PDFGameBuilder: A Reusable Game Engine for Executable PDF Documents

Wilson Ramos do Carmo Junior
Universidade Estadual de Feira de Santana
Feira de Santana, Brasil
wilson.ramoscj@gmail.com

Victor Travassos Sarinho
Universidade Estadual de Feira de Santana
Feira de Santana, Brasil
vsarinho@uefs.br

Abstract

Digital games are typically distributed as standalone applications or web-based systems, while PDF documents remain largely static despite supporting programmable features defined by the ISO 32000-2:2020 standard. This work presents PDFGameBuilder, a reusable game engine that explores the feasibility of using PDF documents as execution platforms for digital games. The proposed architecture separates document generation, implemented in Python, from runtime execution through embedded JavaScript, providing reusable abstractions for rendering, event handling, asset management, and game loop execution. The engine was validated through proof-of-concept games and evaluated in terms of software reuse, code quality, runtime performance, and cross-platform portability. The results demonstrate that interactive games can be encapsulated within a single PDF document and executed consistently in Chromium-based PDF viewers, establishing PDF documents as reusable, lightweight, and portable platforms for interactive applications.

Keywords

Software reuse, Game engine, Interactive applications, PDF games

1 Introduction

Game engines have become a fundamental artifact for digital game development by providing reusable software infrastructures that abstract low-level concerns such as rendering, input processing, asset management, and game execution. These architectures have significantly accelerated the development of interactive applications while promoting software engineering principles including modularity, reuse, abstraction, and separation of concerns [1, 7].

However, most game engines remain strongly associated with specific deployment platforms, such as desktop operating systems, web browsers, mobile devices, and proprietary distribution ecosystems. As a result, deploying interactive applications frequently depends on software installation, platform-specific runtimes, periodic updates, or network connectivity, increasing the complexity of software distribution and maintenance [2, 11].

In contrast, the Portable Document Format (PDF) has become one of the most widely adopted standards for digital document exchange due to the portability and rendering consistency guaranteed by the ISO 32000-2:2020 specification [8]. Although generally regarded as a static publication format, the standard also defines programmable resources, including embedded JavaScript, interactive forms, multimedia elements, and event-driven actions. These capabilities suggest that PDF documents possess characteristics that extend beyond content presentation, motivating the investigation of whether they can systematically support the execution of interactive applications through appropriate software abstractions.

Recent initiatives have demonstrated the expressive power of programmable PDF documents by implementing applications such as Tetris [12], Doom [4], and Linux terminal simulations [5]. Although these projects successfully demonstrate the computational capabilities of the PDF specification, they remain isolated implementations developed specifically for individual applications. Consequently, they provide limited support for software reuse, architectural modularization, or systematic development of interactive systems.

This limitation exposes an interesting software engineering research opportunity. While reusable architectures have already been proposed for document-oriented interactive applications based on the EPUB3 standard [13, 14], comparable software infrastructures have not yet been investigated for the PDF ecosystem. Existing programmable PDF initiatives have primarily focused on demonstrating the computational expressiveness of the format through handcrafted implementations rather than proposing reusable software architectures [3]. As a result, the systematic encapsulation of the programmable resources defined by the ISO 32000-2:2020 standard into reusable software abstractions for interactive application development remains an open research challenge.

To address this gap, this work proposes **PDFGameBuilder**, a reusable software architecture for interactive PDF applications designed to investigate the use of PDF documents as execution environments. The architecture was designed using a three-layered approach that decouples Python-based document generation, embedded JavaScript runtime execution, and application-specific logic into modular components. The framework was validated through two proof-of-concept games, software metrics, and runtime experiments, demonstrating high component reuse, low code complexity, acceptable performance, and correct execution in Chromium-based environments.

2 Background

The design of reusable software infrastructures has played a fundamental role in the evolution of digital game development. Meanwhile, advances in document standards have progressively incorporated programmable resources capable of supporting interactive behaviors beyond static content presentation. This convergence motivates the investigation of document-oriented execution environments, where software engineering principles traditionally adopted in game engines can be applied to programmable documents. This section reviews the concepts underlying this research, including reusable game engine architectures, programmable document formats, and previous efforts toward document-oriented game engines.

2.1 Game Engines as Reusable Software Platforms

Game engines are software frameworks designed to encapsulate functionalities commonly required by multiple games, allowing developers to concentrate on gameplay mechanics and application-specific content rather than low-level implementation details. Typical engine responsibilities include rendering, input processing, asset management, audio control, collision detection, physics simulation, and runtime execution [7].

