

Live Game Design: Prototyping at the Speed of Play

Riemer van Rozen

rozen@cwi.nl

Centrum Wiskunde & Informatica

Amsterdam, The Netherlands

Abstract

Automated Game Design empowers game designers with languages, techniques and tools that automate iterative design processes. However, these tools currently lack suitable input and feedback mechanisms for creating rules and perceiving how changes affect running game prototypes. As a result, iterating takes too long, forming mental models about cause-and-effect relationships is difficult, and learning how to program can be tedious and frustrating. We investigate how Live Programming can accelerate game design iterations, make visual tools more accessible and engaging, and provide immediate feedback that brings code to life. We propose Live Game Design, a novel approach for rapid game prototyping that introduces mini-cycles to help designers of all skill levels explore, learn, and see a prototype come alive. We introduce Vie (pronounced /vi/), a game-making game for simultaneously prototyping and playtesting simple 2D games using Machinations. In an observational study, we evaluate the app during a Game-Based Learning tutorial for children aged 8 to 14. Our results show Vie is accessible to novices and Live Game Design enables prototyping at the speed of play.

CCS Concepts

• **Software and its engineering** → **Domain specific languages; Integrated and visual development environments**; • **Applied computing** → **Computer games**.

Keywords

automated game design, live programming, game mechanics, mixed-initiative design, prototyping, playtesting, game-based learning

1 Introduction

Game development depends on game design, an inherently complex, iterative process focused on improving a game’s quality [35]. At the core of a game’s design are game mechanics, the interactive rules that bring about *gameplay*, experiences such as enjoyment and learning. These rules often describe game-economic mechanisms that determine how players can collect, spend and exchange in-game resources such as coins, crystals or apples [1]. Game designers rely on rapid prototyping for effective playtesting, often using “cardboard prototypes”, to quickly improve the gameplay [11]. For designers, the challenge is to deliver high-quality results within restricted development timelines and limited budgets.

Automated Game Design (AGD) studies how to empower designers with Domain-Specific Languages (DSLs), techniques and tools that help automate game design processes [6, 45]. Various solutions have been proposed for improving a game’s parts, e.g., mechanics [18], levels [21], missions, stories, and virtual worlds. Mixed-initiative approaches leverage state-of-the-art algorithms to help explore design spaces and generate content procedurally [19].

Design experts can leverage these tools to express rules, balance strategies, and predict behaviors with mathematical precision.

Unfortunately, the usability of prototyping tools still leaves much to be desired. Unlike cardboard prototypes, software prototypes are not tangible, easily adjustable, or immediately playable. Many tool formalisms are difficult to learn and hard to master, especially for novices. The large abstraction gap prevents users from intuitively grasping how the rules work. Translating complex designs into working game systems is therefore time-consuming and error-prone, and often leads to loss of design intent. As designs evolve, they rapidly grow too large to comprehend. As Raph Koster once put it: “*a game design should fit on the back of a napkin*” [17].

Accessible design tools, referred to as *casual creators*, have introduced specialized notations for rapidly exploring particular design spaces [4], e.g., board games [7], 2D physics games [12, 18], and procedural imagery and poetry [5]. However, tools that support continuous prototyping and playtesting are not yet available [7, 45]. Modifying rules typically requires recompiling and restarting the game, resulting in the loss of valuable game states. Existing prototyping tools lack suitable input and feedback mechanisms for making gradual changes, perceiving behavioral effects, and evaluating their impact on running game prototypes. As a result, iterating is slow, forming mental models about cause-and-effect relationships is hard, and learning how to program can be tedious and frustrating.

We investigate how Live Programming can accelerate game design iterations, make visual tools more accessible and engaging, and provide immediate feedback that brings code to life [30, 39]. Our focus is on Machinations, a visual language for expressing game economies that has been well-studied, validated and applied in academia, industry and education [1, 9]. We employ design research, an iterative research method for advancing the state of the art, validating theories, and evaluating practical solutions with its users [13]. We propose Live Game Design, a novel approach for rapid game prototyping that introduces mini-cycles to help designers of all skill levels explore, learn, and see a prototype come alive. We introduce Vie (pronounced /vi/), a game-making game for simultaneously prototyping and playtesting simple 2D games using Machinations. Vie is powered by Cascade, a meta-language for change, cause and effect [46]. It builds on a prior evaluation of Godot for creating visual live programming environments [47]. To evaluate the app, we conduct an observational study that applies Vie in a Game-Based Learning tutorial for children aged 8 to 14. Our results show Vie is accessible to novices and Live Game Design enables prototyping at the speed of play. This paper contributes:

- (1) Live Game Design, a novel approach and a set of tool design principles for continuous prototyping and playtesting.
- (2) Vie: a live game design tool for simultaneous prototyping and playtesting simple 2D games using Machinations.

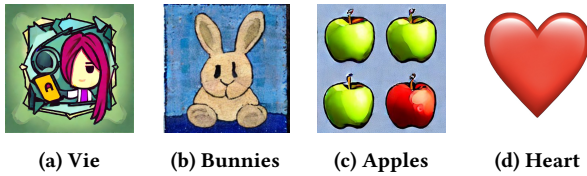


Figure 1: Game elements

2 Machinations

To introduce Machinations, we begin with its historical context, and a tutorial suitable for non-programmers, novices, and children. This tutorial serves as a simple motivating example that illustrates the main challenges we aim to address with Live Game Design. In Section 3, we will relate the needs of designers to challenges and objectives. Section 8 discusses related work in a broader context.

2.1 Background

Machinations is a visual notation and a conceptual game design aid [1, 9]. By foregrounding elemental feedback loops associated with emergent gameplay, designers can prototype how mechanisms work before a game is built. Over the years, Machinations has been applied in research, education and practice, resulting in university courses and industrial case studies. Dormans' initial tool from 2009, which was Flash-based, has been highly prized for its usability, and can still be found on online forums [10]. In 2020, start-up machinations.io reports over 23K users for its cloud service [40].

Micro-Machinations is a programming language that addresses key technical shortcomings of its evolutionary predecessor. For over a decade, we have collaborated with Dutch indie game developers on tools and techniques for game development [50], predictive analysis [16] and content generation [44]. Micro-Machinations has introduced a live programming approach that accelerates game design processes [50]. This approach proposes an embeddable interpreter to power the internal economies of digital games. Designers can simultaneously prototype and playtest mechanisms inside running games. In our most recent efforts, we investigate how game engine technology can be leveraged to create visual Domain-Specific Languages for live and exploratory programming [47]. Vie, the tool described in this paper, continues this branch of research.

2.2 Tutorial

We introduce Vie using a tutorial suitable for novices and children. To ensure no prior coding skills are needed, we use *marble machines* as a game design metaphor. In this metaphor, a game designer creates the marble machine, and decides where the marbles can roll. By introducing places, paths, chutes, and levers, the designer offers the player mechanisms to collect, spend and trade marbles.

There are two assignments. The first introduces design principles step by step, demonstrating how Live Game Design works through reproducible mini-iterations. The second is free-form, encouraging self-exploration, similar to a rapid game jam [12]. Instead of a live demo, we provide the English version of the handout [48]. In Section 6, we will revisit this tutorial in an observational study.

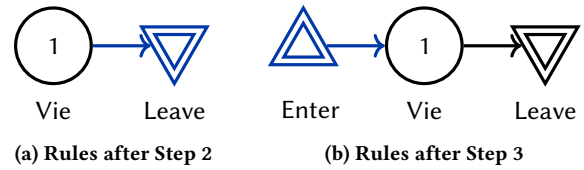


Figure 2: The first versions of the rules

Live Game Design: You Make the Rules

Designing games is difficult and time-consuming. To speed up and simplify the design process, we have developed the Vie app [49]. Vie is a visual programming language for quickly creating 2D game prototypes. After a brief introduction, you can create your own game rules and instantly see your ideas come to life. To get you started, you receive assignments and a handy cheat sheet (Appendix A). There are two assignments, an easy one and a difficult one. We use the Machinations language to express the rules [1]. Using its notation, we create a *marble machine* that works inside a game.

