Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

RESEARCH ARTICLE

Declarative Time-Based Animation using Domain Specific Language

Priyadharsan K[1] | **Srikaanth R**[1] | **Srinivasan S**[1] | **Dr Moratanch N**[2]

[1]Undergraduate Student, Department of Computer Science and Engineering, SRC, SASTRA Deemed University, Kumbakonam, India

[2]Assistant Professor, Department of Computer Science and Engineering, SRC, SASTRA Deemed University, Kumbakonam, India

Correspondence
Corresponding to: Dr Moratanch N, Assistant Professor, Department of Computer Science and Engineering, SRC, SASTRA Deemed University, Kumbakonam,India
Email: authorone@gmail.com

Abstract

Animations are an essential element of the current interactive systems, where the visual behavior is supposed to change with time in a known and predictable fashion. Even with the ubiquitous presence of animation tools, time as a representable and analyzable entity is not easily captured especially where the animation behavior needs to be fine-grained. Most of the currently used methods of animation implicitly represent time by discrete keyframes or using percentages, thereby restricting the ability to infer time. To overcome this drawback, it has been inspired by the GameScript, a domain-specific language that minimizes complexity with minimum programming in the creation of 2D games. GameScript represents every production-level object as a self-reliant entity, known as an Agent, and uses a Per- Agent System to control behavior through time. Based on this idea,the given approach proposes an approach, in which time is a first-class function, and time can be evaluated, serialized, and reasoned about in a systematic manner. Within this context, time is evaluated as an unending operation and systematically translated into values to be implemented. At the time analysis step the expressions written in the DSL are interpreted and converted into representations that can be executed by general animation executor. Consequently, the definition of animation is divided into the two issues: temporal and visual realization. The result of the synthesizing is eventually translated into Cascading Style Sheets (CSS) wherein keyframes and animation properties are automatically created. The system does not require any manual working with keyframes and percentage values by using a compiler-like workflow comprising of lexical, syntactic, and semantic analysis. The abstraction enables the expression of complex, time-constrained animations at a higher level, resulting in increased clarity, accuracy and maintainability and compatibility with existing web rendering pipelines.

KEYWORDS

Animation - Domain Specific Language (DSL); Animation Behaviour; Temporal Sampling; Keyframe Generation; Time-Aware Compilation; Declarative Animation Modeling

1 | INTRODUCTION

Animations are ubiquitous in contemporary digital media, appearing in applications ranging from video games to everyday visual displays such as electronic billboards. As technological advancements have evolved, the process of scripting animations has become increasingly streamlined; however, challenges remain, particularly in achieving precise timing control and fine-tuning animation parameters. Current tools, including CSS animations and AI-driven solutions, often struggle with complex timing adjustments or granular control, especially when users require detailed customization. To address these challenges, we propose a simplified approach to creating CSS animations by integrating GameScripta domain-specific language (DSL)into the development workflow. This integration facilitates the automatic generation of key CSS components, including @keyframes rules and related properties, thereby enhancing efficiency and accuracy in animation development. Originally, GameScript was designed for creating interactive and visual content such as classic arcade-style games including Asteroid Destroyer and Tetris. Its application to the web domain leverages its structured syntax, repeatability, and logical clarity. Importantly, we do not suggest that GameScript or our enhancements are intended

to replace traditional CSS coding; rather, they serve as tools that generate CSS code efficiently. Our focus is on enabling smoother curve transitions, hover effects, focus states, active states, group interactions, property transformations, and peer-related logic. While our approach offers significant advantages, it is important to acknowledge certain limitations inherent to the current architecture of GameScript, such as the implementation of grid-based functionalities. We propose a methodology that converts a DSL built by us, characterized by a user-friendly, IF-ELSE structured syntax similar to natural language, into CSS animations through a series of processing stages. These include parsing, sampling, semantic interpretation, and a critical timing layer component. The timing layer plays a pivotal role in controlling the precise output of animations, enabling users to achieve desired effects with improved accuracy and ease.

2 | RELATED WORKS

This section looks at recent research done in game scripting, domain-specific languages, narrative control, machine assistance, and ease-of-use game development methodologies. Rather than separate or individual literature reviews, the discussion follows the continuity of major concepts in the literature as progression is made with respect to works that account for the weaknesses of earlier approaches and how these extensions contribute to the design of the proposed GameScript (GS).