From a software engineering perspective, game engines promote architectural principles such as modularity, abstraction, reuse, and separation of concerns by isolating reusable infrastructure from game-specific logic [1]. This organization improves maintainability, facilitates software evolution, and reduces development effort through the reuse of common services across different projects.

Originally conceived for entertainment software, modern game engines have evolved into general-purpose development platforms supporting educational applications, simulations, serious games, and interactive storytelling. Consequently, the notion of a game engine extends beyond graphics rendering, representing a reusable software architecture capable of supporting the systematic development of interactive applications [3, 9, 10].

2.2 Programmable Document Formats

Digital documents have traditionally been employed as media for storing and distributing information. However, contemporary document standards increasingly incorporate programmable resources that extend static content with interactive behaviors.

Among these standards, EPUB3 adopts open Web technologies such as HTML5, CSS, and JavaScript, allowing interactive applications to be embedded directly into digital publications. Similarly, the PDF standard, specified by ISO 32000-2:2020, provides support for embedded JavaScript, interactive forms, multimedia resources, annotations, and event-driven actions [8]. Unlike standard web-based HTML5 applications, a PDF provides a self-contained single-file format, offline availability without external dependencies, compliance with the ISO standard, safe fallback to static content when scripts are disabled, and native compatibility with email attachments and digital document repositories. These capabilities considerably expand the expressive power of digital documents, enabling them to react to user interactions while preserving the portability and self-contained nature that have historically made document formats attractive for information distribution.

In practice, the exploration of these programmable resources within the ISO 32000-2:2020 standard has historically been driven by hardcoded, application-specific approaches. These pioneering initiatives focus on testing the format's computational expressiveness by embedding complex logic tailored exclusively for individual scenarios. From a software engineering perspective, however, such implementations operate as monolithic, closed systems where rendering routines and state management are inseparable from the content itself. This lack of a standardized abstraction layer results in high development overhead and prevents code reusability, reinforcing the need to investigate architectures that extend component-based concepts and modularity to the portable document ecosystem.

Another important characteristic of programmable document formats concerns their execution security model. Since embedded scripts are executed within document viewers rather than conventional operating systems, modern platforms employ sandboxing mechanisms to isolate document code from the host environment. In the PDF ecosystem, JavaScript execution is intentionally restricted to a controlled subset of APIs, preventing unrestricted access to local files, operating system resources, or arbitrary network communication [15]. Furthermore, JavaScript execution remains strictly constrained within the viewer's sandbox; if scripts are stripped or blocked by security filters such as email gateways, the PDF safely degrades to a readable static document. Such restrictions reduce security risks while preserving sufficient programmability for interactive behaviors, making the design of reusable software infrastructures dependent on the capabilities exposed by the execution environment rather than the underlying operating system.

2.3 Document-Oriented Game Engines

The possibility of combining programmable document formats with reusable software architectures has motivated recent investigations into document-oriented game engines. Rather than treating documents merely as content containers, these approaches consider them execution environments capable of hosting complete interactive experiences.

Within the EPUB3 ecosystem, the GEnEbook project demonstrated the feasibility of integrating reusable game components into digital books through modular software abstractions [13]. Building upon this concept, EPUBForge proposed a reusable game engine responsible for encapsulating rendering, interaction handling, multimedia resources, and game execution into reusable software modules, allowing developers to create interactive EPUB3 applications without directly manipulating the underlying document technologies [14].

These studies demonstrate that software engineering principles traditionally employed in game engines can also be successfully applied to programmable document formats. However, EPUB3 and PDF adopt substantially different execution models. While EPUB3 is inherently based on Web technologies and browser execution, PDF relies on a document-oriented object model defined by the ISO 32000-2:2020 specification, providing distinct mechanisms for event handling, resource management, rendering, and script execution.

Consequently, reusable architectures designed for EPUB3 cannot be directly transferred to PDF documents. This observation motivates the present work, which investigates how similar software engineering principles can be adapted to the PDF ecosystem through a reusable game engine, which is capable of abstracting the complexity of programmable PDF documents while preserving the portability and self-contained characteristics of the format.

3 Methodology

This study conducts an exploratory research into the interactive boundaries of the Portable Document Format (PDF) ecosystem, operationalized through a methodological artifact built upon Design Science Research (DSR), which focuses on the design and evaluation of software artifacts to systematically solve identified problems. The methodological process consisted of five stages: (i) a structured

literature search to identify existing research on executable PDF applications and reusable game engines; (ii) analysis of the PDF execution model defined by the ISO 32000-2:2020 specification; (iii) design of the PDFGameBuilder architecture; (iv) implementation of the proposed engine and supporting tools; and (v) validation through the development of executable games embedded in a PDF document.