Assignment 1: Vie and her Bunny

We begin with an example about a girl who lost her bunny. Her name is Vie, just like the app. Figure 1 shows images: Vie, Bunnies, Apples and Heart. Step by step, we will use these elements in the game's design. You can play the game immediately.

Step 1: Vie joins in the game

Step 1.1. Goal. We begin by adding Vie to the game.

Action. Start the Vie app. Below Machinations (top left) you can see a circle. Drag the circle and drop it on the center of the screen.

Result. There is now a circle on the screen. In Machinations, this is called a *pool*. It is a place where resources (marbles) can be stored. If there is a marble inside, it means Vie is there.

Step 1.2. Goal. We will make sure that there is just one Vie.

Action. You can use the Node Editor (bottom left) to adjust the pool. Set the values of the At and Max fields to 1.

Result. When you click in the UI Design or Game tabs, you will see that a Sprite (a picture) and a Label (name and value) have been added. When Vie is there, you will see her picture.

Step 2: Vie leaves for a moment

Step 2.1. Goal. We add a rule called Leave for "going away".

Action. Below Machinations (top left) you can also see a triangle that points down. Drag and drop the triangle to the right of Vie. In the Node Editor (bottom left), set the value of When to "User". The Leave rule will become an interactive button in the game.

Result. There is now a triangle on the screen called a *drain*. This is a place where marbles can disappear. From its double lines, you can tell that Leave is an interactive game element.

Step 2.2. Goal. Marbles need a path to roll along. We will add a connection so Vie can leave.

Action. Click on the dot on the right of the Vie pool, and connect the line with the dot on the left of the Leave drain.

Result. There is now a line between Vie and Leave. This is called a *resource connection*. The rules now appear like Figure 2a. We have added the blue elements in Step 2. A Leave Button has also been

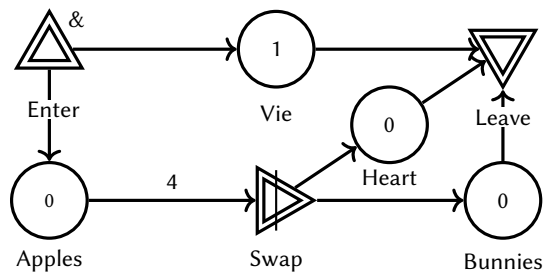


Figure 3: A complete version of the rules

added. If you click that button in the UI Design or Game tabs, Vie will go away. You can try it out.

Step 3: Vie returns on stage

Step 3.1. Goal. We will add an Enter rule for “returning”.

Action. Below Machinations (top left) there is also a triangle that points up. Drag this triangle and drop it to the left of Vie. In the Node Editor, set When to “User”. This also adds a button.

Result. There is now a second triangle on the screen. This is called a *source*, a place where marbles can come from. From its double lines, you can tell Enter is an interactive source.

Step 3.2. Goal. We will add a connection that lets Vie return.

Action. Click on the dot on the right side of the Enter source, and connect the line with the dot left of the Vie pool.

Result. There is now a line between Enter and Vie, another resource connection. The rules now appear as in Figure 2b. We have added the blue elements in Step 3. If you click on the Enter button in the UI Design or Game tabs, Vie will return. Try it out.

Step 4: Vie always brings an apple

Step 4.1. Goal. We will now add Apples to the game.

Action. We again start with Machinations (top left). Drag another circle to a good spot on the screen, e.g., below Enter.

Result. There is now a second pool on the screen. If there are marbles in this spot, these are Apples. A Sprite and a Label have been added in the UI Design and Game tabs.

Step 4.2. Goal. Every time Vie returns, she brings an apple.

Action. Click on the dot on the right side of the Enter source, and connect the line with the dot on the left side of the Apples pool. Next, click on the Enter source. In the Node Editor, set the value of How to “All” to ensure Vie only brings an Apple when she returns.

Result. There is now a resource connection between Enter and Apples. If you now click on the Enter button in the UI Design or Game tabs, then Vie brings an apple when she returns. Try it out.

Step 5: Vie swaps the apples for her bunny

Luckily, Vie can get her bunny back for exactly four apples.

Step 5.1. Goal. We will add Bunnies to the game.

Action. We again start with Machinations (top left). Drag another circle and drop it in suitable place on the screen. In the Node Editor, adjust the value of Max to 1 so there can only be one bunny.

Result. There is another pool on the screen. When there are marbles inside, these are Bunnies. In the UI Design and Game tabs, a Sprite and a Label have been added to see them.

Step 5.2. Goal. We will add a Swap rule and a Swap button.

Action. Below Machinations you can also see a triangle pointing to the right with a vertical line through it. Drag it to a suitable spot.

Result. There is now a new element on the screen. Swap is a converter. Converters can exchange one kind of resource (marbles) for another kind. Because converters are interactive, a Button has also been added in the UI Design and Game tabs that activates it.

Step 5.3. Goal. We add that the bunny costs four apples.

Action. Add a resource connection between Apples and Swap. Click just above the connection, and enter amount 4.

Result. The new source connection indicates that the cost of trading is 4 apples. However, swapping has no benefit yet.

Step 5.4. Goal. We add that for swapping we get the bunny.

Action. Add a connection between Swap and Bunnies.

Result. Now we can exchange apples for the bunny. Click on the Swap converter, or on the Swap button in the UI Design or Game tabs. If you have enough apples you will get the bunny.

Step 6: Happily ever after – or something else!

A game is never completely finished, not even Figure 3. Add rules yourself and try them out. Let a Heart appear, or make up rules about Bananas or Cows. Maybe they’re hungry!

Assignment 2: Climate Change

You can also design complex games with Vie. In this exercise you will make an educational game about climate change.

Goal. Design a game using the elements: Heat, Factories, Trees and CO₂. What can you do to prevent global warming?

Action. Click Menu, select theme “The Climate” and click Done. Design rules and try them out. Tip: discuss and play!

Example. You can also load an example. Click Menu, select “2. The Climate”, and click Load. This is not a very fun game yet. Can you make better rules for improving it?

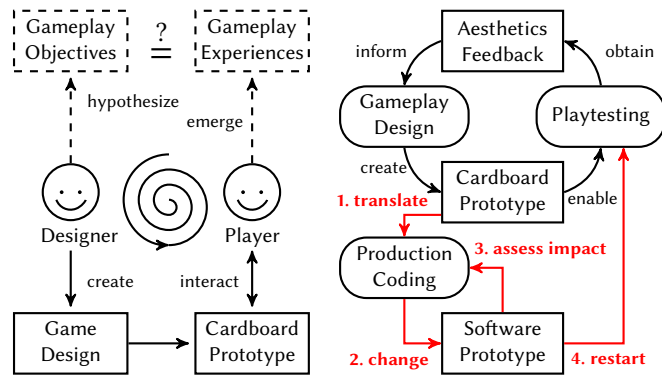
3 Problem Overview

We relate the needs of novice game designers to research challenges and opportunities. To shed light on the problem, we examine research areas with distinct perspectives on how related challenges, objectives and approaches can contribute to a generic solution.

3.1 Automated Game Design

Automated Game Design is a research area that studies how to empower game designers with languages, techniques and tools that offer an increased expressive power over a game’s mechanisms. Using these tools, design experts can create software prototypes and obtain feedback with mathematical accuracy [6, 25, 45].