2.1 | Rule-Based and Declarative Game Scripting

The early research in the domain of game scripting focused on reducing procedural complexity by representing game logics with rules and declarative constructs. Initial studies showed that from structured rule sets, game mechanics and levels could be generated automatically and that expressive gameplay does not necessarily require detailed procedural coding (Paper 1). This work established that rule-based representations would be feasible as a foundation for game logics. As rule-based systems attracted attention, researchers started to investigate their expressive power in practical settings. Investigations on declarative rule-based languages presented that compact formulations of rules can qualitatively record emergent behaviors and complex interactions within games (Paper 16). However, as these systems increase in size, concerns move to script organization, efficiency, and maintainability. Further investigation emphasized poorly designed or architected scripting languages that can hamper both development speed and runtime performance (Paper 5). Later approaches, to gain clarity and, particularly, long-term maintainability, introduced structured declarative models that explicitly separate game state from behavioral rules(Paper 18). Although this separation benefited readability and organization, most of the existing solutions tended to miss a unified and lightweight scripting approach that properly balances simplicity against expressive control. These observations lay the basis for GS, which uses a compact, rule-based scripting approach that is intended to remain readable even as game logic becomes more complex

2.2 | Domain-Specific Languages for Game Development

Based on the shortcomings of general-purpose scripting, there were efforts directed at domain-specific languages (DSLs) as a solution to increase the level of abstraction in game development. Some of the initial attempts in the field of DSLs were the incorporation of game-specific abstractions into host languages, allowing the description of behavior on multiple abstraction levels, yet using the existing execution environment (Paper 10). This was followed by further research work in which extensible game description languages were developed to make changes in game rules and behavior programmable through these languages without requiring changes in game engines (Paper 13). Although this made game development more flexible, this area of research again emphasized the need for well-defined domain concepts to prevent over-engineering. With further advances in DSL research, more emphasis was given to serious games like educational and games for educational institutions. DSL for modeling adventure and educational games successfully applied domain-specific concepts to define gameplay (Paper 14). Further evolution to the language-driven development demonstrated that the definition of game logic before the engine construction allows achieving better modularity and reusability of the code (Paper 15). However, most DSLs still have steep learning curves or rigid syntax. GS addresses this gap by offering a minimal, rule-oriented DSL that maintains abstraction advantages and stays comprehensible and easy to modify.

2.3 | Game Scripting-Engines and Engine Architecture

Game scripting development also incorporates narrative-oriented and interactive story solutions. Structured game scripts have been found early on in research to facilitate branching scenarios and conversations, making it feasible for any individual to develop interactive storylines without

necessarily being programmatically competent (Paper 4). These hypotheses were later verified by game implementations where the scripts handled conversations, puzzle-solving, and storylines in adventure-type games (Paper 7). With the rising interest in Narrative Automation, researchers did look into AI solutions. Some recent work used Large Language Models to automatically create an interactive story sequence (Paper 21). But this work posed its own problems regarding predictability and execution. Some related work examined properties in game dialogues (Paper 22), while others explored ways to model behavioral data to handle non-linear narratives (Paper 25). Although these approaches make progress for narrative flexibility, they frequently do so at the expense of transparency and simplicity. By contrast, GS focuses on explicit, rule-based scripting of narratives, giving designers more control over narrative logic without resorting to complex AI-intensive systems.

2.4 | Narrative, Dialogue, and Interactive Story Scripting

Game scripting development also incorporates narrative-oriented and interactive story solutions. Structured game scripts have been found early on in research to facilitate branching scenarios and conversations, making it feasible for any individual to develop interactive storylines without necessarily being programmatically competent (Paper 4). These hypotheses were later verified by game implementations where the scripts handled conversations, puzzle-solving, and storylines in adventure-type games (Paper 7). With the rising interest in Narrative Automation, researchers did look into AI solutions. Some recent work used Large Language Models to automatically create an interactive story sequence (Paper 21). But this work posed its own problems regarding predictability and execution. Some related work examined properties in game dialogues (Paper 22), while others explored ways to model behavioral data to handle non-linear narratives (Paper 25). Although these approaches make progress for narrative flexibility, they frequently do so at the expense of transparency and simplicity. By contrast, GS focuses on explicit, rule-based scripting of narratives, giving designers more control over narrative logic without resorting to complex AI-intensive systems.

