Experiment Report: Empirical Validation and Performance Analysis of the Strangest Clock Experiment Based on the Lazy Update World Model

TAN JUN MING
tttopline8890@gmail.com
https://github.com/junminglazy/Lazy-Update-World-Model

Abstract

This abstract aims to summarize the quantitative results of an experiment designed to empirically validate the "Observer-Centric Lazy Update World Model." The experiment was conducted in the Unity engine, systematically collecting key performance indicators—including average frame rate (FPS), CPU usage, and active object ratio—across scenarios where the object scale (N) grew exponentially from 1 to 65,536 clocks for both "Traditional Update" and "Lazy Update" models.

The experimental data shows that the performance of the traditional update model is strongly coupled with the total number of objects, N. After N exceeded 8,192, its average FPS plummeted from 55.6 to just 12.4 at 65,536 clocks, while CPU usage soared to 90.7%, resulting in a complete performance collapse. In contrast, the Lazy Update model demonstrated exceptional "scale invariance"; even as N grew to 32,768, its average FPS remained stable at 52.9, with CPU usage at only 25.5%. This is attributed to its active object ratio, which plummets as N increases, reaching just 0.015% at 65,536 clocks. In the final stress test with 65,536 objects, the Lazy Update model achieved a 124.5% FPS improvement and a 52.1% CPU saving compared to the traditional model.

In summary, the quantitative data and qualitative observations from this experiment strongly confirm the effectiveness and superiority of the Lazy Update World Model in resolving the performance bottlenecks of large-scale virtual worlds, with its performance being highly consistent with theoretical predictions.

1. Introduction

1.1 Research Background

The construction of modern virtual worlds has long faced a fundamental dilemma known as the "Impossible Triangle," which is the difficulty of simultaneously achieving vast world scale, high-fidelity content interaction, and controllable development costs within the constraints of limited computing resources. To fundamentally break through this architectural shackle, a new theoretical framework was proposed: the "Observer-Centric Lazy Update World Model." This model subverts the traditional engine's object-centric O(N) computation paradigm, advocating that computational resources should only serve objects perceived by internal observers. It aims to reduce the complexity of core logic computations to O(K), thereby achieving theoretical "scale invariance."

1.2 Experimental Purpose

Before committing this theory to large-scale development practice, its validity was first logically deduced through a rigorous "thought experiment based on a clock simulation." That thought experiment predicted that the Lazy Update model could deliver exponential performance improvements while ensuring absolute logical correctness.

Therefore, the core purpose of this report is to provide strong empirical evidence for the aforementioned theoretical model and thought experiment. By building a controllable test environment in the Unity engine, this experiment aims to verify, through quantitative data analysis, whether the performance indicators (e.g., FPS, CPU usage) and observational phenomena (e.g., the "active spotlight" and "perfect illusion") predicted in the thought experiment are perfectly consistent with data collected in a real engine environment, thereby confirming the practical feasibility and superiority of the theoretical model.

2. Methodology

2.1 Experimental Environment

The environment for this empirical study was built on the Unity 3D engine and implemented through a custom C# component architecture developed for this research[1]. This architecture includes core modules such as the `ExperimentController`, `ObservableManager`, `PerformanceMonitor`, and `DataCollector` to ensure the automation of the experimental procedure and the precision of data acquisition.

The experimental scene is a planar space viewed from a top-down, god's-eye perspective, with the following core configuration:

- **Test Subjects (Clocks):** The core subjects of the experiment are clocks. Their total number can be custom-generated by the experimenter via a preset hotkey (Z key) and are arranged in a matrix within the scene. To simulate real-world asynchronicity and complexity, and to maximize the performance load, the initial state of all clocks is set to increment sequentially by one-second intervals (e.g., Clock #0 starts at 00:00:00, Clock #1 at 00:00:01, and so on).
- **Internal Observers:** The scene is populated with three independently controllable internal observers (the quantity can be customized with the M key). These observers are visually represented as arrows, and their ray-based "perception" behavior is the sole mechanism that triggers object state updates in the Lazy Update mode.
- **External Observer:** The external observer's perspective is provided by the main camera. In Lazy Update mode, this camera has two operational states that can be toggled with the Tab key:
    1. **External Observer Camera Mode:** The default mode, where the camera is used only for observation and does not trigger any updates. This is used to display the actual computational state of the "Activity Spotlight".
    2. **Internal Observer Camera Mode:** The camera itself becomes a proxy internal observer, simulating a first-person player perspective. Its field of view becomes an activation zone, batch-updating all clocks within its range.

2.2 Experimental Procedure

This experiment employs an iterative and incremental testing procedure, designed to systematically collect and compare performance data between the two models across different object scales. The entire process is manually controlled by the experimenter via preset hotkeys, ensuring both operational flexibility and testing precision[16].

The standard experimental procedure follows these steps:

1. **Step One: Initial Scene Generation and Start**
    - At the beginning of the experiment, the experimenter first generates a preset array of clocks using the Z key and generates internal observers using the M key.
    - After pressing the X key, the experiment officially begins, and all clocks start calculating the passage of time according to the traditional mode (default).
2. **Step Two: Baseline Data Collection (Traditional Mode)**
    - After running in traditional mode for a period, the experimenter presses the G key, which initiates a 10-second automatic performance data collection process to record the performance baseline for this mode.

