

Project 2: Rendering and Simulating 3D Scenes

Last Updated: October 14, 2025

In this project you will optimize a graphical N-body gravity simulation program for multicore processors using the OpenCilk parallel programming platform. It is a good idea to fully read this handout and reread the Course Information handout before you get started!

Contents

1	Due dates	1
2	Getting started	2
3	Introduction to the codebase	4
3.1	Running the simulation	4
3.2	Running the reference testing tool	5
3.3	Running Cilk Productivity Tools	6
3.4	Making a simulate test	6
3.5	Making a render test	7
3.6	Modifying the files	7
4	Optimizing the serial code	7
4.1	Guidelines and advice	8
4.2	Rendering through ray tracing	9
4.3	Physics simulation	12
4.4	Floating-point woes	13
4.5	Assumptions you can make	14
4.6	Optimization ideas	14
5	Parallelization	15
5.1	Parallel algorithms	15
6	Evaluation	18

1 Due dates

- ☐ *Team contract:* 10:00 P.M. on Tuesday, October 28, 2025
- ☐ *Beta submission and write-up:* 10:00 P.M. on Thursday, November 13, 2025
- ☐ *Final release:* 10:00 P.M. on Tuesday, November 18, 2025
- ☐ *Final submission and write-up:* 10:00 P.M. on Thursday, December 4, 2025

2 Getting started

Snailspeed Ltd. has been put onto the map due to its newly patented *iRotate* algorithm. Due to your role in Snailspeed's rise to prominence, you have been promoted to CCTO (Co-Chief Technology Officer). Custom engraved door plates and corner offices aren't cheap; and, unfortunately, Snailspeed didn't have enough revenue left over to hire additional engineers. Instead, Snailspeed has hired a small team of corporate management consultants from InchWorm Advisors.

InchWorm Advisors have suggested that Snailspeed pivot towards the development of multicore software in order to remain on the cutting edge. As a result of these discussions, Snailspeed is planning to launch *SnailSim*, a disruptively high-performance graphical *N*-body gravity simulation program designed for multicore processors.

It is up to you and your fellow CCTO to transform *SnailSim* from dream to reality. InchWorm Advisors have suggested that you look into the OpenCilk task-parallel programming platform to parallelize *SnailSim*. OpenCilk may be the key, according to Inchworm Advisors, to transforming Snailspeed Ltd. into a major player in the high-performance computing industry.

Team formation and contract

Teams will be formed by random matching and posted on Piazza. You and your teammate must write a team contract specifying how your team will operate — a set of conventions that you plan to abide by. **Only one copy of the contract needs to be submitted on Gradescope. Please add both teammates to the submission.**

You and your teammate must agree to a team contract. A team contract is an agreement among teammates about how your team will operate—a set of conventions that you plan to abide by. Since your team was randomly formed, taking the time to discuss and agree upon a thorough team contract can save you unexpected disappointments.

The questions below provide some considerations for what might go into your team contract. You should also think back to good or bad aspects of team-project experiences you've already had. The questions are broken into three categories: meeting and communication norms, work norms, and decision making. You needn't address all these questions, which are simply suggestions. Focus on the issues that your team considers most important. A skimpy team contract is a bad idea. The TA's will review all contracts. If you have little in your team contract and subsequently your team has problems working together, you make it harder for the course staff to help you with your plight. All team members should write their names at the end of the contract, to indicate that they agree with it.

Meeting and communication norms

- How often will the team plan to meet outside of class? How long do you anticipate meetings will be? What will you do if things change?

- Where and when will outside-class meetings be held? What will you do if someone fails to show for a meeting?
- How will you communicate outside of meetings? (Email list? Real-time messaging platform? Signed messages on the Bitcoin blockchain?)
- If someone in the group decides to drop the class, what obligations does that person have to inform the remaining team members and complete promised work?

Work norms

- How much time per week do you anticipate it will take to make the project successful?
- How will work be divided among team members? On which parts of the project will you do pair programming?
- What will happen if someone does not follow through on a commitment, e.g., not doing their work? What if someone gets sick?
- How will the work be reviewed? How will you manage your code branches?
- How will you deal with different work habits of individual team members? (e.g., some people like to get work done early, while others like to work under the pressure of a deadline.)

Decision making

- Do you need unanimous consent to make a decision? What process for decision-making will you use if you can't agree?
- How will you prioritize the work to be done? How will you deal with the common situation in which different team members have different optimization ideas?
- What happens if everyone does not agree on the level of commitment? (e.g., some team members want an A, but others are willing to settle for a B.)
- Is it acceptable for some team members to do more or less work than others?

Submitting your team contract

Only one submission is required for each group on Gradescope, and you should include all your team member's name in the team contract. In addition, once your team repository has been created, you should commit a copy of your team contract to the top level of the repo under the name `team-contract.pdf`. Feel free to update the team contract in the repo as needed during the project, as long as all parties agree.

Getting the code

Before beginning to work on Project 2, please update your `apt` packages:

```
$ tellmeroot
$ su -
# sudo apt update && sudo apt upgrade
```

The course staff will release the teams on a spreadsheet through Piazza, notifying you of your `team-name`. You can then get the project code:

```
$ git clone git@github.com:CSE491-fall125/project2_<team-name>.git project2
```

You can also browse the code before that by cloning a read-only repository:

<https://github.com/CSE491-fall125/project2>

Pair programming

We strongly recommend that teams practice pair programming. This style of programming leads to bugs being caught earlier, both programmers always remaining familiar with the code, and fewer repetitive stress injuries (RSI). Pair programming also makes it easy for you to maintain balanced commits among your team members' usernames.

