CSE 491 - Project 2 - Final Submission Writeup

Team: Andrew, Divyalakshmi

Overview and Goal of Project

This project is centred around the idea of developing a high-performance, software-based ray tracer and physics simulator for 3d spheres. The rendering component casts rays from a virtual camera to simulate how light interacts with spheres in a 3d space, accounting for lighting, occlusion, and shading effects. The simulation component models Newtonian gravitational physics and elastic collisions between spheres over time.

The aim was to design a solution that ensures correctness and meets performance requirements when tested on any machine with different architectures (performance portability) and is devoid of memory errors and leaks.

Our beta submission focused on optimising the given serialised rendering and simulation functions, while maintaining accuracy and correctness. We focused the majority of our efforts for the beta submission on improving the given render function and replacing it with an optimised yet still serialised version. In our final submission, we looked deeper into parallelising the way the force-acceleration calculations were carried out and optimising the collision checks between spheres.

Key Features of Our Design

1. *Bounding Box Technique:* To efficiently determine the region of the image where each sphere might appear, we calculate its **axis-aligned bounding box (AABB)** in world space. This is done using the `get_bounding_cube_corners()` function in `render.c`. Unlike naïvely iterating over the whole image for each sphere, this method restricts the expensive ray-sphere intersection tests to a tight box around each object. In our updated version, the bounding cube is computed directly by enumerating all 8 corner offsets from the sphere's centre, avoiding multiple nested for loops (like our previous version) for clarity and potential speed.

2. *Projection of Bounding Box Corners on Plane:* After calculating the 8 bounding cube corners, we **project them onto the image (view) plane** by computing the scalar `k` such that the point lies on the plane defined by `dot(p, W) = 0`, where `W = U × V`.
   This logic is implemented using:
   - `calck()` – computes the scalar `k`
   - `calcProjPoint()` – projects the point using the formula `eye + k * (corner - eye)`

   These projections are essential to find the **2D extent** of the sphere's 3d presence on the screen. We followed this technique, which was mentioned in Recitation 2.1 and Recitation 2.2, to find the projection of 3d spheres onto a 2D viewport.

3. *Conversion of Coordinates to Pixel Values:* Once projected, the 3d coordinates are mapped to **pixel space** using `convert_to_pixel()`. These functions take the `u` and `v` components of each point (based on `U` and `V` basis vectors), scale them using the `pixel_size`, and shift them to centre on the image. This transformation from **projected plane space → pixel coordinates** helps optimise the rendering loop by tightly bounding the set of pixels that may intersect with a sphere.

4. *Hybrid Sort (Insertion and Merge Sort, also known as TimSort) to sort the spheres:* We implemented a **hybrid depth sorting strategy** in `sortInner()` that uses:
    - **Merge sort** for large arrays (n ≥ 512)
    - **Insertion sort** for smaller subsets

    This sorting is based on the **tangent point depth heuristic** (distance² - radius²), ensuring that farther spheres are rendered first. This guarantees correct **occlusion** and avoids overdraw errors.

    The hybrid approach is both **cache-efficient** and **performance-friendly**, leveraging the low overhead of insertion sort on small data and the power of merge sort for large datasets. This is a practical realisation of the lecture's suggestion to optimise sorting logic, particularly when rendering many non-overlapping objects.

5. *Parallelism with Cilk (`cilk_for`):* We briefly explored and introduced parallelism using Cilk in multiple parts of the codebase:
    a) In render.c, the y-loop was parallelised. This allowed pixels inside a bounding box to be computed in parallel, avoiding redundant computations.
    b) In simulate.c, all the heavy lifting loops across different functions - update_accelerations(), update_velocities(), update_positions() - were replaced with cilk_for.

    We also verified that our parallel regions are race-free using Cilksan.

6. *Preserved Structure and Lambert Diffusion Logic for Sphere Shading, Increased Modularity:* We retained the original rendering logic for Lambertian shading to maintain correctness. The code applies Lambert's cosine law to diffuse lighting from all scene lights. We made the rendering code more modular by:
    - Keeping `origin_to_pixel`, `ray_sphere_intersection`, and `set_pixel` as helpers.
    - Moving the projection and bounding logic into reusable functions (`get_bounding_cube_corners`, `convert_to_pixel`).

    This aligns with the project's emphasis on **code maintainability and readability**, and ensures that correctness tests are passed and performance improvements are evident.

7. *Early end before collision check is called:* We used distances between the edges of each direction of the sphere (x, y, z) and used these, as well as the velocity of the spheres, to determine whether or not a collision was possible within the timeframe.

