`.janicre`: A Log-Scale, Reversible Semantic-Commit Manifest for Multi-Agent Software Reasoning

Anonymous ACL submission

Abstract

Large language models (LLMs) promise repository-scale assistance, yet real projects routinely exceed 128 k-token windows and fragment key logic across heterogeneous files. Existing techniques—RAG pipelines, chunked retrieval, or brute-force long-context prompting—either leak code, stumble on cross-file reasoning, or incur quadratic cost.

We realise `.janicre` via an *arbitrary-depth* abstraction pipeline (Stage $1 \dots k$), whose cumulative JSON snapshots plus an HTML semantic map form the final manifest; stopping at any depth trades tokens for detail while the worst-case growth remains $\Theta(\log N)$.

Beyond human-to-LLM use, `.janicre` functions as a **model-agnostic exchange layer**: distinct agents (GPT-4o, Claude, Gemini, *etc.*) can inspect the same manifest, attach `thinking_trace` logs, and negotiate edits—enabling true agent-to-agent (A2A) collaboration without exposing raw source. Coupled with git-like semantic diffs, the manifest becomes a living memory of design rationale that is auditable by both machines and humans.

We formalise the schema, analyse its compression bounds, and outline an empirical plan—HumanEval, MBPP, and delta-stability benchmarks—to compare `.janicre`–augmented reasoning against standard retrieval and full-file prompting.

`.janicre` thus upgrades IR-level code summarisation into a universal, dialogue-ready substrate for large-scale, multi-agent software intelligence.

Highlight: `.janicre` abstracts multi-language codebases into a $\log N$-scale manifest that aligns with transformer reasoning and agent collaboration—without code leakage. It bridges human design intent and LLM reasoning via a real-time semantic interface.

1 Introduction

1.1 Problem Context

LLM assistants have two systemic bottlenecks: (i) *token budgets* ($>$128 k tokens are still dwarfed by modern repositories) and (ii) *fragmentation*—domain logic is scattered across files, packages, and services. Current practice therefore relies on ad-hoc snippets or retrieval pipelines, risking both context loss and code leakage.

1.2 Compression Inspiration: From MP3 and MP4 to Code Semantics

Just as MP3 compresses audio by removing frequencies imperceptible to humans, and MP4 compresses video by encoding delta frames and keyframes, `.janicre` compresses code by extracting only semantically relevant units. These units represent the purpose and behavior of the system in a compact and interpretable form. This mirrors the design goal of enabling LLMs to focus on core logic rather than superficial syntax, providing an efficient input representation aligned with human intuition and transformer attention.

1.3 Limitations of Existing Approaches

Retrieval-augmented generation (RAG) retrieves relevant snippets (Lewis et al., 2020) but struggles with cross-file reasoning due to limitations in context aggregation, such as chunk-based retrieval that fragments inter-file dependencies. Long-context models (Dong et al., 2023) reduce manual selection but incur quadratic computational costs in transformer attention and degrade on gigabyte-scale repositories due to inefficiencies in scaling long sequences.

1.4 Transformer-centric Abstraction

Rather than a fixed three-step recipe, we allow a **variable-depth cascade**: Stage 0 (raw code) $\rightarrow$ Stage 1 (AST) $\rightarrow$ Stage 2 (purpose) $\rightarrow \cdots \rightarrow$