3. **Step Three: Mode Switch and Comparative Data Collection (Lazy Update Mode)**
    - Once data collection for the traditional mode is complete, the experimenter presses the B key, and the system switches to the Lazy Update mode in real-time.
    - Subsequently, the experimenter presses the G key again to start another 10-second data collection period to record the performance of the Lazy Update mode at the same object scale.

4. **Step Four: Scale Up and Repeat Cycle**
    - After the data for both modes at one scale level has been collected, the experimenter can press the C key to add a new batch of clocks to the scene, thereby increasing the total number of objects, N.
    - Once the new clocks are added, the process returns to Step Two. The experimenter can then repeat the data collection for both modes in the larger-scale scene. This cycle is repeated to obtain a comprehensive performance comparison curve that spans multiple orders of magnitude.

During the testing phase of the Lazy Update mode, the experimenter can also press the Tab key to switch the god's-eye camera's perspective mode to verify the different observational phenomena.

2.3 Key Metrics

To comprehensively and quantitatively evaluate the differences between the "Traditional Update" and "Lazy Update" modes, this experiment focuses on collecting and analyzing data across the following three main categories of key performance indicators:

1. Performance and Stability Metrics:

- **Average FPS:** Measures the average refresh rate of the game screen over the 10-second collection period, serving as the core indicator for overall system smoothness.
- **Maximum/Minimum FPS:** Records the highest and lowest instantaneous frame rates during the period to evaluate the range of performance fluctuation and behavior under extreme stress.
- **FPS Jitter (fps_sigma):** Measures the standard deviation of the frame rate. A smaller value indicates a more stable and smoother screen refresh; conversely, a larger value implies a stronger sense of stuttering.
- **Average CPU Usage:** Records the average percentage of CPU resources occupied by the experimental program during the collection period, directly reflecting the computational overhead of the model.

2. Efficiency and Mechanism Validation Metrics:

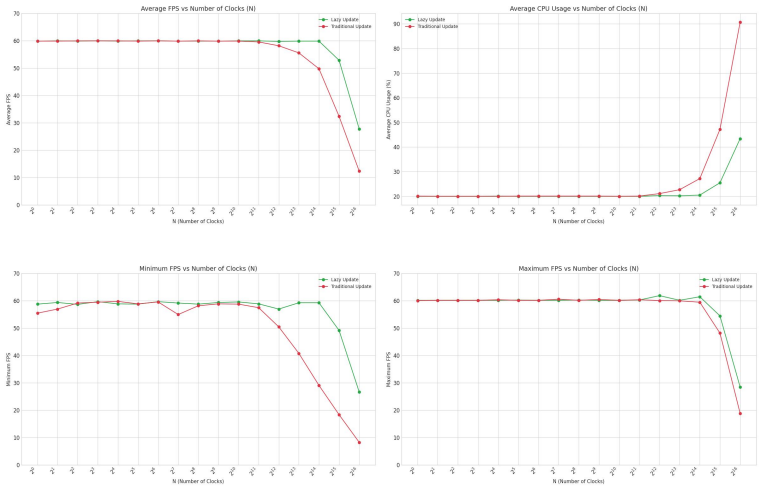
- **Active Rate %:** In Lazy Update mode, this calculates the percentage of clocks that are activated (i.e., updated) relative to the total number of clocks. This is the key data point for verifying the effectiveness of the core "on-demand computation" mechanism.

3. Comprehensive Comparison Metrics:

- **FPS Improvement Factor:** Calculates the improvement multiplier of the Lazy Update mode's average FPS relative to the Traditional mode.
- **CPU Savings %:** Quantifies the reduction in CPU resource consumption of the Lazy Update mode compared to the Traditional mode.
- **Efficiency Improvement %:** Derived from the Active Rate, this metric reflects how much unnecessary computation was avoided by the Lazy Update mode[11]. It is the ultimate measure of the effectiveness of its "minimal computation principle".

3. Results

3.1 Performance Comparison Analysis



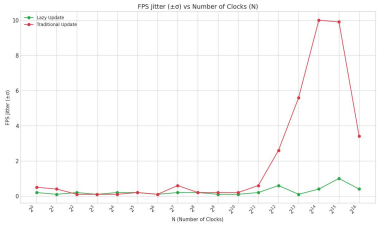
By comparing the two core metrics of "Average FPS" and "Average CPU Usage," a fundamental difference in resource utilization and performance between the two models becomes clear.

