Received: Added at production
Revised: Added at production
Accepted: Added at production
DOI: xxx/xxxx

ARTICLE TYPE

Declarative Time-Based Animation using Domain Specific Language

Srikaanth[1] — Priyadharsan[2,3] — Srinivasan[3]

[1]Department Name, Institution Name, State Name, Country Name
[2]Department Name, Institution Name, State Name, Country Name
[3]Department Name, Institution Name, State Name, Country Name

Correspondence
Corresponding author Mark Taylor, This is sample corresponding address.
Email: authorone@gmail.com

Present address
This is sample for present address text this is sample for present address text.

Abstract

CSS (Cascade Style Sheet) is a Domain Specific Language (DSL) through which diverse styles and animations are implemented, however it fails to evaluate time as a function which is key for animation controls. Inspiration is taken from GameScript by this project, a DSL based on reducing complexity while maintaining minimalism of 2D Video Games by making every production level object a separate entity known as Agent along with the Per-Agent System concept. A progression is proposed where time is assessed as a function and methodically serialized to an appropriate value. The lack of feature of CSS is overcome by Game Script's approach being adopted to solve this problem by encoding a specific DSL which can validate the time functions. During the Time Analysis period the time function is read and methodically serialized to an appropriate value which can be evaluated by CSS, eventually the CSS style sheet is synthesized. This process is executed with the help of compilation as an underlying mechanism such as Lexical, Syntax

and Semantic analysis. Creating complex animations embedded with time constraints is intricate since manually changing multitude keyframes and percentages must be done. In this project the animation is explained at a higher level to the system and it differentiates between the time logic and CSS keyframe generation.

KEYWORDS
Animation - Domain Specific Language (DSL); Animation Behaviour; Temporal Sampling; Keyframe Generation;

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 GameScript 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) system.

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

As high-level scripting and DSLs became commonplace, interest in research output started to shift toward the way such scripts are executed with efficiency within game engines. Performance and memory limits were early points of interest where

lightweight scripting engines, fit for resource-constrained platforms like mobile, were proposed(Paper 2). Such an engine proved that a reduced execution model can drastically minimize the footprint of the resources. Parallel to this, other research focused on more general aspects of the architecture of the engines, focusing on modular design and performance optimization to handle the continuous growth in game logic complexity (Paper 9). Such architectures did improve scalability but at the same time increased system complexity. The long-term maintainability concerns were indeed confirmed in later studies that focused on code quality and sustainability aspects of some of the most popular open-source game engines (Paper 11).In response to these challenges, object-oriented design patterns were used to increase the reusability and flexibility of the game (Paper 28). Yet, this approach tends to move the complexity altogether, making the design understandable only by expert designers. These kinds of advancements imply that simplifying the scripting layer itself, instead of using complex engine architecture, would be more long-term, which GS aims to address.

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 game-based 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.

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.

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 tem-

poral 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