3.1 Structured Literature Search

The first stage of the proposed methodology consisted of a structured literature search to identify existing research on reusable game engines for programmable document formats, particularly PDF. This investigation aimed to determine whether software infrastructures capable of supporting the systematic development of executable PDF games had already been proposed, thereby providing the theoretical basis for the subsequent artifact design stage.

The search was conducted using Google Scholar, ACM Digital Library, and IEEE Xplore, which collectively index a substantial portion of the Computer Science literature. To maximize the retrieval of potentially relevant studies, the search strategy encompassed iterative queries using multiple combinations of keywords, which were dynamically adapted based on the specific search engines. These targeted combinations were applied across different metadata fields, focusing alternately on titles, abstracts, or both, depending on the platform's filtering capabilities. Listing 1 shows a representative example of this search process, illustrating the primary logical grouping of the descriptors used.

```
(("interactive document" OR "interactive pdf" OR
    "programmable document" OR "programmable pdf"
    OR "executable document" OR "executable pdf")
    AND ("game engine")) OR "pdf game engine" OR
    "document game engine"
```

Listing 1: Search string used in the literature search.

The search across the three digital libraries returned a scarce volume of literature addressing interactivity in digital documents, with the few identified studies restricted to the EPUB format, as discussed in Section 2. Notably, the absence of academic publications targeting reusable software infrastructures or game development within the PDF ecosystem highlights a gap in the scientific literature. The existing programmable PDF initiatives are limited to informal, isolated implementations hosted on open-source repositories and media platforms, such as GitHub and YouTube. Consequently, while these community-driven projects demonstrate the feasibility of embedding interactive code in PDF files, they lack the architectural modularity, software engineering frameworks, or systematic reuse models proposed in this work.

3.2 PDF Execution Model

The Portable Document Format (PDF), standardized by ISO 32000-2:2020, adopts a declarative structure in which every document element is represented by interconnected objects and dictionaries. Unlike conventional applications, whose logic is explicitly defined by source code, PDF documents describe both content and interactive behavior through structured objects interpreted by the document viewer.

Internally, a PDF document consists of numbered indirect objects connected through hierarchical references. The document organization begins at the root dictionary (`/Root`), which references the document catalog (`/Catalog`). The catalog provides access to the page tree (`/Pages`), whose page objects (`/Page`) define graphical contents (`/Contents`), rendering resources (`/Resources`), and interactive annotations (`/Annots`). Listing 2 presents a simplified view of this internal organization.

```
1 0 obj << /Type /Catalog /Pages 2 0 R >>
endobj
2 0 obj << /Type /Pages /Kids [3 0 R] >>
endobj
3 0 obj << /Type /Page /Contents ...
   /Resources ... /Annots [4 0 R] >>
endobj
4 0 obj << /Type /Annot /Subtype /Widget >>
endobj
xref ...
trailer << /Root 1 0 R >>
%%EOF
```

Listing 2: PDF document object hierarchy.

Interactivity is implemented through an event-driven model based on Action Dictionaries (`/A`) and Additional Action Dictionaries (`/AA`), which associate predefined events with executable actions. Among the supported action types, embedded JavaScript enables the implementation of application logic triggered by events such as opening the document, loading pages, or interacting with buttons and form fields.

Listing 3 illustrates a simplified example in which JavaScript is associated with the document opening event through the `/AA` dictionary. The field `/S` specifies the action subtype as `/JavaScript`, while `/JS` stores the source code executed by the viewer. Similar structures are used throughout the document to associate interactive widgets with engine functions.

```
/AA <<
   /O << /S /JavaScript /JS (initializeGame();) >>
>>
```

Listing 3: Embedding JavaScript into a PDF Action Dictionary.

Although this programming model provides considerable flexibility, implementing interactive applications requires coordinated manipulation of multiple PDF objects, dictionaries, and event bindings while preserving document consistency. Moreover, embedded scripts are executed within the sandboxed environment provided by modern PDF viewers, restricting access to operating system resources and exposing only a limited set of APIs.

These characteristics significantly increase the complexity of developing executable PDF applications. Consequently, PDFGameBuilder encapsulates these low-level mechanisms into reusable software components, allowing developers to implement game logic without directly manipulating the internal PDF structure.

3.3 PDFGameBuilder Architecture

To address the complexity of direct PDF manipulation, PDFGameBuilder proposes a layered architecture that separates game logic

from document construction and execution details. The architecture is composed of three main layers: the game layer, the engine layer, and the document generation layer.