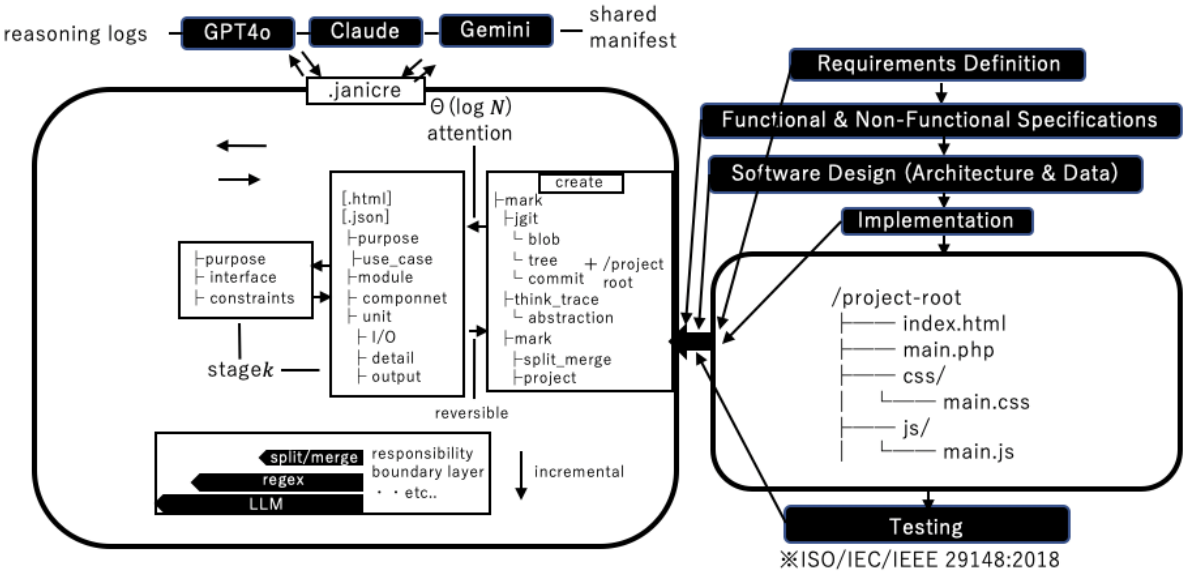

Figure 1: High-level overview of `.janicre`: (left) multi-stage semantic compression with $\Theta(\log N)$ attention; (right) mapping to the classical SDLC V-model.

Stage k (intent, theory). Each step retains only the *attention-anchoring tokens* (names, types, keywords), analogously to how MP3 preserves auditory maskers. Each stage is stored as a reversible semantic commit, enabling loss-bounded round-trip across abstraction depths. Because k can be capped at $O(\log N)$, the pipeline still yields $\Theta(\log N)$ tokens in the worst case, yet lets practitioners exit early when smaller windows suffice.

To enhance transformer alignment, we define a soft attention prior between semantic units based on their abstraction depths. Specifically, we compute the attention weight between unit i and j as:

$$w_{i,j} = \exp(-\alpha \cdot |d_i - d_j|), \quad A_{i,j} = \frac{w_{i,j}}{\sum_j w_{i,j}}$$

where d_i denotes the abstraction depth of unit i, and α is a tunable decay factor. This encourages intra-stage attention and smooths cross-stage transitions, aligning the attention pattern with the underlying semantic hierarchy.

1.5 Our Contribution

A Unified Time-Aware Interface for LLM Reasoning

Unlike static representations, `.janicre` introduces a time-aware, dialogue-based manifest that models not just software structure, but its design rationale across time. This enables LLMs to interpret codebases as evolving conversations—aligning each unit with a reasoning turn and associated expectation.

Moreover, `.janicre` supports *dynamic compression control*, allowing the manifest to adapt its granularity based on context: high-detail traces for chain-of-thought reasoning, and compact structural summaries for fast inference or retrieval. This flow-sensitive abstraction turns `.janicre` into a scalable LLM interface, bridging software design, memory, and attention cost.

Together, these features establish `.janicre` as a temporal-semantic interface between software systems and large language models.

1.6 Our Contribution

1. **Schema.** We define `.janicre`, a manifest that compresses repository intent into $\mathcal{O}(\log N)$ tokens.

2. **CLI.** An open-source generator produces manifests for Python / C++ projects; Rust/Go plugins are planned.

3. **Security.** Only derived metadata crosses the firewall, preventing proprietary code exposure.