3 Introduction to the codebase

In this project, you will be optimizing a 3D *N*-body gravity simulation with rendering. The simulation consists of a 3D virtual environment filled with colored spheres that bounce off one another according to simplified physics.

To get acquainted with the code, start by exploring the `project2` repository, and read the top-level `README.md` and `DESIGN.md` files.

3.1 Running the simulation

The simulation can be executed using several different utilities. Using each is described in the README. To run a simulation and view the result, run

```
$ ./bin/gen_eframes -r -s simulations/300/s simulations/300/r frames.out
$ ./bin/diff2gif frames.out
```

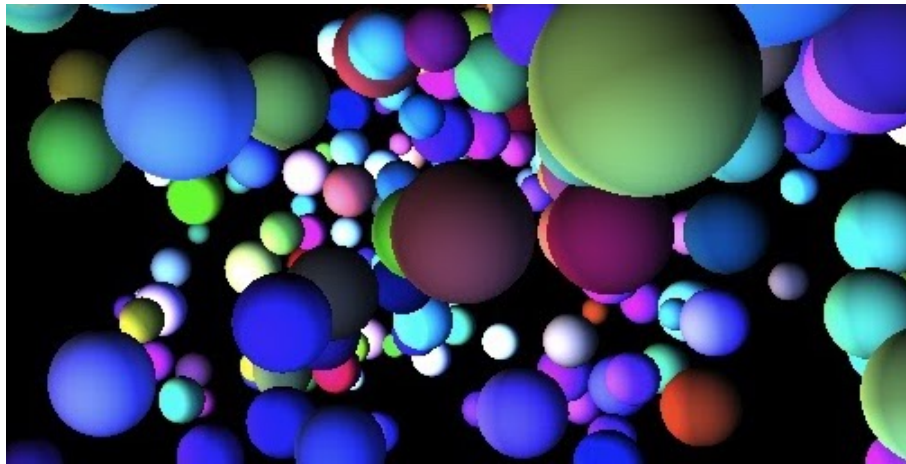


Figure 1: Screenshot of a rendered simulation.

Here the `-r` and `-s` cause this to use your renderer and simulator respectively. Omit one of these flags and it will run that component with the correct initial version we gave you, so you can narrow down whether errors are in your simulator or renderer. The `simulations/` directory contains several (fun!) examples of input files. Figure 1 shows a screenshot from `simulations/300/`.

To test the performance of your code on the 8-core telerun machines:

```
$ telerun --cores 8 ./bin/find-tier
```

You can also run your code locally by running `./bin/find-tier`. The `s` and `r` labels for the timing are the simulation and rendering timings respectively for the tier.

3.2 Running the reference testing tool

To help you get started with reference testing, a method of correctness testing described in Section 4.1, we are providing `ref_test`, a tool to check correctness by comparing implementations. This tool provides a visualization of your error and statistics about your correctness. It also provides the option to use reference simulation or rendering files to help isolate bugs. The execution syntax is as follows.

```
Usage:
./ref-test [-n num_frames] [-r | -s] [-x expected_frames]
          [-o diff_output] sim_spec renderer_spec

[-r | -s]                Use the staff renderer or simulator with your implementation
[-x expected_frames]     Give expected_frames for comparison
[-o diff_output]         Output file for diff visualization
sim_spec                 Initial simulation state file
renderer_spec            Renderer specification file
```

The `ref_test` prints out a few summary statistics about your run. We highly recommend using these statistics to test correctness. A maximum error of 0 implies correctness.

We have also created some helpful scripts in the `scripts` folder to help you check correctness. The command `./scripts/tiers_ref_test <number>` will run the ref test on tiers 0 to the number given inclusive. The command `./scripts/test_all` runs the ref test for each folder in the `simulations` folder using the file names `s` and `r` in each of those folders (if `simulations` has subfolders `t1` and `t2`, the command will run reference testing with `simulations/t1/s` and `simulations/t1/r` for the first test, and `simulations/t2/s` and `simulations/t2/r` for the second test). As this script rebuilds your code with different build flags for testing, pass in `LOCAL=1` if you are running it on your VM.

These tools are far from perfect! Don't be afraid to tweak them for your own purposes or post changes to Piazza. We appreciate all contributions (and they can help your class-contribution grade).

3.3 Running Cilk Productivity Tools

As we will touch on later, we recommend building a solid serial solution before trying to parallelize it. Once you begin to parallelize your code, you will likely find it useful to run some of Cilk productivity tools that Homework 4 will introduce you to, such as Cilkscan and Cilkscale.

3.4 Making a simulate test

You can make a simulation test file by creating a text file in a specific format, and then using that file in the ref-test above for the `sim_spec` file. The format of the text files is as follows:

- the first line is formatted as `g num-spheres`, where `g` is a non-negative floating-point number and `num-spheres` is a positive integer. There must be some whitespace between `g` and `num-spheres`.
- each other line is formatted as `r mass pos-x pos-y pos-z vel-x vel-y vel-z mat-diffuse-red mat-diffuse-green mat-diffuse-blue mat-reflection` where `pos` and `vel` are the position and the velocity of the sphere, respectively, and `mat` is the material of the sphere. `x`, `y`, and `z` refer to the respective components of the vector. `mat-diffuse-red`, `mat-diffuse-green`, `mat-diffuse-blue`, and `mat-reflection` must all be between 0 and 1.
- there must be `num-spheres` lines defining spheres in the text file.