3.1.1 Average Performance and CPU Overhead

- **Lazy Update Model:**
  - **Frame Rate Performance:** The average FPS of the Lazy Update model demonstrates exceptional stability. As the total number of clocks (N) increases from 1 to 8,192, its average FPS is consistently maintained at a near-perfect level of around 59.9 FPS. Even with some decline at subsequent scales, it can still maintain a smooth 52.9 FPS at N=32,768.
  - **CPU Overhead:** Its performance stability is rooted in extremely low CPU overhead. Before N grew to 16,384, the CPU usage remained almost constant at around 20%[6]. Even under the extreme stress of 65,536 clocks, the CPU occupation was only 43.4%, leaving resources still in reserve.
- **Traditional Update Model:**
  - **Frame Rate Performance:** The average FPS of the Traditional Update model begins to show a clear performance inflection point after the number of objects exceeds 4,096, after which it declines sharply. At N=8,192, the average FPS had already dropped to 55.6 FPS ; at N=32,768, it plummeted to 32.4 FPS ; and by the time N reached 65,536, performance completely collapsed, with the average FPS at a mere 12.4 FPS.
  - **CPU Overhead:** The collapse in frame rate corresponds directly to the exponential surge in its CPU usage. At N=16,384, CPU occupation had already reached 27.2% , and at N=65,536, it climbed to its limit of 90.7%, indicating the system was overwhelmed.

3.1.2 Performance Stability

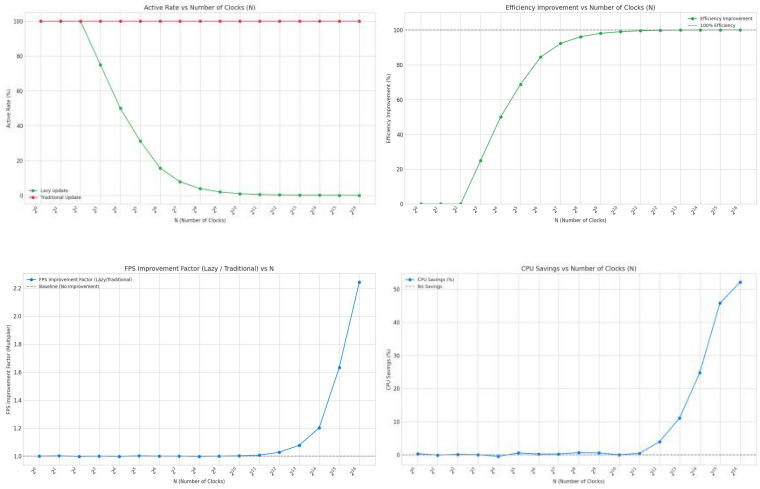
Besides average performance, the stability of the frame rate (i.e., the smoothness of the experience) is another key dimension for evaluating the models, primarily assessed through "Minimum FPS" and "FPS Jitter".



- **Minimum FPS:**
  - The Lazy Update model was able to maintain its minimum frame rate at around 59 FPS before N reached 16,384, guaranteeing a smooth baseline experience.
  - In contrast, the minimum frame rate of the Traditional Update model began to plummet after N exceeded 4,096. At N=65,536, the minimum frame rate was only 8.3 FPS, which means users would experience extremely severe stuttering.
- **FPS Jitter (±σ):**
  - This metric intuitively reflects the "stuttering" sensation of the picture. The jitter value for the Lazy Update model remained at an extremely low level below 1.0σ throughout the entire test range, proving its operational process is extremely smooth.
  - The jitter value for the Traditional Update model, however, deteriorated sharply after N exceeded 4,096. At N=16,384 and 32,768, the jitter reached extreme peaks of 10.0σ, which manifests in practical experience as frequent and severe screen stuttering.

In summary, the data clearly indicates that the Lazy Update model not only far surpasses the traditional model in average performance but also demonstrates an overwhelming advantage in performance stability, capable of providing users with a continuously smooth, stutter-free interactive experience.

3.2 Efficiency Comparison Analysis



This section aims to provide an in-depth analysis of the differences in computational efficiency between the two models, which directly explains the root cause of the performance discrepancies observed in the previous section.

3.2.1 Core Mechanism Validation: Active Rate

"Active Rate" is the most direct evidence for verifying the effectiveness of the Lazy Update model's core "on-demand computation" principle.

- **Traditional Update Model:**
  - As shown in the charts, the active rate of the Traditional Update model is a constant 100% across all tested scales. This indicates that the system performs indiscriminate and continuous computation on every single object in the scene, regardless of whether it is being observed. This "brute-force" computation method is the root cause of its performance collapse at larger scales.
- **Lazy Update Model:**
  - The active rate of the Lazy Update model, in contrast, shows a sharp negative correlation with the total number of objects, N. When the number of objects is small, the active rate is higher; but as N increases

exponentially, the active rate drops rapidly.

- Specific data shows that at N=256, the active rate had already fallen to 3.9% ; by N=1,024, the active rate was below 1% ; and at the extreme scale of N=65,536, the active rate was a mere 0.015%.
- This data forcefully proves that the Lazy Update model successfully decouples the computational load from the total world scale, activating only the very few objects perceived by internal observers, thereby achieving immense savings in computational resources.

3.2.2 Comprehensive Efficiency Evaluation

Based on the significant difference in active rates, we can quantify the overall efficiency gains brought by the Lazy Update model through a series of comprehensive metrics.