The *game layer* contains application-specific logic, including game rules, assets, and narrative or interactive structure (Figure 1). This layer is designed to be independent of the underlying PDF implementation, enabling reuse across different games. Within this scope, game elements are declared through high-level abstractions that encapsulate visual and spatial properties without exposing internal document mechanics. Listing 4 demonstrates the instantiation and spatial initialization of the ball element controlled by the game via update method in a pong game.

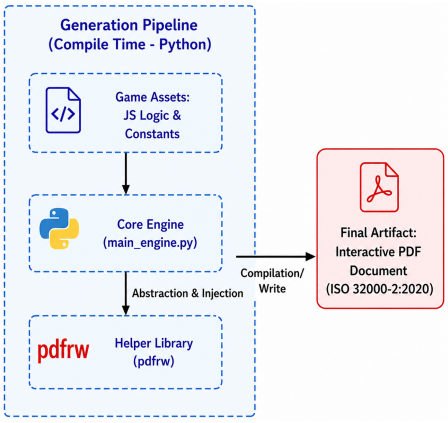

Figure 1: Game and document layer interaction.

```
# Creating the Ball character at game_demo.py file
ball = create_widget(name='ball',
    x=(CANVAS_WIDTH - BAR_WIDTH)/2,
        y=CANVAS_BASE+30,
    width=BALL_WIDTH, height=BALL_HEIGHT,
    r=0.8, g=0, b=0.8)
fields.append(ball)
-----------------------------------------------------
# Using gameBall and update() at game_demo.js file
var gameBall = new GameObject('ball',0,0,
        BALL_WIDTH, BALL_HEIGHT);
...
function update() {
    if (!newGameActive) return;
    checkCollision();
    gameBall.x += gameBall.dirX * speed;
    gameBall.y += gameBall.dirY * speed;
}
```

Listing 4: Initialization of a game object widget.

The *document generation layer* is responsible for transforming the abstract game representation into a valid executable PDF file (Figure 1). This process includes the creation of the PDF object structure, the definition of interactive annotations embedded with JavaScript runtime functionalities, and the insertion of graphical resources. This layer is implemented using Python and the `pdfrw` library, which allows programmatic manipulation of PDF elements.

The creation of interactive elements capable of representing the PDF object structure is centralized in the method `create_widget()` within the `main_engine.py` module. This function abstracts the complexity of `/Annot` and `/Widget` dictionaries, allowing the developer to instantiate buttons, text areas, or input sensors through a unified interface. Mechanically, this method builds the foundational layout by defining spatial coordinates (`Rect`) and background rendering properties (`MK`). As demonstrated in the simplified implementation in Listing 5, internal conditional blocks then determine whether the element behaves as an interactive text field (`Tx`) or as a functional button (`Btn`) embedded with JavaScript actions.

```
def create_widget(name, x, y, width, height, r=1,
    g=1, b=1, opaque=None, field_type="text",
    **kwargs):

  # Base PDF /Annot and /Widget layout
  widget = PdfDict(Type=PdfName.Annot,
      Subtype=PdfName.Widget,
      T=PdfString.encode(name),
    Rect=PdfArray([x,y,x+width,y+height]),
    MK=PdfDict(BG=PdfArray([r, g, b])
    if opaque else PdfArray([])))
  # TEXT
  if field_type == "text":
    widget.FT, widget.V = PdfName.Tx,
        PdfString.encode(kwargs.get("value", ""))
    # ... omitted #
    if kwargs.get("on_key_stroke"):
      widget.AA = # ... omitted #
  # BUTTON
  elif field_type == "button":
    widget.FT, widget.Ff = PdfName.Btn
    # ... omitted #
    if kwargs.get("js_action"):
      widget.A = create_js_action(...))
  return widget
```

Listing 5: Interactive field instantiation (simplified).

To integrate behavioral logic with the PDF structure without code redundancy, the engine centralizes script binding within the `create_js_action()` method (Listing 6). This generic method encapsulates raw JavaScript strings into native PDF action dictionaries compliant with the ISO 32000-2:2020, specifically populating the required `/S` (Action Type) and `/JS` (JavaScript Stream) keys.

```
# Encapsulates JS into PDF actions
def create_js_action(js_code):
    return PdfDict(S=PdfName.JavaScript,
        JS=PdfString.encode(js_code))
```

Listing 6: Unified method for PDF JavaScript actions.

