Advanced Game Engine Wizardry for Visual Programming Environments

Elisabeth Kletsko
University of Amsterdam
Amsterdam, The Netherlands
elisabeth.kletsko@student.uva.nl

Riemer van Rozen
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
rozen@cwi.nl

Abstract

Visual programming environments empower end-users with interactive input and feedback mechanisms that support live and exploratory programming. For creating these environments, language engineers require enabling technology. Language workbenches and meta-programming languages support rapid construction of interpreter back-ends. Game engines are specifically created for rich interactive user experiences and have the potential to augment this technology even further with maintainable front-ends. However, these technologies presently live in separate technological spaces.

We aim to automate the creation of visual programming environments by integrating the two. We propose Raven, a meta-framework for rapidly prototyping visual editors that exposes key 2D features of Godot in a programmable user interface. Using Raven's declarative notation, language engineers can concisely express the structure and styling of tree-based editors. We demonstrate our approach by: 1) integrating Raven into the Rascal language workbench, and 2) creating two editors for the live state machine language.

CCS Concepts: • **Software and its engineering → Visual languages**; **Domain specific languages**; **Integrated and visual development environments**.

Keywords: programming environments, game engines, language workbenches, domain-specific languages, live programming

ACM Reference Format:
Elisabeth Kletsko and Riemer van Rozen. 2024. Advanced Game Engine Wizardry for Visual Programming Environments. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments (PAINT '24), October 22, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3689488.3689992

PAINT '24, October 22, 2024, Pasadena, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1212-8/24/10
https://doi.org/10.1145/3689488.3689992

1 Introduction

Visual programming environments have the potential to make programming more accessible to programmers of all backgrounds and skill levels. Domain-Specific Languages (DSLs) empower non-programmers with notations and abstractions aimed at particular problem domains [25]. DSLs have been shown to help non-programmers raise their productivity and improve the code quality, e.g., in robotics [10], banking [20], and digital forensics [23]. We aim to support the development of visual programming environments for DSLs. Specifically, we study how to create interactive input and feedback mechanisms that bring the code to life.

Language workbenches and meta-programming languages provide techniques and approaches that support prototyping DSLs, e.g., compilers and interpreters [5]. However, generic language technology often has limited support for creating advanced visual programming environments. As a result, creating user-friendly and aesthetically pleasing prototypes is costly and time-consuming. Language engineers need tools and techniques that accelerate the development of prototypes that are easy to deploy, maintain, and extend.

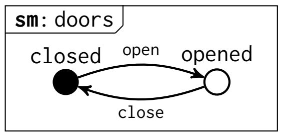
Game engines have been specifically designed to create rich interactive user experiences, e.g., games, apps, and other visualizations. Using powerful tool sets, developers can easily maintain and deploy user-friendly, feature-rich visual applications across platforms. Unfortunately, language workbenches and game engines are currently separate technological spaces. Until now, the combination of this technology in generic solutions for language engineering has not yet been explored. As a result, the potential of game engines for prototyping visual editors for DSLs is still largely unknown.

We aim to bridge the gap between these technologies by automating the creation of visual programming environments using game engines. A pilot study on this topic shows that Godot in particular offers a solid foundation [2, 27]. Godot comes with out-of-the-box support for C, C++, and C#, and recently JVM-based languages such as Kotlin or Java [3]. We investigate how Godot can be leveraged in generic language technology for prototyping of visual editors for DSLs.

We propose Raven, a novel meta-framework that exposes key 2D functionalities of Godot in a reusable, programmable user interface. We illustrate its potential in two ways. First, we integrate Raven into the Rascal language workbench [13]. Next, we create a programming environment for the Live

```
machine doors
  state closed
    open => opened
  state opened
    close => closed
```

(a) Textual doors program



(b) Visual doors program

Figure 1. LiveSML Example (appears in van Rozen [26])

```
MachCreate(1);
MachSetName(1, "doors");
StateCreate(2, 1);
StateSetName(2, "closed");
StateCreate(3, 1);
StateSetName(3, "opened");
TransCreate(4, 2, 3);
TransSetTrigger(4, "open");
TransCreate(5, 3, 2);
TransSetTrigger(5, "close");
```

(a) Creating the program

```
MachInstCreate(5, 1);
MachInstTrigger(5, "open");
```

(b) Running the program

```
MachInstPrint(5);
machine doors
  [close]
  state closed: 1
  state opened: 1 (*)
```

(c) Inspecting the program

Figure 2. REPL commands for syntax edits and user actions

State Machine Language, and we measure its performance using a simple programming scenario. Our initial results show promise. Using Raven, language engineers can concisely express the structure and styling of fast and aesthetically pleasing visual editors for DSLs. In future work, we will further extend and validate Raven, e.g., with graph editors. This paper contributes: 1) an initial prototype of Raven, a meta-framework and rapid prototyping of visual environments for DSLs; and 2) Raven-Rascal, an extension to Rascal.

2 Background

We study how to develop DSLs that support live and exploratory programming [19, 21]. In particular, we aim to develop generic language technology for rapidly prototyping visual programming environments. We will investigate how game engines can be leveraged. First, we relate the needs of programmers to challenges and research opportunities.