2.5 | Educational Games and Gamification Systems

The requirement for simple scripting tools is particularly important in education-based as well as gamification-based systems. The initial research proved that knowledge modeling through scripting can be useful in structuring gamebased learning activities in education-based games (Paper 3). Simplified scripting engines with a reduced instruction set were later developed to facilitate scripting activities even for beginners, allowing students to conduct scripting activities in games without having in-depth knowledge in programming (Paper 6). On top of this, the development of authoring systems followed to enable the creation of gamified learning activities by high-level scripting (Paper 8). Research continued with collaborative learning, team evaluation, and games for inquiry-based learning (Papers 24, 26,27, and 30). Although the emphasis of these studies now lies on pedagogical outcome rather than technical design, the significance of understandable representations of game logic remains, thus upholding the importance of GS.

2.6 | Model-Driven and Visual Game Design Methods

Alongside script-based solutions, model-based and graphical abstractions have been investigated to further minimize development time. Domain-specific modeling languages and model-driven development allowed showing the capability to transform high-level models directly into game executables with less reliance on programming (Papers 17 19). Graph DSLs furthered this concept to enable designers to represent game concepts in graphical forms (Paper 20). Even as they provide strong abstraction over low-level details, they can be rather restrictive in terms of expressivity. Such a trade-off between abstraction and expressibility serves to emphasize GS as a middle path that combines scripting expressibility with simplicity.

2.7 | Game Design and Conceptual Planning

Some research concentrates on supporting conceptual and nascent designs as opposed to executable logic. Game sketching assists in quickly evaluating designs (Paper 23), and game design documents aid in maintaining consistency and theme integrity (Paper 29). It can be seen that these contributions are useful for the purpose of planning, but they do not cover the specification and implementation of game behavior. Lastly, research on the use of blockchain and game theory for federated learning incentivization mechanisms (Paper 12) can be considered irrelevant to this project since it neither relates to game scripting nor to methodologies on developing games.

Summary of Related Work

- Previous work involved rule-based and declarative scripting to alleviate procedural complexity while preserving expressive game behavior.
- Domain-specific languages enhanced abstraction in game development but tended to impose inflexible syntax or learning overhead.
- Game scripting engines concentrated on efficiency and modularity, sometimes adding architectural complexity.
- Narrative and educational scripting paradigms prioritized usability but compromised transparency and predictability.
- Model-driven and visual approaches made design simpler but restrictive in expression, thus underlining the need for a lightweight scripting solution.

3 | PROPOSED METHODOLOGY

We introduce a conceptual framework that will make the creation of animations easier and allow performing more granular control over animation parameters. The methodology entails the consumption of the temporal information as one of the essential input parameters in a domain-specific language (DSL) syntax that comprises the full descriptive semantics of animation sequence desired. This DSL input is then manipulated and converted through a methodical conversion process to machine readable and understandable numerical values in accordance with CSS property values. The transformed values are then systematically injected into the CSS rendering pipeline, thus, producing the animation with built-in time control, which in turn is able to provide the ability to time, sequence and synchronise control mechanisms that are part of the animation cycle.