Regarding the insertion of graphical resources, the engine utilizes a configuration dictionary (`config`) to manage the game's visual identity. The `insert_image()` method (Listing 7) facilitates this injection through a two-stage overlay technique. First, it generates a temporary PDF canvas (*pdf_temp*) to convert raw images (PNG/JPG) into a compatible PDF object. Subsequently, it utilizes the `PageMerge` utility to dynamically fuse this rendered asset into

the base document's page structure at defined coordinates. This approach ensures visual fidelity while maintaining the integrity of the existing PDF document tree.

```python
def insert_image(input_pdf, image_path, width,
    height, x=0, y=0):
    # Generates temporary canvas
    pdf_temp = "_temp_imagem.pdf"
    c = canvas.Canvas(pdf_temp, pagesize=(...))
    c.drawImage(image_path, x, y,...)
    c.save()
    # Merges image asset
    base = PdfReader(input_pdf)
    imagem = PdfReader(pdf_temp)
    PageMerge(...).add(...).render()
    # Cleans temporary files
    PdfWriter().write(input_pdf, base)
    if os.path.exists(pdf_temp): os.remove(...)
```

Listing 7: Image overlay function (simplified).

To establish a highly responsive, event-driven mechanism, the *engine layer* maps user interactions directly onto structural PDF components (Figure 2). Keyboard inputs are captured by routing keystroke actions through hidden text fields (`/Tx`), mouse clicks are processed via immediate action triggers bound to button annotations (`/Btn`), and coordinate tracking is achieved by intercepting cursor entry and exit events over mouse tracker fields, thereby decoupling player input from the underlying system. This interactive layer feeds into a reusable runtime infrastructure executed within the PDF environment through embedded JavaScript.

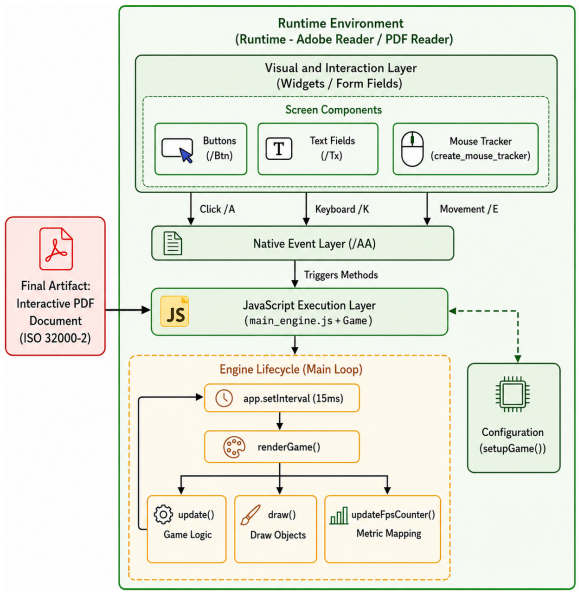

Figure 2: Runtime game loop architecture.

In terms of lifecycle and execution management, JavaScript execution is orchestrated by the `main_engine.js` centralized loop. Upon loading the document, the global function `initialize()` (Listing 8) verifies the system state through the `global.initialized` variable to ensure that the initialization process is executed only once. After completing the game configuration, the architecture starts a continuous execution loop managed by the native timer `app.setInterval`, which invokes the central `renderGame()` routine every 15 milliseconds. During each iteration, `renderGame()` coordinates the game lifecycle by sequentially calling the `draw()` function to render the objects on the screen, followed by the `update()` function to process game logic and state transitions.

```javascript
function initialize() {
    if (global.initialized) return;
    global.initialized = true;
    // Executes the setup of the loaded game
    if (typeof setupGame === "function")
        setupGame();
    // Activates the global rendering loop
    global.gameLoop =
        app.setInterval('renderGame()', 15);
}
```

Listing 8: Centralized initialization in `main_engine.js`.

4 Obtained Results

The functional validation of the proposed architecture was performed through the development of two proof-of-concept games (Figure 3) implementing distinct interaction paradigms and yielding lightweight document file sizes. The first is a *Pong-inspired* game (2,282 KB) controlled by continuous mouse interaction, in which the player moves a paddle to keep the block in play. The second is an *Apple Collector* game (719 KB) based on keyboard interaction, requiring the player to move a character to collect apples.

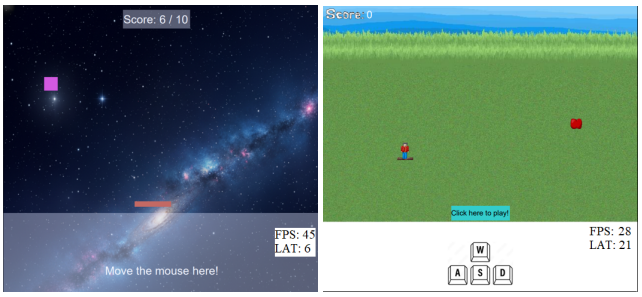

Figure 3: Pong-style and Apple-catching PDF games with embedded indicators highlighting the Frames Per Second (FPS) and processing latency (LAT) sustained during execution.