Here's an example test file with 2 spheres:

```
9.8 2
1 2 3 4 5 6 7 8 0.5 0.5 0.5 0.6
10 20 30 40 50 60 70 80 0.6 0.4 0.3 0.7
```

3.5 Making a render test

You can also make a render specification text file, using the specification format given below. Section 4.2 explains each of these parameters in more detail.

- the first line is formatted as `resolution viewport-size`, where `resolution` and `viewport-size` are integers. There must be some whitespace between `resolution` and `viewport-size`.
- The next three lines contain the position of the eye, and the basis vectors u and v of the projection plane, in that order. The projection planes are unit vectors. Each vector is specified by the `x y z` components in that order on the line, with whitespace separating each.
- The fifth line is the number of lights.
- The remaining lines specify the lights with each line in the format `x y z r g b`, where `x y z` give the location of the light, and `r g b` give the intensity of the light in each color.
- There must be number of lights lines at the end defining lights in the text file.

Here's an example test file with 3 lights:

```
256 256
800.0 100.0 0.0
0.0 -1.0 0.0
0.0 0.0 1.0
3
0.0 240.0 -100.0 1 1 1
3200.0 3000.0 -1000.0 0.6 0.7 1
600.0 0.0 -100.0 0.3 0.5 1
```

3.6 Modifying the files

Your grade is only affected by changes to files in `libstudent/`. Please make sure that your project compiles and runs with the original staff-provided folders with your changes to `libstudent/`.

When you submit your code, the course staff will revert all other files to their original states. Feel free to explore the codebase, but please keep in mind that changes to files not in the above list will be reverted for grading.

4 Optimizing the serial code

The first part of the project is to make algorithmic changes to the `render()` function to reduce the total work performed by the serial code. Currently, the rendering code uses an extremely inefficient algorithm to determine the color of a given pixel. At each time step, the algorithm

iterates through all pixels, and for each pixel, it iterates through all spheres to see if that pixel should take the color of any sphere. This exhaustively iterative method is expensive, requiring $\Theta(MN)$ ray-sphere intersection tests for a size- M image and a scene containing N spheres.

Begin by implementing a new rendering algorithm that uses the mathematical concept of a projection to reduce the total amount of spheres that must be checked for each pixel. Please do **not**, at this point, complicate matters by using OpenCilk (or any other parallelization method) to parallelize your rendering algorithm. Parallelization will happen in the second part of this project, described in Section 5. Generally, when parallelizing code, the course staff advises that you first exploit all the serial performance enhancements you can, while preserving your options for parallelization later.

4.1 Guidelines and advice

Before you modify the implementation, please consider the following guidelines and advice:

- *Fast math:* You may **not** use the compiler flag `-ffast-math`. This flag allows the compiler to reorder certain mathematical operations, which may result in nondeterministic floating-point results. Since the course staff needs to grade the correctness of your code, nondeterminism is not okay. Although use of this flag is disallowed for your submission, feel free to play with it. (If you do, please report on your experiments with `-ffast-math` in your write-ups.)
- *Reference testing:* You will find it useful in this project to check the correctness of your optimizations using *reference testing*: a method of correctness testing that compares the execution of two different implementations of a function to ensure that the implementations behave identically. We have provided a tool called `ref_test` for this purpose.

Let's say that you have modified the `render()` function in `libstudent/`. There are two ways to test. One is to write unit tests. The other is to compare two versions of the function: the original implementation `libstaff::render` and your new implementation `libstudent::render`. The reference testing tool relies on the latter approach. By preserving the original behavior in `libstaff/` and developing new code in `libstudent/`, you will be able to use `ref_test` to perform reference testing.

- *Serial optimization:* You may be tempted to jump directly into parallelizing your code as a means to gain performance. Generally, however, the best parallel programs are those that are first highly optimized serially and then made parallel afterwards. We therefore recommend making the serial version of the program as fast as possible before parallelizing.
- *Determinacy-race freedom:* Your code must be free of determinacy races. This means you may not use parallel constructs involving locking, memory synchronization, or atomics. Stick to the `cilk_scope`, `cilk_spawn`, and `cilk_for` keywords for your parallelization.

Remark: The course staff takes no responsibility for giving you bad advice! You must make your own decisions and take responsibility for them. We do not give out intentionally bad advice, but we

have been wrong in the past — not often, but occasionally. Every term, there is almost always one piece of advice that we provide that some student discovers is wrong. (And we're delighted to find out about it!) In the vast majority of cases, however, our advice is sage. But since it will be on you if you accept bad advice from us, you must think carefully about the optimization decisions you choose to explore.

4.2 Rendering through ray tracing

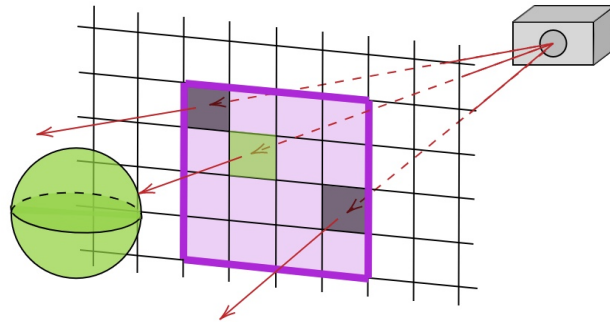
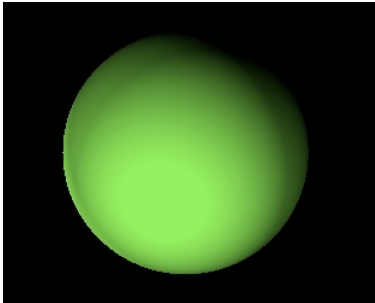
To render a scene, the code uses a highly simplified version of ray tracing. Ray tracing is a rendering algorithm that mimics the behavior of the human eye in order to create realistic images. As with human vision, there are three key parameters we must set to fix the 2D image we see when presented with a given 3D scene: the position of the eye, the plane onto which we project the image, and the size of the image. Since there are many different ways to represent these objects, let's look at the specifics as they are written in the code.

