All in One: Rapid Game Prototyping in a Single View

Eva Krebs
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
eva.krebs@hpi.de

Tom Beckmann
Hasso Plattner Institute
Potsdam, Germany
tom.beckmann@hpi.uni-potsdam.de

Leonard Geier
Hasso Plattner Institute
Potsdam, Germany
leonard.geier@student.hpi.uni-potsdam.de

Jonathan Grenda
Hasso-Plattner-Institut
Potsdam, Brandenburg, Germany
jonathan.grenda@student.hpi.uni-potsdam.de

Stefan Ramson
Hasso Plattner Institute
Potsdam, Germany
stefan.ramson@gmail.com

Robert Hirschfeld
Hasso Plattner Institute
Potsdam, Germany
robert.hirschfeld@hpi.uni-potsdam.de

Abstract

Creating games involves frequent prototyping to quickly obtain feedback. In this paper, we explore the impact of removing a traditional game engine's separation of scene and game logic that supports scalability to large projects and, instead, combine scene and game logic in a single view. In our tool, *Pronto*, designers connect game objects with visual representations of behavior to define game logic in the scene view, thus exposing any concern of the prototype to the designer within one click. To explore the implications of the trade-off between scalability and speed of access, we conducted a cognitive walkthrough and an explorative user study comparing prototyping in the Godot game engine and in *Pronto*. Godot's separate views made it appear more structured and reliable to users, while *Pronto*'s scattered game logic accelerated editing and gave users the impression of progressing faster in their implementation.

CCS Concepts

• **Human-centered computing** → *Empirical studies in interaction design*; **Systems and tools for interaction design**; • **Software and its engineering** → **Visual languages**.

Keywords

game programming, game prototyping, visual programming

ACM Reference Format:
Eva Krebs, Tom Beckmann, Leonard Geier, Jonathan Grenda, Stefan Ramson, and Robert Hirschfeld. 2025. All in One: Rapid Game Prototyping in a Single View. In *CHI Conference on Human Factors in Computing Systems (CHI '25), April 26–May 01, 2025, Yokohama, Japan.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3706598.3714251

1 Introduction

Games improve through iteration: as a rule of thumb, when designers get to iterate more and obtain feedback on their decisions faster, the resulting game mechanics will improve [33]. Often, designers choose a digital prototyping tool for that purpose, as it allows them to create prototypes with the same interactions as the final product. Common to digital game prototyping tools is that they draw a deliberate line between game objects and game logic: objects are defined in a scene view, and game logic is defined in a separate code view that references the objects of the scene.

This separation allows games to scale in complexity. Complex game logic can be spread across multiple files or abstractions and can be applied to multiple game objects in the scene. In contrast, if the game logic is combined with game objects in the same scene, the logic would be tied to individual objects in the scene, making reuse and abstraction more difficult. However, by separating game logic and scene view, the engine necessarily introduces context switches, navigation, and complexity when users have to map between objects and game logic in the different views.

In this paper, we investigate trading the potential to scale to complex projects for speed of access to all prototype parts. To this end, we designed *Pronto*, an adaptation of the Godot game engine to facilitate the rapid creation of throw-away prototypes of game mechanics. *Pronto* builds on the following concept:

> Game logic is expressed as visual connections between game objects in the game scene. Functionality without inherent visual representation in Godot is made visual through special game objects we call *Behaviors*.

Consequently, all game logic and game objects are in a single view, allowing designers to reach any part of their design they may want to adapt in a single step. This removes the otherwise present navigation and mapping overhead, but it does so at the expense of the tool's ability to scale to complex games.

We conducted two studies to understand the trade-offs resulting from the behavior concept and *Pronto* in general. First, a cognitive walkthrough based on the *Cognitive Dimensions of Notations* [10], and second, an explorative user study that compares the creation of game prototypes in *Pronto* and in Godot. We find that *Pronto* supports users in prototyping rapidly and encourages trying out new ideas. At the same time, Godot offered better documentation and appeared more beginner-friendly.

In the remainder of the paper, we first present methods of game prototyping and relate them to our approach (section 2). We then describe the behavior concept and example projects for *Pronto*

CHI '25, Yokohama, Japan

ACM ISBN 979-8-4007-1394-1/25/04
https://doi.org/10.1145/3706598.3714251

(section 3). We report our cognitive walkthrough (section 4) and our user study (section 5). We discuss our findings and potential future work (section 6).

2 Background and Related Work

As outlined in section 1, prototyping is essential to the game design process. Prototyping comes into play for various aspects of a game. The "elemental tetrad" of game design [33] divides a game into four aspects: aesthetics, technology, mechanics, and story. Designers may create prototypes for any of the four aspects to externalize their ideas or to mitigate risk in their design process by obtaining feedback or assessing feasibility [20].