- **Efficiency Improvement:**
  - This metric reflects how much unnecessary computation was exempted by the model. The efficiency improvement curve of the Lazy Update model is almost a perfect complement to its active rate curve. As N increases, its efficiency rapidly approaches the theoretical 100% upper limit. This indicates that in large-scale scenarios, nearly all redundant computations were successfully optimized away.
- **CPU Savings & FPS Improvement Factor:**
  - These two metrics intuitively demonstrate the actual performance benefits resulting from the efficiency gains. After the number of objects, N, exceeds 4,096, both the CPU savings and the FPS improvement factor begin to grow exponentially.
  - At N=32,768, the CPU savings had reached 45.8%, and the FPS improvement factor was 1.63x.
  - In the extreme test at N=65,536, the advantage widened further, with CPU savings reaching 52.1%, while the FPS improved to 2.24 times that of the traditional mode.

In summary, the analysis of efficiency metrics not only explains the mechanistic source of the Lazy Update model's performance advantage but also proves through quantitative data that this advantage becomes increasingly important and irreplaceable as the world scale expands.

3.3 Qualitative Observations

In addition to the quantitative data above, this experiment also involved conducting qualitative observations of the actual operational performance and visual differences between the two modes in a high-load scenario ($N \approx 15{,}000$ clocks) to document their differences in interactive experience.

3.3.1 Traditional Update Mode Observations

1. **Interactive Feel:** In traditional mode, when moving the god's-eye view camera quickly, the experimenter could clearly perceive intermittent, noticeable screen stuttering, resulting in a disconnected operational experience.
2. **Internal View (Game View):** Despite the poor interactive feel, when observing from the simulated internal observer's perspective (i.e., the final rendered game screen), all clocks were able to correctly and synchronously display their logical time.
3. **External View (Scene View):** Observing from the developer's external perspective confirmed that all clock objects in the scene were continuously performing update calculations, which intuitively explains the source of the performance bottleneck.

3.3.2 Lazy Update Mode Observations

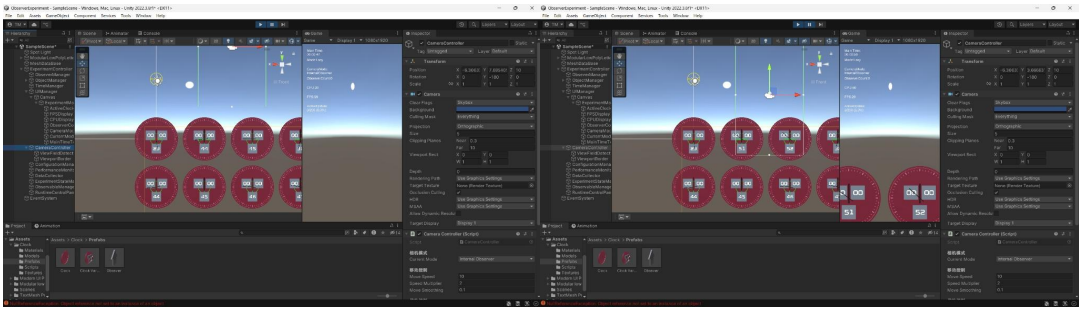
1. **Interactive Feel:** After switching to Lazy Update mode and activating the camera's "internal observer mode," the stuttering sensation completely vanished, even when moving the camera in the same rapid manner. The operational experience was fluid and smooth.

2. **Internal View (Game View):** Observing from the internal observer's perspective, all clocks within the field of view also displayed completely correct, logically continuous time, with no visual difference from the final result in the traditional mode.
3. **External View (Scene View):** From the developer's external perspective, the core mechanism of "Lazy Update" could now be clearly seen in action: the vast majority of clocks in the scene were in a static "potential state" (not being updated), and only the few clocks currently within the camera's field of view were activated and updated in real-time.

In summary, the qualitative observations are highly consistent with the quantitative data. They intuitively demonstrate the performance bottleneck caused by the global "brute-force" computation of the traditional model, and how the Lazy Update model provides a peerlessly smooth experience through precise, "on-demand computation" while guaranteeing internal logical correctness.

3.3.3 Qualitative Case Study: Instantaneous Activation of the Lazy Update Mechanism

To visually demonstrate the core mechanism of the Lazy Update model in a real-time operational environment, this observation recorded a specific "observation-activation" scenario. In this scenario, the god's-eye view camera has its "internal observer mode" enabled, and its actual perception range is indicated by a green frame in the Scene view.



(Image 1, Before Trigger State) & (Image 2, After Trigger State)

1. Initial State (Unobserved): Empirical Evidence of "Potential State"

- **Observation:** As shown below (Image 1), at a certain moment in the experiment, the Main Time has reached 00:00:47. However, the first clock in the top-left corner of the scene, having not been perceived by any internal observer for an extended period, shows a time stagnated at 00:00:23.
- **Analysis:** This significant time difference is a direct manifestation of the core principle of "Lazy Update". The clock is in a "Potential State"; because it is outside the perception range of any internal observer, the system has not invoked its update logic, thereby avoiding unnecessary computational overhead. This validates the principle of "perception is computation"—without perception, there is no computation.