8. *Optimized Gravitational Acceleration Computation:* We optimized sphere acceleration updates by structuring `update_accel_sphere()` to manually compute distance cubed (`dist³ = d * d * d`) and apply negated displacement vectors, reducing floating-point overhead compared to `pow()`. In `update_accelerations()`, we fused the reset and update phases into a single `cilk_for`, minimising synchronisation and improving cache locality. Each thread resets its sphere's acceleration and immediately computes updated forces, maximizing parallel efficiency without introducing races

Stats on Correctness and Performance

Performance: We were able to reach **64 tiers on Telerun.** The optimisations we made to reduce redundant force computations in simulate.c helped us reach from our beta tier 59 to tier 64.

Correctness: Our render and simulate algorithms pass all correctness tests (We ran the correctness hashes for all tiers up to 64). We also checked for memory leaks and errors using Valgrind and the Address Sanitiser, which gave a green signal too. No race conditions were reported by Cilksan either.

Failed Methodologies and Challenges

For the beta submission, our main challenge was reworking the render technique to use bounding boxes. We found that this was very hard to debug for us and had many points of failure. At certain points, we were getting a good amount of speed up (around 53 tiers but were failing all the correctness tests. We were able to independently verify some of the aspects of the algorithm, such as creating the bounding boxes and 3d point projection onto a 2D plane. We then broke these down into specific modular functions, which became easier to use and helped us identify our logical errors. We also constantly verified that the outputs we got were in fact a solution to the system of equations representing the scenario given. Another methodology that failed for us was the parallelisation of the main simulate function. We think this didn't work because this only allowed a few threads at the same time, while we parallelised the other subfunctions in this. So this caused more overhead while also reducing the effectiveness of our parallelisation in the other functions.

For the final submission, we tried to vectorize some of the helper functions in misc_utils.c by hand but were running into a lot of compilation issues due to local architecture differences and once we got past that we also were failing most of the correctness tests, so we decided

to revert to the pre-vectorised version of the code. We also tried more parallelism with functions in render.c and simulate.c that didn't yield any potential speed-ups.

Breakdown of the Aesthetic Test Created

The aesthetic test we created involves many overlapping spheres at many different distances that move towards and away from each other. The reason we chose to make it like this is because it can detect issues in a render function involving sorting of spheres, or the order in which spheres are chosen to render. Another difficulty in this test is that the spheres are of many different colours, and the placement of lights is also across various positions in space. For example, there is a sphere in the middle that ends up rendering as multicoloured due to having a strong red light above it, a green light in front of it, and a blue light under it. This sphere also changes the gradient distribution of these colours as it moves across. As well, there's a blue sphere in the back which moves very quickly towards the eye and moves past all the lights, causing the edges of the spheres to not be visible. We also test the collision of spheres in our test. We chose a multitude of factors and parameter values to make our aesthetic test robust and effective.

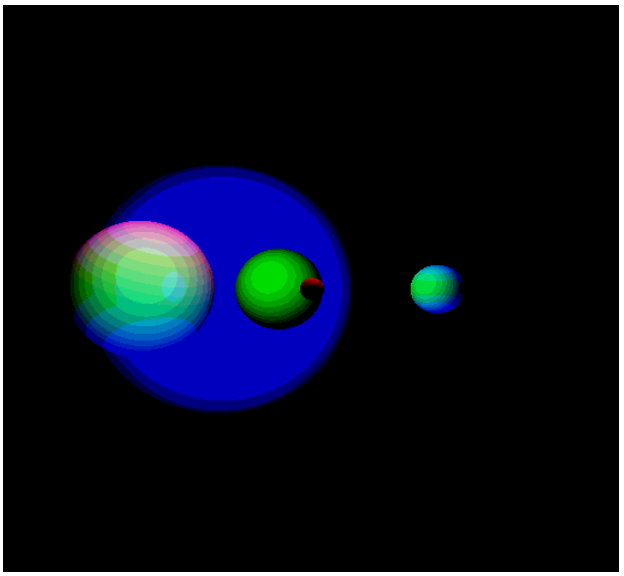

Figure 1: A GIF showing the rendered output of our aesthetic test with multiple colliding spheres at various positions.

Project Log

Date	Start time	Duration - Divya	Duration - Andrew	Description
3/11/2025	7:00 PM	40 mins	40 mins	Drafted and submitted the Team contract.
3/19/2025	7:00 PM	1 hour 20 mins	1 hour 20 mins	Worked on getting the better rendering to work with the bounding boxes, need to get projections to work now.