Prototypes take various shapes. To understand aesthetics, designers may create mood boards. To understand the design of their story, they may create storyboards or dialog scripts. To understand technical risks, they may employ technical spikes that implement the bare minimum to attempt to demonstrate a technical capability [9, 17].

Pronto's focus is on creating prototypes that assess mechanics. Here, designers commonly start with paper prototyping, a method that allows rapid iteration as rules are agreed upon by the tester and designer and can thus be changed on a moment's notice [33]. In contrast, in digital prototypes, rules are programmed into the game and thus require changing the game logic to adapt. However, there are game mechanics that cannot easily be expressed through paper: if the prototyping goal is to assess how well players can use reflexes to react to visible changes in the game response times must be in real-time [1]. As an example, a designer may want to validate if attack moves in a combat game like Super Smash Bros.[1] are not too fast to be able to dodge or block. If a human test conductor would have to simulate the full game on paper, it will necessarily feel very different when compared to the perfectly even and consistent movements of a computer-controlled character.

When creating a prototype for mechanics, designers will either modify an existing game or draft relevant of a game from scratch [17]. General-purpose game engines such as Godot [2] or Unreal [3] facilitate this process but may also require more technical decisions than is desirable to answer the prototype designer's question. Reuse of existing games is an often used method, but it requires write-access to an existing game that serves as an adequate base, as is possible through modding of many popular games. Even given write access, it is still a consideration whether working from scratch as opposed to modding serves the prototyping goal better: the designer may deem the size of changes too large for modding to make sense to realize their vision.

Specialized game engines or programming environments can facilitate rapid creation of prototypes but are limited to certain types or genres of games. They offer higher-level primitives that are targeted at common challenges within their domain. For instance, RPG Maker [4] includes facilities specifically for the creation of 2D role-playing games. For story-focused games, other tools facilitate the rapid iteration of conversation trees [7].

Other environments support the rapid creation of 2D games more general way, including systems such as GameMaker [5], Scratch [28], Snap [6], or AgentSheets [27]. In the 3D space, games that allow creating games include Dreams [7] and Roblox [8]. Here, players can enter a creation mode that typically allows them to select objects, open a programming editor, and define rules for their behavior. Other systems facilitate creating games for XR [35], tangible environments [21], or mobile devices [32]. Often, these systems also incorporate visual programming elements, such as nodes-and-wire programming, to remove technical hurdles faced by designers.

As an alternative, prior work has explored using high-level modeling techniques to specify behavior [30, 31]. Here, game designers specify state charts or similar models to define the interaction between objects. A generator then derives source code for an executable game. Using models allows some design decisions to be more easily changed, supporting faster prototyping.

We designed *Pronto* to evaluate the conscious intertwining of game elements with visual programming elements in one scene. This contrasts the previously mentioned specialized game engines: except for limited special cases, behavior is defined as separate windows, popups, or panes. *Pronto*'s design attempts to close the gap at the expense of a more cluttered scene design view. Some programming systems have considered similar steps. For example, Boxer [6] or Lively Fabrik [16] also mix functional user interface elements with programming primitives that define their behavior in one view. Both aim to support user comprehension and direct access by end-users. In this way, their goal differs from *Pronto*'s, where the combination in a single view is designed to facilitate rapid prototyping. Still, some design aspects align, such as the immediate access to the entire system's functionality and use of visual representations that suit the users' mental model better.

In the context of graphical user interface (GUI) development, a variety of tools and approaches exist that are designed to support the rapid development of interactive elements. For instance, Rapid Application Development (RAD) [2] is a development approach in the domain of business applications aimed at applications of high interactivity but low computational complexity, where iteration speed is prioritized [2]. RAD advocates the use of visual tools for purposes of accelerating the development process, for instance by leveraging code generation or GUI builders [3]. GUI builders typically establish a link between a visual interface designed for user interface creation and an underlying code base through named references. Integrated development environments, such as QtCreator, support code generation based on interactions in the GUI builder to further facilitate the transition between visual and textual.

