Game Engine Wizardry for Programming Mischief

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

Abstract

Programming grants individuals the capability to design, create, and bring ideas to life. To improve their skills, programmers require powerful languages and programming environments for understanding the impact of gradual code changes. We investigate how modern game engine technology can be leveraged for creating visual input and feedback mechanisms that drive exploratory and live programming.

In this paper, we report experiences on creating a visual programming environment for Machinations, a domain-specific language for game design. We share initial findings on how to automate the development of graph- and tree-based editors in Godot, an open source game engine. Our results show that today's game engine technology provides a solid foundation for future programming language research.

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

Keywords: programming environments, game engines, language workbenches, live programming

ACM Reference Format:
Riemer van Rozen. 2023. Game Engine Wizardry for Programming Mischief. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments (PAINT '23), October 23, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3623504.3623570

1 Introduction

Visual programming environments have the potential to make programming more accessible to programmers of all backgrounds and skill levels. For instance, Domain-Specific Languages (DSLs) have been shown to help non-programmers raise their productivity, and improve the quality of their work [25]. DSLs offer specific abstractions and notations that provide increased expressiveness over particular problem domains, e.g., banking, digital forensics and game design.

PAINT '23, October 23, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0399-7/23/10.
https://doi.org/10.1145/3623504.3623570

Language workbenches and meta-programming languages provide techniques and approaches that support rapid prototyping of DSLs [9], especially compilers and interpreters, e.g., based on Visual Studio Code and the Language Server Protocol (LSP). However, generic language technology often has limited support for visual programming environments, e.g., projectional editors [29], block-based editors [19], or web interfaces [30]. Language engineers require tools and techniques to speed up and simplify the development of visual programming environments that are user friendly, aesthetically pleasing, and easy to deploy, maintain and extend.

Game engines are collections of software libraries, toolkits and cross-compilers that have been especially designed for developing portable visual applications, games in particular. These engines represent the state of the art in 2D and 3D frameworks for creating visual simulations, Heads-Up Displays (HUDs) and immersive interactive experiences.

To date, only a limited number of authors have explored using game engines for creating DSLs [18, 26, 33]. We see a research opportunity to bridge the gap between the technological spaces of game engine technology and Programming Language (PL) research, language workbenches in particular.

We hypothesize that game engines are well-suited for the automated development of interactive programming environments, especially visual DSLs. In particular, we aim to learn how game engines can be used for creating visual input and feedback mechanisms that support exploratory programming, live programming and creative tinkering.

To shed light on this matter, we conduct a feasibility analysis and carry out a pilot study. First, we create a concise overview of a limited number of well-known game engines in Section 2. We assess strengths, weaknesses, opportunities and threats. Based on our initial positive analysis, we select Godot, a free open source game engine described in Section 3, for further study. We investigate how Godot can facilitate creating visual programming environments with graph- and tree-based projectional editors.

We develop a live programming environment for Micro-Machinations, a visual DSL for game design in Section 4. This work adds a visual front-end based on Godot to the existing language back-end based on the Cascade meta-language [27]. We report positive experiences and reflect on lessons learnt in Section 5. Finally, we describe related work in Section 6, and conclude with final remarks in Section 7.

This paper contributes: 1) a concise overview and analysis of well-known game engines as a means for creating

Table 1. Game Engines integrate programming languages

Engine	Visual scripting	Language support
CryEngine V	Flow Graph	C++, Lua, C#
GameMaker	visual scripting tool	GML
Godot Engine 4	VisualScript*	GDScript, C, C++, C#, ...
Open 3D Engine	Script Canvas	Lua
Unity	Bolt visual scripting	C#
Unreal 5	Visual Blueprints	C++

visual programming environments; and 2) an assessment of Godot in the development of Vie, a tiny Live game engine for Machinations, specifically for a) realizing its preliminary requirements, and b) creating its visual front-end.

Our results show that today's game engine technology, Godot in particular, provides an excellent foundation for future research on visual and interactive programming environments. Game engines provide us with the means, motive and opportunity for programming mischief.

2 Game Engines

Language workbenches and meta-programming languages offer tools and techniques for developing meta-programs, programs that work on other programs, e.g., compilers and interpreters [9]. These tools are often based on Java, e.g., Rascal, Spoofax and Jetbrains MPS [9]. We wish to learn if and how game engines can be integrated with this generic language technology for developing visual DSLs. We conduct a feasibility analysis to assess if game engines are suitable for developing visual programming environments.

To accomplish this, we create a concise overview of well-known game engines about: a) support for programming languages; b) features for creating UIs; and c) software licenses. We weigh the pros and cons of applying engines, and assess strengths, weaknesses, opportunities and threats.