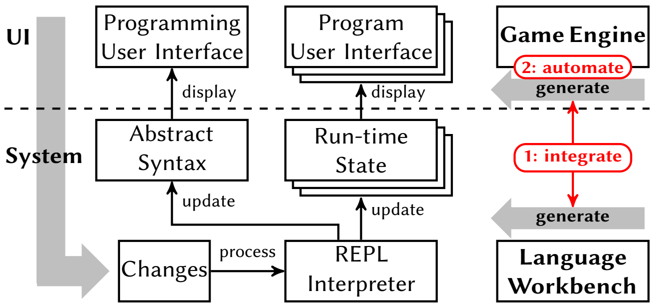
2.1 Live State Machine Language

The Live State Machine Language (LiveSML) is a DSL for simultaneously creating and running state machines [4, 26]. We describe an illustrative scenario that shows how a programmer can create running state machines step by step.

First, we create a machine called *doors*, add *opened* and *closed* states, and create *open* and *close* transitions between them. Figure 1 shows a visual and a textual representation of the resulting SML program. To run the machine, we create an instance, which starts in the *closed* state. Next, we trigger *open* and observe a transition to the *opened* state.

When modifying a running machine, users can learn cause-and-effect relationships between changes to the syntax and running instances. For instance, when we delete the *opened* state, we can observe the current state becomes *closed*.

The changes to the syntax (coding actions) and to the run-time state (user interactions) can be expressed as a sequence



(a) Live Programming Environment **(b)** Enabling Technology

Figure 3. Leveraging Game Engines for Visual Programming

of Read-Eval-Print-Loop (REPL) commands [22]. The first parameter of a command is a Unique Universal Identifier (UUID) of an object. A REPL interpreter evaluates these commands and updates the corresponding program elements. Figure 2 shows the commands for creating the doors program (2a), running it (2b), and printing its run-time state (2c).

By offering immediate feedback with every change, a visual programming environment can help its programmers form mental models. Each iteration offers a learning opportunity for understanding the impact of coding actions. We will use LiveSML as an illustrative running example.

2.2 Leveraging Language Workbenches

Creating visual programming environments for DSLs such as LiveSML is complex. Language workbenches and meta-programming languages provide a generic infrastructure [5].

We focus on a change-based approach, illustrated by Figure 3, that leverages this technology. According to the Model View Controller (MVC) paradigm, this approach separates the program state and execution logic from the user interface.

First, language engineers use an existing language workbench to create a REPL interpreter as the DSL back-end (Figure 3a). They can add liveness to its design by introducing cause-and-effect relationships between changes to the syntax and the run-time state [26]. Next, they create a UI that connects to this interpreter and serves the REPL visually.

2.2.1 Web-Based UI Frameworks. In recent years, many web-based solutions have been created that leverage browsers, e.g., LionWeb, Freon [29] and Rascal's Salix [24]. Expressing front-ends using the Document Object Model (DOM), these solutions incrementally update the UI using HTML5, CSS, and JavaScript. This update loop combines well with REPL interpreters that allow interaction through a web-based UI, e.g., via button presses and textfield inputs. A key benefit is that HTML5 libraries are widely available, e.g., for diagrams, figures, and editors. However, web-based solutions are no silver bullet for long-term compatibility, deployment and maintenance. Challenge 1 is integrating game engine technology into language workbenches as an alternative.

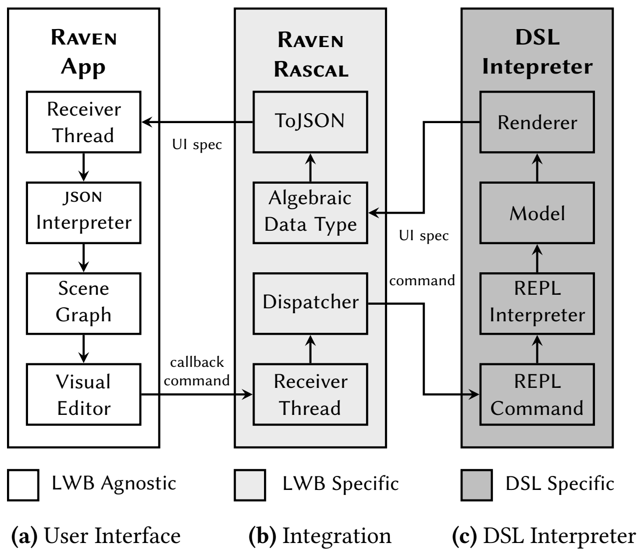


Figure 4. Overview of the Raven framework

2.2.2 Game Engines. Game engines are collections of software libraries, cross-compilers, and toolkits for creating interactive 2D and 3D applications [9]. Especially suitable for game development, these engines have also successfully been applied in other areas, e.g., health care [15], construction [6], and geography [12]. Developers can leverage engines to create responsive, user-friendly, and aesthetically pleasing applications that are easy to deploy and maintain. However, unlike web apps, these cannot easily be reprogrammed at run time. As a result, creating visual programming environments using game engines is costly and time-consuming. Challenge 2 is automating this process by exposing 2D features of the Godot engine in a web-like manner. Next, we introduce Raven, a novel approach to tackle these challenges.

3 The Raven Framework

Raven is a meta-framework for prototyping visual programming environments for DSLs. The proposed approach, illustrated by Figure 4, consists of three main parts.

The first part, shown in Figure 4a, is the Raven front-end application (or Raven for short), a programmable user interface based on Godot. Raven is a fully reusable language workbench-agnostic solution that operates as a generic visual REPL. Similar to a web browser, Raven requires a host environment. Its UI immediately updates when it receives a new UI specification from this host. When users interact, e.g., by pressing a button, it returns signals via associated callbacks. We describe the design of Raven in Section 4.

