Project Title: *Oops! All Collisions*
Team Name: *.DS_STORE*
Team Members and Github Usernames:

- Elle Burkhalter, *elleburkhalter*
- Logan Dapp, *lognd*
- Derrick Davison, *mcnairrobotics*

Github Repository: https://github.com/elleburkhalter/Oops-All-Collisions.git
Video Demonstration: https://youtu.be/fO_JSi4XxXk

Problem:

In large-scale games with high entity counts, such as any tower defense game, efficient collision handling is essential to sustaining real-time performance. A common but inefficient method for detecting collisions is the naive linear approach, where every entity, such as a player or projectile, is checked against every other potentially collidable object. While simple to implement, this method scales quadratically ($O(n^2)$) with the number of entities, making it impractical as the entity count grows into the tens or hundreds of thousands.

Motivation:

To illustrate the severity of this problem, imagine a game with 10,000 active projectiles and 10,000 dynamic entities. A naive approach would result in approximately 100 million pairwise checks per frame, a number that is computationally infeasible even on modern hardware. Games would suffer massive frame drops and input lag if they relied solely on this method, especially during high-action moments when performance is most critical.

The naive linear method also fails to take advantage of spatial coherence—the fact that most entities only interact with those nearby. By ignoring this, it wastes computational resources on unnecessary comparisons between distant objects. This inefficiency becomes even more pronounced in open-world or large map-based games, where entities can be spread over vast areas yet still require frequent updates.

Implemented Features:

Our project functions as a sandbox environment for experimenting with a variety of data structures and techniques for broad-phase and narrow-phase collision detection. It is designed to support rapid implementation and testing and comparison of different approaches. A visualization system displays entities moving in real time, with visual cues such as changing colors and velocities to indicate collisions.

Data Description:

All data used in our project was pulled from a uniform normal distribution and merely served as the foundation for simulating a test environment. Each dynamic entity is assigned an initial position using x and y-coordinates and an initial velocity in both directions. This data is not static. Positions and velocities are continuously updated in every frame to reflect motion, collisions, and physics-based responses. Together, these values contribute to a dataset of over ten

thousand unique and continuously changing data points which we handle at a framerate of nearly thirty frames per second. This allows for realistic, large-scale performance testing of collision-handling algorithms under unpredictable conditions.

In the future, other colliders are possible, all the collisions are intelligently handled and dispatched through *ColliderInterface*, allowing for easy implementation of more complex colliders, so long as the necessary functions are generated.

Tools/Languages/APIs/Libraries:

Our project uses *C++23* and *CMake* for the core build. Visualization is done primarily using the *raylib* library. We also used *Range-v3* for type-erased iterator containers. Unit testing was done using *Catch2 v3.* The external dependencies are automatically fetched using CMake's *FetchContent,* so a quick inspection of the top-level *CMakeLists.txt* and the *tests/CMakeLists.txt* produces a complete view of the build process.

Implemented Algorithms/Data Structures:

We implemented the following structures and algorithms: (1) non-ordered array-based entity storage and naive quadratic comparisons, i.e. every pair is tested; (2) sorted array-based entity storage and the *sweep-and-prune* algorithm commonly used in game-engines today, with the idea of projecting bounding boxes on one axis, keeping a sorted list of interval endpoints, and sweeping once to find overlapping intervals (candidate pairs); (3) spatial hashmap mapping "cell" locations to a std::list of entities and intelligent cell-based coarse look-up, and (4) a multi-level Quadtree where data can be stored in every node and spatial partitioning when nodes become too full.

Roles:

- *Logan Dapp*: data structure, collision, and renderer implementations and video
- *Derrick Davison*: renderer implementation, project setup, project proposal, general bug fixing and computer support.
- *Elle Burkhalter*: test cases, additional programming, report

Analysis:

After our initial proposal, we refocused the scope to emphasize performance, correctness, and core data-structure design over gameplay. We originally planned a game-like scenario with towers and projectiles (dedicated Tower/Projectile classes), but we removed these to avoid coupling gameplay logic to the experimental harness. This simplification allowed us to: stress-test each broad-phase in isolation, compare apples-to-apples with identical motion fields, and debug deterministically without game loop side effects.

We were also initially undecided on data input. We chose randomized initialization (positions/velocities) with autonomous, non-user-controlled entities to ensure high variability and to reveal worst-case pockets (clusters, voids) that challenge each structure differently. This change improved repeatability and made performance deltas between methods more obvious.

Finally, we switched visualization from *SFML* to *raylib*. Raylib's straightforward API and low overhead made it a better fit for drawing large numbers of moving debug primitives and instrumentation overlays. The simpler build surface and good CMake stories reduced friction for teammates and graders, and the responsive draw loop helped us maintain ~30 FPS with 10k+ entities in typical configurations.