2 Prompt-oriented DSLs

Syntactic IRs. Early LLM-for-code studies rely on *syntactic* representations such as ASTs (CodeBERT) or CFGs (GraphCodeBERT). These IRs expose parse-tree structure but seldom encode *intent*, *semantics*, or cross-module relationships.

Prompt-oriented DSLs. More recent work shifts to prompt engineering DSLs (e.g., agent scripts, Toolformer graphs). While flexible, these DSLs are *imperative* and often tied to a specific runtime, limiting reuse across projects.

`.janicre`: a semantic, declarative IR. Our schema occupies the gap between the two trends:

- **Semantic layer** – captures *purpose*, *I/O contracts*, and *module hierarchy*, not just syntax.
- **Declarative form** – JSON/YAML manifest aligns with transformer attention, enabling token-efficient reasoning ($\mathcal{O}(\log N)$ tokens).
- **LLM-friendly graph** – unit-level nodes and explicit edges mirror token-level attention (Vaswani et al., 2017).

Recent frameworks such as PDL (Chen et al., 2024), LangGPT (Wang et al., 2024), and DocCGen (Pimparkhede et al., 2024) define declarative layers for modular LLM interaction and document-conditioned code generation. These approaches improve prompt composability, yet typically remain imperative, runtime-tied, or file-local. In contrast, semantic-commit and time-aware intermediate representations—structured to persist abstraction history—remain largely unexplored.

The `.janicre` schema is not merely a descriptive manifest, but a structurally optimized intermediate format for LLM-based reasoning.

First, we choose JSON/YAML as the base format for the following reasons:

- **LLM Compatibility:** Transformer-based LLMs attend over token sequences, and key–value structured formats (like JSON) yield stable and predictable attention paths, helping models form logical groupings during inference.
- **Multilingual Metadata Extraction:** Languages such as Python, JavaScript, Rust, or Go expose ASTs that can be naturally mapped to JSON, enabling unified structural representation.
- **Declarative Abstraction:** Unlike imperative code, this schema allows systems to be described independently of runtime execution—capturing semantics and modular relationships.

Moreover, we treat all software as ultimately a composition of input-output transformations. This aligns with the principle that LLMs operate effectively when provided with structured descriptions of units, each explicitly named and ordered. For example:

```
"units": ["startGame", "endGame", "
    spawnEnemy"]
``` | A simplified excerpt is shown
    below:
```json
"units": ["startGame", "endGame", "
    spawnEnemy"],
"commit_meta": {
  "timestamp": "2025-05-20T12:34Z",
  "depth": 3
}
```

Such structure enables LLMs to attend across unit boundaries and predict interactions more accurately.

This design echoes attention-based reasoning, as introduced in (Vaswani et al., 2017), and adapts it to LLM–software interfaces. Rather than prompt full source files, we create interpretable hooks for the model to follow.

Moreover, while the core `.janicre` schema is formally defined, we consider any simplified variants (e.g., flat representations), partial modular forms, or implementation-specific adaptations to be legitimate extensions under the same conceptual framework. These derived or restructured formats remain within the scope of our proposal as long as they preserve the semantic objectives and hierarchical abstraction principles defined herein.

2.1 Preliminary Observation

A manually structured Shogi engine (˜2GB) suggests a **97% token reduction** when encoded via `.janicre`, compared with full-file prompting.

Schema Overview. The `.janicre` manifest describes a software system's purpose, components, and logic in a JSON/YAML format optimized for LLM reasoning. Each manifest includes fields such as `global_objective`, `code_sharing`, and `code_logic`, which summarize the architecture, I/O contract, and computational units.

A simplified excerpt is shown below:

```
"units": ["startGame", "endGame", "
    spawnEnemy"]
```

The manifest's complete structure—including `global_objective`, `code_sharing`, and

detailed `code_logic` units—is provided in the supplementary file `example.janicre.json`.