Using Raven comes at a cost. The second part of the approach, illustrated by Figure 4b, involves integrating Raven into existing meta-programming languages or language workbenches. As an example, we integrate Raven into Rascal [13] by mapping its generic JSON format to an Algebraic Data Type (ADT). We describe and assess this effort in Section 5.

The third part of the approach, illustrated by Figure 4c, is applying Raven and its host language in the creation of visual programming environments for DSLs. Language engineers create REPL interpreters using Rascal's advanced features, e.g., pattern matching, `visit` statement, and origin tracking. Using its declarative notation, they can concisely express a front-end as a so-called *renderer*. Based on the syntax and the run-time state (or model), a renderer generates the structure and styling of tree- or form-based editors. Each time a renderer runs, Raven immediately shows the result. Instead of a textual REPL, Raven supports live and exploratory programming *visually*. We demonstrate how this works in a case study on LiveSML in Section 6.

We begin by formulating functional requirements and technical challenges Raven addresses in Sections 3.1 and 3.2.

3.1 Requirements

Language engineers require a means for rapidly prototyping visual programming environments. They need a reusable front-end, reminiscent of a browser, that can easily be reprogrammed to create visual mechanisms for conveying coding actions, user interaction, and feedback. In particular, this front-end should expose the functionality of REPL interpreters using the expressive power of game engines. For programming the UI, language engineers need a minimal set of 2D elements to express:

R1 Informative output via *labels* showing a specified text.
R2 Interactive input controls: a) *buttons* that show a particular text and describe a callback message for executing a specified command when pressed; and b) *option buttons* that define a list of elements, and describe a callback for signaling when an element is selected.
R3 Structured views consisting of nested elements (or containers) with a specific spacing that are arranged: a) hierarchically in panels; b) horizontally in an HBoxContainer; c) vertically in a VBoxContainer; or d) side by side in tabs that can be viewed one at a time.

To control and adjust the appearance, engineers need to:

R4 Express the *styling* of: a) individual elements in terms of size, color, and spacing; and b) multiple elements at once to control the overall look and feel.

To support live and exploratory coding, programmers need immediate feedback. When programming, DSL users should see the consequences of their actions *immediately*. We formulate the following non-functional requirement.

R5 The entire REPL-loop must complete in under 100ms, which is regarded as instantaneous in HCI research.

Finally, language engineers may be bound by restrictions or have personal preferences that dictate which host language must be used. Therefore, they need a language workbench-agnostic solution that is straightforward to integrate using simple APIs. These APIs enable:

```
{"HBoxContainer": {"children": [{
  "Button": {
    "text": "Create Machine",
    "callback": "MachCreate(1)",
    "styles": [{"FontSize": {"font_size": 24}}]}}]}}
```

Listing 1. Example scene graph containing a Button

R6 Sending the UI state to render it.
R7 Receiving interactive REPL commands via callbacks.

Of course, these requirements are still preliminary. For now, we omit functionality for pictures and icons and graph-based editors, including drag-and-drop functionality.

3.2 Technical Challenges

To realize the requirements of Section 3.1, we identify the following technical challenges Raven must address.

T1 **UI Language.** To expose 2D game engine elements (requirements R1–R4), Raven should offer a declarative notation for concisely expressing: a) the state of the UI and b) callbacks that connect to DSL back-ends.
T2 **Rendering Engine.** To render the UI state quickly (requirement R5), Raven must integrate an interpreter that parses UI specs (T1), renders them, and facilitates executing the read-eval-print-loop visually.
T3 **Communication Protocol.** To easily integrate Raven with DSL back-ends (R6), it offers simple APIs that support asynchronous communication via sockets. For updating the UI, Raven receives UI specs. To propagate change, it returns the associated callbacks.

Next, we describe how Raven's design addresses the technical challenges. In Section 5, we will discuss how to integrate Raven into Rascal.

4 The Raven Application

A Raven application is the visual part of the framework, as illustrated by Figure 4a. We tackle the technical challenges of Section 3.2 as follows. Raven offers a language for describing structured UIs (T1), a rendering engine (T2) that updates the view, and a communication protocol (T3) for integrating Raven with its host language. Next, we will describe how these components work.

4.1 Raven UI Language

Godot-based applications contain a so-called scene graph. Similar to the browser's DOM tree, the scene graph consists of nodes that determine what the UI looks like. Raven offers a JSON notation to program the UI. The JSON schema expresses the following categories of nodes.

A *Label* displays a specified text. *Container Nodes* control the layout and alignment of child nodes. For instance, HBoxContainer and VBoxContainer align child nodes horizontally and vertically. ScrollContainer enables scrolling

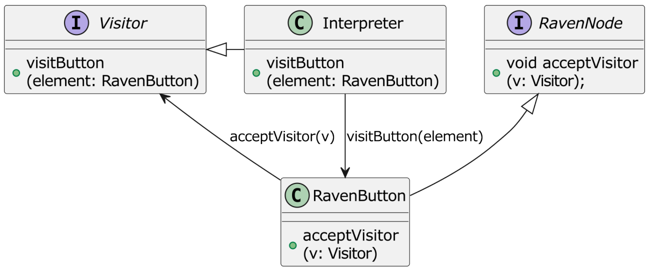

Visitor
visitButton (element: RavenButton)
Interpreter
visitButton (element: RavenButton)
RavenNode
void acceptVisitor (v: Visitor);
acceptVisitor(v)
visitButton(element)
RavenButton
acceptVisitor (v: Visitor)

Figure 5. Exposition of the Godot node Button in Raven

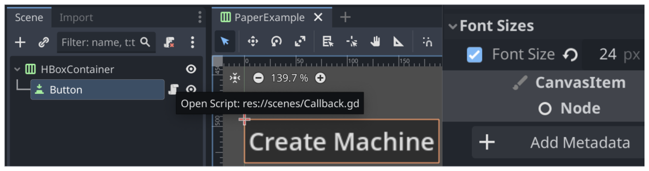


Figure 6. Overview of Minimal Example Creation in Godot and an attached callback script

them vertically. MarginContainer adjusts the spacing, and TabContainer shows Tabs side by side.