- Viewpoint e , named `eye` in the codebase, is defined by a 3D position.
- Projection plane P is defined by two 3D vectors, u and v , that lie in P . (These vectors are named `proj_plane_u` and `proj_plane_v` in the codebase.) We define u and v to be orthonormal for computational ease. Remember that the term *orthonormal* refers to vectors that are orthogonal to each other and of unit size. Points on P lie in 3D space, and therefore require three coordinates to fully specify their position. Note that P always passes through the origin, and images are always centered at the origin.
- Image I , viewed from e , is then defined by a square of side length `viewport_size` units on the projection plane P , subdivided into `resolution` \times `resolution` pixels. Points on the image lie in 2D space relative to the projection plane, and therefore require only two coordinates to fully specify their position.

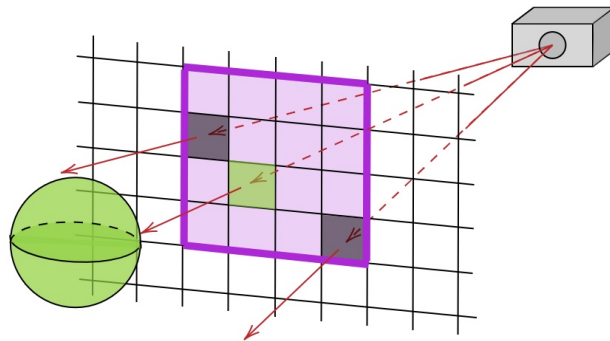
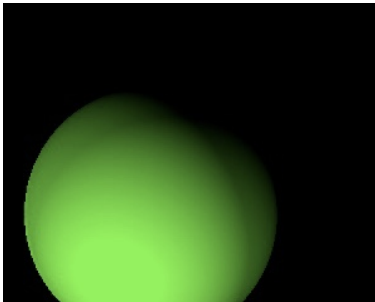
In our case, a scene is composed of N spheres indexed from 0 to $N - 1$. Each sphere is defined by position, velocity, acceleration, radius, mass, color, and reflectivity. To see how modifying the rendering parameters changes a rendered image, let's consider a scene consisting of a single green sphere i , rendered with projection plane P , eye location e , and image size M , as illustrated in Figure 2(a). What is the impact of changing the rendering parameters?

What happens, for example, if you change the position e of the eye? This parameter is fairly intuitive to understand. Imagine staring in the same direction as you walk side to side or sit down and stand up without changing the direction in which you're looking. The section of the scene you're looking at shifts as your position shifts. In Figure 2(b), observe the result of rendering sphere i when you replace e with a new position e' , representing an upwards shift of the eye, while keeping P and M constant.

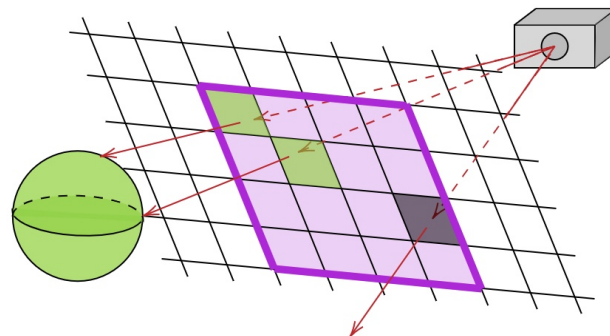
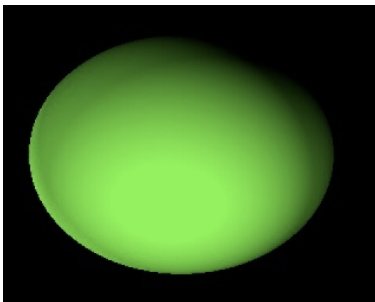
Return again to Figure 2(a), where we render the sphere with eye position e and projection plane P . What happens if you change P to P' , representing a rotation of the plane, while keeping



(a) The image (left) of a scene containing a single sphere and its corresponding ray-tracing diagram (right) showing the eye (box, top right), the projection plane (grid), and the subset of the plane that constitutes the image (shaded purple). A sample of rays from the eye to the image are shown in red.



(b) The scene from (a) with a new viewpoint. As compared to (a), the eye and the corresponding image (shaded purple) have shifted up.



(c) The scene from (a) with a rotated projection plane.

Figure 2: A comparison of the results of rendering the same scene under different parameters.

e and M fixed? The resulting scene is shown in Figure 2(c). This image corresponds to keeping the position of the eye fixed, but changing the angle from which the eye looks. Since you are no longer looking at the sphere head-on, Figure 2(c) renders an ellipse instead of a circle.