Similarly, image-based development environments [12], such as many Smalltalk or Lisp dialects, are designed for an exploratory programming [26] approach to software development. For instance, in Self's Morphic [18, 19], logic and (visual) application exist in the same process and are displayed in the same window. Programmers edit textual code through dedicated inspector windows that can be opened on an object. Connections among objects and between

[1] https://www.smashbros.com/, last accessed: 2025-02-11
[2] https://godotengine.org, last accessed: 2024-09-10
[3] https://www.unrealengine.com, last accessed: 2024-09-10
[4] https://www.rpgmakerweb.com/, last accessed: 2024-12-09
[5] https://gamemaker.io, last accessed: 2024-09-10
[6] https://snap.berkeley.edu, last accessed: 2024-09-10
[7] https://www.playstation.com/en-us/games/dreams/, last accessed: 2024-09-10
[8] https://www.roblox.com/, last accessed: 2024-12-09

objects and code are thus not modeled explicitly through connections as in *Pronto* but implicitly through references in variables. The Etoys [13] programming system similarly targets interactive, visual applications, and further pushes intertwining of visual application and logic: references to objects are established through a context menu on an object or by dragging it, yielding a block that represents the object. Code is placed in the scene as objects and, while open, will even be considered for collision detection of moving objects. In this way, Etoys emphasizes a "sandbox-style" development environment designed for experimentation, often employed in educational contexts [8].

An early prototype of *Pronto* has been discussed at the Psychology of Programming Interest Group [15]. This work significantly extends the scope of *Pronto* through high-level components that tackle recurring aspects of game prototyping, such as implementing a health bar or platform controls. Further, an important new aspect concerns bridging the gap between scene editor and game view: the *Live Value HUD* allows developers to make adjustments to their prototype directly in the game while playing the game. The early prototype had not been been evaluated beyond an initial experience report. This paper adds an evaluation of the system through a cognitive walkthrough and an explorative user study.

3 A Prototyping Tool for Godot

Pronto is an extension of the Godot game engine. *Pronto* forms a superset in terms of features: users of *Pronto* have access to all functionality of Godot. In this section, we will first outline user interactions within Godot and then how interactions in *Pronto* work to demonstrate how they differ.

3.1 The Godot Game Engine

At the core of Godot is a scene tree of nodes. Godot offers an extensive library of node subclasses for various purposes, such as showing an image, moving a node in 2D space according to simulated physics, or playing spatial sound. Designers create games by composing instances of node subclasses in the scene tree or creating their own subclass of a node class.

Nodes are always placed within a scene. A scene can be instantiated in another scene. Modifying the original scene will propagate changes to all instances. One scene is the main scene that will be displayed when the game starts.

Godot heavily uses the composition of nodes to derive relationships implicitly: adding a collision shape node as a child of a character body node assigns this shape to the body. When moving nodes in the scene, children move with their parent.

Godot's user interface is made up of five major parts:

(1) The scene tree is shown as a *tree view* with collapsible subtrees. This view allows selecting nodes and changing the hierarchy, such as moving a subtree to be a child of another node.
(2) The *scene view* shows a 2D or 3D preview of the game scene. The scene view contains visual representations of all nodes that have a position in space. It omits nodes without a position in space, such as a node for ubiquitous background music, which can only be accessed from the scene tree.

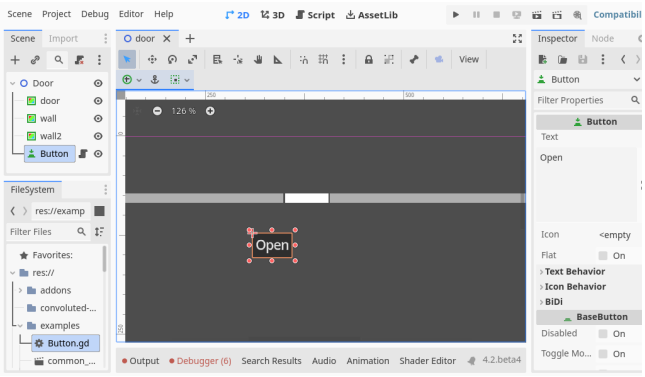

Figure 1: The scene view of the Godot game engine. The tabs at the very top allow switching the scene view to the code editor where users define game logic.

(3) The *inspector* allows editing properties of selected nodes, as determined by the node's class. These include the color of a rectangle, the velocity of a custom car node, or the text of a label.
(4) The *code editor* is accessible as a separate tab that replaces the scene view, or in a separate window. It displays the code for a game's custom nodes or the documentation of any built-in node.
(5) The *game view* opens in a separate window and shows the running game.

The tree view and inspector appear as sidebars on the left and right of the scene view or code editor, as seen in Figure 1.

When working on a game, users typically switch between three views: the scene view, the code editor, and the game view. As an example of the workflow in Godot, we want to create a door that disappears when a button is clicked, or the "A" key is pressed. For that purpose, we first set up the scene as illustrated in Figure 2. We create three rectangles for the walls and a door, name them appropriately, and color them gray and white respectively in the inspector.