Input Nodes are interactive nodes that consume user input and call a corresponding callback function. Button displays a text and activates its callback when pressed. Listing 1 shows an example of a scene graph expressed in the JSON format. When pressed, the button sends its callback message, a LiveSML REPL command that creates a new state machine.

LineEdit and TextEdit, respectively, show single and multi-line text input, whose callback is the text input. An OptionButton defines a pull-down menu that returns the selected option. Finally, every node can be optionally modified with style information, e.g., color or font size. Listing 1 shows the Button has a font size of 24.

4.2 Raven Engine

The Raven engine parses the JSON format of UI specifications and updates the internal scene graph accordingly. Scene nodes are integrated as follows.

The engine employs an interpreter pattern to traverse the abstract syntax trees of the JSON format. Each node implements an accepting visitor function. For instance, Figure 5 illustrates how the Button element is integrated. When the interpreter visits a JSON node, it instantiates the corresponding Godot Button node, connects the corresponding signal (OnButtonPressed), and adds it to the scene graph. These nodes are then decorated with additional properties, such as style information.

Adding a Godot node to Raven is straightforward. To add a node, language engineers can create a scene graph by hand and add a script that handles the callback. When calling the associated visitor, the interpreter will instantiate the node. Figure 6 shows an example scene graph.

```
ravenButton(
  "Create Machine",
  "MachCreate(1)",
  settings=[])
```

(a) Raven Node in Rascal

```
{"Button": {
  "text": "Create Machine",
  "callback":
    "MachCreate(1)", ...
}}
```

(b) Raven Node in JSON

Figure 7. Mapping from a Rascal-specific Algebraic Data Type to the generic Raven JSON format

4.3 Communication Protocol

Raven offers a communication protocol to integrate its functionality with a host environment, e.g., a language workbench or DSL interpreter. Raven receives JSON data and sends callback messages via a TCP socket. The receiver thread handles messages in a producer-consumer fashion.

When it receives the `VIEW_UPDATE` message from its host environment, it passes the contents (of type string) to the engine for updating the view. The engine interprets the JSON and updates the UI.

When a user activates interactive nodes in the Raven UI, the Raven sends a message of type `CALLBACK`. This message passes the contents of the callback field back to the host environment. After evaluating the message, the host can again update the view.

5 Integrating Raven into Rascal

For applying Raven in a host environment, language engineers have to integrate its functionality. Here, we describe how to integrate Raven into the Rascal meta-programming language [13]. Figure 4b gives a general overview of the required components. Next, we will describe how to create these components and also assess the implementation effort.

5.1 Describing Editors

Programming UI specifications of visual editors in JSON directly is cumbersome. Instead, language engineers can generate these specifications by mapping its notation to language features of the host language.

Algebraic Data Types (ADT) are Rascal's composite types for defining custom types. We use the ADT feature to offer a more convenient notation for defining Raven scene trees. Listing 2 shows the definition of this type. For conciseness, we omit the style data. Each of the exposed 2D features has its own constructor and parameters. Using the ADT, language engineers can define the desired structure of an editor view. For instance, Figure 7a shows the definition of a button with the text "Create Machine" and a callback REPL command.

Of course, Raven cannot process this ADT directly. Instead, we create a generator that transforms the ADT into JSON. Figure 7b shows the generated JSON equivalent.

```
data RavenNode
= ravenNode2D(list[RavenNode] children, bool root)
| ravenNode2D(list[RavenNode] children)
| ravenLabel(str text, list[Setting] settings=[])
| ravenButton(str label, str callback, list[Setting] settings=[])
| ravenTextEdit(str content, str callback, ...)
| ravenLineEdit(str content, str callback, ...)
| ravenOptionButton(list[str] options, str callback, ...)
| ravenHBox(list[RavenNode] children, ...)
| ravenVBox(list[RavenNode] children, ...)
| ravenTab(str nodeID, str name, list[RavenNode] children, ...)
| ravenTabContainer(list[RavenNode] children, str callback="",...)
| ravenMarginContainer(list[RavenNode] children, ...)
| ravenPanelContainer(list[RavenNode] children, ...)
| ravenPanel(list[RavenNode] children, ...)
| ravenScrollContainer(list[RavenNode] children, ...);
```

Listing 2. Algebraic Data Type of the Raven Scene Tree

Table 1. Components that integrate Raven into Rascal

Component	SLOC: Rascal	Java
Algebraic Data Type	22	
JSON Generator	301	
Dispatcher	9	89
Sending Utils	2	12
Receiver Thread		174
SUM	334	275

5.2 Handling Incoming Callbacks

Visual views consist of nodes that can be annotated with callback functions in the form of a string. To parse and execute the callbacks, the Raven-Rascal integration provides a simple interface for the initialization. A higher-order function takes in a (dispatcher)-function as an argument with one parameter, a (callback)-string. The body of this function should include *parsing* and *dispatching* logic that handles incoming callback strings correctly.