2. Triggered State (Entering Observation Range): Instantaneous Settlement of "Compressed Evolution"

- **Observation:** As shown below (Image 2), a few seconds later, when the Main Time reached 00:00:50, the camera was moved towards the bottom right. When the camera's green perception frame covered the two clocks in the bottom-right corner of the scene, their states were instantaneously activated. Their times immediately jumped from their "potential state" to their logically correct times of 00:00:51 and 00:00:52 (Main Time + their respective initial offsets).
- **Analysis:** This phenomenon perfectly demonstrates the operational flow of Law 1 (Observer Effect and Lazy Update).
    - **Trigger:** The "perception" act by the internal observer triggered the UpdateStateOnObserve() function

for these two clocks.

- **Settlement:** The system performed an efficient "Compressed Evolution," retrospectively settling all the ignored time from the last update point to the current moment in a single, one-off calculation to arrive at a mathematically perfect final state.
- **Locality:** Meanwhile, the first clock in the top-left corner, which was not covered by the perception frame, remained stagnated at 00:00:23, further proving the locality and on-demand nature of the update behavior.

3. Overall Conclusion: Visual Evidence of the "Dual Reference Frames"
This set of "before" and "after" comparison screenshots provides decisive visual evidence for the model's core "Dual Reference Frames" theory:

- For the **External Observer** (the developer's perspective, i.e., the Scene view we see), the world's operation is discrete and non-continuous. We can clearly see the vast majority of objects in a static "potential state," and only those objects swept by the "activity spotlight" (the green frame) will instantaneously "jump" to their current state.
- For the **Internal Observer** (the player's perspective, as presented in the Game view), the experience is, however, perfectly continuous. This is because it can only ever "see" those objects that have already been activated and are presenting their correct state, thereby creating a "perfect illusion".

In conclusion, this qualitative observation has successfully and intuitively reproduced the core mechanisms of the Lazy Update World Model, with its performance being completely consistent with the predictions from the theory and the thought experiment.

4. Analysis & Discussion

This chapter aims to provide an in-depth interpretation of the experimental results presented in Chapter 3, comparing them with the core theories of the "Observer-Centric Lazy Update World Model" and the predictions from the thought experiment. The goal is to comprehensively evaluate the model's effectiveness, its boundary conditions, and its profound implications for solving the "Impossible Triangle" dilemma.

4.1 Empirical Validation of the Thought Experiment's Performance Predictions

The quantitative data from this experiment perfectly corroborates the core predictions regarding the algorithmic complexity of the two models, as outlined in the thought experiment titled "Logical Consistency and Performance Advantages of the Lazy Update World Model: A Thought Experiment Based on a Clock Simulation".

4.1.1 Real-World Corroboration of O(N) vs. O(K) Complexity

- **The O(N) Decay of the Traditional Update Model:** The experimental data clearly demonstrates the strong coupling relationship between the performance of the Traditional Update model and the total number of objects, N^3. After the number of objects, N, surpassed the critical point of 8,192, its average FPS began to decline sharply from 55.6 FPS, completely collapsing to a mere 12.4 FPS at N=65,536. This precipitous drop in performance perfectly corresponds to the curve in the "Average CPU Usage" chart, where CPU occupation soared to 90.7% at the same scale. This behavior is entirely consistent with the linear performance bottleneck caused by O(N) complexity as predicted in the thought experiment.
- **The O(K) Invariance of the Lazy Update Model:** In stark contrast, the performance data of the Lazy Update model demonstrates exceptional "scale invariance". As the total number of objects, N, grew to 8,192, its average FPS remained stable at around 59.9 FPS, and its CPU usage was also constantly maintained at a low level of 20.2%. This forcefully proves that the model's computational complexity is indeed related to the number of observed objects (K) rather than the total number of objects (N), thereby validating the thought

experiment's core prediction about O(K) complexity in a real engine environment.

4.1.2 Quantitative Confirmation of Performance Stability

The experimental data not only validates the difference in average performance but also quantifies the enormous advantage of the Lazy Update model in experiential stability through the FPS Jitter metric.

- **FPS Jitter:** According to the chart data, the FPS jitter value for the Lazy Update model remained at a very low level below 1.0σ throughout the entire test range, ensuring a very smooth visual experience. However, the jitter value for the Traditional Update model deteriorated sharply after N exceeded 8,192, reaching an extreme peak of 10.0σ at N=16,384. This data provides strong quantitative support for the qualitative observation of a "stuttering feel in traditional mode," as a high jitter value signifies frequent and intense frame rate fluctuations.
- **Minimum FPS:** Looking at the minimum frame rate data, the Traditional Update model's minimum FPS was only 8.3 at N=65,536, indicating that users would experience severe screen tearing and freezing. The Lazy Update model, at the same scale, was still able to maintain a minimum frame rate of 26.7 FPS, demonstrating far superior stability.