Next, we attach a script to the door game object. When creating the script, the code editor replaces the scene view. In the code editor, we type the code also shown in Figure 2. The code connects the button's "pressed" signal to an anonymous function that addresses the door by name in the scene tree and deletes it. On input, the code checks if a key press occurred and whether the pressed key was "A", and if so, proceeds to do the same. To test our game, we launch it twice and try out the two ways of interacting with the door in the game view.

3.2 *Pronto*

Pronto extends Godot with two essential additions: *connections* and *behaviors*. We want to create the same door mechanic in *Pronto* as shown in Figure 2. The complete walkthrough is illustrated in Figure 3.

We begin with the same game object setup in the scene view without creating a script. We select the game object triggering the relevant signal, the button, and hover *Pronto*'s connection list (1).

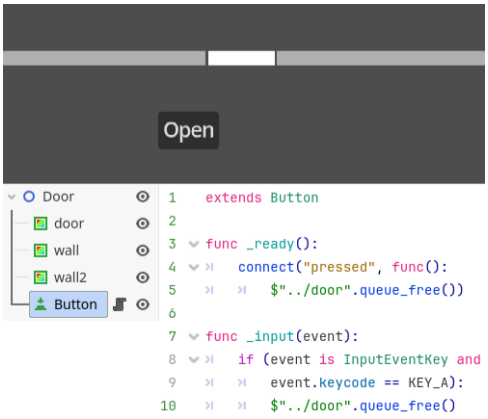

Figure 2: At the top, we show an excerpt of the scene view with our game objects setup. In the bottom left, the scene tree illustrates the hierarchy of game objects. In the bottom right, we see the code view with the script responsible for managing the door.

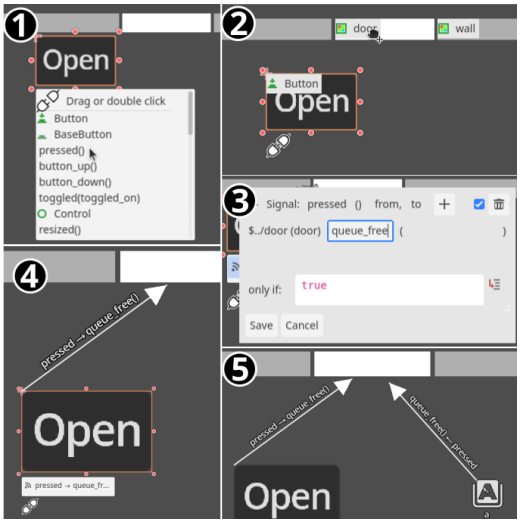

Figure 3: Constructing a door that opens when a button or key is pressed by creating connections. The same initial scene is shown at the top of Figure 2.

Next, we start dragging the "pressed" signal and drop it onto the door (2). While dragging a signal, game objects show markers with their names as drop targets. These are grouped when many objects are close to one another to prevent overlaps that would make targeting difficult.

Once the user drops the signal on the door, a connection dialog opens (3). Here, we configure the method to be called on the door when the signal triggers, `queue_free`, which deletes the object. In the dialog, we may also configure a condition that must be true for the method to be called, or we can turn off the connection entirely.

When compared to the workflow in Godot, the connections in the scene view provide spatial immediacy [37] for the expression of logic next to the game objects that it is relevant for. Through the connection dialog, we make the details of the connection visible without requiring the user to switch views. By opening multiple dialogs of different connections, users can compose views relevant to their current use case, even if the concerns are spread across multiple game objects and would thus have ended up in different Godot script files. Once a dialog is closed, its connection remains visible as an arrow labeled with the signal and method (4). It thus provides an entry point for quickly modifying or comprehending any part of the game's logic.

Next, we also would like the door to open when the "A" key is pressed. Godot does not have a built-in node that signals keystrokes, instead relying on code as shown in Figure 2.

Instead, we instantiate a *Pronto* behavior, a Godot node subclass that appears as an icon in the scene view. *Pronto* behaviors are a visual representation of an intangible aspect relevant to the formulation of game logic, such as keyboard input. Being Godot nodes, behaviors possess methods, signals, and properties. For instance, the *Timer* behavior has a property configuring its timeout duration, a signal when the duration has elapsed, and a method to start the timer.

As shown in (5), we instantiate a *Key* behavior that emits a signal whenever a user-configured key is pressed. For visual clarity, we position the *Key* behavior close to the door and create a connection that calls `queue_free` as we did for the button.

3.3 Abstraction-level of Behaviors

Behaviors in *Pronto* augment what game logic can be expressed through *Pronto* connections as they make intangible functionality visible in the game scene and thus addressable through connections. The library of behaviors is thus extensible and should serve the needs of a specific prototype. In our exploration, we strived to create a small library of composable behaviors to facilitate the creation of complex games without having to learn a large list of specialized behaviors.