2.1 Overview of game engines

Game engines are software libraries, tools, and compilers for creating interactive 2D and 3D simulations [13]. Especially suitable for game development, these engines can also be used to create other interactive visual programs with fast, user-friendly, and aesthetically pleasing interfaces. Figure 1 shows well-known game engines and their support for programming languages. We briefly describe each game engine.

2.1.1 CryEngine. CryEngine, originally developed around the 3D shooter Far Cry, is a proprietary game engine that includes many visual tools [12]. The toolset includes a terrain editor, behavior trees, and flow graphs, to name a few.

2.1.2 GameMaker. GameMaker is the odd one out in our short list. Intended for making 2D games only, this proprietary platform by YoYo games is aimed at indie game developers and educators alike [17, 20]. The GameMaker Language a C-like language for adding behaviors.

2.1.3 Godot. Godot is a free 2D and 3D game engine for cross-platform game development [16]. Godot comes with support for C, C++ and C#. Godot also includes a script language designed for novices called GDScript, a dynamically typed language that resembles Python. Due to lack of interest, its visual language VisualScript was discontinued at v3.0. The sources of Godot are released under the MIT license.

2.1.4 Open 3D Engine. Open 3D Engine (O3DE) is a 3D game engine, a free open source continuation of Amazon Lumberyard now developed by the Open 3D Foundation [8]. O3DE includes Lua [14], a multi-paradigm embeddable language that is often used for scripting in games. Supported by the Linux foundation, its sources are available under the Apache 2.0 license.

2.1.5 Unreal Engine 5. Unreal Engine 5 is a real-time 3D engine and creation tool for "visuals and immersive experiences" owned by Epic Games [11]. Originally developed for Unreal, a 3D shooter, this engine has seen many iterations and is widely applied.

2.1.6 Unity. Unity is a commercial game engine with support for creating 2D and 3D applications [24]. Unity remains a popular choice game among developers for its intuitive APIs and cross platform compilation.

2.2 Weighing pros and cons

Game engines have compelling benefit. Each of these engines, except perhaps GameMaker, has the necessary features for creating multi-faceted tools and programming environments. We relate key technical concerns to applied research needs.

2.2.1 Documentation. Engines often have high quality and up to date documentation with examples and explanations. For instance, show cases illustrate examples that demonstrate engine capabilities and features. Usability is a prime concern, and so is educating junior developers.

2.2.2 2D and 3D. Engines usually offer state of the art 2D and 3D frameworks. A projectional editor will mostly use the 2D APIs, and can potentially integrate 3D visualizations.

2.2.3 Toolkits. Engines offer rich toolkits to support design and development processes, e.g., in asset stores. This includes programming environments, visual editors and asset editors created using the engine itself. Some are DSLs in their own right. These are evidence that supports our hypothesis. These examples make DSL development feasible.

2.2.4 Cross-compilation. Most engines have compilers that can target various platforms, e.g., Windows, Linux, iOS and Android. Developers have a single point of maintenance and enjoy platform independence. Using game engines, cross-platform compilation comes at no additional cost.

2.2.5 Mobile devices. Engines have built-in support for gestures and touch APIs for tablets and mobile phones.

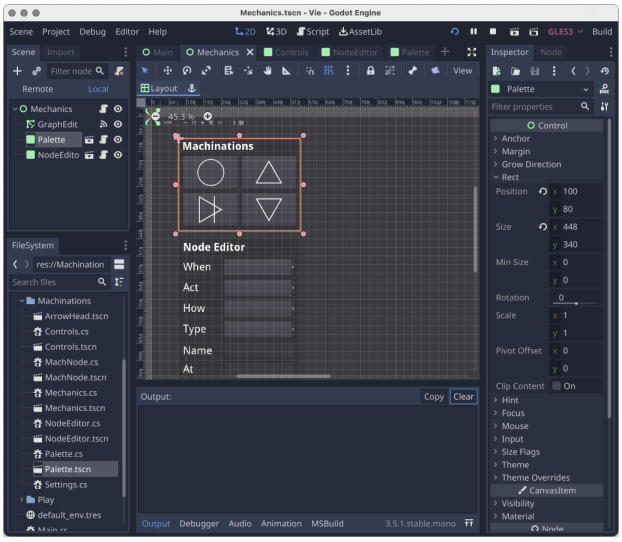


Figure 1. Godot showing a scene graph built from parts

2.2.6 Deployment. For commercial software, time to market is essential. Most engines therefore integrate with app stores and market places, enabling rapid deployment.