However, Automated Game Design is no silver bullet for creating better games more quickly. Cardboard prototyping and software prototyping are usually nothing alike. Creating software prototypes adds technical complexity. As a result, the design iterations take too much time. The main reasons, illustrated by Figure 4, are:



(a) Cardboard prototyping is fast (b) Software prototyping is precise but lacks mathematical accuracy but design iterations are too slow

Figure 4: Game design requires prototyping and playtesting

- (1) **Abstraction gap.** Designers need to be able to express a game’s mechanisms independently. However, the available formalisms are too difficult to learn and understand. As a result, translating designs into code is complex, time-consuming and error-prone.
- (2) **Lack of continuous change.** Improving a prototype requires constantly changing its mechanisms. However, changing how the mechanisms of a running prototype work simply is not possible. Many changes require updates to the user interface to again make the prototype playable, which requires a restart.
- (3) **Lack of immediate feedback.** Designers need immediate feedback to form mental models, and learn cause and effect relationships. However, they lack a means for assessing the behavioral impact of changes on running prototypes.
- (4) **Interrupted playtesting.** The playtesting process halts every time a digital prototype needs to be recompiled and restarted. Designers lose the valuable run-time state in the process.

In this paper, we aim to address these challenges. We introduce Vie, a tool for prototyping and playtesting simultaneously. Our approach combines Game-Based Learning with Live Programming.

3.2 Game-Based Learning

Game-Based Learning is an area that studies how to leverage games to teach subject matter [29]. Using applied games (or serious games), educators can offer gamified experiences that help learners acquire skills in a fun and exploratory manner [28]. Like game design itself, learning how to create digital game prototypes is an iterative process that hinges on feedback [35]. Facilitating the learning process is complex because dedicated teaching tools are still mostly missing. In particular, the following challenges remain unaddressed.

- (1) **Game design metaphor.** Designers, like novices and children, are often non-programmers who lack mental models about the relationships between code and play. Acquiring new skills is difficult due to a lack of suitable abstractions that appeal to the imagination. For bridging the abstraction gap (Challenge 1), learners need a powerful metaphor for game design.
- (2) **Educational programming environments.** To acquire new skills, learners need to engage with the subject matter. However, educators lack a means to create design spaces learners

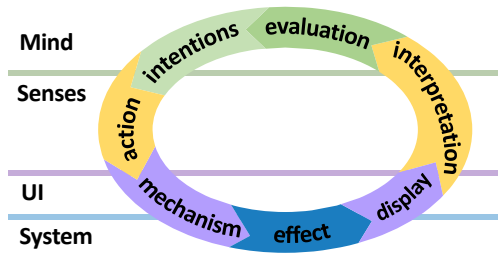


Figure 5: Feedback loops drive Human-Computer Interaction (adapted from van Rozen [46])

can safely explore. They lack tools for creating interactive tutorials and coding exercises for learning with trial and error.

In this paper, we investigate how to create a game-making game that addresses these challenges. Our objective is to support educators and learners with an educational programmable environment for creating interactive tutorials and exploring programmable design spaces. The tutorial of Section 2 exemplifies our objectives and introduces marble machines as a game design metaphor. It illustrates a style of programming that explicitly relates gameplay goals to coding actions, and immediately playable results. Each step is a mini-cycle that offers learners opportunities to learn, construct mental models, and see the subject matter come alive. To introduce these cycles, we investigate how to leverage Live Programming.

3.3 Live Programming

Live Programming caters to the needs of programmers by providing immediate feedback about the effect of changes to the code [30, 39]. Our objective is to combine Live Programming with Automated Game Design in a novel approach called Live Game Design. The tutorial scenario of Section 2 illustrates key requirements a live programming environment must fulfill to cater to a designer’s needs.

Like all of Human-Computer Interaction, programming is driven by feedback loops [27]. Figure 5 illustrates this. However, prototyping and playtesting are two distinct but interdependent loops. We aim to integrate them in a single programming environment for:

- (1) **Prototyping.** Realize intentions by interacting with: a) *input mechanisms* for performing the effects of coding actions; and b) *feedback mechanisms* that display changes for perceiving those effects, and evaluating if actions were successful.
- (2) **Playtesting.** Assess gameplay with input and feedback mechanisms for performing and evaluating player actions.

Next, we will describe Live Game Design, our solution proposal for addressing these challenges.

4 Live Game Design

We propose Live Game Design, a combination of Automated Game Design and Live Programming that empowers designers with languages and tools for prototyping and playtesting simultaneously. The main idea behind our approach is seamlessly integrating these activities and accelerating the feedback loops. Figure 6 illustrates how both loops are integrated into a single design environment.

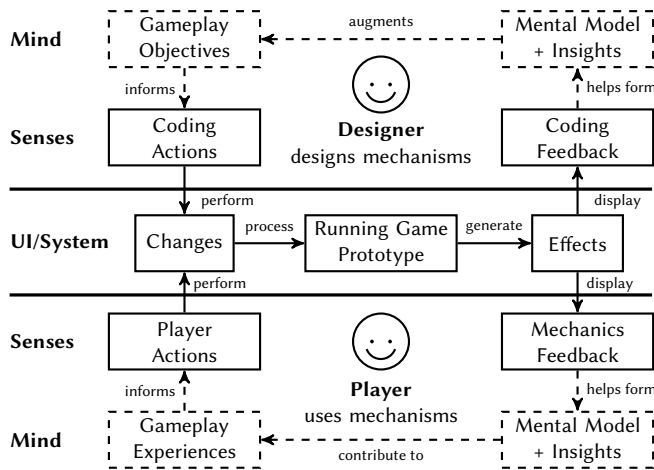


Figure 6: Live Game Design integrates prototyping (top) and playtesting (bottom) cycles with input and feedback mechanisms that provide immediate feedback about the incremental effects of coding and player actions on each other

4.1 Design Principles

In a Live Game Design process, each step is a mini-cycle designed to explore, learn, and see a prototype come alive. To introduce these mini-cycles, we propose principles of Live Game Design. We follow the design research methodology [13], informed by the state-of-the-art in game design tools [45], and insights from Cognitive Load Theory [24, 43]. In Sections 5 and 6, we iteratively validate, evaluate, and refine the descriptions. We propose the following principles:

- P1 **Observable effect.** An action results in observable effects and yields informative feedback whenever possible.
- P2 **Immediate feedback.** Feedback on an action is always immediate to preserve the integrity of a mini-cycle.
- P3 **Learnable meaning.** Every action has a predictable and ultimately learnable effect governed by the language semantics.
- P4 **Meaningful change.** Every interaction, for play or design, results in a meaningful change that feeds into the next cycles.
- P5 **Continuous processes.** Design activities, such as prototyping and playtesting, are continuous and uninterrupted processes.
- P6 **Universal playability.** Designs and prototypes are always playable. Instead of compile- or run-time, *now* is the only time.
- P7 **Minimalist design.** Tools offer a minimal set of elements and features that focus activities and reduce the cognitive load.
- P8 **Ongoing discovery.** Uninformed actions result in effects and feedback that support learning. There is no right or wrong.
- P9 **Moldable design.** Changes to a game design are free-form. The results are sculptable and moldable artistic expressions.
- P10 **Formal semantics.** Domain-Specific Languages (DSLs) span well-defined game design spaces that can freely be explored.
- P11 **Generic tool.** A generic tool supports making changes for exploring these design spaces in *any* direction.
- P12 **Specific themes.** Specific design themes that focus the design space exploration are a means to introduce subject matter.

P13 **Captivating examples.** Educational design themes empower novices and learners with recognizable examples that help them engage, e.g., a lost bunny or climate change.

Next, we design a tool to validate and evaluate of these principles.

5 Vie: A Tool for Live Game Design

We introduce Vie (pronounced /vi/), a novel tool for creating 2D game prototypes using Machinations. We formulate requirements, explain the main design decisions, and describe its implementation.

5.1 Requirements

Based on prior studies, workshops, and experiences, we formulate the following functional requirements. These extend an earlier version that appeared as part of a pilot study, a positive suitability analysis of Godot for creating visual programming environments [47].