The behaviors that emerged during that exploration can be categorized in five groups:

(1) Actions: making more methods accessible,
(2) Triggers: making more signals accessible,
(3) Data: managing state in the scene view,
(4) Visualization: visualizing an aspect to the player of the prototype, and
(5) Utility: aiding the developer of the prototype in debugging.

Actions. Action behaviors cause some side effects when triggered. For instance, the *Move* behavior moves its direct parent in the cardinal directions or toward a point when triggered, or the *CameraShake* behavior applies a decaying shake to the viewport. All methods of Godot nodes can be used as actions, such as the `queue_free` method used to delete a node. In addition, a *Scene* behavior makes methods available to *Pronto* that exists on Godot's scene singleton, such as restarting or quitting the game.

Triggers. *Pronto* includes three default temporal triggers: a *Timer* behavior, an *Always* behavior, which triggers on every frame, and a *Ready* behavior, which triggers when its parent node first enters the game as part of its scene tree. Two triggers act on spatial conditions:

a *Collision* behavior, which triggers when something collides with its parent, and a *Query* behavior, which allows composing distance filters, type filters, and code filters and triggers when nodes start or stop matching the query. Finally, there are input triggers for the mouse and keyboard. Both trigger signals when buttons are just pressed, held, or released. As described, all signals of Godot nodes can also be used as triggers, such as the `animation_finished` signal of the *AnimationPlayer* node.

Data. For managing state, *Pronto* offers a *Value* behavior for defining configurable constants, which shows a slider in the scene view for fast access. A *Store* behavior is a dictionary for storing arbitrary dynamic variables. A *Bind* behavior binds the value of one property to another property with an optional transform expression. A *StateMachine* behavior allows users to define states and transitions; signals trigger when states are left or entered, which can be used to trigger other actions.

Visualization. A *Placeholder* behavior visually represents a game object and automatically communicates collision information. It can take the form of predefined shapes, such as rectangles, circles, or triangles, or appear as icons or images. *Pronto* includes a library of generic game icons for communicating common purposes of objects. In addition, a label can be drawn across the shape or icon. Second, a *Line* behavior draws a line between two nodes.

Utility. In a *Watch* behavior, users can enter arbitrary GDScript expressions that are continuously evaluated, and the result is shown underneath the watch. Similarly, an *Inspect* behavior shows the current runtime value of any of its parent node's properties. To help organize the scene view, a *Group* behavior draws an outline around all its child nodes and can display a label, allowing users to segment their scene visually.

Trade-offs for Specialized Behaviors. During user testing, designers created the same combinations of behaviors and game objects repeatedly. This concerned, in particular, character movement in a side-scrolling platformer. As a solution, we introduced a *Platformer-Controller* behavior as a ready-to-use combination of a *Move* and *Controls* behavior, including the required three connections for left, right, and jump movement. Similarly, it became apparent that some state that users were placing in *Store* behaviors had an inherent visual representation in the game. This insight led to the addition of a *HealthBar* behavior, which stores a maximum and current number of health points and displays a corresponding visualization similar to a progress-bar optionally labeled with the current value. These specialized behaviors demonstrate the tension in the design goal of keeping a small, composable number of behaviors—similarly, designers working on many prototypes in a specific genre may want to create a specialized behavior that facilitates a specific aspect of that genre.

3.4 Composing Larger Scenes

Even prototypes will often require a moderate level of complexity that makes working through copy-and-paste intractable. Godot facilitates the instantiation of the same type of game object by allowing users to create multiple scenes and instantiate them from one another. However, this forces users to switch between scene views and requires a level of indirection when addressing them, for example, through a tag that every instance of the game object carries.

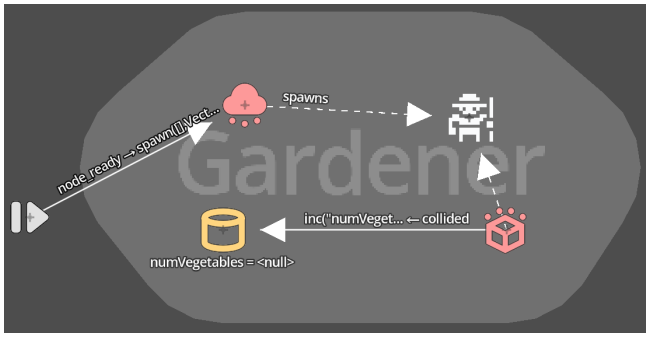

Figure 4: On game start, this scene will spawn a gardener. The gardener has a *Store* behavior remembering the number of vegetables collected by that gardener. On collision, we increase the number.