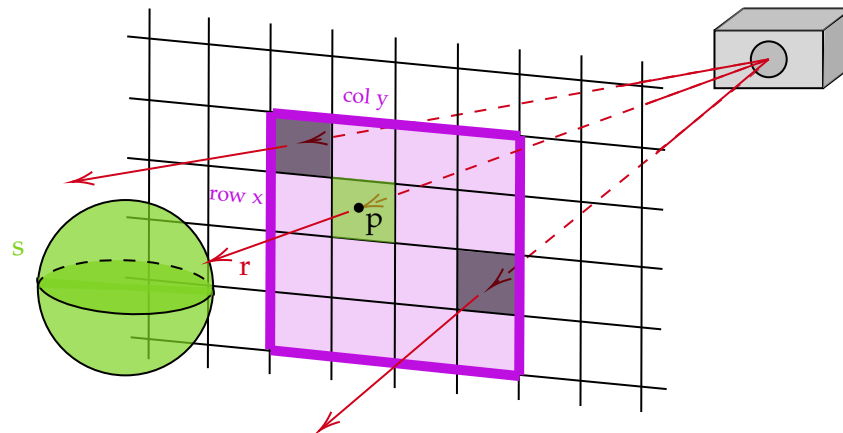


Figure 3: A visual example of the basic ray tracing algorithm showing the eye, the projection plane, the subset of the plane that constitutes the image, and a sphere. Each red ray from the eye to a pixel determines the color of that pixel from the objects in the scene that the ray intersects. An example for ray r , which intersects pixel (x, y) and corresponding 3D point p , is labeled.

Given a scene to render, you must fix the eye position, the projection plane, and the image size in order to fully specify the image. Now, what happens after you've fixed your parameters? How do you go from an infinite 3D scene to a finite 2D image? This is where the idea of ray tracing comes in. Follow along with Figure 3 to understand the procedure.

Consider a particular pixel in the 2D image you want to produce. Denote this pixel as the ordered pair (x, y) representing the pixel's x - and y -coordinates within the 2D image (labeled in purple in Figure 3). Since the image you want to produce is a subset of the projection plane, there is some known 3D point p (labeled in black in Figure 3) on that plane that corresponds to pixel (x, y) . To determine the color of pixel (x, y) , define a ray r (labeled in red in Figure 3) from the eye to point p . Any sphere that intersects ray r has a 2D projection that includes pixel (x, y) . In other words, some part of the sphere is mapped to (x, y) when you project the sphere from 3D to 2D. Of course, with multiple spheres, because spheres closer to the eye occlude spheres farther away, you actually only use pixel (x, y) to render the sphere closest to the eye that intersects ray r .

Let's review. To determine the color of pixel (x, y) , you must find point p on the projection plane, calculate ray r from the eye to point p , and find the sphere closest to the eye that intersects ray r , if such a sphere exists. If this sphere exists, pixel (x, y) adopts the color of that sphere (realistically shaded according to [Lambert diffusion](#)). If no such sphere exists, the pixel adopts the background color (black). The image is rendered by repeating this process for each pixel, as in Figure 3. For a scene with N spheres, since we check all N spheres for intersections with r , the same ray-sphere intersection calculation is repeated N times per pixel. This strategy is inefficient.

Fortunately, this problem contains spatial locality that you can exploit. Consider some sort of projection algorithm to replace the current ray tracing algorithm. Here's the main idea: instead

of iterating through the spheres for each pixel to determine the pixel's color, iterate through the spheres only once in order of increasing distance from the eye. For each sphere, compute the projection of the sphere onto the projection plane, and color in the appropriate pixels within this projection as you go. Since you iterate in order of increasing distance, once a pixel is colored, you need never color it again as you continue iterating through the spheres. This way, you need only visit each sphere once and color each pixel once, yielding an algorithm with runtime $\Theta(M + \sum_{i=1}^N M_i)$ instead of $\Theta(MN)$ as before, where N is the number of spheres, M is the size of the image, and M_i is the size of the projection of the i^{th} sphere, which is much less than M in practice.

As you test your code, you may want to consider a few key questions for the write-up:

- What speedup did you achieve? Was this performance what you expected?
- What design choices did you make while rewriting the render function to use the idea of projections? For example, how did you determine which pixels to color for each sphere?
- Does this implementation seem parallelizable? Do you foresee any determinacy races?

Remember that the color of each pixel and the position of each sphere in the new code should be the same as in the old code. Verify your code using `ref_test` or by writing your own tests.

4.3 Physics simulation

The function `simulate()` implements the physics of the spheres as they interact. There are two main components to the 3D physics of the simulation: gravitational forces between every pair of the N spheres and elastic collisions between spheres. This simulation aspect of the project is just as important as the rendering aspect, although we do recommend tackling simulation second.

Each sphere `i` has the following properties related to the physics of the simulation: `i.accel`, `i.vel`, `i.pos`, and `i.mass`. Acceleration, velocity, and position are updated after each time step. These calculations are handled in the functions `update_accelerations()`, `update_velocities()`, and `update_positions()`, which you can find in file `simulate.c` of the starter code.

The simulation discretizes time by partitioning the time line into small intervals. A *time step* is a point in time connecting two adjacent intervals. At each time step, the simulation updates the acceleration, velocity, and position of all spheres based on their values at the previous time step. Let's denote the acceleration of sphere i at time t as $a_i^{(t)}$, velocity as $v_i^{(t)}$, and position as $p_i^{(t)}$. For a given time step t , let $t - \Delta t$ be the time of the previous time step.