It is up to the language engineer to provide these facilities. When a message of type `CALLBACK` arrives, the registered dispatcher evaluates the callback command from the DSL back-end. At the end of execution, an updated JSON view is returned. The Raven-integration sends then the view back to the Raven application. In the case of Rascal, a Java function is invoked to send the generated JSON to the server, with the message type `VIEW_UPDATE`.

5.3 Starting a Raven-Rascal Application

We add functionality to Rascal that makes it convenient to apply Raven. To use Raven, a language engineer has to update a configuration file and provide it with the entry point of the DSL program (usually a `main` function) that contains the registration of the dispatcher. After this, they can start the application from Rascal by calling the `main` function. The responsibility of the `main` function is to initiate the communication peer, which then registers the dispatcher function as the receiver callback.

5.4 Implementation Effort

Integrating Raven into Rascal is straightforward. Table 1 displays the number of Source Lines of Code (SLOC) for each component we have implemented[1]. The source code of Raven and Raven-Rascal is available on GitHub[2].

Rascal's advanced meta-programming features enable concise definitions. The ADT, the dispatcher, and communication protocol integration together count just 43 SLOC. The largest component is the JSON generator 301 SLOC.

The asynchronous communication protocol is instead implemented in Java. Conveniently, Rascal functions can invoke Java methods annotated with a `@java` annotation. The Java code amounts to 275 SLOC. Notably, the receiver thread has been reused from Raven.

6 Case Study

To explore Raven's potential, we create and analyze two visual editors for LiveSML, the DSL introduced in Section 2.1. Our objectives are to: 1) investigate how Raven can be used to quickly prototype alternative views; 2) assess the level of effort required to implement these views; and 3) determine the viability of Raven applications for visual programming.

6.1 LiveSML Revisited

LiveSML is a prototype REPL interpreter that processes sequential commands. To use the REPL, programmers have to express changes to the syntax and user interactions textually. However, DSL users may instead require visual views that offer distinct ways of understanding SML programs.

Tree Editors make the structure of the abstract syntax tree apparent through indentation. *Tabular Editors* are a popular choice when working with numeric values, such as invoices. Because state transitions can also be displayed in a table, both of these representations may be suitable.

We investigate how Raven can be used for rapidly prototyping visual programming environments that support live and exploratory programming. Our analysis has two stages.

First, we create tree and tabular editors using Raven and Rascal. We assess the implementation effort by measuring volume in source lines of code. Second, we apply the newly created editors in a realistic live programming scenario. We reproduce the scenario described in Section 2.1 and measure the performance. The results show that visual prototypes can be created with relatively little coding effort, and that the editors support live programming with immediate feedback.

6.2 Visual Live Programming Environment

We formulate requirements for a visual programming environment with coding and debugging facilities that offers:

1. a *navigation view* for creating programs, running them, opening new views, and navigating between tabs.

[1]We measure SLOC using CLOC v2.00 – https://github.com/AlDanial/cloc
[2]https://github.com/liza-kl/raven-project

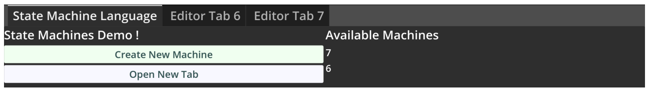


Figure 8. Initial Screen showing Navigation facilities

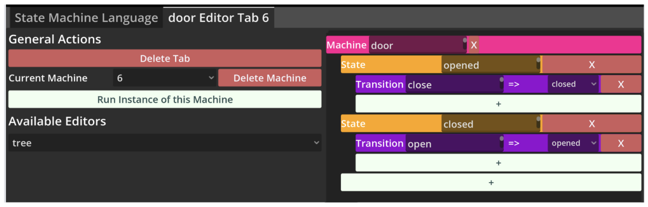


Figure 9. Tree Editor showing the doors program

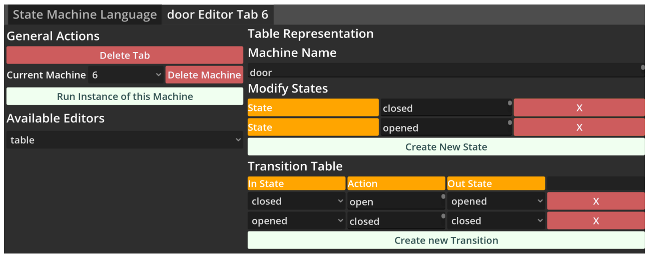
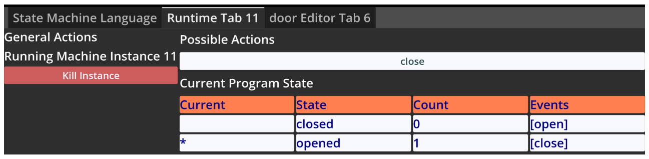


Figure 10. Tabular Editor showing the doors program



Current	State	Count	Events
	closed	0	[open]
*	opened	1	[close]

Figure 11. Run-time View showing a running doors program

2. *tree-* and *tabular editors* for altering the program's state visually, e.g., for adding, removing, and modifying state machines, states, and transitions.
3. a *runtime* view for running state machines, observing the run-time state; and triggering state transitions.

We use Raven and Rascal to create a visual programming environment. For navigating between views, we design a familiar browser-like view, including an *initial screen* and *tabs*, as shown in Figure 8. The *tree* and *tabular* views offer controls that map directly to REPL commands that work on the syntax. To create them, we implement renderers that generate the user interface based on the program state. Figures 9 and 10 shows how Raven renders an example program. These editors share an identical left-hand side. This allows the user to navigate between views smoothly with the "Available Editors" drop-down. Finally, the *runtime* view, shown in Figure 11, enables interaction with running machine programs. The source code of the LiveSML, including the visual editors and the REPL interpreter, is available on GitHub[3].

