model on this great equation was practically impossible (at least for me), nor that I could take derivative and write a program to equate it to 0.

After the failure of my second approach, I decided to continue with simulation of the system. Simulating the system was already in my mind but I believed that I could find a much more efficient way than that. Therefore, this challenge was also a great reminder about the fact that you should never get pretentious.

4 Simulation Approach

Iteratively simulating the system was the last option I had in my hands. As I said, I wasn't willing to implement a solution at the beginning, however after my other trials failed, I reminded myself that this challenge was about testing my programming skills and the way I think so I somehow accepted my fate and started to think on a simulation model which is as efficient as possible in terms of time complexity.

The biggest question at the beginning was; when will I terminate the simulation? How could I guarantee that while computing the closest distance between any sphere at any time, I found the best result and it won't get better when I continue iteration. Since the pair of spheres could easily oscillate but also get closer with each period, what if they get closer when time goes to infinity? This has been the biggest question mark at that stage. To be honest, it took a good amount of time until I came up with a great realization: angular velocities were given as integer! This directly answered my question as in 360 units of time, system would reset to its initial state in any condition. Any sphere will return back to where it started after 360 units of time passed. For example, if a sphere has an angular velocity w degrees per unit time, after 360 units of time, it would travel $360 * w$ degrees. Since w is integer, $360 * w$ is a multiple of 360. Hence, iterating for 360 units of time was the answer to my question.

Then, I got back to the question about finding the position of a sphere at time t. At first, I wanted to calculate displacement somehow, but for this purpose, I needed to have too small step sizes to stay on the path. Having that small step sizes kills the time efficiency, as you would guess. So I started to look that up and while I was diving deep on the math pages, I saw rotation matrices and suddenly remembered the topics of computer graphics class. I watched a few videos about 3D transformations and rotation matrices and realized that this would come very handy in my situation. The best part is that rotation matrix does not change over time. You calculate it once for each sphere and then use it for any time t. Since I was iterating over each t, this was perfectly suitable for my case.

The last thing left for clarification was that would I really need to compare each pair of spheres. I needed to find a condition to eliminate pairs since it was gonna converge to $O(n^2)$ otherwise which I didn't really want. Then I remembered that the best possible case for two spheres is that they get as close as the radial distance between their orbits. This greatly helped me to eliminate most of the pairs since at the beginning I sorted the vector of spheres according to radius and I broke loops whenever radial distance got bigger than current closest distance computed.

5 Implementation

I was finally ready to implement the solution after I answered all the questions in my mind. To summarize the implementation, I have written a C++ program which gets 3 arguments: path to the input file name, number of divisions for 1 unit of time and number of maximum allowable threads. Then, program reads spheres from given input file. Each sphere is represented by 1 line in the input file. Each line consists of 7 values:

$r\ \theta\ \phi\ \omega\ u_x\ u_y\ u_z$

r is the radius of the orbit of the sphere (distance from the origin) which is an integer value
θ is a component of the spherical coordinate of the sphere. It is in radians and denotes the angle with x-axis in the x-y plane
ϕ is a component of the spherical coordinate of the. It is in radians and denotes the angle with z-axis.
ω is the angular velocity of the sphere in degrees per unit time. It is an integer value (thankfully!)
u_x, u_y, u_z are components of the unit vector containing the tangential direction in which the sphere is travelling. They are all float values.

Second parameter given to the program, number of divisions per 1 unit time is used to denote the step size of the simulation. For example, if we use 10 for this parameter (which I mostly used in my experiments), program will advance $1/10 = 0.1$ unit times per each iteration.

The last argument is number of maximum allowable threads. This value is used together with *SPHERES_PER_THREAD* macro defined in *Utilities.h* file to determine number of threads to be invoked. You can refer to *Simulation.cpp* to learn more about calculating number of threads.

