Zachary Savage

Performance One Pager

10/28/2025

So games similar to mine have three major performance concerns. The first example would be the loading json files or similar NPC heavy files with lots of data points and info. If these assets are large or stored inefficiently, startup time can noticeably increase. Repeated parsing of full data files or loading unnecessary assets also adds delay. The second most common example of performance issues often arise from how the game saves and retrieves player progress. Writing to disk or a database after every small action can slow the game, especially as save files grow. Frequent file access or inefficient serialization compounds over time. Finally each player choice requires evaluating outcomes based on hero stats, difficulty levels, and past results. If this logic is deeply nested or overly complex, response times increase as calculations scale. Inefficient conditional structures or unoptimized randomization can create noticeable lag. While these performance issues are significant in larger games, my game suffers to a lesser degree for now.

To reduce startup delays, essential data should be prioritized during the initial load while secondary elements are loaded asynchronously or on demand. Implementing lazy loading ensures that only the data required for the current session or screen is processed immediately. Caching frequently used assets such as challenge templates or hero archetypes further minimizes file reads. Using compact data formats and precompiled configurations also shortens parse times. For ongoing sessions, maintaining key data in memory rather than reloading from disk each time creates a smoother, faster startup experience. Improving save and load performance begins with reducing how often the system writes to disk or a database. Instead of saving after every action, a queued or timed save system can bundle updates and commit them at regular intervals or specific milestones. Lightweight, structured data formats such as binary serialization or compressed JSON can improve both write and read speeds. In-memory caching of the player’s active state reduces the need for constant I/O operations. Implementing autosave checkpoints and efficient rollback mechanisms ensures reliability without sacrificing responsiveness. To make game logic more efficient, conditional structures should be streamlined and redundant calculations eliminated. Frequently used outcomes can be precomputed or stored in lookup tables, reducing the need for repeated calculations at runtime. Modularizing the decision system—separating randomization, difficulty scaling, and event evaluation—helps isolate performance-heavy operations and makes optimization simpler. Profiling tools can identify logic bottlenecks, enabling targeted improvements. With well-structured logic and predictable decision trees, the game can resolve player choices quickly, even as complexity increases.

I am definitely going to be working on the loading and game file loading. The problem is that my json file has a 120 complex objects with nested outcomes. It’s still not a very long load time, but the more choices I add will add to the load time. In order to handle the loading time I am going to use in-memory caching. So basically I am going to load the safe file, but subsequent loads shouldn’t take as long. In fact it would cut down on the individual load times to nearly nothing. I want this because longer load times will interrupt the flow and experience.