In *Pronto*, users instead use a special *Spawner* behavior as seen in Figure 4. The *Spawner* is placed like every other behavior in the single scene view. Using the *Spawner*, users can create linked copies of the *Spawner*'s subtree and either place them in their scene at edit-time or use a connection to trigger a method on the *Spawner* to instantiate a linked copy while the game is running. Users can still modify the properties of instantiated copies manually or through code. A property modified on a copy will no longer be kept in sync with the original.

The *Spawner* also facilitates local state, as illustrated in Figure 4. Connections local to a *Spawner* subtree are instantiated in their entirety. By instantiating and connecting to a local *Store* behavior, users can state local to the subtree. Connections that enter or leave the *Spawner* subtree are also duplicated. Consequently, users can establish direct connections between local and global state, thus facilitating top-down and bottom-up data flow without the need for inversion of control, as would be the case with Godot's default scenes that use signals.

To further facilitate fast access, *Pronto* includes a set of utility functions that facilitate access to local properties in the scene tree. A `closest(condition)` function will locate the closest node that fulfills the given condition. To find what we call the closest node, we start at the node that called the function and traverse the node's local surrounding tree, starting with its own children in a breadth-first manner and then continuing with its parents and their children.

3.5 Designing For a Single View

Godot users primarily switch between three views, as outlined in subsection 3.1: the scene editor, the code editor, and the running game. In *Pronto*, we combine scene editing and behavior definition into one view.

Code is brought as close as possible to the place where it causes the effect: connections between game objects and behaviors define all game logic. *Pronto* encourages users to place all game objects and behaviors in a single scene, so changing any aspect is only one click away. Consequently, game prototypes reach an upper limit of

complexity that can still fit well in a scene. Otherwise, users end up with the dreaded "spaghetti" of node-and-wire programming. As *Pronto* was designed for rapid iteration of throw-away prototypes that are for validating a single or very few mechanics at a time, the design makes little effort to address the scaling issue, prioritizing speed of access over hierarchies of logic that may scale better.

As one exception, the running game appears in a separate view. In the running game, behaviors and connections are invisible, showing only the game objects, allowing one to play the game without distraction.

Editing the state of and playing the game in the same view would be an ambiguous operation: When users edit the game state in the engine, they edit the runtime state s_0 of the game. As soon as the game is started, the simulation modifies the game state, moving the runtime state from s_0 to some s_i. Any modification of the game state through the user in state s_i will have to be mapped back to s_0, which is a necessarily ambiguous operation from the user's perspective.

To bridge the gap less ambiguously, *Pronto* visualizes information from the runtime into the scene view and exposes selected information for editing in the game view. In terms of visualization, *Pronto* shows the state of any *Store* underneath its representation. Similarly, as described before, *Watch* and *Inspect* behaviors visualize specific, user-requested runtime state. In addition, connection arrows flash in red whenever their signal is fired.

In terms of editing, *Pronto* collects all *Value* behaviors on game start, which represent any constants the other behaviors are using. These are then gathered in a collapsible overlay on the game view as sliders or input fields. Through this overlay, users can tweak values without switching focus. Any changes in the game view to a constant are propagated back to the engine, modifying s_0. While the game is still running, users can choose to reset all values to the values they had at the game's launch. In this way, *Pronto* constraints the editing operations to those that are easier to reason about. At the same time, the state can still diverge: if, for example, users decrease the jump distance while in-game, they may have been able to reach a point of the game map that would not have been possible had the value been as low from the start of the game session.

To further support fast feedback loops, *Pronto* automatically deploys the prototype scene as a web export through a single button press. This way, playtesting can occur within a minute of a change on anyone's phone or laptop.

3.6 Prototyping Prototyping: Example Games

Initially, the authors validated that the core concept of connections would work for creating prototypes. Then, we organized several seminars, during which both we, the authors, and computer science students taking our seminar used *Pronto* to prototype games.

Throughout the seminars, we asked students to work on games with varying topics to test the limits of *Pronto*. Participants built prototypes for game domains such as simulation, racing, platformer, or turn-based games. While some prototypes only focussed on a single mechanic, some participants tried to make an entire playable game, including nice-to-have features such as a menu. Here, the limits of *Pronto* in terms of the visual complexity of the scenes became apparent. However, students still managed to create clones of *Ridiculous Fishing*[9](see Figure 5) and *Geometry Dash*[10].

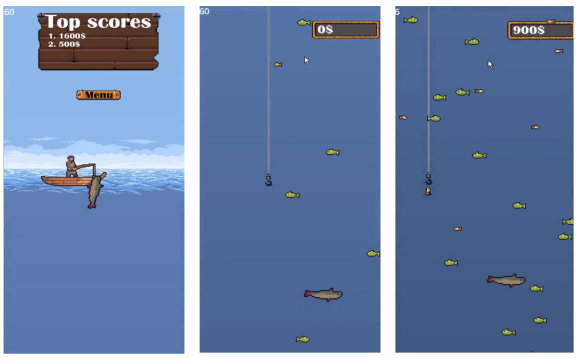

Figure 5: A clone of Ridiculous Fishing made with *Pronto*.