Although the games employ different gameplay mechanics, both reuse the same engine services responsible for document initialization, rendering, event handling, collision detection, state management, and execution control. Consequently, only the game-specific rules, assets, and update logic required implementation by the developer. The successful execution of both prototypes demonstrates that PDFGameBuilder supports real-time rendering, keyboard and mouse input processing, collision detection, object animation, score management, scene transitions, and dynamic manipulation of graphical resources directly inside executable PDF documents. These results suggest that PDFGameBuilder successfully

encapsulates low-level PDF mechanisms into reusable components, allowing developers to implement 2D interaction logic through a structured, predictable, and maintainable programming model for interactive PDF documents.

Beyond functional validation, the software quality of the proposed architecture was evaluated through static analysis using the Radon tool[1]. Table 1 summarizes the Cyclomatic Complexity (CC), Maintainability Index (MI), Halstead Difficulty, and estimated bugs obtained for the main PDFGameBuilder Python modules.

Table 1: Python code complexity and quality metrics.

Module	Cycl.	MI	Diff.	Bugs
main.py	A (2)	A	1.00	0.005
core/config.py	A (2)	A	1.76	0.043
core/generator.py	A (4)	A	0.66	0.004
core/main_engine.py	C (14)	A	5.30	0.086
games/game_allegro.py	A (4)	A	3.00	0.086
games/game_demo.py	A (4)	A	2.43	0.066

The results indicate low structural complexity across most modules, with the majority of components classified as Complexity A. The only module exhibiting higher complexity is the engine core (`core/main_engine.py`), reaching Complexity C (14) within its interactive field instantiation method, where the reusable execution services are intentionally concentrated. Nevertheless, this module achieved the highest rating in the Maintainability Index (Rank A), indicating that the updated architecture preserves high readability and modularity despite centralizing the heavy lifting of the engine infrastructure. Similarly, the Halstead difficulty of 5.30 for the engine core represents a low semantic effort required to read or modify the code, which directly implies a highly reduced probability of introducing errors during future software maintenance.

To evaluate the effectiveness of the reusable abstractions, measurements focused on JavaScript code components were performed using the *ES6-Plato Code Analyzer*[2]. Table 2 summarizes the Source Lines of Code (SLOC) and Cyclomatic Complexity (CC) generated for the core runtime JavaScript components.

Table 2: JavaScript runtime obtained metrics (ES6-Plato).

Component File	SLOC	CC
main_engine.js	114	11
game_demo.js	135	14
game_allegro.js	68	7

To assess the degree of reuse achieved by the developed games, the *reuse leverage* metric (R_{lev}) [6] was adopted. This metric is defined as the ratio between reused objects and the total number of objects built for a system, expressed as $R_{lev} = OBJ_{reused}/OBJ_{built}$. Since the available measurements are based on SLOC, the metric is operationalized in this study using the number of reused engine lines relative to the total number of lines comprising each game (Table 3).

Table 3: Reuse leverage metrics obtained for the PDF games.

Game	R_{lev} (SLOC)
Pong-inspired	$114/(114+135) = 45.78\%$
Apple Collector	$114/(114+68) = 62.63\%$

The results demonstrate a substantial degree of reuse across the developed applications. The reusable engine accounts for 45.78% of the implementation of the *Pong-inspired* game (*game_demo.js*) and 62.63% of the *Apple Collector* game (*game_allegro.js*), indicating a higher degree of reuse in the latter.

To establish a direct baseline for comparison, the *Apple Collector* game originally developed by the Allegro.js library was identically replicated using PDFGameBuilder. This comparison assesses whether PDFGameBuilder affects software complexity or preserves the simplicity of the original programming model. Table 4 summarizes the obtained software metrics.

Table 4: Software metrics comparison for the *Apple Collector* game according to respective game engines.

Metric	PDFGameBuilder	Allegro.js
Code Volume (SLOC)	68	78
Cyclomatic Complexity (CC)	7	6
Maintainability Index	76.98	70.36
Estimated Errors (Halstead)	0.47	0.37

The PDFGameBuilder implementation provides empirical evidence that it is possible to support the PDF execution environment without substantially increasing the amount of game-specific code when compared to Allegro.js (68 vs. 78 SLOC). Although the framework introduces a slight increase in Cyclomatic Complexity (7 vs. 6) and estimated Halstead errors (0.47 vs. 0.37), these differences reflect the architectural abstractions required to manage PDF events. Even so, PDFGameBuilder achieved a higher Maintainability Index (76.98 vs. 70.36), indicating that this abstraction does not compromise code maintainability. Overall, the results demonstrate that executable PDF applications can achieve software quality comparable to traditional web-based game engines.