Of course, reusing this digital infrastructure can also be beneficial for deploying open source software. The deployment of language technology and DSLs can be a time consuming process. Using game engines, this process can become more straightforward.

2.2.7 Long term support. Maintenance and software dependencies are key concerns. As substantial investments, engines likely receive extensive maintenance and support.

2.2.8 Vendor lock-in. A tight coupling between game engines and software poses the risk of vendor lock-in. Our intended use, generating UIs, aims to decouple DSL specifications from the engine code.

2.2.9 Language support. A challenge is that many language workbenches are based on Java which is not supported by these game engines. The Godot community has added support for Rust, Nim, Haskell, Clojure, Swift, and D. Java is not yet supported, but might be added.

2.2.10 Proprietary products. Many engines are proprietary paid products with royalty systems that have long term costs. Godot and O3DE are free open source projects, which makes them attractive for community efforts.

2.3 Generic Language Technology

We aim to leverage game engines in language-parametric technology for developing visual programming environments. To investigate if this is feasible, we carry out a pilot study.

We select Godot for a deeper investigation for its extensive 2D support, clear documentation and open source license. We apply Godot to the development of a live programming environment for Machinations, a visual DSL for game design.

3 Godot Game Engine

Before we assess Godot as a platform for developing visual programming environments we first give a concise overview.

The entity component model is a well-known solution for varying modular structures that originates from game development [2]. This pattern ensures a loose coupling between entities and components for flexible and extensible designs. We summarize Godot's model as follows.

The main data structure of a program is its scene graph. Godot includes a visual scene editor, shown in Figure 1. This graph consists of graphical components that can be flexibly added and removed (top left). Control nodes are the base type of every interactive 2D component, e.g., Label, Panel, TextEdit, LineEdit and Tabs. The center view shows what the scene looks like. Using the editor, programmers can compose components and modify properties, including positions and spacing (right), into scenes with reusable interfaces and behaviors. Scenes can express components of UIs, e.g., editor menus, graph editors or debug windows.

Scenes can be packed into "prefab" components. At run time, packed scenes can be efficiently instantiated. Of course, the graphs can still be modified, pruned and extended.

Scene nodes, specifically control nodes, communicate via events. These can register event handlers on the graph to receive notifications, and decorate the graph with timers and callbacks. The engine determines when events trigger, hiding the control flow from the programmer. For handling notifications, programmers can connect scripts.

For scripting, Godot provides GDScript, a script language specifically designed for novice programmers. Godot also offers C# support with the .NET SDK and the .NET-enabled version of Godot. Earlier versions shipped Mono. As an alternative, developers can use JetBrains Rider, a powerful C# IDE together with Godot. Godot has excellent documentation that is continually updated as new features are added.

Next, we apply Godot in the creation of an interactive visual programming environments for Machinations. Please note that we have used Godot 3.5.1 for compiling a prototype for iOS. At the time of writing, the current version is 4.2.

4 Machinations

Machinations is a visual notation for game design that foregrounds elemental feedback loops associated with emergent gameplay [1]. Micro-Machinations (MM) is a textual and visual programming language that addresses several technical limitations of its evolutionary predecessor [28]. In particular, MM introduces a live programming approach for accelerating the game development process with an embeddable interpreter that enables modifying digital games at run time.

We investigate how Godot can be leveraged to create a visual programming environment that integrates this interpreter. In particular, we assess its uses for designing visual input and feedback mechanisms that drive live programming.

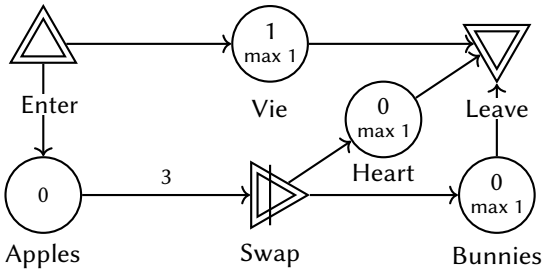


Figure 2. Micro-Machinations diagram of Vie and her bunny

We describe Vie, a tiny live game engine for simultaneously prototyping and playtesting a game's mechanisms. The prototype demonstrates that Godot offers the necessary foundations for creating visual editors of graph based DSLs.

4.1 Micro-Machination

Micro-Machinations programs, or diagrams, are directed graphs that control the internal economy of running digital games. When set in motion through runtime events and player interactions, the nodes act by pushing or pulling economic resources along its edges.

We introduce the notation by means of a design theme intended for young children. The theme features a girl, bunnies and apples. Game designers and children can explore this design space together, guided by creative tinkering and imagination. Figure 2 shows an example game economy.