Modular Composition. The `.janicre` schema is inherently composable. Multiple sub-manifests—e.g., `ui.janicre` for HTML layout and `backend.janicre` for Python/JS logic—can be referenced or included by a higher-level `core.janicre`. This enables distributed teams or tools to independently define interface, logic, and algorithms, yet produce a unified LLM-interpretable manifest. A merging protocol resolves shared objectives and unifies component trees and unit indexes across manifests.

Merge respects commit order; higher-depth units are deterministically regenerated from lower-depth blobs, guaranteeing reversible abstraction quality.

2.2 From Git Commits to Dialogue Units

Traditional version control systems like Git decompose software into a series of commits—each capturing a structural diff and accompanied by a descriptive message. These commits form a linear or branching history, enabling developers to track the evolution of code over time.

`.janicre` draws inspiration from this idea, but shifts the focus from syntactic diffs to semantic intent. Each `unit` in a manifest acts like a commit: it is named, scoped, and justified. The optional `thinking_trace` serves as a semantic commit message, recording the rationale, prompt context, and decision steps behind the unit's creation or modification.

Unlike Git, which records lines of code, `.janicre` records units of purpose and reasoning. This structure provides an audit trail for LLMs—not just for what changed, but why—and enables model-aware tools to interpret, revise, or simulate development at the semantic level.

In this sense, `.janicre` is a Git-like protocol for software semantics, optimized for transformer-based reasoning systems. It captures code not merely as syntax but as a living dialogue of design decisions.

2.3 Conversational Interface Inspiration

Why Meaning Emerges Only in Dialogue. Transformer-based LLMs operate by predicting the next token, but prediction alone does not guarantee semantic coherence. Given an underspecified prompt such as *"Write about time"*, RLHF-tuned models tend to regress toward a *single, safest average* response that maximizes annotator agreement. This phenomenon—semantic convergence toward a median output—arises precisely because the input lacks *directional intent*. **The less a prompt specifies expectation, the more the model collapses meaning into neutrality.**

Dialogue format resolves this degeneracy. Each conversational turn embeds an expectation shaped by context: the prior utterance narrows the manifold of plausible continuations and forces the model to align with a specific semantic target. **Only in dialogue can an LLM both imagine and justify a precise value—because dialogue provides expectation.** We thus view conversation as a *convergence protocol*—a structure that transforms token-level prediction into context-aligned reasoning.

Structural Properties of Dialogue. Beyond convergence, dialogue also supplies inductive biases beneficial to transformer models:

- **Turn-taking** constrains the scope per utterance, preventing attention from overextending.
- **Intent segmentation** frames each utterance as a discrete, nameable semantic unit.
- **Contextual grounding** maintains a shared, evolving state across steps.

These properties are not cosmetic—they *scaffold* model behavior in alignment, memory, and structured inference.

Empirical Support. This perspective is supported by empirical studies. Ouyang et al. demonstrate that instruction-tuning via conversational prompts improves model stability and alignment with human preferences (Ouyang et al., 2022). Christiano et al. show that turn-based feedback via reinforcement learning yields more robust policies than one-shot or batch supervision (Christiano et al., 2017). These results underscore that conversation is not merely interaction—it is optimization.

Design Implication for `.janicre`. This insight motivates the structural choices in `.janicre`. Each unit—explicitly named, scoped by input/output, and embedded in a hierarchical context—acts as a semantic utterance in a system-level dialogue. Rather than presenting code as flat syntax, `.janicre` frames it as

intent-driven discourse. **Software becomes a dialogue among components, and LLMs can reason over it not as text, but as communicative structure.**

Why Meaning Emerges Only in Dialogue. Transformer-based LLMs operate by predicting tokens—but prediction alone does not guarantee semantic coherence. In unconstrained text, the output space is vast and amorphous; without external structure, the model's generation lacks a center of gravity.