Complementing the static quality analysis, the framework's runtime performance was evaluated through a built-in routine designed to capture Frames Per Second (FPS), to measure visual fluidity and frame rate stability, and processing latency (LAT) in milliseconds, to determine the time delay between user inputs and interface updates. The experiments were conducted on a workstation equipped with an Intel Core i3-7100 CPU (3.9 GHz), 16 GB DDR4 RAM, and an NVIDIA GeForce GTX 1050 Ti (4 GB) GPU, running Windows 10 Pro (version 22H2). Performance was evaluated across multiple PDF rendering environments, including Google Chrome (v147.0.7727.103), Opera (v130.0.5847.58), Brave (v148.0.7778.179), and Vivaldi (v8.0.4033.26). Each game prototype underwent 10 experimental runs per viewer, with each trial lasting approximately

[1]Radon is a Python tool that computes various metrics from the source code, including Cyclomatic Complexity and Halstead metrics, was performed. Available at: https://radon.readthedocs.io/

[2]ES6-Plato is a visualizer for JavaScript source code complexity analysis, specialized in ES6+ syntax. Available at: https://github.com/the-simian/es6-plato

Table 5: Compatibility across PDF viewers.

Feature	Chromium*	Edge (Win)	Adobe
Background rendering	✓	✓	✓
Widget rendering	✓	×	×
Game loop execution	✓	×	×
Keyboard/Mouse input	✓	×	×
Dynamic widgets	✓	×	×

*Chrome, Opera, Brave and Vivaldi on Windows/Linux/macOS. Microsoft Edge on Linux/macOS.

15 seconds to measure performance. During the experiments, the *Pong-inspired* game, featuring mouse-driven mechanics, maintained between 44 and 53 FPS with latency ranging from 4 to 7 ms. In contrast, the *Apple Collector* game achieved an average of 24 FPS with latency varying between 21 and 36 ms. This lower frame rate occurs because rendering image files introduces additional processing overhead compared to the lightweight, solid-color shapes used in the Pong-inspired game. Nevertheless, the framework maintained execution rates suitable for casual document-based interactions across different platforms.

Finally, since PDFGameBuilder relies on the JavaScript interpreter provided by the PDF viewer, portability was evaluated across different operating systems and document readers. Table 5 summarizes compatibility for rendering, widget manipulation, game loop execution, input processing, and dynamic interface updates. All tested Chromium-based[3] viewers fully supported these functionalities except Microsoft Edge on Windows. Google Chrome, Opera, Brave, Vivaldi, and Microsoft Edge on Linux and macOS successfully executed the proposed architecture, confirming their suitability for executable PDF games. The only incompatibility among Chromium-based browsers occurred in Microsoft Edge on Windows, where an internal JavaScript object responsible for widget rendering was not properly instantiated, preventing some dynamic engine services from executing. Adobe Acrobat Reader rendered static document content successfully but could not execute the complete runtime due to restrictions in its JavaScript implementation. Therefore, the compatibility limitations (Table 5) are attributable to differences among PDF viewers rather than to the proposed engine.

5 Conclusion and Future Work

This work presented PDFGameBuilder, a reusable software architecture for interactive PDF applications that enables games to be developed as executable PDF documents. Rather than simply demonstrating PDF programmability, the proposed approach formalizes a modular structure that separates document generation, runtime execution, and game-specific logic, allowing developers to create interactive experiences without directly manipulating the internal structures of the PDF specification.

The experimental evaluation demonstrated the feasibility of the proposed architecture from multiple perspectives. Functional validation showed that different game genres can be implemented using the same runtime infrastructure. Static analysis indicated low software complexity and high maintainability of the engine modules, while reuse metrics confirmed that PDF-specific mechanisms are encapsulated within a compact core. Comparison with an equivalent Allegro.js implementation further showed that the abstraction layer preserves software quality while promoting code reuse. Finally, compatibility experiments demonstrated that executable PDF games can be deployed across modern Chromium-based viewers, confirming the practicality of the proposed execution model. Overall, these results indicate that PDFGameBuilder extends PDF from a static document format to a portable execution environment for interactive software applications.