3/24/2025	8:00 PM	-	3 hours	Worked on math for a better rendering technique. I think I am almost there, will try to finish tomorrow, then put it into code.
3/26/2025	8:00 PM	30 mins	-	Worked on creating a function to create bounding boxes for all the spheres. Pushed it to GitHub.
3/27/2025	1:00 PM	-	2 hours	Finished the projection and started working on colouring pixels.
3/28	3:00 PM	-	5 hours	Tried to get the rest of it working, doesn't quite work
3/28	20:00	1 hour	1 hour	Meeting to debug the new render() function logic with the correct projection math.
3/28	9:00 PM	5 hours	-	Focused on drawing out diagrams to visualise the scenarios. Worked on implementing the correct projection function into the existing render() function. Worked on a 3-D point projection on a 2-D U-V plane. Brought down max diff to 0.711. Passed tier 48 but failing all correctness tests
3/29	1:00 PM	2 hours	-	Worked on a 3-D point projection on a 2-D U-V plane.
3/29	8:00 PM	3 hours		Reduced max difference to 0.00002. Tier 40 passing on telerun. For some reason, when running the reference tests and some of the simulations (15 and 30), the output GIF didn't have any spheres showing up.
3/29	10:00 PM	-	2 hours	Implemented hybrid merge sort and testing out other optimisations on the original sort() code. Speed from 6 to 9 tiers.
3/30	8:00 PM	3 hours	-	Fixed the issue of spheres not rendering in the output GIFS. There was an issue with the ordering of the operations and calculations inside the render function. Organised code into separate functions. Max difference is still the same (0.0000201811..). Tier 43, but correctness is still not passing.
3/30	11:00 PM	-	2 hours	Fixed new renderer up to t42 from base, 43 including merge sorting.
3/31	3:00 PM	6.5 hours	-	Worked on optimising update_accel_sphere() and parallelised some loops in simulate. Tier 50 reached in performance. Correctness passing. Parallelised outer loop in render(). Tier 59 reached.
4/1	11:00 PM	-	2 hours 30	Added test, changed sort, verified we met conditions for the beta
4/2	1:30 AM	1 hour	1 hour	Meeting to discuss aesthetic test cases worked on, run valgrind and the address sanitiser.
4/20	5:30 PM	2 hours	-	Tried implementing the bounding boxes, prune, sweep, and sort algorithm for collision check. Got it to 62 tiers. correctness failing.
4/20	11:30 PM	4 hours	-	Reached tier 62 with correctness. Optimised and parallelised the update_accelerations() and update_accel_sphere(). Tried vectorising some of the helper functions in misc_utils.c but didn't yield anything, so haven't committed it to GitHub yet.

4/22	8:00 PM	-	1 hour	Started working on trying to parallelise the main simulate function
4/23	10:00 PM	-	2 hours	Almost finished simulate and tried to figure out any other optimisations
4/24	2:00 PM	-	8 hours	Finished simulate parallelisation, but it provided a speed down, then worked on and finished a system to end early and not call the collision check at all, which works by checking if the distances between the edges are close enough to have a collision at all within the timestep. The current version fails tier 74 as the only failure and we will not reach it, so this isn't an issue

Acknowledgement of Help from Course Staff and Classmates

The recitation and lectures were helpful and gave us a good idea of how we should be approaching the project. It also introduced us to the concepts of parallelisation, Cache-Optimizations, vectorisation, and other optimisation techniques, which we experimented with. We'd also like to thank our peers who shared the roadblocks they were facing, as well as the speedup they received from the various optimisation techniques they had experimented with. We received help from the course staff in the form of the recitations provided to us, which gave a lot of help on the new rendering technique. We also received help from Hank Murdock, Nihar Saketh Bolareddy, Jathin Sabbineni, and Dheeraj Thota, who we discussed implementation and the math of the bounding box-projection technique to optimize our render function briefly.

We thank our classmates whose beta submission codes were shared on Piazza, which gave us a head start on optimising the redundant force calculations in simulate.c. This gave us the required inspiration to gain a good 3-tier speed-up for our final submission.

Citations

1. Chatgpt and other generative AI tools to help us with C syntax, writing comments/docstrings, drafting an overview for our writeup, and proofreading our beta and final submission writeups.

2. TimSort (Hybrid Merge-Insertion Sort Algorithm): https://en.wikipedia.org/wiki/Timsort

3. Recitation 2.1 Slides: https://software-performance-engineering.github.io/spring25/calendar/Recitation%202_1%202025%20v2.pdf

4. Recitation 2.2 Slides: https://software-performance-engineering.github.io/spring25/calendar/Recitation%202_2%202025.pdf

5. OpenCilk Programming Documentation: https://cilk.mit.edu/programming/

6. Lambert's Cosine Law: https://en.wikipedia.org/wiki/Lambert%27s_cosine_law

7. Recitation 2.3 Slides: https://software-performance-engineering.github.io/spring25/calendar/Recitation%202_3%202025.pdf

8. Recitation 2.4 Slides: https://software-performance-engineering.github.io/spring25/calendar/Recitation%202.4.pdf