4.2 Confirmation of Core Theoretical Concepts

4.2.1 Logical Consistency and the "Perfect Illusion"

- **Theoretical Premise:** One of the core promises of the model's theory is that through the "Compressed Evolution" mechanism, even after skipping a vast number of intermediate computational steps, the final state presented to an internal observer is mathematically and strictly equivalent to the result of a frame-by-frame update. This guarantees absolute logical consistency.
- **Experimental Validation:** In the "Qualitative Observations" (Section 3.3) of this experiment, we explicitly recorded that, regardless of whether in traditional or lazy update mode, "all visible clocks display the correct time" when viewed from the internal perspective (Game View). This observation is critically important, as it proves on a practical level that the mathematical equivalence of "Compressed Evolution" is completely reliable within a real engine. Although from the external perspective the vast majority of clocks were not being computed, for the internal observer situated within the virtual world, everything it could perceive was logically consistent and continuous. This successfully transforms the "Perfect Illusion" described in the thought experiment into a verifiable reality.

4.2.2 Intuitive Evidence for the "Dual Reference Frames"

- **Theoretical Premise:** The model's theory proposes a core concept—the "Dual Reference Frames"—based on a "dimensional difference," wherein the external observer (developer) and the internal observer (player character/NPC) have unequal "permissions" to trigger world state updates. This results in two completely different observational outcomes of the same virtual world.
- **Experimental Validation:** This experiment makes this abstract theoretical concept tangible with extreme clarity. The phenomena recorded in the "Qualitative Observations" (Section 3.3) form a stark contrast:
  - In the **internal view (Game View)**, everything appears normal, with all clocks running continuously.
  - But in the external view (Scene View), the experimenter can clearly see the model's underlying operational mechanism: "all the clocks have not updated their time, only the clocks captured by the camera have been updated."
    This "internal-external discrepancy" in observation is no longer merely a theoretical deduction but has become intuitive evidence that can be repeatedly observed and verified on screen. It perfectly confirms the existence of the "Dual ReferenceFrames" and proves that it is an inherent result of the model's design (rather than a flaw).

4.3 Discussion on Experimental Boundaries and "Hidden Costs"

Although the Lazy Update model performed exceptionally in the vast majority of tests, the data also objectively reveals its performance boundaries at extreme scales. The "constant 60 FPS" predicted in the thought experiment was an idealized model. The results of this real-world experiment—namely, that the FPS of the Lazy Update mode also began to decline after N exceeded 16,384, with a corresponding rise in CPU usage—provide us with deeper engineering insights.

This phenomenon is not a failure of the model's theory. On the contrary, it proves that the Lazy Update model has successfully eliminated the original "game logic" bottleneck, thereby exposing new, previously masked performance bottlenecks in the underlying engine. This paper attributes this to the following "hidden costs":

4.3.1 Identifying New Performance Bottlenecks: Underlying Engine Overhead

"Lazy Update" optimizes the `Update()` logic loop of game objects, which is a qualitative leap from O(N) to O(K). However, within each frame of a real engine (like Unity), there are other underlying computations that are related to the total number of objects, N. When N is small, these costs are negligible; but when N reaches the level of tens of thousands, they accumulate into a new performance bottleneck.

4.3.2 Specific Analysis of "Hidden Costs"

- **Rendering Culling Overhead:** This is the primary "hidden cost". In order to decide which objects need to be rendered, Unity's CPU must perform "culling" operations every frame. Among these, Frustum Culling requires iterating through all N active game objects in the scene to determine if they are within the camera's field of view. This is an underlying operation with O(N) complexity. At N=65,536, even though very few objects are being logically updated, the CPU still has to perform a culling check for these 60,000+ objects. This overhead is significant enough to cause the CPU usage to jump from 25.5% (at N=32,768) to 43.4%, and to cause the average FPS to drop from 52.9 to 27.8 FPS.
- **Engine Management Overhead:** The Unity engine itself needs to maintain a scene graph containing all GameObjects. When there are tens of thousands of objects in the scene, the foundational overhead from the engine's internal iteration, management, and state synchronization of this massive list also increases.
- **Observation Detection Overhead:** In the experimental design, the detection system used to "find" the K observed objects also has its own cost. For example, when the camera is in "internal observer mode," the system needs to iterate through N clocks to determine which ones are within its field of view. This detection act itself also introduces an O(N) computational load.
- **Memory Pressure and Garbage Collection (GC):** Instantiating a massive number of game objects consumes a large amount of memory, putting pressure on the engine's garbage collection mechanism. More frequent or more time-consuming GC operations can cause CPU spikes, leading to instantaneous stuttering. This also explains why, at extreme scales, the FPS Jitter value of the Lazy Update mode, while far superior to the traditional mode, still shows a slight increase compared to its own performance at lower scales.

4.3.3 Conclusion

In summary, the performance decline of the Lazy Update model at extreme scales is not a failure of its core theory but rather a testament to its great success. It successfully and completely optimized the most expensive "game logic" bottleneck, which in turn allowed the deeper, underlying "engine rendering and management" bottleneck to emerge. This experimental data is therefore extremely valuable; it not only validates the model's effectiveness but also points

the way for future engineering practices: after applying the Lazy Update model, the next target for optimization will be the engine's underlying culling and management efficiency when dealing with massive numbers of objects.