[3]https://github.com/liza-kl/raven-project/tree/main/raven-rascal-example/raven-core/src/main/rascal/lang/sml

Table 2. Components of LiveSML created in Raven-Rascal

Part	Component	Description	SLOC
General	Environment	Stores objects (syntax/run-time state)	30
Syntax	Model	Objects that define the abstract syntax	40
	Command	REPL commands that update syntax	24
	REPL	Evaluates REPL commands	158
	TreeRenderer	Renders the tree view	77
	TableRenderer	Renders the tabular view	57
Runtime	Model	Objects that define the run-time state	25
	Command	commands that update run-time state	16
	REPL	Evaluates REPL commands	112
	Callbacks	Binds Raven callbacks	58
	RTRenderer	Renders the run-time view	55
Control	Command	UI actions as REPL commands	13
	Model	Objects that define the UI state	3
	REPL	Evaluates REPL commands	149
	Callbacks	Binds Raven callbacks	43
Total			1177

6.3 Live Programming Scenario

We reproduce the live programming scenario of Section 2.1 to validate the design and measure its performance. Table 3 shows the steps and the recorded response times.

We start the visual programming environment (Step 1) and see the initial screen Figure 8. To create a new machine, we click on the button "Create New Machine" (Step 2). The ID of the machine will appear under "Available Machines". To alter its definition, we click the "Open New Tab" button (Step 3). We then click on the tab (Step 4) and select the machine we want to edit (Step 5). Using the tree editor, shown in Figure 9, we rename the machine to "door" in the Machine text field (Step 6). We add states closed and opened (Steps 7–10) and transitions open and close between them (Steps 11–16). Next, we open a tabular view using "Available Editors" on the left-hand side (Step 17). Figure 10 shows the tabular view.

We then run the machine by clicking on the button "Run Instance of this Machine" and selecting the newly opened tab (Steps 18–19). A runtime view appears as in Figure 11. The initial state of the machine is closed. To open it, so click on the button "close", triggering the transition (Step 20).

We delete the running machine by clicking on the button "Kill Instance" (Step 21). This also closes the runtime view and returns us to the initial view. To return to the previous editing view, we click on the Tab "Editor <machine ID>" (Step 22). Finally, we close the machine editing tab by clicking the button "Delete Tab" (Step 23) and end up in the initial view.

6.4 Analysis

Creating a visual programming environment for LiveSML with Raven is straightforward. Most of the work has been lifted by the Raven-Rascal integration, which includes the JSON generator. This allows for a concise definition of view structures using the Rascal ADT, as Table 2 shows. The system consists of three modules that each contain small components of Raven front-ends and REPL interpreters back-ends. The modules syntax, runtime and control respectively modify the abstract syntax, run-time states, and editor states.

Table 3. Programming scenario and rendering times in *ms*

#	Step	BE	FE	Total
1	Start the programming environment	3185	216	3401
2	Create a new state machine	104	23	127
3	Open a Tree-Editor in a new ab	129	50	179
4	Navigate to the opened tab	64	14	78
5	Select the newly created machine	75	29	104
6	Rename the machine to "door"	70	31	101
7	Add new state	70	22	92
8	Rename the state to "closed"	74	22	96
9	Add new state	76	35	111
10	Rename the state to "opened"	76	32	108
11	Add new transition to state "opened"	100	28	128
12	Rename its trigger event to "open"	92	56	148
13	Set its target state to "opened"	78	28	106
14	Add a new transition event to state"closed"	93	30	123
15	Rename its trigger to "close"	117	64	181
16	Set its target state "closed"	90	35	125
17	Change the view to Table	74	27	101
18	Run a state machine instance	134	38	172
19	View the running state machine	126	58	184
20	Trigger the "open" transition	94	41	135
21	Delete the Machine Instance	108	66	174
22	Navigate back to the Machine Editing Tab	106	44	150
23	Close the Machine Definition View	21	6	27

The prototype is fast enough for live programming. Table 3 shows the editor's response times during the programming scenario, for the front-end (FE) and back-end (BE). The combined rendering time is close to 100ms. This is the threshold the Nielsen Norman Group suggests for giving users a feeling of instantaneous response [18]. The main performance bottleneck resides in the unoptimized interpreter back-end.

7 Discussion

The Raven framework offers generic language technology for prototyping visual programming environment of DSLs. Of course, Raven is currently still an academic prototype that has has not yet been thoroughly validated and evaluated in practice. We discuss costs, benefits and threads to validity.

7.1 Costs, Benefits and Threats to Validity

Raven exposes a compact yet robust set of Godot nodes. Language engineers can integrate Raven into their preferred language workbench or meta-programming language.

Instead of manually creating a front-end using Godot, language engineers can use these nodes to create and test different programming front-ends *programmatically*, directly from a language-workbench. Raven's rendering engine design allows them to bypass the typically required compilation process for Godot applications. Instead of recompiling the

UI every time, the scene trees are constructed at run time. Unlike plug-in approaches [7], Raven does not rely on existing communication protocols and debug APIs for its operation.

Using Raven comes at a cost. Extending a language workbench requires components to communicate with a Raven app, and a mechanism to define the scene tree in a Raven-JSON format. Depending on the language workbench, Raven makes it possible to quickly and concisely implement different views. Boilerplate parts of the JSON can be extracted into a generator, and a UI language can be implemented with custom data types. The rapid prototyping speed opens up opportunities to test different views and their suitability for various user groups.