After spheres are read from the file, they are sorted by their radius. Then, each thread gets a portion of the vector containing the spheres. Each thread iterates over the vector in given range and calculates distances between current sphere and its successors in the entire vector and updates the best result found if a better one is calculated. However, if radial distance between current sphere and its successor gets bigger than current best result, current sphere is done and iteration continues with the next one. I always prefer pseudo-code over explanations since explanations always get confusing so here is a pseudo-code for the routine of a thread:

Algorithm 1 Main routine for a thread

```
$i \leftarrow$ startIndex
**while** $i <$ endIndex **do**
    $j \leftarrow i + 1$
    **while** $j <$ sphereCount **do**
        **if** $radialDistance(sphere_i, sphere_j) >$ bestDistance **then**
            **break**
        **end if**
        $d \leftarrow closestDistance(sphere_i, sphere_j)$
        **if** $d <$ bestDistance **then**
            update best result
        **end if**
        $j \leftarrow j + 1$
    **end while**
    $i \leftarrow i + 1$
**end while**
```

After all threads are finished, best result according to current configuration of the program is calculated and printed to *stdout*.

Closest distances between spheres are calculated using list of positions as cache. In other words, at the first time a sphere's *closestDistance* method is invoked or a sphere is given as a parameter to another one's *closestDistance* method, its positions are calculated for each possible t between 0 and 360 and stored in a member vector of positions. Then, this vector of positions are used to compute the closest distance between spheres. Since we are storing these lists in memory, program can use huge amounts of memory. It saves a lot of time though, especially when there are a lot of threads running around, calculating positions again and again for each sphere would take too long due to matrix operations. Caching this positions dramatically increased computational speed. Theoretically, this should not be a problem since no memory constraint is specified in the challenge as far as I read. However, combining too many spheres with a small step size could end up consuming HUGE amount of memory.

6 Usage

6.1 Building from Sources

Since *cmake* is used as build tool, you need to have *cmake* installed on your system in order to build the project from sources. If you have it installed, you can simply run *build.sh* and it will carry out necessary steps to build the project. After build is completed, it runs unit tests. If you see "*All tests passed!*" in your screen, you are good to go. You should be able to see 2 binaries under */bin* directory. *run_tests* is for running unit tests and *dark_orbit* is the main program.

6.2 Running the Program

After building the binaries, you can run the simulation by providing it 3 arguments as described before. For example, *./bin/dark_orbit /fixtures/simple.txt 10 8* will run the program with input file */fixtures/simple.txt* to read observation of spheres, will divide 1 unit of time into 10 intervals and spawn up to 8 threads. Result will be printed to *stdout*.

6.3 Running the Tests

You can run unit tests provided by running *./bin/run_tests* easily. This program does not require any arguments and will run unit tests sequentially and report the result after execution.

7 Results

Time and memory usage for inputs with different sizes and different step sizes can be found below:

Number of spheres	Number of intervals	Time elapsed	Memory used
1000	10	34 ms	65 MB
1000	100	130 ms	250 MB
10000	10	189 ms	250 MB
10000	100	1182 ms	700 MB
100000	10	2442 ms	4 GB
100000	100	18.4 sec	20 GB
1000000	10	58.1 sec	26 GB

Table 1: Results of experiments with different configurations and input sizes

Note that this experiments are run on a MacBook Pro with Apple M1 Pro chip with invoking 10 threads.

8 Future work

My implementation has a lot of room for optimizations. For memory constrained systems, an option could be given to the program for disabling caching. This would have a huge impact on execution time but will prevent the program from seeing crazy amounts of memory usage (like the one in 1M sphere case). Also, while writing this report I found out that I don't always need to iterate until $t = 360$. Instead of this, I could've taken least common multiple of periods of two spheres and iterate until that time. This will always result in a faster execution and should be added to the implementation (it's 2.30 a.m. at the moment though :D). Also, cache of spheres investigated could be freed to open space on the memory. This may restrict maximum amount of memory usage.