The acceleration $a_i^{(t)}$ of sphere i at time t is provided by the Newtonian law for the forces of gravity. Let $d_{ij}^{(t)}$ be the vector difference of the positions of spheres i and j at time t , let G be the gravitational constant, and let m_j be the mass of sphere j . The net acceleration of sphere i due to the gravitational forces from the other $N - 1$ spheres at time t is then given by the formula

$$\vec{a}_i^{(t)} = \sum_{\substack{j=0 \\ j \neq i}}^{N-1} \frac{Gm_j}{|\vec{d}_{ij}^{(t)}|^3} \cdot \vec{d}_{ij}^{(t)}.$$

The acceleration calculation involves $\Theta(N^2)$ work to update all spheres. There are methods in the literature, including the Barnes-Hut algorithm and the Fast Multipole Method, for reducing the work asymptotically. We do **not** recommend that you implement one of these methods. The constants in these algorithms are unlikely to make them faster than the naive $\Theta(N^2)$ for the relatively small problem instances we'll be dealing with, and they will wreak havoc on your ability to control floating-point accuracy. As further reasoning, the course staff would like you to know that none of our optimized implementations contain such algorithms or other advanced ideas such as k -d trees. You are free to do as you wish, but our advice is to mine the many other sources of performance before risking much or any time implementing such algorithms.

After calculating $\vec{a}_i^{(t)}$, the simulation algorithm finds $\vec{v}_i^{(t)}$ and $\vec{p}_i^{(t)}$:

$$\begin{aligned}\vec{v}_i^{(t)} &\leftarrow \Delta t \cdot \vec{a}_i^{(t)} + \vec{v}_i^{(t-\Delta t)}, \\ \vec{p}_i^{(t)} &\leftarrow \Delta t \cdot \vec{v}_i^{(t)} + \vec{p}_i^{(t-\Delta t)}.\end{aligned}$$

The simulation handles collisions between spheres as follows. Suppose that the time of the current time step is t and the time of the next time step is t' . If a collision will happen between t and t' as determined by a geometric collision checker in `simulate.c`, the collision checker finds the time t'' of the first collision that occurs between times t and t' . Then, the code breaks the interval $[t, t']$ into two *mini* time intervals: $[t, t'']$ and $[t'', t']$. The effects of the collision are calculated after updating acceleration, velocity, and position for time t'' . The code then iterates on interval $[t'', t']$, again subdividing it if a collision will occur before t' .

4.4 Floating-point woes

One particularly hairy aspect of N -body simulation and rendering is dealing with floating-point numbers. Floating-point arithmetic is not associative. In particular, there might be faster ways of computing things like velocity, but unless the same operations are performed in the same order, floating-point error *can* occur, which might cause you to fail the reference test. Thus, while you are allowed to modify formulas and reorder operations, you should proceed with caution when doing so. **A good practice is to always run the correctness tests whenever you modify a formula.** You can also use `ref_test` with `diff2gif` to visualize the locations of floating-point error.

The code uses double-precision floats for calculations to mitigate the effects of round-off. A rule of thumb is that reordering calculations that are done using double-precision floats is less likely to cause floating-point errors, because while round-off errors can accumulate, the error is unlikely to change the final result after conversion back to single-precision floats. However, this is not a guarantee — always make sure to run the correctness tests!

More subtleties arise when writing “shortcut” code. For example, you may use the fact that if the bounding cubes of two spheres do not overlap, then they cannot collide. Taking advantage of this observation can reduce the number of pairs of spheres for which you perform a collision check. Since this operation for collision detection is different than the condition used in the original code, however, it is possible, due to round-off error when computing the bounding box, for the original code to report a collision while the bounding cubes do not intersect.

4.5 Assumptions you can make

So far, we have not fully defined the behavior of the code in a few edge cases, such as when the eye position is located within a sphere. We will not test on these cases — you may assume the following in all of the test cases we will test your code on:

- There are no overlapping spheres in the initial state of the simulation.
- During the course of a simulation, the eye and lights will never be located inside a sphere.

4.6 Optimization ideas

Look for opportunities for serial optimization within the `render()` and `simulate()` functions and describe what you did. Here are some optimization ideas to help you get started:

- Replace the ray tracing algorithm by implementing a projection algorithm that maps 3D spheres to 2D shapes. Iterate through the spheres and color in the projection for each sphere as you go. The course staff highly recommends this optimization as a place to start, but you’re free to make your own decisions (especially if our advice is bad).
Hint: Think about how to use a bounding volume for a 3D sphere to find the bounds of its projected 2D shape.
- A large number of calculations are repeated in each time step. For example, the rendering code recalculates the color of each sphere, even though the color never changes. Consider precomputing these calculations and storing results for reuse.
- Many calculations are repeated even within the same loop. For example, when calculating the force of gravity between spheres i and j , we effectively calculate the force of gravity between spheres j and i . Take advantage of this symmetry.
- Although the method given for testing whether two spheres collide is fairly efficient, you may wish to explore more efficient ways of testing for sphere collision. Take care to thoroughly reference test your implementation to avoid inadvertently changing `simulate()`’s behavior.
- You are allowed to use AVX intrinsics and Clang `__builtin` functions. If you are looking for a challenge, perhaps investigate vectorizing `render()` by hand.

5 Parallelization

Once you have optimized your serial execution, you can begin parallelizing with OpenCilk. Parallelization can be tricky because race conditions and nondeterminism can arise if you implement parallelism incorrectly. Careful parallelization is key to improving the performance of your code.

Profiling the serial code

Profiling your application can help determine where you should focus your time when parallelizing your code. The goal is to identify a region of code that comprises a large percentage of the total execution time but is amenable to a potentially large degree of coarse-grained parallelism. Run `telerun perf record` and `perf report` to see the profiling statistics.

Parallelizing the application

Parallelize your code by including `cilk_scope`, `cilk_spawn`, and `cilk_for` keywords appropriately in the regions of code you have identified as worth parallelizing. If you do not see much code suitable for parallelization using the Cilk primitives, you may want take another look at both the `render.c` and `simulate.c` files.