Raven still has important limitations. The design of Raven does not support easy implementation of end-to-end (E2E) tests or direct inspection of the user interface (UI), as is typically possible for web apps. However, developers can write tests by generating JSON objects and intercept the TCP communication, e.g., using tools such as Wireshark.

The case study represents only a limited validation. Furthermore, we acknowledge that Raven has only been integrated into Rascal, and as a result, we cannot yet draw general conclusions. Further investigation is necessary to determine to what extent Raven's concepts and tools can be integrated into other language workbenches too. Ultimately Raven provides an extensible approach. By describing its steps, we have laid the groundwork for future case studies.

7.2 Raven's Potential for Future Research

We discuss Raven's potential for future research.

7.2.1 Validation. In this paper, we demonstrated one concrete use case of Raven: how to retrofit a textual REPL of LiveSML with a visual programming environment. In future work, we plan to validate Raven further in user-driven studies that explore how to create UI variations for DSLs.

7.2.2 Integration and Deployment. Currently, each part of Raven runs on a separate JVM instance. However, we have selected the Java/Kotlin version of Godot for its potential to integrate Rascal with Godot more tightly. In principle, they could share one JVM instance, which could streamline the development, maintenance and deployment. In future work, we will investigate how to deploy DSLs to mobile devices.

7.2.3 Extensions. At the moment, Raven exposes a minimal set of Godot elements. However, Godot has many more interesting features that are not yet included. For instance, using its GraphNode/GraphEdit framework, language engineers could also create diagrammatic (or graph-based) editors. In future, work we plan to add these features. Raven's extensible design applies Strategy and Visitor patterns. As a result, Godot nodes can be added with minimal effort.

7.2.4 Rendering speed. The case study shows Raven's performance is already adequate. However, Raven updates its entire UI every time it receives an update, which slows down the response time unnecessarily. This behavior is comparable to `window.reload()` in the browser. The rendering speed can be further optimized by sending and evaluating differences instead, keeping existing visual elements intact.

7.2.5 Live Programming. Live programming is an area with open challenges that revolve around change [4, 21], e.g., run-time state migration, distributed editing and versioning. The Raven framework represents a step towards generic change-based solutions for tackling these challenges. We plan to further investigate how the create the necessary REPL interpreters in a principled manner, e.g., using Cascade [26].

8 Related Work

Game engines have been applied in many professional areas for creating interactive visual applications, e.g., in health care [15], construction [6], and geography [12]. We approach game engine technology from a language engineering perspective, which is still relatively uncommon [27]. Instead of creating visual programming environments "by hand", Raven supports rapid prototyping by generating them. To the best of our knowledge, Raven is the first approach that leverages a game engine in a language-parametric manner.

We relate our work to language workbenches [5], and tools and techniques that help create visual programming environments for DSLs. Tung et al. provide an overview of recent developments in this sphere [11].

Eclipse-based technology provides mature ways to define visual editors, predominantly graphical ones, e.g., Sirius [28], Eugenia [14], Modigen [8], and CINCO [17]. The UI specifications of these editors are tightly bound to the Ecore model, limiting the tools to the Eclipse ecosystem. In contrast, Raven is intended as a light-weight reusable solution not bound to one ecosysem. Any host environment adhering to its JSON schema can integrate the Raven framework.

Web-based technology provides a flexible way for creating portable visual editors that can choose from a plethora of libraries and styling options. In comparison, Raven is a much more focused approach. By offering a minimal set of elements, Raven provides a means for describing the structure of visual editors. Unlike web apps, game-based applications are not limited to the browser. Engines offer a reusable, portable and maintainable alternative. Raven offers flexibility by exposing sufficient Godot styling properties and having multi-platform features inherent to Godot's design.

Freon [29] is a web-based language workbench that allows language engineers to define their language and automatically generate a projectional editor. Kogi generates block-based editors, based on Google Blockly, directly from context-free grammars [16]. The Sandblocks system uses Squeak/Smalltalk as its foundation to derive structured editors from Tree-sitter grammars [1].

Rascal includes Salix, a web-based solution for the visualization, including DSLs [24]. Inspired by Elm, Salix helps to define and render a UI in the browser. To define the UI, Salix offers an internal DSL and a datatype. This convenience notation enables expressing UI elements as "statements" separated by semicolons. Raven builds on the experience of Salix. Raven still relies solely on a datatype, and does not yet support this convenience feature. However, by leveraging Godot, Raven adds speed, portability, touch interfaces, etc.

9 Conclusion

Visual environments have the potential to make programming more accessible. However, creating visual programming environments costs time and effort. Language engineers need enabling technology that supports rapid prototyping. We propose accelerating the development process by integrating game engine technology into generic language technology. We have introduced Raven, a meta-framework for creating visual programming environments. To illustrate the approach, we have integrated Raven into Rascal, and created visual editors for LiveSML. Although Raven is still an academic prototype, the initial results are promising. Using Raven, language engineers can concisely express the structure and styling of fast and aesthetically pleasing visual editors for DSLs. In future work, we plan to further extend and validate Raven, e.g., with graph-based editors. Raven opens possibilities for future studies on visual programming.

Acknowledgments

We thank the reviewers for their insightful comments that helped improve this paper. We thank Tijs van der Storm for his feedback and advise on the design of Raven.