The game introduces Vie (pronounced /vi/), a girl who has lost her bunny. She can get him back by collecting and exchanging apples. Four *pool* nodes, shown as circles, abstract from the in-game resources: Vie, Apples, Heart and Bunnies. The integers inside represent current and maximum (max) amounts. The edges are *resource connections* that define the rate at which resources can flow between source and target nodes. Three interactive nodes, indicated by double lines, define interactive mechanisms a player can activate.

Every time Vie enters, she brings an apple. Enter is an interactive *source node*, shown as a triangle pointing up, the only place where resources can originate. When activated, it produces Vie and an apple.

Vie has to leave before she can collect more apples. Leave is an interactive *drain node*, shown as a triangle pointing down, the only place where resources can disappear. When activated, it consumes the Vie resource.

When Vie has collected three apples, she can exchange them for her bunny. Swap is an interactive *converter* node, appearing as a triangle pointing right with a vertical line through the middle. Converters can be rewritten as a combination of a drain, a trigger and a source. When the drain consumes the costs of the conversion, the trigger activates the source, which then produces the benefits. When activated, this converter consumes three apples and produces a heart and a bunny. All is well that ends well when Vie leaves reunited with her bunny.

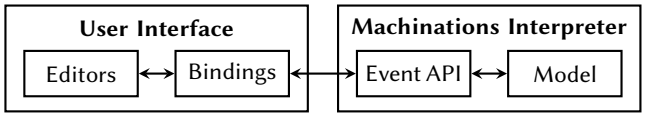


Figure 3. Vie tool architecture and components

4.2 Vie: a tiny live game engine

Micro-Machinations has introduced a live programming approach that accelerates game design [28]. This approach proposes an embeddable interpreter to power the internal economies of digital games. Designers can simultaneously prototype and playtest mechanisms inside running economies, e.g., by modifying the diagram and by activating nodes.

We address the need for a visual live programming frontend that integrates visual game prototypes. We define preliminary requirements of Vie: a tiny live game engine.

4.2.1 Functional requirements. Designers require a live programming environment for simultaneously prototyping and playtesting a game's mechanisms. They need immediate and continuous feedback for understanding the relationships between mechanisms, UI elements and gameplay. We formulate preliminary requirements for each of these concerns.

Mechanics Editor. Designers need to prototype mechanisms before assessing the gameplay. They require an editor to flexibly design, modify, and test the dynamics of game-economic mechanics. The editor allows designers to:

- R1 Create diagrams by adding nodes and edges on a canvas, by moving elements, and zooming in and out.
- R2 Modify and edit the properties of a nodes and edges.
- R3 Activate nodes for enacting their effects.
- R4 Observe visual feedback about the success of nodes, triggers and the flow of resources in a diagram.

UI Editor. Designers prototype user interfaces to make the mechanics playable. They require an editor and a minimal set of visual components to design, bind, place and position:

- R5 Buttons that activate interactive nodes.
- R6 Labels that show current amounts of resources in pools.
- R7 Sprites that relate imagery to current pool amounts.

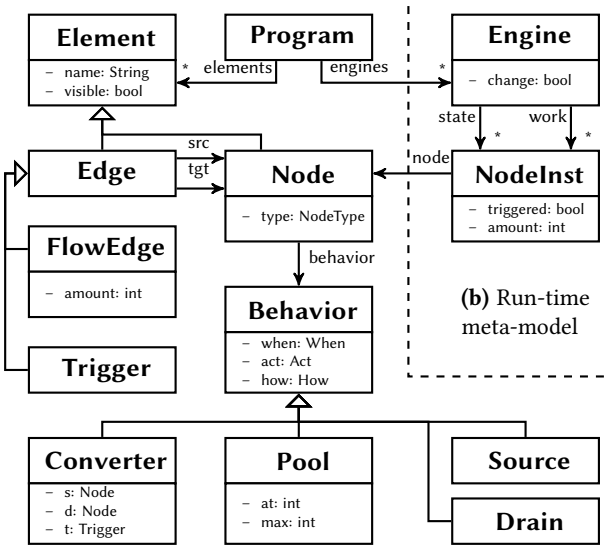
Interactive Game Prototype. For playtesting a game's mechanisms, designers require a game prototype that facilitates experiencing gameplay. The tool provides an integrated game simulation that enables designers and players to:

- R8 Press buttons for activating mechanisms.
- R9 Observe labels with textual amounts.
- R10 Observe sprites representing game state.

Next, we describe technical challenges for creating the Vie programming environment.