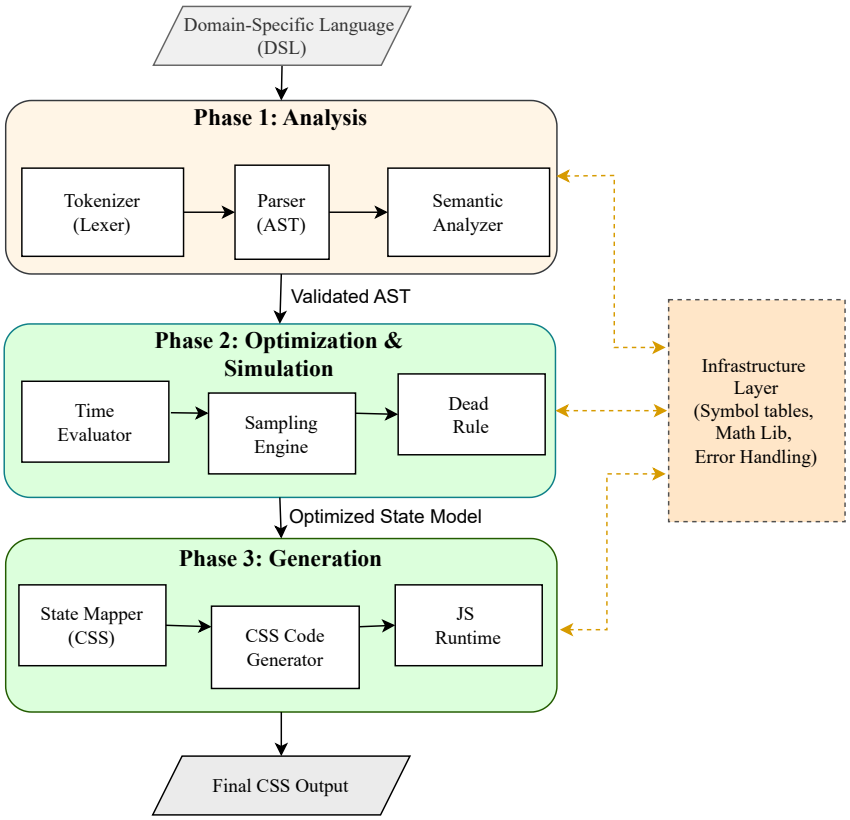

FIGURE 1 Architecture diagram of the GameScript compilation pipeline

3.1 | Temporal Analysis

The Domain-Specific Language (DSL) requires the end-user to define a temporal parameter that establishes the specific timelines in which the action should be executed, and the specific set of operations that should be instantiated at such timelines. It shares the underlying computational logic of a conditional (if-then) construct whereby the active modulation of the animation parameters is the predicated parameter, that is, the parameter of opacity, based on continuous and periodic functions represented as trigonometric sine and cosine functions. The choice of these functions is based on their intrinsic simplicity, periodicity and determinism that, together, allow the production of repeatable and predictable visual effects needed to produce consistent animation effects in graphical user interfaces. Opacity, represented as a time-varying constant, O(t) is represented as a continuous function of time t, and is gradually increasing as the animation advances through the animation phase. All these functions make transitions between visual levels of transparency smooth, and they are the workings of developing the animated element within the temporal domain. Other functions like tangent are not included on purpose because of their predisposition to asymptotic discontinuities and unstable behavior that cannot be trusted to maintain animation fidelity. Even though linear interpolation models may be used, it is possible that the interpolation may create sharp or abrupt transitions hence compromising visual smoothness. Although the sine and cosine functions form the main modeling technique, more complicated parametric curves, including Bezier splines, can be combined in the case they are needed to provide more detailed or stylistically rich animation effects. The temporal profile T (t) obtained through a sine-based calculation provides a parameterizable temporal profile that can be directly inserted into Cascading Style Sheets (CSS) animation keyframes and transition definitions, compatible with web-based rendering engines and making it easy to control the transitions between animated changes in opacity.

3.2 | Implementation Computational Toolkit

The Domain-Specific Language (DSL) requires the end-user to define a temporal parameter that establishes the specific timelines in which the action should be executed, and the specific set of operations that should be instantiated at such timelines. It shares the underlying computational logic of a conditional (if-then) construct whereby the active modulation of the animation parameters is the predicated parameter, that is, the parameter of opacity, based on continuous and periodic functions represented as trigonometric sine and cosine functions. The choice of these functions is based on their intrinsic simplicity, periodicity and determinism that, together, allow the production of repeatable and predictable visual effects needed to produce consistent animation effects in graphical user interfaces. Opacity, represented as a time-varying constant, O(t) is represented as a continuous function of time t, and is gradually increasing as the animation advances through the animation phase. All these functions make transitions between visual levels of transparency smooth, and they are the workings of developing the animated element within the temporal domain. Other functions like tangent are not included on purpose because of their predisposition to asymptotic discontinuities and unstable behavior that cannot be trusted to maintain animation fidelity. Even though linear interpolation models may be used, it is possible that the interpolation may create sharp or abrupt transitions hence compromising visual smoothness. Although the sine and cosine functions form the main modeling technique, more complicated parametric curves, including Bezier splines, can be combined in the case they are needed to provide more detailed or stylistically rich animation effects. The temporal profile T (t) obtained through a sine-based calculation provides a parameterizable temporal profile that can be directly inserted into Cascading Style Sheets (CSS) animation keyframes and transition definitions, compatible with web-based rendering engines and making it easy to control the transitions between animated changes in opacity.