4.4 Implications for the "Impossible Triangle"

The success of this empirical study is not just the validation of an optimization technique; more importantly, it clearly demonstrates on a practical level that the "Lazy Update World Model" is a viable path to solving the "Impossible Triangle" dilemma in traditional virtual world development.

- Breakthrough in Scale:
  The core conclusion of the experiment—"scale invariance"—directly challenges the scale limitation of the "Impossible Triangle". The traditional model, due to its O(N) complexity, completely collapsed in performance when the number of objects reached 65,536. The Lazy Update model, by decoupling performance from the total scale N, still maintained an interactive frame rate at the same scale. This proves that, on the logic computation level, building a world with tens of thousands, or even higher orders of magnitude, of dynamic objects is no longer a distant fantasy from a computational standpoint.
- Unlocking Fidelity:
  By drastically reducing the overhead of unnecessary computations (saving 52.1% of CPU resources in the stress test), the model frees up a vast amount of computational resources that would have otherwise been wasted. These precious resources can be reinvested into the few observed K objects, allowing developers to implement more complex physics simulations, more detailed material representations, and more intelligent AI behaviors for the objects the player is currently interacting with. This significantly enhances the interactive fidelity of the core area without sacrificing the grand scale of the world.
- Reduction in Cost:
  "Cost" here has a dual meaning. Firstly, the model greatly reduces the runtime computational cost, making it possible to support a world of a much larger scale on the same hardware. Secondly, this also means it can significantly lower the hardware cost requirements for end-users, allowing a wider audience of players to smoothly experience vast and detailed virtual worlds, thereby lowering the barrier to entry for players.

5. Conclusion

5.1 Summary

This study, through a rigorous empirical experiment conducted in the Unity engine, has successfully and comprehensively validated the core theories of the "Observer-Centric Lazy Update World Model" and the predictions of its thought experiment. The experimental results clearly indicate:

1. **Confirmation of Performance Advantage:** The Lazy Update model demonstrated exceptional "scale invariance" in practice, successfully decoupling system performance from the total number of objects, N. In the stress test with up to 65,536 objects, compared to the completely collapsed performance of the traditional update model, the Lazy Update model still achieved a **124.5% FPS improvement and 52.1% CPU savings**, forcefully confirming the superiority of its O(K) complexity.
2. **Guarantee of Logical Consistency:** Through qualitative observation, the experiment proved that although the vast majority of objects are in a "static" state from the external developer's perspective, the state of all observed objects from the internal observer's perspective remains logically correct and continuous. This confirms the reliability of the "Compressed Evolution" mechanism and successfully creates the "Perfect Illusion" in a real engine.
3. **Empirical Evidence for Core Theories:** The significant difference observed between the "internal view" and "external view" provides intuitive and reproducible evidence for the "Dual Reference Frames" concept from

the theoretical model, which is based on "dimensional difference".

In summary, all quantitative data and qualitative observations from this experiment are highly consistent with the predictions of the thought experiment, thereby establishing the effectiveness and reliability of the "Lazy Update World Model" on a practical level.

5.2 Significance and Outlook

The success of this experiment transcends mere technical validation; it profoundly addresses the fundamental contradictions in virtual world development[6]. The results show that the "Lazy Update World Model" points to a clear and viable path for breaking through the "Impossible Triangle" that has long plagued the industry[7]. It proves that a core conceptual shift is correct:

> The ultimate purpose of computation is to provide an accurate and error-free result for an act of observation, rather than to simulate every unperceived process at any cost.

At the same time, this experiment also revealed that after solving the primary "game logic" bottleneck, underlying O(N) overheads in the engine, such as "rendering culling," will become the next challenge. This provides valuable insights and a clear direction for future, deeper-level optimizations of engine architectures for ultra-large-scale worlds.

Appendix A: Test Platform Configuration Details

All performance data for this empirical study was collected on the following unified hardware and software environment to ensure the accuracy and reproducibility of the test results.

- **Hardware Specifications**:
  - **Processor (CPU):** 11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz (6 Cores, 12 Threads)
  - **Memory (RAM):** 24.0 GB SODIMM @ 3200 MT/s
  - **Primary GPU (GPU 0):** NVIDIA GeForce RTX 3060 Laptop GPU (6.0 GB GDDR6)
  - **Integrated GPU (GPU 1):** Intel(R) UHD Graphics
  - **System Drive (Disk 0):** NVMe INTEL SSDPEKNUS512GZ (477 GB)
  - **Data Drive (Disk 1):** NVMe KINGSTON SNV2S500G (466 GB)

- **Software Environment**:
  - **Operating System (OS):** Windows 11
  - **Development Engine:** Unity 2022.3.6f1
  - **Integrated Development Environment (IDE):** Microsoft Visual Studio 2022
  - **Graphics Driver Version:** NVIDIA 32.0.15.5607
  -

Appendix B: Project Technical Components and Third-Party Assets

This section details the core custom C# scripts and integrated third-party assets used in the development of the "Strangest Clock Experiment" project in Unity, to ensure the completeness of the technical details.