5.1.1 Game Mechanics. Designers need to prototype game mechanics before assessing the gameplay. They require an editor to flexibly design, modify, and test the dynamics of game-economic mechanisms. The editor offers a dedicated view that lets designers:

- R1 Create diagrams by adding nodes and edges on a canvas, by moving elements, and zooming in and out.
- R2 Modify and edit node properties: name, type, behavior modifiers (when, act, how), and for pools only: its starting (start) and maximum (max) amounts.
- R3 Modify the flow rate of a resource connection by modifying its expression. The default (no expression) is one. Adjustments enable whole amounts (4), pool references (Apples), and composite clauses with multiplication, division, addition and subtraction ((Apples*2)-1).
- R5 Observe visual feedback about the success of nodes triggers and the flow of resources in a diagram.
- R6 Activate an interactive node to evaluate its effects.

5.1.2 UI Design. Designers need simple 2D user interfaces to make the mechanics playable. They require an editor and a minimal set of visual components for associating mechanisms with UI elements. The editor enables them to design, place and move:

- R7 Buttons that display a text and activate an interactive node.
- R8 Labels that display a specified text and enable observing the current amount of resources inside a pool node.
- R9 Sprites that display an image indicating the current amount of resources inside a pool node.

5.1.3 Interactive Game Simulation. For playtesting, designers and players need a running game prototype. They need game simulations that project the UI design, and enable them to:

- R10 Press buttons for activating mechanisms.
- R11 Perceive labels that show amounts textually.
- R12 Perceive sprites that represent the game state visually.

5.1.4 Educational Themes. Educators require a means to introduce subject matter, such as the lost bunny or climate change, into a game's design. To set educational parameters, and to help focus exploratory design processes, educators need to:

- R13 Create educational design themes that associate specified verbs to the names of pools, sources drains and converters.

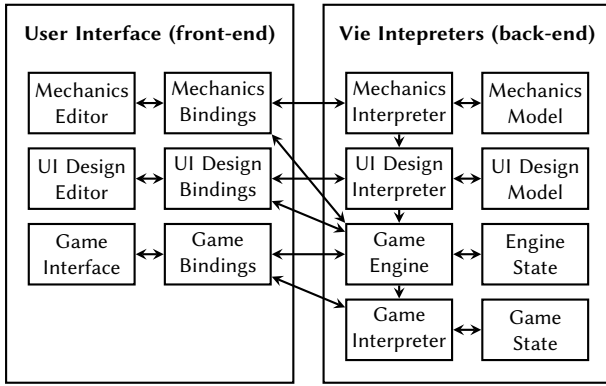


Figure 7: Vie: tool architecture and components

R14 Integrate a theme with the editors for creating mechanisms.

In addition, we formulate a non-functional requirement.

R15 Actions shall have an immediate effect. A response time of 100ms is regarded as instantaneous in HCI research [26].

Next, we explain how the language design realizes these functions.

5.2 Tool Architecture

The architecture of Vie adheres to the Model View Controller (MVC) paradigm. Figure 7 shows an overview of its components. The arrows denote event-driven communication between programs (Model), UI components (View) and the interpreters (Controller). Vie integrates three interconnected Domain-Specific Languages.

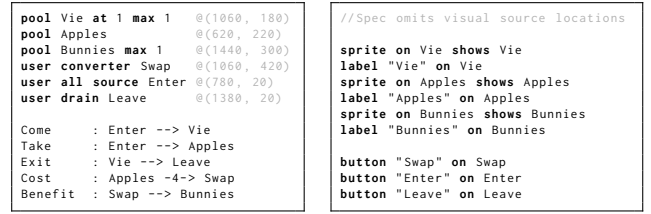
- (1) **Machinations**: expresses internal game economies. Figure 8a illustrates the textual storage format, and shows Step 5 of the tutorial. The Mechanics Editor displays programs visually.
- (2) **UI Design**: expresses visual 2D elements of a game. Figure 8b shows the generated specification for the tutorial. Users edit UI designs visually with the UI Design Editor.
- (3) **Design Themes**: specify verbs and nouns that determine the names of Machinations nodes, and focus the design process. Figure 9 shows specifications of the lost bunny and the climate change themes of the tutorial. These are omitted in the diagram because an editor is still missing.

In addition, the Game Simulation displays a running game, enabling players to click buttons. In prior work, we have addressed the challenge of creating a user interface [47]. Here we address the challenge of supporting the principles of Live Game Design.

To introduce mini-cycles, we design and integrate so-called Read-Eval-Print-Loop (REPL) interpreters. We create these using the Cascade Meta-Language [46] and the C# version of the Godot game engine [22]. Vie integrates four back-end interpreters:

- (1) **Mechanics Interpreter**: evaluates changes to Machinations.
- (2) **UI Design Interpreter**: evaluates changes to UI Designs.
- (3) **Game Engine**: evaluates Machinations behaviors and runs a single game simulation.
- (4) **Game Interpreter**: maintains a game simulation view.

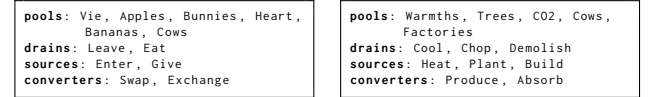
Cascade expresses DSLs and run-time transformations, and compiles to C#. The event-based runtime serves as a controller that manages the abstract syntax and the run-time states. The C# interpreters have an event-driven design. They offer a scheduling



(a) Machinations after Step 5

(b) Generated UI Design spec

Figure 8: Domain-Specific Language storage formats



(a) Lost Bunny theme

(b) Climate Change theme

Figure 9: Domain-Specific Language for game design themes

API for making gradual changes, e.g., for adding nodes or edges, deleting them, or changing the type of a node. Next, we describe how we introduce mini-cycles into the interpreters.

5.3 Language Design

The language design of Vie is based on the meta-models of Figures 10 and 11. These UML class diagrams show the structure of the abstract syntax and the run-time states. Machinations programs, or Abstract Syntax Graphs (ASGs), are instances of the static meta-model of Figure 10a. Run-time states, on the other hand, are instances of the run-time meta-model of Figure 10b. UI designs are instances of the static meta-model of Figure 11a. Game simulations are instances of the run-time meta-model of Figure 11b.

5.3.1 Mini-cycles. The language design of Vie introduces mini-cycles that offer designers opportunities to learn, construct mental models, and see the subject matter come alive. When a designer modifies 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. The views update in response. Next we describe the inter-dependent mini-cycles.

5.3.2 Mechanics Editor. Users can drag, drop, and reposition a node from a palette onto a canvas whose type is pool, source, drain or converter. Adding resource connections between nodes requires connecting dots on the node boundaries (left is input and right is output). Using the node editor to change the properties of a selected node also updates its visual appearance and behavior.

5.3.3 Mechanics implement design themes. The currently loaded game design theme determines which name a node receives. Every time a name is used, the editor moves to the next one in the list. This saves the designer the trouble of typing them in manually.

5.3.4 Mechanics remain intact. The Mechanics Editor maintains the program integrity. Deleting a node also deletes all the edges connected to it. Changing a node name only works when the input is of type string and not duplicate. Otherwise, the text becomes red.