In order to benchmark your parallel code, use:

```
$ telerun --cores 8 ./bin/find-tier
```

The `telerun --cores 8` command will run your code on an 8-core machine. **Warning:** If you benchmark using `telerun` (no `--cores 8`), it will run on a single core machine, and you will get inaccurate results.

For any changes you make, verify that your code reports the same coloring of pixels and positions of spheres as before. Additionally, verify that the code is race-free by running it through the Cilksan determinacy-race detector. **Note:** Run the Cilksan binary *locally* on your VM.

Profiling the parallel code

You can determine the work and span of your parallel code by executing with CilkScale. How much parallelism do you see? Vary any parameters your algorithm may take as well as any other spawn cut-offs you used. What is the maximum amount of parallelism you can achieve?

5.1 Parallel algorithms

The serial algorithm currently implemented for finding force between bodies `i`, `j` using some function `f()` looks like this (see `update_accelerations()`):


```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        if (i != j) force[i] += f(i, j);
    }
}
```

You can get an easy factor of 2 speedup by traversing only unique pairs like so:

```
for (int i = 0; i < N; i++) {
    for (int j = i+1; j < N; j++) {
        force = f(i, j);
        force[i] += force;
        force[j] -= force;
    }
}
```

One especially tricky portion of the simulation comes with parallelizing these pairwise effects. Naively, to parallelize the above code, one would just replace the outer loop with a `cilk_for`:

```
cilk_for (int i = 0; i < N; i++) {
    for (int j = i+1; j < N; j++) {
        force = f(i, j);
        force[i] += force;
        force[j] -= force;
    }
}
```

This modification, however, produces a race condition since `force[j]` may be updated over many different values of `i` at once. If reads and writes occur out of order, we may get an incorrect result.

One clever way to parallelize this block of code is due to Matteo Frigo. His method uses a recursive algorithm, whose recursive calls can be written as independent `cilk_spawn`'s. *Warning: The course staff does not promise that this algorithm is the best way to parallelize this code!* You must evaluate for yourself whether the overheads introduced by the complexities of Matteo's algorithm are sufficiently small for it to be worthwhile to be used in practice. But since it is a clever parallel algorithm, let's understand how it operates — whether it's a good idea for this project or not.

First, we divide the triangular iteration space described by the double loop

```
for (int i = 0; i < N; i++) {
    for (int j = i+1; j < N; j++) {
        // ...
    }
}
```

into two smaller triangular regions and a single rectangular region, as in Figure Figure 4. (Although you might think that the rectangular region would always be square, it may be a more

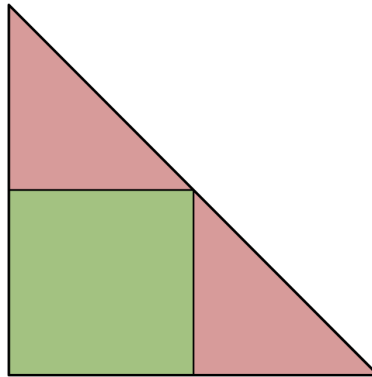


Figure 4: A figure representing the recursion of a triangular iteration space. The red subtriangles can be processed in parallel, and the green rectangle can be processed either before or after. Green and red cannot be processed at the same time, however, without creating a determinacy race.

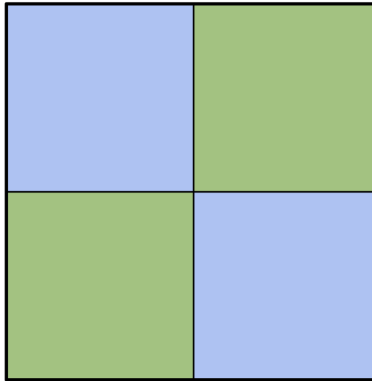


Figure 5: A figure representing the recursion of a rectangular iteration space. The blue subrectangles can be processed in parallel, and then the green subrectangles can be processed in parallel. Green and blue cannot be processed at the same time, however, without creating a determinacy race.

general rectangle when N is not an exact power of 2.) The two triangular regions can be then run in parallel because those regions update `force[i]` and `force[j]` for different values of i and j , and so there are no races. Since the rectangle races with both triangles, however, it must be processed either before or after the triangles.

Fortunately, by further dividing the rectangle into four subrectangles as in Figure 5, we observe that the rectangle can also be processed in parallel. Subrectangles that are diagonal from each other can run in parallel since they update `force[i]` and `force[j]` for different values of i , j , causing no races. If all four subrectangles were to run in parallel, however, there would again be a race.

This process continues recursively for both triangular and rectangular regions. A two-level recursion is illustrated in Figure 5.