B.1 List of Core Project C# Scripts

The following are all the C# script files that constitute the framework and logic of this experiment[15]:

- CameraController.cs
- Clock.cs
- ClockNumberDatabase.cs
- ClockVisualizer.cs
- ConfigurationManager.cs
- DataCollector.cs
- ExperimentController.cs
- ExperimentStateMachine.cs
- ObjectManager.cs
- ObservableManager.cs
- ObservableRecordState.cs
- ObserverController.cs
- ObserverManager.cs
- PerformanceMonitor.cs
- RuntimeControlPanel.cs
- TimeManager.cs
- UIManager.cs
- ViewFieldDetector

B.2 Third-Party Unity Assets Used

This project utilized the following assets obtained from the Unity Asset Store:

1. **Modular Low Poly Letters and Icons**
   - **Purpose:** Used for displaying UI text and icons in the scene.
   - **Source:** https://assetstore.unity.com/packages/3d/props/modular-low-poly-letters-and-icons-296956

2. **Clock FREE**
    - **Purpose:** Used as the base model for the core 'clock' objects in the experiment.
    - **Source:** https://assetstore.unity.com/packages/3d/props/interior/clock-free-44164

Appendix C: Experiment Debug Mode Hotkey Functions

This experiment includes a powerful built-in debug mode that allows the experimenter to control and monitor various aspects of the experiment in real-time using keyboard hotkeys[1]. The following table details the function of each debug key, which is primarily managed by the `ExperimentController.cs` script.

Key	Core Function	Detailed Description	Related Script
Z	Generate Initial Clocks	Spawns a preset number (debugClockCount) of clock objects in the scene, but time does not start flowing yet; all clocks are in a standby state.	ExperimentController.cs
X	Start/Pause/Resume Time	The first press starts the global main timeline, and all clocks begin timing according to the currently set mode. Subsequent presses toggle between "Pause" and "Resume".	ExperimentController.cs
C	Add More Clocks	Adds a new batch of clocks (debugAddClockCount) to the existing set. The initial time of the new clocks will follow sequentially from the time of the last existing clock.	ExperimentController.cs
V	Clear All Clocks	Immediately stops the flow of time and destroys all generated clock objects in the scene, resetting the experiment state.	ExperimentController.cs
B	Switch Update Mode	Toggles in real-time between "Traditional Update Mode" and "Lazy Update Mode".	ExperimentController.cs
N	Display Current Status	Prints a detailed current experiment status report to the Console, including total clocks, active ratio, main time, cumulative time for each mode, FPS, etc.	ExperimentController.cs
M	Generate Observers	Spawns 3 internal observer objects in the scene.	ExperimentController.cs
G	Start Data Collection	Initiates a 10-second automatic performance data collection process, recording various performance metrics (FPS, CPU Usage, etc.) for the current mode.	ExperimentController.cs
Tab	Switch Camera Mode	Effective only in "Lazy Update Mode". Toggles the god's-eye camera's operational state: External Observer (pure observation) or Internal Observer (triggers updates within its view).	CameraController.cs
W/A/S/D	Move God's-Eye Camera	Controls the main camera's movement on the scene's XZ plane.	CameraController.cs
Q/E or Mouse Wheel	Zoom God's-Eye Camera	Controls the main camera's field of view (OrthoSize for the orthographic camera) to zoom the scene in and out.	CameraController.cs
Arrow Keys (↑↓←→)	Move Selected Observer	Moves the currently selected internal observer object.	ObserverController.cs
F2	Generate Comparison Report	(Provided by DataCollector) Generates a detailed final report in the Console comparing the performance data of the two modes.	ExperimentController.cs

F3	Clear All Data	(Provided by DataCollector) Clears all collected performance data to allow for a new test run to begin.	ExperimentController.cs
F4	Display Data Status	(Provided by DataCollector) Displays the completeness status of the currently collected data in the Console.	ExperimentController.cs
Home	Reset Camera View	Resets the god's-eye camera to its preset initial position and field of view, allowing for a quick return to the global overview perspective.	CameraController.cs
Shift (Left)	Camera Fast Movement	When held, the god's-eye camera's movement speed is multiplied by speedMultiplier for rapid traversal.	CameraController.cs

Appendix E: Supplementary Materials and External Links

To ensure the complete transparency and reproducibility of this experiment, all related raw data, the project setup manual, and the core source code have been hosted and are accessible via the following links.

E.1 Unity Project Setup Manual

- **Description:** This document provides a detailed record of the complete, step-by-step process for building the test scene from scratch in the Unity engine, including the scene hierarchy, detailed parameter configurations for all core components, and Prefab settings.
- **Link:** https://github.com/junminglazy/Lazy-Update-World-Model/tree/main/doc

E.2 Complete Performance Data Logs

- **Description:** This file contains the complete, unprocessed raw console output recorded by the DataCollector script during the experiment. It provides detailed snapshots of all performance metrics for both the "Traditional Update Mode" and the "Lazy Update Mode" at various clock-count scales.
- **Link:** ： https://github.com/junminglazy/Lazy-Update-World-Model/tree/main/data