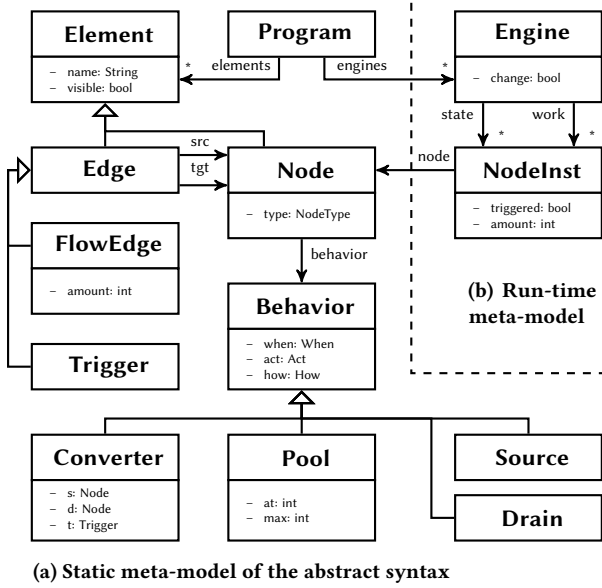


Figure 10: Meta-model of Micro-Machinations (appears in van Rozen [47])

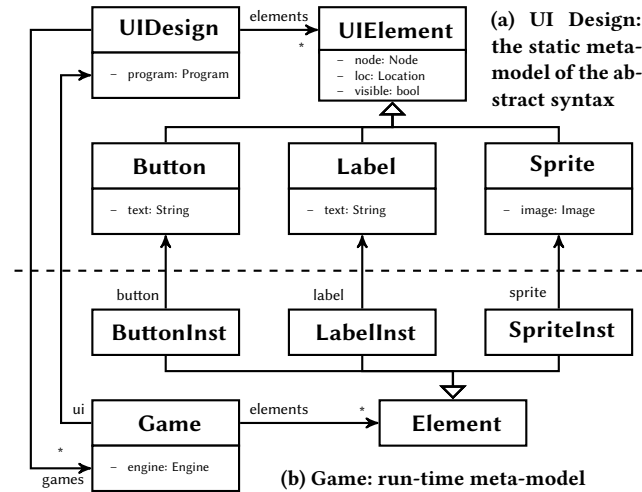


Figure 11: Meta-model of Vie: UI Design and Game packages

Edge expressions can be modified by clicking just above an edge. This reveals an input text field. Changing an edge expression only works if the input parses, and all variable references can be resolved. Otherwise, a tooltip shows the error, and the text becomes red.

5.3.5 Mechanics integrate a running simulation. One game simulation always runs. We introduce the following mini-cycles between the mechanics and the engine. Creating a pool gives it a current amount; and changing the starting amount updates the current amount. Deleting a pool removes its current amount.

Users can activate interactive nodes by clicking on them. The diagram shows visual feedback when nodes: a) activate (blink yellow); b) succeed (blink green); c) fail (blink red); d) update pool amounts (display current amount); and e) redistribute resources along the resource connections of the diagram (edges blink green).

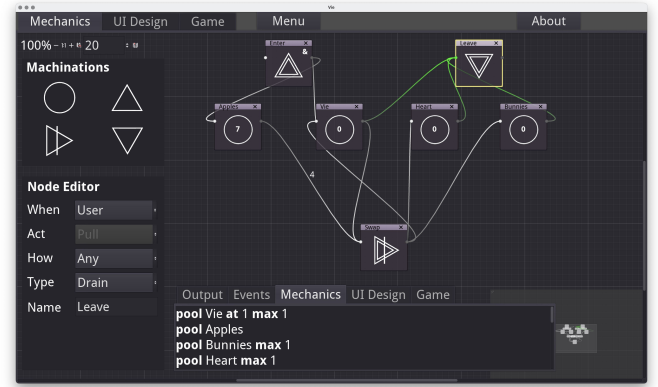


Figure 12: Mechanics Editor displaying the Lost Bunny example, showing feedback after the Leave drain is activated

5.3.6 Mechanics make the UI design playable. We introduce the following mini-cycles between the mechanics and the UI design. Creating a pool node also creates a Sprite and a Label. Making a node interactive also adds a Button. To reduce the perceived distance, these elements are positioned in the UI design at the same visual coordinates as their nodes in the mechanics view.

Deleting an interactive node (or changing its when modifier) severs the connections with Buttons that refer to it. Deleting a pool node severs connections with Labels and Sprites.

5.3.7 UI Design. Users can drag and drop a node from a palette onto a canvas whose type is Button, Label or Sprite. Using an editor on the left, they can modify the: a) associated interactive node of a Button; b) associated pool node of a Label or Sprite; or c) text of a Label; or toggle the visibility of an element in the game simulation.

5.3.8 UI Design integrates a running simulation. Just like the mechanics, the UI design projects the running simulation. Pressing a button activates its interactive node. Labels and sprites update, showing a pool's name and current amount textually, or visually.

5.3.9 Game simulation. The game simulation projects the UI design, but is not an editor. Creating, editing and repositioning elements in the UI design view updates the running game. Changing the game mechanics updates the behavior.

The game engine always runs one game simulation. When a user activates an interactive node, the engine evaluates the program until the fixpoint computation completes. This computation generates the run-time events shown in the other views.

5.4 Implementation

The implementation of Vie consists of three main parts: the front-end editors, the back-end interpreters, and the bindings between their APIs. The front-end includes the Mechanics Editor, the UI Design Editor and the Game Simulator. Figures 12, 13 and 14 illustrate these integrated views. To indicate the implementation effort, we measure the code volume in Source Lines of Code (SLOC) using cloc [8]. We created the editor scenes using Godot. The total volume of the tscn files is 2852 SLOC. The back-end interpreters are compiled from a Cascade specification of 2605 SLOC. The resulting

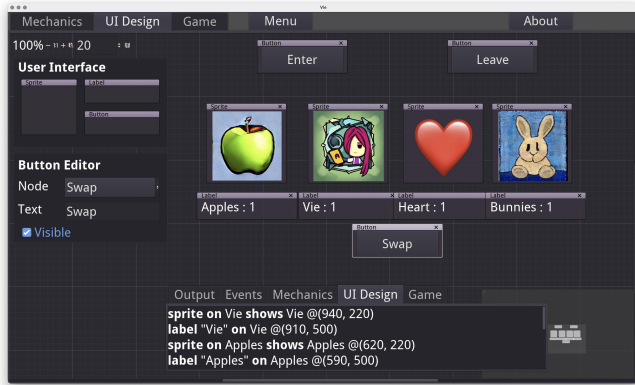


Figure 13: UI Design Editor showing the game state

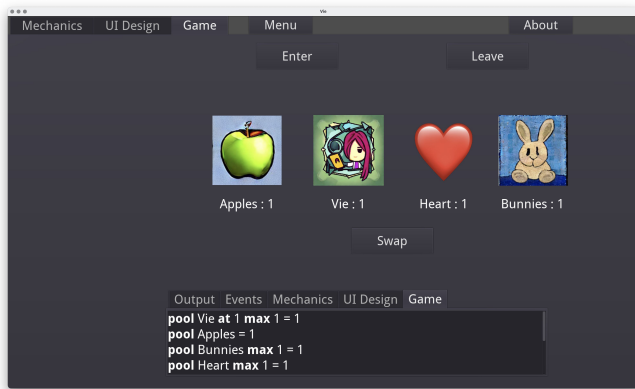


Figure 14: Game Simulator projecting the running game

engine is mainly generated, and consists of 32.9 KSLOC of C# code. The bindings, written by hand, consist of 3330 SLOC of C#.

The Vie app (v0.0.7) is available for Windows, macOS and Linux under the 2-clause BSD license [49]. Additionally, iOS and Android versions are planned. We have tested the prototype, and it works well on personal computers, laptops, tablets and mobile phones.

6 Observational Study

To evaluate Vie and assess the design principles we have formulated in Section 4.1, we conduct an informal observational study. The aim of the study is to perform an initial qualitative analysis about how novices interact with Vie. Our objectives are to: 1) report observations; 2) reflect on successes and failures; 3) gain insights; and 4) make improvements. The results inform future design research.

6.1 Setup

We report on a workshop for children organized at the open day of Centrum Wiskunde & Informatica on two occasions. The workshop took place between 12:00 and 17:00, continually offering sessions of approximately 30 minutes. The goal was to convey this main message: “programming is fun, can be visual too, and is for everyone.”