4.3 Objectives

We aim to create a visual live programming environment for Micro-Machinations based on the Model View Controller



(a) Static meta-model

Figure 4. Partial meta-model of Micro-Machinations (diagram appears in van Rozen [27])

(MVC) paradigm , as illustrated by Figure 3. We study how to leverage Godot in the creation of Vie, a tiny live game engine that satisfies the requirements of Section 4.2.1.

In particular, we investigate how to integrate the Micro-Machinations interpreter, which has been created using the Cascade framework [27]. Cascade expresses DSLs and run-time transformations, and compiles to C#. The event-based runtime serves as a controller that manages the abstract syntax and the run-time states (Model). The technical challenge we have to address is adding a user interface (View).

4.3.1 Technical challenges.

Technical challenges:

- T1 Create user interfaces by defining scene graphs of visual components for mechanics, UI, and gameplay.
- T2 Modify programs, and activate run-time behaviors, using event APIs of the Cascade interpreter.
- T3 Render the view by connecting callbacks that trigger when models are updated to modify the scene graphs.
- T4 Display visual feedback about the effects of events by highlighting behaviors (activation, success, failure, triggers and flow) using colors and timers.

4.4 Tool Design

The language design of Micro-Machinations is based on the meta-model of Figure 4. This class diagram shows the structure of its abstract syntax and run-time states. Programs, or Abstract Syntax Graphs (ASGs), are instances of the static meta-model. Run-time states, on the other hand, are instances of the run-time meta-model.

The C# interpreter has an event driven design. Based on the Cascade framework, it offers a scheduling API for making gradual changes, e.g., for adding nodes or edges, deleting them, or changing the type of a node. Several advanced features support live programming, including run-time state migrations [27]. When designers modify the program, the interpreter migrates the run-time state by updating current amounts of pool nodes. A publish-subscribe mechanism allows registering external observers that trace events and side-effects. When events happen, the interpreter notifies these components that changes have occurred.

4.4.1 Tool.

We design a visual tool that consists of tabs and an output window. Figures 5 and 6 show the different tabs of the tool. We explain them one by one.

4.4.2 Mechanics Editor.

We design a Mechanics Editor that realizes requirements R1–R4 described in Section 4.2.1. We create the following scene graphs, and add C# code that binds them to the APIs of the interpreter.

Mechanics. Mechanics is the tab containing a GraphEdit, Palette and NodeEditor components. GraphEdit is Godot's reusable graph editor component, which combines with GraphNode. The MachNode scene inherits from GraphNode, and contains labels and sprites for displaying Machinations nodes. A script enables observing a specific node, and updating the visibility and values based on callbacks. The center of these nodes contains a button that binds the API for scheduling node activations. We also bind events to timers that temporarily change visual appearances to signal: 1) node success and failure; 2) triggers; and 3) resource flow. Figure 5 shows an example of visual feedback.

NodeEditor. The NodeEditor scene is a Panel consisting of RichTextLabels, OptionButtons and LineEdit components. A script enables observing a selected node, which binds the API to the interpreter. The controls schedule events to the interpreter back-end. Properties become visible as appropriate, and values update based on callbacks.

Palette. The Palette scene consists of four Buttons with Sprites on them. Clicking a button creates exactly one new node of that type in the GraphEdit area by clicking a location.

4.4.3 User Interface Editor.

The UI editor that satisfies requirements R5–R7. Like the mechanics editor, the HUD editor consists of a panel, an editor and a GraphEdit area. Here, three classes inherit from GraphNode for sprites, buttons and labels. These nodes can be placed on the GraphEdit area. Their 2D location determines where these elements appear in the game simulation. The UI contains sprites for Vie, Apples, Heart and Bunnies.

4.4.4 Game.

The game simulation satisfies requirements R8–R9 of Section 4.2.1. The Game tab consists of a Panel with Buttons, RichTextLabels and Sprites. The Mechanics and UI Design define its looks and behavior.