4 Cognitive Walkthrough

Pronto is designed to explore the impact of combining game logic and scene objects in a single view described in section 1 on the prototyping workflow. To evaluate the impact, we designed a two-staged explorative evaluation similar to prior work [40]. In the first stage, we analyzed *Pronto* using a cognitive walkthrough, followed by a user study described in section 5.

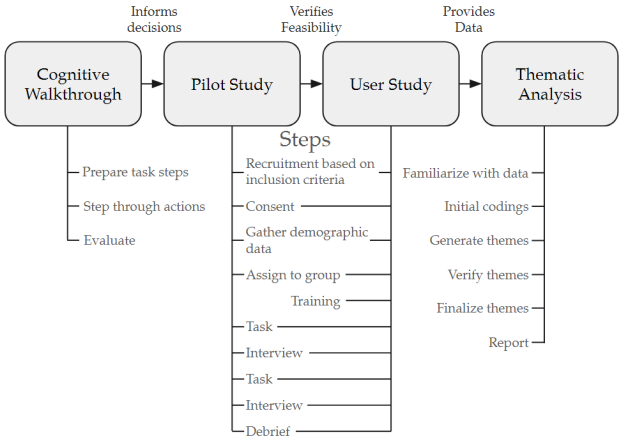

Figure 6: Overview of our evaluation concept based on prior work [5, 14]. A detailed list of actions for one cognitive walkthrough is available in Appendix B.

4.1 Setup

To perform the cognitive walkthrough, we designated an expert, a member of the *Pronto* development team, as an evaluator. That evaluator performs a task and analyzes each step based on predetermined questions. The evaluator performed this process twice

[9]https://en.wikipedia.org/wiki/Ridiculous_Fishing, last accessed: 2024-09-06
[10]https://store.steampowered.com/app/322170/Geometry_Dash/, last accessed: 2024-09-06

over three days to capture any details that may have been missed in the first walkthrough. For this evaluation, the following questions were asked for each step of the task:

- Will the user try to achieve the right effect?
- Will the user notice that the correct action is available?
- Will the user associate the correct action with the effect they are trying to achieve?
- If the correct action is performed, will the user see that progress is being made toward a solution to their task?

In addition, we considered the cognitive dimensions of notations, a set of heuristics on the usability of notations. For each of the dimensions, we asked whether any restrictions or benefits are attributable to it.

Definition of User. As learnability was not our focus, we described the assumed user as an experienced *Pronto* user. However, the user does not know the task to be completed and cannot prepare for later sub-tasks while solving earlier ones.

Definition of Task. As a task, we selected a realistic scenario we observed students perform when creating their prototypes. The user is trying to create a rudimentary driving and sliding mechanic for a top-down car game. We include checkpoints where the user wants to playtest the prototype to evaluate their progress but otherwise choose the most direct path to completion. The task involves three subgoals:

- Implement a driving mechanic that allows the user to accelerate, steer left and right, and break and make sure that friction affects the car's speed.
- Next, the user adds icy surfaces that prevent the user from controlling their car while on top.
- Lastly, the user fine-tunes constants and the implementation to ensure that the race course can be completed.

We summarize abstract steps required to solve the three sub-tasks in Figure 8. In total, our procedure includes 64 concrete steps.

4.2 Results

We considered 17 questions per concrete action—four as shown above and a selection of thirteen cognitive dimensions of notations [10]—for a total of 1,088 possible data points. Of those, we noted impacts for 115 data points. The others showed no notable deviation from the expected outcomes. In the following, we describe the insights gained from the cognitive walkthrough. Where applicable, we relate the insights to the relevant cognitive dimensions of notations [10].

Insight 1: Behavior granularity. Pronto's behavior library allowed us to represent the functionality required for our task easily, contributing to *Closeness of Mapping* (CDN), where we consider the match between the notation and the problem domain. However, it is not always straightforward to identify the right behaviors. For input in our task example, *Pronto* offers a *Controls* and a *Key* behavior. The former exposes input using the classic two-axis WASD keys, while the latter exposes input for arbitrary single keys. The overlap based on their name and functionality is an issue of *Role Expressiveness* (CDN), where users may struggle to identify the right component of our tool to use.