References

[1] Tom Beckmann, Stefan Ramson, Patrick Rein, and Robert Hirschfeld. 2020. Visual Design for a Tree-Oriented Projectional Editor. In *Art, Science, and Engineering of Programming*. ACM.
[2] The Godot Community. 2024. Godot documentation. https://docs.godotengine.org/
[3] The Godot Community. 2024. Godot Kotlin. https://godot-kotl.in
[4] Jonathan Edwards, Tomas Petricek, and Tijs van der Storm. 2023. Live & Local Schema Change: Challenge Problems. *CoRR* abs/2309.11406 (2023). arXiv:2309.11406
[5] Sebastian Erdweg, Tijs van der Storm, et al. 2015. Evaluating and Comparing Language Workbenches: Existing Results and Benchmarks for the Future. *Comput. Lang. Syst. Struct.* 44 (2015).
[6] Ali Ezzeddine and Borja García de Soto. 2021. Connecting Teams in Modular Construction Projects Using Game Engine Technology. *Automation in Construction* 132 (2021).
[7] Leonard Geier, Clemens Tiedt, Tom Beckmann, Marcel Taeumel, and Robert Hirschfeld. 2022. Toward a VR-Native Live Programming Environment. In *International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments, PAINT 2022*. ACM.
[8] Markus Gerhart and Marko Boger. 2016. Modigen: Model-Driven Generation of Graphical Editors in Eclipse. *AIRCC's International Journal of Computer Science and Information Technology* (Oct. 2016).
[9] Jason Gregory. 2018. *Game Engine Architecture*. AK Peters/CRC Press.
[10] Matteo Iovino, Edvards Scukins, Jonathan Styrud, Petter Ögren, and Christian Smith. 2022. A Survey of Behavior Trees in Robotics and AI. *Robotics Auton. Syst.* 154 (2022).
[11] Aníbal Iung, João Carbonell, Luciano Marchezan, Elder Rodrigues, Maicon Bernardino, Fabio Paulo Basso, and Bruno Medeiros. 2020. Systematic Mapping Study on Domain-Specific Language Development Tools. *Empirical Software Engineering* 25, 5 (Sept. 2020).
[12] Julian Keil, Dennis Edler, Thomas Schmitt, and Frank Dickmann. 2021. Creating Immersive Virtual Environments Based on Open Geospatial Data and Game Engines. *KN-Journal of Cartography and Geographic Information* 71, 1 (2021).
[13] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *SCAM*. IEEE Computer Society.
[14] Dimitrios S. Kolovos, Antonio García-Domínguez, Louis M. Rose, and Richard F. Paige. 2017. Eugenia: Towards Disciplined and Automated Development of GMF-based Graphical Model Editors. *Software & Systems Modeling* 16, 1 (Feb. 2017).
[15] Stefan Marks, John Windsor, and Burkhard Wünsche. 2008. Evaluation of Game Engines for Simulated Clinical Training. Canterbury U.
[16] Mauricio Verano Merino, Tom Beckmann, Tijs van der Storm, Robert Hirschfeld, and Jurgen J. Vinju. 2021. Getting Grammars into Shape for Block-based Editors. In *Software Language Engineering*. ACM.
[17] Stefan Naujokat, Michael Lybecait, Dawid Kopetzki, and Bernhard Steffen. 2018. CINCO: A Simplicity-Driven Approach to Full Generation of Domain-Specific Graphical Modeling Tools. *International Journal on Software Tools for Technology Transfer* 20, 3 (June 2018).
[18] Jakob Nielsen. 1993. Response Times: The 3 Important Limits. https://www.nngroup.com/articles/response-times-3-important-limits/
[19] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2019. Exploratory and Live, Programming and Coding - A Literature Study Comparing Perspectives on Liveness. *Art Sci. Eng. Program.* 3, 1 (2019), 1.
[20] Jouke Stoel, Tijs van der Storm, Jurgen J. Vinju, and Joost Bosman. 2016. Solving the Bank with Rebel: On The Design of the Rebel Specification Language and its Application Inside a Bank. In *Industry Track on Software Language Engineering, ITSLE@SPLASH 2016*. ACM.
[21] Steven L. Tanimoto. 2013. A Perspective on the Evolution of Live Programming. In *Proceedings of the 1st International Workshop on Live Programming, LIVE 2013*. IEEE Computer Society.
[22] L. Thomas van Binsbergen, Mauricio Verano Merino, Pierre Jeanjean, Tijs van der Storm, Benoit Combemale, and Olivier Barais. 2020. A Principled Approach to REPL Interpreters. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2020)*. ACM.
[23] Jeroen van den Bos and Tijs van der Storm. 2011. Bringing Domain-Specific Languages to Digital Forensics. In *International Conference on Software Engineering, ICSE 2011*. ACM.
[24] Tijs van der Storm. 2023. *usethesource/salix-core: v0.1.0-RC2*. https://doi.org/10.5281/zenodo.8094140
[25] Arie van Deursen, Paul Klint, and Joost Visser. 2000. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices* 35, 6 (2000).
[26] Riemer van Rozen. 2023. Cascade: A Meta-Language for Change, Cause and Effect. In *Software Language Engineering, SLE 2023*. ACM.
[27] Riemer van Rozen. 2023. Game Engine Wizardry for Programming Mischief. In *Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments, PAINT 2023*. ACM.
[28] Vladimir Viyović, Mirjam Maksimović, and Branko Perisić. 2014. Sirius: A Rapid Development of DSM Graphical Editor. In *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*.
[29] Jos Warmer and Anneke Kleppe. 2022. Freon: An Open Web Native Language Workbench. In *Software Language Engineering, SLE 2022*. ACM.