Conversational format resolves this. Each turn imposes an expected response, compressing the space of plausible outputs and forcing the model to converge toward semantically meaningful values. **It is only within dialogue that an LLM can both imagine and justify a specific value—because dialogue embeds expectation.**

In essence, the act of dialogue is a convergence protocol: a structural constraint that transforms token-level prediction into intent-aligned imagination. We posit that *conversation is not merely a UX design, but the minimal viable structure under which large language models can produce grounded meaning*.

This motivates our design of `.janicre`. By treating each software unit as a scoped utterance—with name, purpose, and contract—it mirrors the same structure that enables LLMs to think. Software becomes a dialogue between components, and LLMs can reason about it not as code, but as communicative structure.

3 Reference Implementation & Multi-Stage Pipeline

3.1 Pipeline Overview

We implement a generator `janicre-cli` that emits *Stage-i* JSON files ($i = 1 \dots k$), each representing a progressively abstracted view of the codebase—from raw ASTs to high-level semantic units. This cascade allows developers to tune token granularity, stopping at any Stage-j to fit context limits while preserving structural clarity.

Language-Agnostic Design. Unlike language-specific tools, `janicre` is not bound to any particular syntax or ecosystem. It supports Python, JavaScript, HTML, Vue, and even domain-specific DSLs. Each Stage-i manifest shares a unified schema capturing purpose, input/output contracts, and modular hierarchy, enabling LLMs to reason consistently across heterogeneous systems. Future extensions may support Rust, Go, and declarative formats like YAML or Terraform.

HTML Semantic Map. All Stage-i JSONs are merged into an interactive HTML canvas powered by Vue. Clicking a unit reveals its `source_ref`, editing its `purpose` live-updates the underlying JSON, and exporting yields a consolidated `.janicre`. This interface offers both machine-parsable structure and human-navigable semantics—bridging reasoning and authoring within a single workflow.

4 Semantic Commits & Thought Trace

To extend `.janicre` beyond static manifest, we introduce `thinking_trace`—a per-unit log of reasoning steps and LLM prompts that shaped each commit.

This allows:

- traceable origin of unit definitions (e.g., spawn logic);
- comparison across models (e.g., GPT vs Claude);
- intent continuity under refactoring or future evolution.

Just as Git preserves structural diffs, `.janicre` maintains conceptual diffs across units. This enables semantic scaffolding for LLM pretraining, where prompt-response pairs can serve as alignment records.

Git-style Correspondence. We model each unit as a semantic commit, allowing `.janicre` to function as a time-aware versioning interface for LLMs. The correspondence is as follows:

- **Commit message** $\rightarrow$ `thinking_trace` rationale
- **Diff** $\rightarrow$ semantic delta between abstraction depths
- **Timestamp** $\rightarrow$ temporal ordering of abstractions

By merging structure with discourse, `.janicre` becomes not just a snapshot—but a living memory of design evolution.

5 Token and Security Analysis

Let c ($0 < c < 1$) be the empirical geometric decay factor, empirically observed as $c \approx 0.25$ in our corpus. We model the token count at abstraction depth i as:

$$T_i = c^i N$$

where N is the original token count at Stage 0. This geometric decay reflects progressive semantic pruning. At the maximum depth k, the token count converges to:

$$T_k \approx \Theta(\log N), \quad \text{for} \quad k = \left\lceil \log_{1/c}(N) \right\rceil$$

Practitioners can choose any $j \leq k$ depending on context window constraints. Given a maximum token budget B (e.g., 32k tokens for GPT-4), the minimum required stage depth $k^\star$ can be computed as:

$$k^\star = \left\lceil \log_{1/c} \left(\frac{N}{B} \right) \right\rceil$$

This enables automated tradeoffs between semantic fidelity and budget, making the system adaptive to various model capacities.

Notably, the compression is not merely a pruning—it preserves transformer-aligned units such as attention anchors and semantic landmarks, retaining the system's functional semantics at every level.