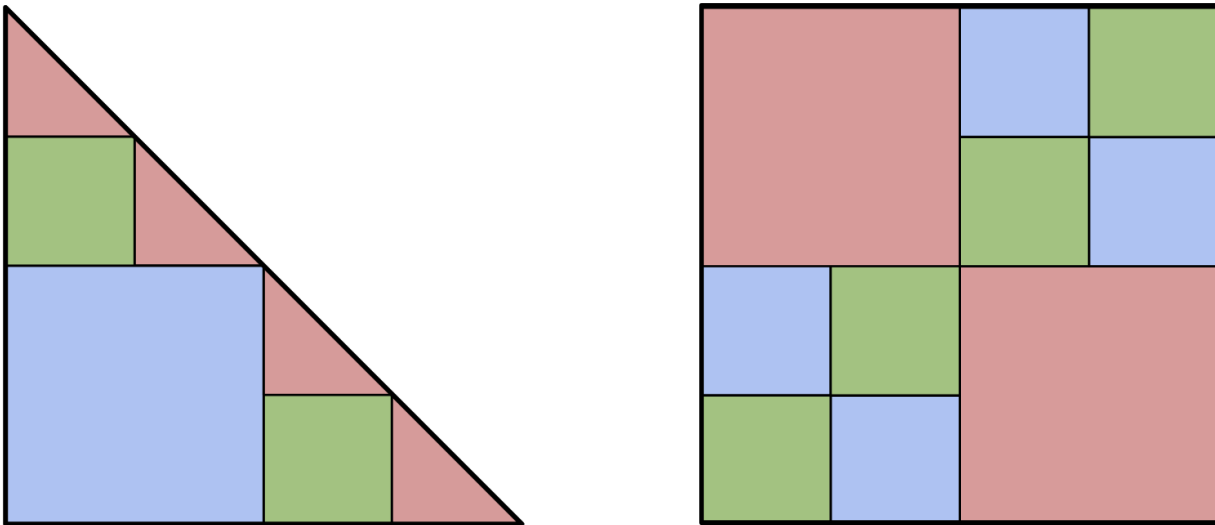


Figure 6: A figure representing the recursion of `triangle()` and `rectangle()` two levels down. A rectangle is recursively broken into subrectangles, and a triangle is broken into two subtriangles and a rectangle, which are then recursively subdivided. Same-colored regions may be spawned concurrently without racing.

Parallelizing render

The rendering algorithm can be sped up with parallelization as well. In the 2D-projection algorithm, each pixel can be colored independently of the other pixels, making it fairly straightforward to parallelize. As you change and optimize the rendering algorithm, the course staff recommends that you ensure that this independence remains in the presence of more complex data structures, although, once again, you are responsible for making your own decisions.

Performance tuning

In your write-up, describe any decisions, trade-offs, and further optimizations that you made, including which parts of the code you parallelized using the divide-and-conquer algorithm explained above. While we will be grading on a machine similar to `telerun --cores 8`, we expect your code should not rely on `CILK_NWORKERS=8` to perform correctly.

6 Evaluation

Please remember to add all new files to your repository explicitly before committing and pushing your final changes.

Evaluation measures

Your grade will be based on all of the following:

Performance (of correct code)

As with Project 1, your final performance grade will be computed by comparing the performance of your submission against a baseline to be announced. Your code will be benchmarked against inputs not made available to you. Therefore, you should keep your code general and be careful not to over-optimize to any specific type of simulation configuration. In addition, you should keep your code as processor-oblivious as possible as a matter of portability. You can use Cilkscale to explore whether your parallel code scales well or not. As a reminder, Cilkscale runs your code serially to compare scalability so you may see the following `telerun --cores 8` output:

```
==== Standard Error ====  
Forcing CILK_NWORKERS=1.
```

For correctness, your code should:

- Match the frames of the starter code pixel-for-pixel for the entire duration of its execution.
- Contain no memory leaks, as checked by AddressSanitizer and Valgrind.
- Contain no determinacy races, as checked by the Cilksan race detector.

Remember that you can run correctness tests using the commands `./scripts/test_all` and `./scripts/tiers_ref_test <number>`, setting the parameter `<number>` as appropriate.

Beta group write-up

You are required to submit on Gradescope a beta group write-up discussing the work that you performed. Please follow the same template for your project write-up as with Project 1. **Important: Please also push your write-up to your Git repository as a `.pdf` or readable `.txt` file.**

Similarly to Project 1, as part of your write-up, you must submit a log indicating how you spent your time on the project. You may choose any method that produces an easy-to-interpret log.

Final group write-up

You are required to submit on Gradescope a final group write-up, which builds on the beta group write-up and additionally discusses the work you performed since the beta release. Similarly to Project 1, in your final write-up, please include the following information:

- An overview of changes you made to your beta release for the final, their impact, and what motivated you to make those changes.
- An updated project log documenting how you spent your time on the final.

Team contract

You are required to submit on Gradescope your team contract, which sets forth conventions on how you and your teammate will collaborate. Remember that you should only submit one copy of the team contract on Gradescope and add both teammates to the submission.

Aesthetic test inputs

The staff has provided a few example input files in the `simulations/` directory. A small portion of your grade is based on the quality of the new inputs you create to test and benchmark your code. These new inputs will be judged based on whether they satisfy the following criteria: (i) the input has challenging performance or correctness properties; and (ii) the input is aesthetically interesting or entertaining. We will curate the most interesting inputs and show them in class.

Grade breakdown

We will grade your project submission based on the following point distribution:

	<i>Beta</i>	<i>Final</i>
Performance (of correct code)	40%	45%
Write-up	5%	5%
Team contract	3%	—
Aesthetic test inputs	2%	—
<i>Total</i>	50%	50%

As with Project 1, this point distribution serves as a *guideline* and not as an exact formula. The staff will also review your Git commit logs to assess the dynamics of your team.

Before you submit

Before you submit your code, complete the checklist below.

- ☐ **Important:** Run `./scripts/test_all` to verify correctness on the test cases in `simulations`, and run `./scripts/tiers_ref_test <number>` to verify correctness up to the given tier.
- ☐ Run the Cilksan race detector to confirm that you have no determinacy races.
- ☐ Run AddressSanitizer and Valgrind to confirm that you have no memory leaks.
- ☐ Run the style checker with the following command:

```
$ find . -regex '.*\.(c|h)' -exec clang-format -style=file -i {} \;
```

- ☐ Ensure that you didn't modify any files that will be reverted for testing (see Section 3.6).
- ☐ Add your aesthetic test input file to the `simulations/` directory.