4 | PROCESSING PIPELINE

The architecture of the proposed system is described in this section. It is designed in a classical compiler pipeline framework, except that it is made to accept time-based animation specification in the form of a domain-specific language. The system takes a DSL input and analyzes it at the front-end stage of analysis, optimizes it with time-awareness, and produces animal code. The architecture is divided into three key steps, namely, the front-end processing, time evaluation optimization, and code generation in the back-end. The stages are tasked with the responsibility of gradually converting the input specification into a form that can be executed.

4.1 | Domain-Specific Language

A domain-specific language is used as the input of the proposed system. The DSL is intended to describe animation behavior through time in a explicit manner, which is hard to describe using the standard constructs of CSS animation. The language permits the user to define time intervals,

define state transitions in between time intervals, and states the connections among various animation elements, e.g. sequential or parallel execution. The DSL is declarative and thus the system can reason about animation behavior without having to make reference to the details at the implementation level.

4.2 | DSL Program Analysis Pipeline

The input is processed by lexical analysis, which receives the DSL input as a stream of characters and generates it into a stream of tokens. These tokens are time literals, identifiers, keywords and operators which are language defined. This step makes sure that the input is compliant with the lexical rules of the DSL and indicates errors, e.g. invalid literals or symbols. The result is a stream of tokens utilized by the parser. Syntactic analysis is used to ensure that there is adherence to the grammar of the DSL by the token stream. At this stage, the structural connections among the definitions of animation and time-scoped blocks are determined. The parser will form an abstract syntax tree which will show the hierarchical structure of the input program, and syntactic errors e.g. missing elements or invalid nesting are identified. Semantic analysis verifies logical correctness of the structure that has been parsed. These involve time interval verification, animation property consistency and reference verification of animation elements. Conflicts (e.g. various incompatible transformations being made on the same property in overlapping time ranges) are also identified during the semantic phase, and the output is an annotated representation that represents the structural and temporal information.

4.3 | Time Evaluation and Optimization

Following the semantic validation phase, the program advances to the optimization and time evaluation phase, which explicitly interprets temporal aspects and prepares the represented data for efficient code generation. The time evaluation component scales individual time values and constructs a unified timeline encompassing all animation components, translating declarative time specifications into evaluable representations to ensure consistent interpretation within the system. Subsequently, sampling discretizes continuous time-based behavior, with the sampling engine estimating animation states at fixed intervals to generate discrete states suitable for CSS keyframes, thereby balancing temporal accuracy with operational feasibility. Finally, optimization rules are applied to reduce redundancy and enhance performance by combining compatible animation segments, eliminating unnecessary keyframes, and resolving overlaps according to predefined precedence rules, while preserving the semantics of the DSL and improving execution efficiency.

4.4 | Code Generation

Following the optimization phase, the back-end converts the optimized representation into executable code. In state mapping, abstract animation states are translated into concrete states compatible with CSS, converting high-level animation intent into precise CSS properties and state transitions according to the analyzed time scale. During CSS generation, the system creates CSS code, including keyframe specifications and animation statements, conforming to standard CSS specifications and reflecting the temporal behavior defined in the DSL. Finally, runtime support manages dynamic triggering and synchronization of animations where lightweight runtime is required, complementing the generated CSS without duplicating its functionality.

Summary

- The system follows a compiler-inspired pipeline that processes the DSL input through lexical, syntactic, and semantic analysis, producing an annotated representation of animations.
- Time evaluation and sampling scale and discretize animation timelines, translating continuous behavior into discrete states suitable for CSS keyframes while maintaining temporal accuracy.
- Optimization rules reduce redundancy, resolve overlaps, and combine compatible animation segments, enhancing performance while preserving DSL semantics.
- The code generation and runtime support convert the optimized representation into CSS-compatible states, keyframes, and dynamic animation management, ensuring efficient, accurate, and maintainable execution.