6 Planned Evaluation

We will benchmark HumanEval and MBPP to measure (i) accuracy gain versus snippet baselines, (ii) token savings, and (iii) zero-leakage assurance by manual review.

Strategic Value of Model-Agnostic Benchmarking. By aligning `.janicre` evaluation with standard LLM benchmarks such as SWE-bench and ToolBench, we enable consistent, model-agnostic assessment of code reasoning performance. Crucially, this compatibility allows users to switch between LLM backends (e.g., GPT-4o, Claude, Gemini) without restructuring the prompt interface or modifying the underlying manifest.

This interchangeability reduces vendor lock-in and amortizes the cost of adopting `.janicre`—transforming initial investment in manifest structuring into a long-term asset reusable across evolving model ecosystems. We argue that this flexibility plays a pivotal role in enterprise-scale LLM deployment, where rapid model turnover and API discontinuities are frequent. As such, benchmark alignment thus becomes not merely an evaluation methodology, but a cornerstone of long-term operability, cost mitigation, and architectural agility in evolving LLM ecosystems.

Delta Stability under Code Evolution. Inspired by MP4's delta frame model, we plan to measure the stability of `.janicre` manifests across codebase changes. Specifically, we will benchmark whether small code edits (e.g., bug fixes or new functions) result in local diffs within the manifest. This helps evaluate the long-term viability of `.janicre` as a persistent abstraction layer over evolving repositories.

We will also benchmark abstraction round-trip fidelity (shallow $\rightarrow$ deep $\rightarrow$ shallow) to quantify reversible commit accuracy and ensure semantic equivalence across depth transformations.

Depth-Sweep Study. For $j = 1 \ldots k$, we feed $\{\text{Stage} \leq j\}$ into GPT-4o-32k and measure task accuracy against token count. This allows us to verify empirical geometric decay, validate compression efficiency, and identify the "elbow" depth—i.e., the optimal tradeoff point between token budget and semantic fidelity.

Limitations

While `.janicre` enables efficient representation and reasoning, several limitations remain.

First, it is designed for static manifest generation. Dynamic behaviors—such as `eval`-based execution, reflection via `getattr`, or runtime mutation of control flow—cannot be fully captured through static analysis alone. As such, runtime-specific logic may require complementary trace-based instrumentation.

Second, current support is limited to statically typed or semi-structured languages; dynamically reflective systems may require additional adaptation for full semantic fidelity.

Third, the compression quality depends on developer-authored metadata such as docstrings and function modularity. Poorly documented or monolithic code may yield underspecified or noisy manifests, reducing the effectiveness of downstream LLM reasoning.

Finally, `.janicre` guarantees loss-bounded round-trip across depths, though regenerating full source from the highest abstraction may be lossy

in syntactic details.

7 Potential Risks

We discuss three main risks. (1) Information-leakage risk: even though `.janicre` removes raw source, a manifest may still expose sensitive architectural metadata; we propose configurable redaction levels and cryptographic hashes to mitigate this. (2) Misuse by malicious agents: an attacker could inspect the manifest to craft targeted exploits; we suggest access-control wrappers when `.janicre` is generated from proprietary code. (3) Over-reliance on compressed views: developers might miss security-critical edge-cases that were pruned; we therefore recommend fall-back to deeper stages or full static analysis for safety-critical code.

8 Conclusion

`.janicre` compresses software intent and structure into a single manifest, enabling LLMs to assist on large, fragmented codebases within token and privacy constraints.

Beyond practical usage, the schema may serve as a candidate format for future LLM pretraining corpora. With sufficient `.janicre` samples accumulated across projects, models could learn modular abstraction, intent classification, and high-level architectural reasoning—analogous to how code search engines or doc2vec embeddings are built from structured repositories today.

We are drafting a formal JSON Schema for `.janicre` v2.1 to enable validation and integration into IDE tooling.