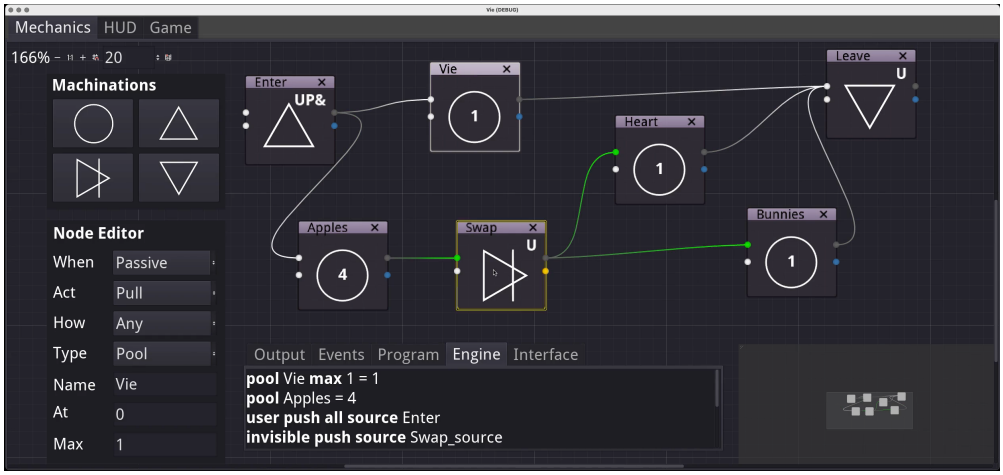


Figure 5. Vie mechanics editor showing Swap activated and succeeding, generating flows of resources

(a) User interface editor showing the UI design

(b) Running game showing Vie with her bunny

Figure 6. Vie user interface editor and a live game simulation

4.5 Live Programming Scenario

We revisit the theme of Vie and her bunny, and use the prototype to explore this design space. We recreate the diagram of Figure 2, which is just one of many potentially fun scenarios we can explore. After all, Vie affords exploring any design we like. We can perform the following activities in any order.

Designing Mechanics. Using the Mechanics editor, shown in Figure 5, we design an internal game economy. By first tapping in the Machinations panel on the left, we then tap on the canvas on the right to add the nodes. We drag the nodes into suitable places. We add resource edges using the tip of our finger on the connectors appearing on the nodes. Using the Node Editor, we set maximum amounts and make nodes interactive.

Designing a User Interface. Designing the UI, shown in Figure 6a, is straightforward. Several nodes are added automatically by the interpreter as convenient side-effects. For instance, the interpreter automatically ads labels and sprites for pool nodes. When we create an interactive node, the interpreter adds a buttons if it does not yet exist. We drag and drop these controls into suitable locations.

Table 2. Vie's UI packages, file counts and volume in SLOC

Package	C# bindings		Visual scenes	
	Files	SLOC	Files	SLOC
Tool	1	171	1	19
Mechanics Editor	6	1234	5	434
User Interface Editor	7	1085	9	883
Gameplay Simulator	4	407	4	469
sum	18	2897	19	1805

Playing the Game. Meanwhile, one game instance is always running. By pressing the buttons in sequence, we playtest if Vie can collect apples and if she can exchange them for her bunny. Figure 6b shows this is indeed the case.

4.6 Implementation

The implementation of Vie consist of three main parts: the scene graphs, the MM interpreter, and the bindings between the APIs of the two. Table 2 shows volume in Source Lines of

Lines Of Code (SLOC) of the scene graphs and the bindings[1]. The Godot project also integrates the existing sources of the MM interpreter. These are compiled from a Cascade specification of 2330 SLOC, and measure 33.6 KSLOC of mostly generated C# code. We created the scenes visually, but we also show the volume of the `.tscn` files, 1805 SLOC total. The bindings consist of 2897 SLOC of hand-written C#.

We have compiled Vie for MacOS and iOS and tested the prototype works well on a laptop and a mobile phone.

4.7 Analysis

Creating the UI of Vie in Godot is not hard. We spent a few days on learning how to use Godot, and creating the design. Most of the work went into the C# binding the event API of the interpreter to the UI elements. Creating these was straightforward because both Godot and Cascade have event APIs. The technical difficulty is in the MM interpreter.

GraphNode and GraphEdit can be used for creating graph-based interfaces. The mini-map, the grid, and zoom and snap functionality are built-in and customizable. However, there are also limitations. Edges are not represented as first class components, but as integers in a list, which complicates mapping the view to the model.

In addition, edges are not arrows. The type and direction of an edge is determined by specific connection points on the GraphNodes. Due to this limitation, we represent triggers and flow edges with different colors and connection points.

5 Discussion

5.1 Projectional editors

Of course, we have just described the creation of one visual programming environment in Godot. We are currently investigating how to do this for other DSLs too.

The Vie prototype only includes graph-based editors and tree editors of fixed sizes. Other languages require editors for dynamically sized trees. In Godot, such an editor can be created by adding components in a vertical container, and adjusting their left-hand margins based on the nesting level in the tree. A cursor can be introduced for navigating the list, and selecting, adding and removing elements.

5.2 Metaprogramming

In addition, the conceptual gap between Cascade, the meta-language of the MM interpreter, and Godot is relatively narrow. Both frameworks are event-driven and integrate C#.