Despite these contributions, the proposed approach presents inherent limitations associated with the PDF execution environment. Since PDFGameBuilder relies on the JavaScript interpreter provided by PDF viewers, portability depends on each viewer's capabilities. Differences among viewers may prevent the execution of interactive widgets or runtime services, limiting cross-platform compatibility. Moreover, the PDF specification lacks native continuous keyboard and mouse listeners, requiring the engine to emulate real-time interaction through event bindings and timer-based execution. Multimedia support also remains limited, particularly for audio playback, whose availability varies across PDF viewers and may be restricted for security reasons. Consequently, the current version is better suited to games based primarily on graphics, animations, and user input than to rich audiovisual applications. Finally, separating document generation in Python from runtime execution in embedded JavaScript introduces additional engineering complexity, requiring synchronization between generated PDF resources and execution logic.

Future work includes extending the engine with reusable components, improving compatibility across PDF viewers (e.g. mobile devices), and investigating alternative multimedia mechanisms (e.g. audio playback and richer interaction models). We also plan to conduct empirical evaluations with software developers, perform scalability stress tests with larger and more complex games to determine capacity limits for simultaneously rendered interactive objects, explore gray literature for additional references, and benchmark PDFGameBuilder implementations against development without a game engine. Finally, expanding the engine to additional game genres and other document-oriented interactive applications may further demonstrate the generality of the proposed architecture.

Artifact Availability

The complete source code of the experimental game engine and the developed prototypes are available at https://github.com/wilsonramosnanalista/research-project/tree/master as research artifacts to support reproducibility. The repository includes the full engine implementation, source code for the validation games, detailed usage instructions, and a demonstration video showcasing real-time gameplay and dynamic behavior within the PDF environment.

Acknowledgments

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001. The authors would also like to thank the Universidade Estadual de Feira de Santana (UEFS) for supporting this research.

[3]Chromium is an open-source web browser project whose source code provides the foundational engine for platforms such as Google Chrome, Microsoft Edge and Opera .

References

[1] Len Bass. 2012. *Software architecture in practice.* Pearson Education India.

[2] Eric Blancaflor and Jan Martin Gomez San Miguel. 2022. Analyzing Digital Game Distribution in Gaming Industry: A Case Study. In *Proceedings of the 2nd Indian International Conference on Industrial Engineering and Operations Management.* IEOM Society International, 674–681. doi:10.46254/IN02.20220232

[3] M. Brown. 2018. The Impact of Game Development Tools on Game Quality. *International Journal of Game Development* 5, 1 (2018).

[4] Allan Ding. 2025. DoomPDF. https://github.com/ading2210/doompdf.

[5] Allan Ding. 2025. LinuxPDF. https://github.com/ading2210/linuxpdf.

[6] Nasib S Gill. 2003. Reusability issues in component-based development. *ACM SIGSOFT Software Engineering Notes* 28, 4 (2003), 1–5.

[7] Jason Gregory. 2018. *Game Engine Architecture* (3rd ed.). CRC Press.

[8] International Organization for Standardization. 2020. ISO 32000-2:2020 - Document management — Portable document format — Part 2: PDF 2.0.

[9] David Kushner. 2003. *Masters of Doom.* Random House.

[10] M. Li, X. Feng, Z. Wu, J. Bai, and F. Yang. 2025. Game engine-driven synthetic point cloud generation method for LiDAR-based defect detection in sewers. *Tunnelling and Underground Space Technology* (2025).

[11] Tanushree Mukherjee. 2023. Videogame Distribution and Steam's Imperialist Practices. *Journal of Games Criticism* 5, A (2023). https://gamescriticism.org/wp-content/uploads/2024/08/mukherjee-5-a.pdf

[12] Thomas Rinsma. 2025. Tetris in a PDF. https://th0mas.nl/2025/01/12/tetris-in-a-pdf.

[13] Victor Travassos Sarinho. 2021. GEnEbook: A Game Engine to Provide Electronic Gamebooks for Adventure Games. In *XX Simpósio Brasileiro de Jogos e Entretenimento Digital (SBGames).* Sociedade Brasileira de Computação, Porto Alegre, RS, Brasil.

[14] Victor Travassos Sarinho. 2024. EPUBForge - A Simplified Game Engine Based on Features for the Production of Ebook Games. In *XXIII Simpósio Brasileiro de Jogos e Entretenimento Digital (SBGames) - Anais Estendidos.* Sociedade Brasileira de Computação, Porto Alegre, RS, Brasil. doi:10.5753/sbgames_estendido.2024.240421

[15] The Chromium Authors. 2025. Chromium Sandbox. https://chromium.googlesource.com/chromium/src/+/main/docs/design/sandbox.md Accessed: 2026-07-04.