Additionally, by abstracting software logic into a model-independent format and aligning it with benchmark-based evaluation, `.janicre` ensures long-term interoperability across evolving LLM architectures. This enables seamless model substitution, minimizes prompt engineering overhead, and mitigates risks of vendor lock-in or platform-specific obsolescence.

In doing so, `.janicre` delivers not only structural and reasoning efficiency, but also economic sustainability—transforming initial LLM integration efforts into a durable, reusable asset that preserves value amid a rapidly shifting model landscape.

Appendix A: `.janicre` Manifest Blueprint

The fragment below illustrates the full conceptual scaffold that `.janicre` is designed to capture. It can be read as a "check-list" for repository-to-LLM interaction.

global$_{objective}$

- **purpose**: compute closed-form solutions to symmetrical tiling problems
- **technical**$_{category}$: $discrete - mathalgorithms, CLIscripts$

- **code**$_{sharing}$

  - **language**: language-agnostic (supports Python, JavaScript, HTML, etc.)
- **structure.external**$_{links}$: $none$

- **code**$_{logic}$

- **overview.core**$_{algorithm}$: $combinatorialenumerationunderinversionsymmetryoverview modulardecompositionwithseparationofUIandlogic$

Frontend:

- **module**: `ui.vue`
- **component**: `gameUI`
- **units**: `startGame`, `endGame`, `spawnEnemy`

Unit Definitions:

- **name**: `startGame`
  - **type**: method
  - **purpose**: Initialize and start game loop
  - **detail.input**: -
  - **detail.output**: Running game state
  - **detail.steps**:
    1. Initialize game flags and score
    2. Reset player and bullet arrays
    3. Setup and play background music
    4. Launch game loop with `requestAnimationFrame`
  - **detail.formula**: -
  - **detail.theory**: UI state initialization and animation loop
  - **detail.application**: Browser-based arcade games
- **name**: `endGame`
  - **type**: method
  - **purpose**: Stop the game and display results
  - **detail.input**: -

  - **detail.output**: Stopped state and final score shown
  - **detail.steps**:
    1. Pause background music
    2. Set `gameOver = true`
    3. Show result UI
  - **detail.formula**: -
  - **detail.theory**: State transition in game UI
  - **detail.application**: Game-over screen logic
- **name**: `spawnEnemy`
  - **type**: method
  - **purpose**: Create a new enemy on canvas
  - **detail.input**: -
  - **detail.output**: Enemy rendered in game space
  - **detail.steps**:
    1. Randomize spawn coordinates
    2. Push enemy object to array
    3. Render in next animation frame
  - **detail.formula**: -
  - **detail.theory**: Procedural enemy generation
  - **detail.application**: Game difficulty balancing

research$_{notes}$

- Inspired by grid symmetry in 2023 Shogi tile patterns
- Based on known complexity bounds in tiling theory

References

Tianhao Chen, Weijia Xu, and 1 others. 2024. Pdl: A declarative prompt programming language. *arXiv preprint*.

Paul F Christiano, Jan Leike, Tom B Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. Deep reinforcement learning from human preferences. In *Advances in Neural Information Processing Systems*, volume 30, pages 4299–4307.

Zican Dong, Tianyi Tang, Lunyi Li, and Wayne Xin Zhao. 2023. A survey on long text modeling with transformers. *arXiv preprint*.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *arXiv preprint*.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, and et al. 2022. Training language models to follow instructions with human feedback. *arXiv preprint arXiv:2203.02155*.

Sameer Pimparkhede, Mehant Kammakomati, Srikanth G. Tamilselvam, Prince Kumar, Ashok Pon Kumar, and Pushpak Bhattacharyya. 2024. Doccgen: Document-based controlled code generation. *arXiv preprint*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, pages 5998–6008.

Ming Wang, Yuanzhong Liu, and 1 others. 2024. Langgpt: Rethinking structured reusable prompt design framework for llms from the programming language. *arXiv preprint*.