6.1.1 Focus group. Intended for Dutch kids between 8 and 14, the workshop also welcomed younger kids and parents. Participants were exceptionally talented, not a general subset of the population.

6.1.2 Activities. Using Vie, participants engaged in live and exploratory programming. After a brief introduction (10 minutes), they received a demo (5 minutes), and tried the tutorial of Section 2. We encouraged free exploration and having fun, like in a game jam.

6.1.3 Results. On October 7th 2023 and October 5th 2024, we counted 30 and 60 participants. We made notes, but for privacy reasons, we did not store programs. Next, we report our observations.

6.2 Observations

We have made the following noteworthy observations.

6.2.1 Beginners. Many kids experimented very quickly, filling the screen with symbols in the blink of an eye. When asked about the ideas behind the rules, their explanation was typically incomplete. The nodes were all connected by resource connections, but the rules only partially worked. Discussing the design, e.g., by explaining that marbles can never flow from a drain, typically helped.

6.2.2 Prior experience. On both occasions, we observed several kids could quickly create rules and play with them. When asked about their prior experiences, most reported starting high school and using Scratch [32]. They had a noticeable advantage.

6.2.3 Young children. On both occasions, we noticed several very young participants, no more than four or five years old, with no prior programming experience. One boy used a wheel mouse for the first time. His father showed him how. Very determined, he managed to create working rules. For the youngest ones, simply getting the bunny to appear created a moment of pride and achievement.

6.2.4 Engaging design themes. We observed several groups having vivid discussions among themselves about climate change. In three cases, we observed girls changing the name of Vie into their own name. When we asked a participant what her aim was, she explained she wanted to move the image between places, and that this did not work. We could help realize this. What prevented movement was that pools are variables that cannot have identical names.

6.2.5 Saving and loading designs. Most of the participants played for at least half an hour. In three cases, participants continued for several hours, much to the frustration of their parents.

Only in these instances, where the designs represented significant effort, participants asked to save them for later. Not once did a participant ask why there was no compile or run button.

6.2.6 Exploring design spaces. In several instances, we observed participants reuse images for different purposes. For instance, two brothers, who were already in high school, created an applied game about health. Much to their enjoyment, they identified how to reuse the images of heart, cows and trees for health, burgers and pickles.

6.2.7 Prototyping and playtesting. Almost every participant frequently switched between views for prototyping and playtesting.

In addition, we have not observed any crashes or unexplainable behavior. However, based on our observations, we have gained new insights. We have also made several improvements and bug-fixes.

6.3 Reflections and Insights

We have gained the following insights about improvements.

6.3.1 Beginners. Novices lack a mental model of how the rules work. However, having no prior experience did not prevent them from exploring. Vie’s minimalist interface supports the process of discovery, in accordance with Principles P7, P8 and P9.

6.3.2 Engaging with subject matter. Based on our observations, we conclude the subject matter helped at least some participants engage, supporting the principles of specific themes (P12) and captivating examples (P13). However, for educators, identifying suitable examples still poses a significant challenge.

6.3.3 Abstraction. Our marble machine analogy has a flaw. Designers often need to abstract from resource locations and movement in game economies. As witnessed by the duplicate names, this is hard for novices. However, experts also find this difficult, and meaning can be experienced in different ways. Vie has gameplay too.

6.3.4 Learnable effects. With the proper guidance, the tool’s interface provides a usable starting point for beginners. Of course, feedback is essential for learning, but interpretation and evaluation require time and practice. Just like a good game, a game designed to create other games should be easy to learn and hard to master.

6.3.5 Continuous processes. The frequent switching between views indicates that prototyping and playtesting are seamless processes (Principle P5). Adding buttons, labels and sprites automatically is a particularly useful feature. The principles of continuous processes (P5) and universal playability (P6) seem to come naturally to novices. We hypothesize they are accustomed to using apps that simply always and immediately work, restoring the state automatically.

6.3.6 Universal playability. Automatically adding buttons, labels and sprites worked especially well for playability. However, in many cases rapid changes resulted in visual clutter. Because elements may represent conscious design decisions, Vie does not automatically clean them up. To address this issue we propose a *mental investment* design principle. Interactive elements may be automatically added for playability. These elements can outlive their usefulness when they lose their function, and be removed if never seen or edited.

6.3.7 Meaningful change. There are a few cases, specific to textual input, where we cannot avoid errors. For instance, we now clearly mark duplicate names locally in red. We observe the principles of observable (P1), immediate (P2) and learnable (P3) feedback. In addition, the game continues to function (P6). However, the solution is not perfect because it violates the principle of meaningful change (P4). Our observations suggest novices need guidance to realize meaningful change. Future versions of Vie could provide explanations and suggest coding actions.

6.3.8 Generic tool. We see evidence that Vie is a generic tool for exploring design spaces (Principles P10 and P11). This is witnessed by examples where novices reused images for designs outside of the boundaries of the themes. An important limitation is that Vie currently only offers a limited set of sprites. Leveraging generative AI could further improve the uses of Vie as a generic tool.

6.3.9 Usability improvements and bug-fixes. We added drag and drop functionality because clicking twice was not intuitive. In addition, the editor palettes now show icons instead of text. We fixed an incorrect boundary condition that could cause a pool to overflow.

7 Discussion

We discuss the costs and benefits of Live Game Design, reflect on the limitations and threats to validity of our work.

Applying Vie comes at a cost. Despite our best efforts to make programming easier, game design is intrinsically complex, and learning how to program remains difficult. For educators, creating design themes to guide explorations requires time and effort. A separate editor for creating them is not yet available.

Vie also has compelling benefits. Using Vie, designers can prototype and playtest simultaneously. Vie is accessible to novices. By introducing mini-cycles in its design, users receive immediate feedback about the impact of gradual changes on running software. This is fundamental for learning cause-and-effect relationships.

Although Machinations is already well-validated, the evaluation of Vie is still limited. The observational study is biased toward highly skilled participants. As a result, we cannot yet draw general conclusions. Further study will also require quantitative analyses.

Save and load functionality is currently limited to Machinations diagrams. Instead of loading the syntax directly, Vie distills coding actions. When the interpreter executes these, a playable UI design is generated automatically. Currently, modified UI designs cannot be stored. Vie is a generic design tool. However, it currently only offers a limited set of sprites. We do not yet leverage image generators.

The design principles we have proposed have provided useful guidelines for reasoning about Live Game Design. However, the qualities we have formulated are merely a solution proposal and a first step towards empirically evaluated theoretical foundations.

7.1 Challenges and Opportunities

We discuss open research challenges. Visual interfaces support correct modifications to the syntax. Textual interfaces allow for temporarily incorrect input until coding actions can be distilled. A key challenge is integrating these two coding styles.

Integrating generative AI into game design processes is another challenge. Prompt engineering is inherently unpredictable, which violates our design principles. However, the *chain of thought* aligns perfectly with the cause-and-effect chains that power Vie.

Tutorial generation is an unexplored area. A key challenge is how to generate exercises tailored towards an individual learner’s expertise and needs. Because Vie is a change-based programming environment, its built-in version history records every coding action, user interaction and side effect. This presents an opportunity for studying interaction patterns empirically.

8 Related Work

The contributions of this paper intersect at three main areas: Automated Game Design, Live Programming, and Educational Programming Environments for Game-Based Learning.

8.1 Automated Game Design

Various languages, techniques and tools have been proposed for creating, generating, analyzing and improving a game’s parts. Much of the work in this area has originated from the AI and games community [25, 41]. Design tools offer interactive user interfaces that support prototyping [25], sketching designs [21], automating play testing, and exploring design spaces. Early approaches were

logical formalisms [25, 36]. Ludi, an evolutionary game design system with a Lisp-like notation, famously generated the fun board game Yavalath [2]. Its successor Ludii has been used to study ancient games, generate puzzles, and support AI competitions [3].