Of course, this not generally the case. For instance, some languages operate on immutable data or are not inherently event-driven. These cases require first computing differences before rendering updates to the views. Naturally, this can be automated too, e.g., using GumTree [10].

In addition, many meta-programming languages require Java. None of the popular engines we discussed support Java. Creating a binding between an engine and a programming language represents a substantial amount of work. Work has been done to integrate Kotlin 1.7.10, which supports desktop (Linux, MacOS, Windows) and Android applications[2].

5.3 Towards generic language technology

Based on our preliminary findings, we assert that Godot can provide a solid foundation for developing visual programming environments. Since other engines have similar capabilities, those too may be suitable.

We have not yet automated the development of visual programming environments. We created the UI of the Vie prototype by hand. Further investigation is necessary to ascertain to what extent programming environments can be generated, and how generic language technology can support this. Open challenges are:

1. Generating API bindings that exchange events between the DSL back-end and the UI front-end.
2. Introducing scene templates that can be mapped to DSL features for generating scenes.
3. Defining variation points for tweaking default views.

Since sizes, offsets and sprites are key, the generated programming environments will likely have limitations. However, the approach supports rapid prototyping and deployment of unpolished but usable prototypes.

6 Related work

Game engines are a topic in related work. Andrade gives a limited overview of game engines [3]. Zhu and Wang give a model-driven engineering perspective on games [32]. In a much larger study, van Rozen sheds additional light on the topic, discussing model-driven engineering, automated game design and meta-programming [26].

There is a rich tradition of tools for modifying (or modding) games [22]. Some games also integrate forms of end user programming. For instance, in "Baba is You" players modify mechanisms and solve puzzles by pushing program blocks into place [6]. Similarly, we aim to integrate programming environments with the applications they can modify.

Alternatives to game engines are 2D frameworks. These include Qt [5], Tcl/Tk [31], Eclipse Modeling Framework (EMF) [23], to name a few. While appropriate for certain technological spaces, or maintenance scenarios, these frameworks are incomparable to the feature sets of game engines. Another avenue to achieve portability is creating an interactive web site. Browser frameworks include Angular, Vue, Svelte, Elm, and many more[3].

Generic language technology aims to facilitate the creation of interactive visual programming environments. Cicchetti et al. survey multi-view modeling approaches [7]. Several generic approaches integrate web-based UI frameworks with

[1] We measured SLOC using `cloc-1.96` – https://github.com/AlDanial/cloc

[2] https://godot-kotl.in – Last visited July 14th 2023.

[3] https://angular.io. https://vuejs.org. https://svelte.dev. https://elm-lang.org

meta-programming environments to support interactive experiences. For instance, the Salix framework of Rascal [15] performs Elm-like rendering and updating[4], similar to the game loop. Freon uses Svelte for creating projectional editors [30]. Kogi instead derives block-based editors, based on Google Blockly, from context-free grammars [19]. In contrast, the Sandblox system automatically derives structured editors from Tree-sitter grammars on Squeak/Smalltalk [4]. Unlike web libraries, game engines are not limited to the browser. Engines offer a reusable, portable and maintainable alternative for developing generic solutions.

An important facet of visual programming environments is rapidly providing understandable feedback. Live programming studies how to speed up programming cycles by providing immediate and continuous feedback [21]. The Cascade framework enables creating interpreters of DSLs whose users enjoy the benefits of live programming, but still lacks visual interfaces [27]. This pilot study has performed initial steps towards generic language technology for visual live programming environments.

7 Conclusion

Language engineers need tools and techniques that speed up and simplify the development of visual programming environments. We have studied how game engines can be used for creating visual input and feedback mechanisms that support exploratory and live programming, and creative tinkering. To find out, we have conducted a feasibility analysis and carried out a pilot study. We have investigated how Godot facilitates creating a live programming environment for Micro-Machinations, a visual DSL for game design. Our results show that today's game engine technology, Godot in particular, provides an excellent foundation for future research on visual and interactive programming environments.

References

[1] Ernest Adams and Joris Dormans. 2012. *Game Mechanics: Advanced Game Design.* New Riders.

[2] Toni Alatalo. 2011. An Entity-Component Model for Extensible Virtual Worlds. *IEEE Internet Comput.* 15, 5 (2011).

[3] António Andrade. 2015. Game Engines: A Survey. *EAI Endorsed Trans. Serious Games* 2, 6 (2015).

[4] Tom Beckmann, Patrick Rein, Stefan Ramson, Joana Bergsiek, and Robert Hirschfeld. 2023. Structured Editing for All: Deriving Usable Structured Editors from Grammars. In *Conference on Human Factors in Computing Systems, CHI 2023.* ACM.