Time Complexities:

n = total number of entities

c = average number of entities per cell

NaiveLinear:

- get_collisions(const EntityInterface&); // O(n); scan every entity.
- get_all_collisions(); // O(n^2); (calls get_collision for every entity).
- update_structure(); // O(1); (does nothing).
- (... other methods are O(1) getters or are wrapped in the collision methods)

SweepAndPrune:

- get_collisions(const EntityInterface&); // NOT IMPLEMENTED; not what SAP is for!
- get_all_collisions(); // O(n); like NaiveLinear except only a SINGLE sweep needed.
- update_structure(); // avg: O(n); worst: O(n^2); (insertion sort on nearly sorted array)
- (... other methods are O(1) getters or are wrapped in the collision methods)

SpatialHash:

- get_collisions(const EntityInterface&); // O(c); hashing independent of n, collision only dependent on avg. number of entities in cells.
- get_all_collisions(); // O(cn); loops over every entity, calling get_collisions.
- update_structure(); // O(cn); loops over every entity, potential calling erase or push_back on a std::list, both of which are constant time.
- (... other methods are O(1) getters or are wrapped in the collision methods)

MultiLevelGrid:

- get_collisions(const EntityInterface&); // avg: O(log(n)); worst: O(n); descend a tree-data structure, cutting off 75% of the search space each time; worst case is when the tree is very unbalanced, which is rare.
- get_all_collisions(); // avg: O(n log(n)); worst O(n^2); calls get_collisions n times.
- update_structure(); // avg: O(n log(n)); worst O(n^2); crossing cell-boundaries requires a reinsertion, which with parent pointers is fast, although still dependent on the height of the tree in the case where a root boundary is crossed. The height in the average case is O(log(n)) and O(n) in the worst case.

Logan's Reflection:

Our team found the project both challenging and rewarding, beginning with a game-like vision and converging on a focused sandbox for collision detection that emphasized correctness, performance, and clean design. The live visualization loop made results tangible and helped us see trade-offs between broad-phase methods that weren't obvious from theory alone. This scope adjustment ultimately improved the quality of our comparisons and the clarity of our code.

In terms of challenges, working from different states made alignment difficult because we couldn't "whiteboard" together and clarify assumptions quickly. Misunderstandings around ownership rules, reinsertion policies, and invariants occasionally led to rework, and we wrestled with C++ pitfalls such as iterator invalidation during erasure, smart-pointer cycles, and quadtree split/merge behavior. Despite these hurdles, the friction became a catalyst for better documentation, clearer interfaces, and more disciplined testing.

If we restarted, we would first write a short alignment document that fixes ownership, invariants, and terminology, then lock in benchmark scenes and metrics so results are directly comparable. We would also add lightweight CI with unit tests, formatting, and sanitizers, and choose designs that reduce structural churn, such as loose quadtrees with hysteresis and seeded randomness for reproducibility. Establishing a predictable cadence of brief standups and occasional pairing sessions would further reduce miscommunication.

The project deepened our understanding of design patterns, custom iterators, and smart pointers while forcing us to confront practical trade-offs in Sweep-and-Prune, spatial hashing, and quadtrees. Logan strengthened skills in broad-phase architecture, iterator ergonomics, and renderer instrumentation; Derrick refined renderer structure, build tooling with CMake/FetchContent, and separation of rendering from simulation; and Elle expanded testing discipline and scenario generation.

The most important lesson was the value of shared mental models up front and data-driven instrumentation throughout. Aligning early on invariants and ownership, then measuring consistently with clear benchmarks, turns disagreements into experiments and prevents costly rewrites, especially when asynchronous communication makes rapid clarification harder.

Derrick's Reflection:

For me, this project was a tremendous learning experience. Though I have worked on group projects before, it was nothing like the scale and complexity of this project. Right from the start I was researching all the different data structures and implementations that we were going to end up using. From the abstract base class and interface setup, to importing Raylib into our cmake as a parameter, to learning how the different data structures like spatial hashing worked, this was nothing short of new material to wrap my brain around. This project, while not only introducing me to new approaches for comparing collisions, deeply strengthened my understanding of the use cases of C++ and the complexity and sophistication such a language brings. It was also a unique challenge to try to converse between Logan Dapp and Elle Burkhalter, who came from two completely different backgrounds of C++. I had to make sure they were able to understand each other's concerns while building team bonds so that they could feel accomplished in what they were doing while not running into each other or accidentally rewriting each other's code. If I could do this project again, I would definitely be much more

efficient myself due to the new information I now possess that I did not know at the start of the project.

Elle's Reflection:

The experience I had with this project was extremely valuable for me. I had the opportunity to work with people whose experience and skill far outweighed my own, and I was able to learn a lot from their help. I had many challenges at the beginning of this project, most of which stemmed from my inexperience working with repositories and *CMake*. I had never worked on a project of this scale before and had to rethink the way I do a lot of things, specifically including project setup, building, and overall workflow. The physics component of the project also required me to step out of my comfort zone and do extra research. If I were to start over again, I would pay more attention to the work that other group members were putting into the project. Not only would I have learned even more from my group, this would have also made integration of certain components of the project easier for me. I could have built off of what they had already done much better and been a more helpful addition to the team. Finally, I would begin certain aspects of the project, such as rendering, much earlier. Even without other data structures and class implementations fully complete, I could have begun working on the visual aspect of the project sooner and added other unique features to the main simulation. Overall, this project gave me a much deeper appreciation for project development and has inspired me to improve my own coding abilities in this area.