Mixed-initiative approaches offer a conversational interaction style where users and computers take turns adjusting content [19], e.g., the level generation of Tanagra [37]. Persuasion and procedural rhetorics have also been influential, e.g., in the micro-rhetorics of Game-o-Matic [42], and the high-level constraints of Germi-nate [18]. Casual creators are tools that support an easy, pleasurable, and expressive design space exploration [4]. Tracery, for instance, has been used for procedural chat bots, poetry, and imagery [5]. PuzzleScript is an online engine and a textual programming language created specifically for puzzle game design [20].

Vevva is a *fluidic game*, moldable through play and design [12]. In rapid game jams, kids explore predefined design spaces. After setting visual design parameters, e.g., for physics and scoring, they press play to start the game. Vie shares its aims of accessible and rapid exploration, and reduced cognitive load.

Puck is a game design tool that integrates continuous creativity with an exhaustive form of procedural content generation [7]. Puck and Vie occupy different design spaces. In Puck, design revolves around generating 2D tile maps, and evaluating single and multi-player games. In Vie, generative responses to edits help ensure playability. Evaluation primarily involves manual playtesting.

The principles of Live Game Design introduce a new way of reasoning about and evaluating game design tools. Vie is the first tool that takes the form of a visual live programming environment. Accessible enough to be a casual creator, its mini-cycles focus on learning to code. Vie also aims for quality and productivity.

8.2 Live Programming

We identify three distinct types of Live Game Design. The first involves updating online games by hot-swapping components. The second features designing games in live, on-stage performances, which is akin to live coding. This may include recompilation and restarts. The third, proposed in this paper, is a form of Live Programming that enables simultaneous prototyping and playtesting without restarts, making it well-suited to support the other two.

Live Programming is a style of programming that revolves around continuous change and immediate feedback [30, 39]. Game engines usually integrate some liveness features in their editors [30], e.g., not requiring restarts on changes to scene graphs or variable values.

However, in academia the combination of games and Live Programming is a relatively new area [34, 50]. Until recently, a lack of enabling technology prevented developers from creating live programming environments that could migrate run-time states [47]. As a result, how to introduce inter-dependent mini-cycles has not been studied before. Moreover, the dimensions of live programming are not yet well understood [39]. The design principles we have formulated represent a first step towards a foundational theory. Vie is the first live programming environment of its kind.

8.3 Educational Programming Environments

Educational programming environments can trace their origins back to the dawn of personal computing, and the early history of

Smalltalk [15]. Logo is an educational programming language that is well-known for its “turtle”, which can be steered using commands for drawing vector graphics [38]. Turtle graphics offer a powerful metaphor for drawing. In this paper, we have identified another space. We propose marble machines as a metaphor for game design.

Scratch is a visual environment for creating, designing and remixing interactive stories, games, animations, and simulations [32]. Using Scratch, children between 6 and 12 years old learn creative thinking, logic and programming concepts. Scratch is a block-based language whose syntactic constructs fit together as puzzle pieces. Vie instead offers a domain-specific notation that is graph-based.

Kodu is a visual programming language for young children that supports learning by means of independent and playful exploration [23]. In Kodu, rules control robots in a real-time 3D gaming environment. Instead of having a fixed design theme, Vie introduces subject matter through programmable themes.

Agentsheets is a visual tool for creating agent-based games and simulations, also used for teaching game design [31]. Agents simulate a game and provide syntactic and semantic feedback. GameStar Mechanic is an RPG-style online game for teaching the fundamentals of game design to children aged 7 to 14 [33]. GameDevDojo is a visual programming environment that provides a game-based learning approach to teach foundational game development concepts, such as mechanics, gameplay, sprites, and levels [14]. Vie stands out as a visual live programming environment, designed specifically to speed up prototyping and bring the code to life.

9 Conclusion

In this paper we have proposed Live Game Design, a novel approach that combines Automated Game Design with Live Programming for prototyping and playtesting simultaneously. We have introduced Vie, a visual programming environment that enables quickly creating simple 2D games using Machinations. To evaluate the app, we have conducted an observational study on a tutorial for children aged 8 to 14. Our results show that Vie is accessible to novices and that Live Game Design enables prototyping at the speed of play.

Acknowledgments

We thank the workshop participants, and Daria (Dasha) Protsenko for co-hosting the second workshop. We also thank our partners in the Live Game Design RAAK-MKB project, funded by NWO/SIA from 2016 to 2019. Special thanks to Joris Dormans (Ludomotion), Loren Roosendaal (Knowingo), Paul Brinkkemper (Money Maker), and Anders Bouwer (Amsterdam University of Applied Sciences) for their continued collaboration and support over the years. Finally, we thank the anonymous reviewers for their feedback and suggestions, which helped improve this paper.









References

- [1] Ernest Adams and Joris Dormans. 2012. *Game Mechanics: Advanced Game Design*. New Riders.
- [2] Cameron Browne. 2012. Yavalath: Sample Chapter from Evolutionary Game Design. *J. Int. Comput. Games Assoc.* 35, 1 (2012).
- [3] Cameron Browne, Matthew Stephenson, Éric Piette, and Dennis J. N. J. Soemers. 2019. A Practical Introduction to the Ludii General Game System. In *Advances in Computer Games, ACG 2019 (LNCS, Vol. 12516)*. Springer.
- [4] Kate Compton and Michael Mateas. 2015. Casual Creators. In *Proceedings of the Sixth International Conference on Computational Creativity, ICCO 2015*. computationalcreativity.net.