[5] Jasmin Blanchette and Mark Summerfield. 2006. *C++ GUI programming with Qt 4.* Prentice Hall Professional.

[6] M. Charity, Ahmed Khalifa, and Julian Togelius. 2020. Baba is Y'all: Collaborative Mixed-Initiative Level Design. In *Conference on Games, CoG 2020.* IEEE.

[7] Antonio Cicchetti, Federico Ciccozzi, and Alfonso Pierantonio. 2019. Multi-view Approaches for Software and System Modelling: a Systematic Literature Review. *Softw. Syst. Model.* 18, 6 (2019).

[8] Open 3D Engine Contributors. 2023. O3DE Documentation. https://docs.o3de.org/ (CC BY 4.0) Last visited: July 10th 2023.

[9] Sebastian Erdweg, Tijs van der Storm, et al. 2013. The State of the Art in Language Workbenches. In *Software Language Engineering, SLE 2013 (LNCS)*, Vol. 8225. Springer.

[10] Jean-Rémy Falleri et al. 2014. Fine-grained and accurate source code differencing. In *Automated Software Engineering, ASE '14.* ACM.

[11] Epic Games. 2023. Unreal Engine 5.2 Documentation. https://docs.unrealengine.com/ Last visited: July 10th 2023.

[12] CRYTEK GmbH. 2019. CRYENGINE V Manual. https://docs.cryengine.com/ Last visited: July 10th 2023.

[13] Jason Gregory. 2018. *Game Engine Architecture.* CRC Press.

[14] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. 2007. The Evolution of Lua. In *History of Programming Languages Conference (HOPL-III).* ACM.

[15] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Source Code Analysis and Manipulation, SCAM 2009.* IEEE.

[16] Juan Linietsky, Ariel Manzur, and the Godot community. 2023. Godot Engine 4.2 documentation. https://docs.godotengine.org/

[17] YoYo Games Ltd. 2023. GameMaker Manual. https://manual.yoyogames.com Last visited: July 10th 2023.

[18] Sonja Maier and Daniel Volk. 2008. Facilitating Language-Oriented Game Development by the Help of Language Workbenches. In *Future Play 2008.* ACM.

[19] 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.

[20] Mark H. Overmars. 2004. Teaching Computer Science through Game Design. *Computer* 37, 4 (2004).

[21] Patrick Rein, Stefan Ramson, et al. 2019. Exploratory and Live, Programming and Coding - A Literature Study Comparing Perspectives on Liveness. *Art Sci. Eng. Program.* 3, 1 (2019).

[22] Walt Scacchi. 2011. Modding as an Open Source Approach to Extending Computer Game Systems. *Int. J. Open Source Softw. Process.* 3, 3 (2011).

[23] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: Eclipse Modeling Framework.* Pearson Education.

[24] Unity Technologies. 2023. Unity User Manual 2022.3 (LTS). https://docs.unity3d.com/ Last visited: July 10th 2023.

[25] Arie van Deursen, Paul Klint, and Joost Visser. 2000. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Notices* 35, 6 (2000).

[26] Riemer van Rozen. 2021. Languages of Games and Play: A Systematic Mapping Study. *ACM Comput. Surv.* 53, 6 (2021).

[27] Riemer van Rozen. 2023. Cascade: A Meta-Language for Change, Cause and Effect. In *Software Language Engineering, SLE 2023.* ACM.

[28] Riemer van Rozen and Joris Dormans. 2014. Adapting Game Mechanics with Micro-Machinations. In *Foundations of Digital Games.* SASDG.

[29] Markus Völter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. Towards User-Friendly Projectional Editors. In *Software Language Engineering, SLE 2014 (LNCS)*, Vol. 8706. Springer.

[30] Jos Warmer and Anneke Kleppe. 2022. Freon: An Open Web Native Language Workbench. In *Software Language Engineering.* ACM.

[31] Brent B Welch, Ken Jones, and Jeffrey Hobbs. 2003. *Practical programming in Tcl and Tk.* Prentice Hall Professional.

[32] Meng Zhu and Alf Inge Wang. 2020. Model-driven Game Development: A Literature Review. *ACM Comput. Surv.* 52, 6 (2020).

[33] Meng Zhu, Alf Inge Wang, and Hallvard Trætteberg. 2016. Engine-Cooperative Game Modeling (ECGM): Bridge Model-Driven Game Development and Game Engine Tool-chains. In *Advances in Computer Entertainment Technology, ACE 2016.* ACM.

Received 2023-07-17; accepted 2023-08-07

[4] https://github.com/usethesource/salix