- [5] Kate Compton and Michael Mateas. 2015. Tracery: An Author-Focused Generative Text Tool. In *Foundations of Digital Games, FDG 2015*. SASDG.
- [6] Michael Cook. 2020. Software Engineering For Automated Game Design. In *IEEE Conference on Games, CoG 2020*. IEEE.
- [7] Michael Cook. 2022. Puck: A Slow and Personal Automated Game Designer. In *Proceedings of the Eighteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2022*. AAAI Press.
- [8] Albert Danial et al. 2012. cloc-1.96. <https://github.com/AlDanial/cloc>
- [9] Joris Dormans. 2009. Machinations: Elemental Feedback Patterns for Game Design. In *GAME-ON-NA. EUROISIS*.
- [10] Joris Dormans. 2009. Machinations Tool. <https://discussions.unity.com/t/any-offline-machinations-like-tool/762019> Last visited: October 24 2024.
- [11] Tracy Fullerton. 2014. *Game Design Workshop: A Playcentric Approach to Creating Innovative Games*. CRC press.
- [12] Swen E. Gaudl, Mark J. Nelson, Simon Colton, Rob Saunders, Edward Jack Powley, Blanca Pérez Ferrer, Peter Ivey, and Michael Cook. 2018. Rapid Game Jams with Fluidic Games: A User Study & Design Methodology. *Entertainment Computing* 27 (2018).
- [13] Alan R. Hevner et al. 2004. Design Science in Information Systems Research. *MIS Quarterly* 28, 1 (March 2004).
- [14] Michael Holly, Lisa Habich, and Johanna Pirker. 2024. GameDevDojo - An Educational Game for Teaching Game Development Concepts. In *Foundations of Digital Games, FDG 2024*. ACM.
- [15] Alan C. Kay. 1993. The Early History of Smalltalk. In *History of Programming Languages Conference (HOPL-II)*. ACM.
- [16] Paul Klint and Riemer van Rozen. 2013. Micro-Machinations: A DSL for Game Economies. In *Software Language Engineering, SLE 2013 (LNCS, Vol. 8225)*. Springer.
- [17] Raph Koster. 2013. *Theory of Fun for Game Design*. O'Reilly Media, Inc.
- [18] Max Kreminski, Melanie Dickinson, Joseph C. Osborn, Adam Summerville, Michael Mateas, and Noah Wardrip-Fruin. 2020. Germinate: A Mixed-Initiative Casual Creator for Rhetorical Games. In *Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2020*. AAAI Press.
- [19] Gorm Lai, Frederic Fol Leymarie, and William Latham. 2022. On Mixed-Initiative Content Creation for Video Games. *IEEE Transactions on Games* 14, 4 (2022).
- [20] Stephen Lavelle. 2015. PuzzleScript. <https://github.com/increpare/PuzzleScript>
- [21] Antonios Liapis, Georgios N. Yannakakis, and Julian Togelius. 2013. Sentient Sketchbook: Computer-Aided Game Level Authoring. In *Foundations of Digital Games, FDG 2013*. SASDG.
- [22] Juan Linietzky, Ariel Manzur, and the Godot community. 2025. Godot Engine 4.4 documentation. <https://docs.godotengine.org> Last visited: March 26 2025.
- [23] Matthew B. MacLaurin. 2011. The Design of Kodu: A Tiny Visual Programming Language for Children on the Xbox 360. *SIGPLAN Not.* 46, 1 (Jan. 2011).
- [24] Richard E. Mayer and Roxana Moreno. 2003. Nine Ways to Reduce Cognitive Load in Multimedia Learning. *Educational psychologist* 38, 1 (2003).
- [25] Mark J. Nelson and Michael Mateas. 2007. Towards Automated Game Design. In *Artificial Intelligence and Human-Oriented Computing, AI*IA 2007 (LNCS, Vol. 4733)*. Springer.
- [26] Jakob Nielsen. 1993. Response Times: The 3 Important Limits. (1993). <https://www.nngroup.com/articles/response-times-3-important-limits/>
- [27] Donald A. Norman. 1986. Cognitive Engineering. *User centered system design* 31 (1986), 61.
- [28] Paul Pivec. 2009. Game-Gased Learning or Game-Based Teaching. *British Educational Communications and Technology Agency (BECTA), corp creator* (2009).
- [29] Meihua Qian and Karen R. Clark. 2016. Game-based Learning and 21st Century Skills: A Review of Recent Research. *Comput. Hum. Behav.* 63 (2016).
- [30] 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.
- [31] Alexander Repenning and Tamara Sumner. 1995. Agentsheets: A Medium for Creating Domain-Oriented Languages. *Computer* 28, 3 (1995).
- [32] Mitchel Resnick, John H. Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay S. Silver, Brian Silverman, and Yasmin B. Kafai. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (2009).
- [33] Katie Salen. 2007. Gaming Literacies: A Game Design Study in Action. *Journal of Educational Multimedia and Hypermedia* 16, 3 (2007).
- [34] Anthony Savidis and Alexandros Katsarakis. 2021. Game Development as a Serious Game with Live-Programming and Time-Travel Mechanics. In *Entertainment Computing, ICEC 2021*. Springer.
- [35] Jesse Schell. 2008. *The Art of Game Design: A Book of Lenses* (1st ed.). CRC press.
- [36] Adam M. Smith, Mark J. Nelson, and Michael Mateas. 2010. LUDOCORE: A Logical Game Engine for Modeling Videogames. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games, CIG 2010*. IEEE.
- [37] Gillian Smith, Jim Whitehead, and Michael Mateas. 2011. Tanagra: Reactive Planning and Constraint Solving for Mixed-Initiative Level Design. *IEEE Trans. Comput. Intell. AI Games* 3, 3 (2011).
- [38] Cynthia Solomon, Brian Harvey, Ken Kahn, Henry Lieberman, Mark L. Miller, Margaret Minsky, Artemis Papert, and Brian Silverman. 2020. History of Logo. *Proc. ACM Program. Lang.* 4, HOPL (2020).
- [39] Steven L. Tanimoto. 2013. A Perspective on the Evolution of Live Programming. In *Workshop on Live Programming, LIVE 2013*. IEEE.
- [40] Machinations.io Dev. Team. 2024. Machinations.io. <https://machinations.io> Last visited: March 26th 2025.
- [41] Julian Togelius and Jürgen Schmidhuber. 2008. An Experiment in Automatic Game Design. In *Computational Intelligence and Games, CIG 2009*. IEEE.
- [42] Mike Treanor, Bryan Blackford, Michael Mateas, and Ian Bogost. 2012. Game-O-Matic: Generating Videogames that Represent Ideas. In *Workshop on Procedural Content Generation, PCG 2012*. ACM.
- [43] Jeroen J. G. van Merriënboer and John Sweller. 2005. Cognitive Load Theory and Complex Learning: Recent Developments and Future Direction. *Educational Psychology Review* 17, 2 (2005).
- [44] Riemer van Rozen. 2015. A Pattern-Based Game Mechanics Design Assistant. In *International Conference on the Foundations of Digital Games, FDG 2015*. SASDG.
- [45] Riemer van Rozen. 2021. Languages of Games and Play: A Systematic Mapping Study. *Comput. Surveys* 53, 6 (2021).
- [46] Riemer van Rozen. 2023. Cascade: A Meta-Language for Change, Cause and Effect. In *International Conference on Software Language Engineering (SLE 2023)*. ACM.
- [47] Riemer van Rozen. 2023. Game Engine Wizardry for Programming Mischief. In *International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments, PAINT 2023*. ACM.
- [48] Riemer van Rozen. 2024. Live Game Design: You Make the Rules. In *CWI Open Day, Science Weekend, Oktober 5 2024*. Tutorial, demo and workshop.
- [49] Riemer van Rozen. 2025. Vie - v0.0.7. (March 2025). <https://vrozen.github.io/Vie>
- [50] Riemer van Rozen and Joris Dormans. 2014. Adapting Game Mechanics with Micro-Machinations. In *Foundations of Digital Games, FDG 2014*. SASDG.

A Cheat Sheet

Machinations is a visual language for designing a game's rules. Vie uses a variant of this notation. Diagrams (or programs) are graphs that consists of two kinds of elements: nodes and edges. Both can be adjusted with extra information. These elements determine how resources (marbles) are step by step redistributed (roll) along the paths of the diagram (the marble track). This cheat sheet describes the main language elements.

 Apples	A <i>pool</i> is a node with a name that can contain resources (marbles) such as coins, crystals or apples. A pool appears as a circle with a number inside that represents the current amount, and the starting amount (at). The maximum capacity (max) determines when a pool is full, and no more marbles can be added.
	A <i>resource connection</i> is an edge with an associated amount that represents the rate at which marbles roll between source and target nodes. Every step, each node can work once by redistributing marbles along the resource connections of the marble track. The inputs of a node are the resource connections on the left, and the outputs are on the right.
	The <i>activation modifier</i> (when) determines when a node can work. By default, nodes are <i>passive</i> (no symbol). <i>User</i> nodes (double line), represent interactive elements offering actions users can activate. Automatic nodes (*) work automatically.
 Enter	Nodes work (act) either by <i>pulling</i> marbles along their inputs (default, no symbol) or by <i>pushing</i> marbles along their outputs (p). They work in two ways (how). Nodes that have the <i>any</i> modifier (default, no symbol) interpret the amounts of their source connections as upper bounds and move as many marbles as possible. For nodes that instead have the <i>all</i> modifier (&), these are strict requirements, and the associated flows either all happen or not at all.
	A <i>source</i> , a node shown as a triangle pointing up, is the only element that can generate marbles. Sources can be seen as a pool with an infinite amount of marbles. They can always provide sufficient marbles.
	A <i>drain</i> , a node shown as a triangle pointing down, is the only element where marbles can disappear. Drains can be seen as pools with an infinite negative amount of marbles. They can always consume more.
	A <i>trigger</i> is a connection whose value is a multiplication sign (*). The source node of a trigger activates the target node when sufficient marbles roll for each resource connection on which that node acts.
	Converters are nodes shown as triangles pointing to the right with a vertical line through the middle. They consume one type of resource and produce another. Converters only work if all the required marbles are available at the inputs.