Adding behaviors works the same as adding other Godot nodes, contributing to *consistency* (CDN), where parts of the notation work according to users' expectations according to their existing knowledge. Some behaviors act on their parent in the node hierarchy, which *Pronto* indicates through an arrow in the scene view from the behavior to the parent it acts on. However, as users change the position in the node hierarchy, the position remains the same in the scene view, a feature of Godot nodes to facilitate reorganizing scenes. As users have to manually adapt the behavior's position in the scene view as well, this may introduce *Viscosity* (CDN), where barriers in the notation make changes more effortful.

Insight 2: Refactoring connections. For the second subtask, the controls are supposed to be suspended while on an ice surface. We need to edit multiple connections and add a condition for that purpose. In code, we would likely introduce a new method that encapsulates the condition and can be used wherever needed. In contrast, in *Pronto* we need to duplicate the condition or add a *Code* behavior, which appears as overhead for a single line of code. Subsequent edits of that condition require us to revisit the duplicated copies, manifesting as *Repetition Viscosity* (CDN), where one intended change has to be carried through multiple actions. On the flip side, scattered code snippets contribute to the *Visibility* (CDN) between cause and effect, where parts of the notation are made identifiable and accessible. When we want to edit a functionality, we locate the concerned game object and find the relevant behavior through its large icon.

Insight 3: Game in a separate view. Changing a value in the *Live Value HUD* syncs it back to the editor. While this facilitates tweaking and persisting values experimentally in-game, it may also surprise users if they meant to simply experiment with values, especially since the game opens in a separate window. Further, the separate window for the running game introduces a context switch when game logic is changed that is not exposed in the *Live Value HUD*.

Insight 4: Drag-and-drop. Many relevant operations in *Pronto* rely on drag-and-drop using a mouse, for instance, creating new connections. While drag-and-drop is a direct and efficient way of connecting elements in a user interface, it is also prone to erroneous inputs and fatigue [34]. The user study in section 5 will analyze whether this has a practical impact.

Insight 5: Quality of life. Several minor quality-of-life issues were uncovered as well. For instance, documentation for behavior functionality, methods, and signals is missing. Further, the node triggering the signal or the signal name to which a connection is connected cannot be edited, forcing users to create a new connection and copy over existing values. These aspects were not relevant enough to be addressed during the development of *Pronto* but surfaced during the cognitive walkthrough.

4.3 Threats to Validity

As the analysis has only been performed by a single expert, the results may be skewed as no consensus or discussion took place to align disagreements. The task design included 64 actions and should thus give a broad overview of interactions with *Pronto*. However, of

the built-in behavior nodes, only 6 of 32 were required to complete the task. In addition, we did not include any errors or usage slips.

The concrete list of steps that we analyzed splits high-level operations that require multiple actions, potentially creating blind spots for an operation's overall usability. To mitigate this, we structured the actions required to solve the task hierarchically, divided into abstract and concrete steps. This allowed analyzing potential issues in both high-level intents and their execution.

5 User Study

We conducted a user study with eight participants to validate the insights of the cognitive walkthrough and collect new insights beyond those. In the study, participants worked on two game prototypes, once in *Pronto* and once in unmodified Godot. We then took observations and results from an ensuing interview and performed a thematic analysis to identify the impact and trade-offs that our behavior concept implies for the participants' prototyping workflow.

5.1 Study Design

We designed our study following best practices suggested by Ko et al. [14] and illustrated the concept in Figure 6. Each run took around 2.5 hours. Participants received 35€ as reimbursement for their time. The study's procedure is visualized in Figure 7. We began with welcoming the participants, obtaining their consent, gathering demographic data, and finally providing a 15-minute training session. Next, the two tasks followed, each allocated for one hour, separated by a five-minute break. We ended with a debrief to allow participants to ask questions or leave last remarks about the tools, tasks, or study.

Participants worked on one task in Godot and another in *Pronto*, resulting in four total task-tool combinations as shown in Table 1. We thus have two independent variables: the tool used and the task. The dependent variable is the usability feedback captured in the interviews.

To account for potential differences in skill, we opted for a within-subjects analysis. To mitigate learning effects, we created two different but structurally comparable tasks. With these restrictions, a regular Latin Square cannot be employed. We manually assigned tools and tasks, controlling for roll-over effects. This creates four valid task-tool combinations as depicted in Table 2, where every combination is used twice, given our eight participants.

5.1.1 Tasks. Both tasks are structured into three subtasks. We designed the tasks to present participants with new challenges throughout, requiring them to adapt and tweak their implementation. Further, we invited them to adapt the results to suit their preference to encourage additional iterations. To present changed conditions that the participants must adapt to, we prepared a new base scene for each subtask, which we call *environment* in the following.

We reused the task we analyzed in the cognitive walkthrough as described in subsection 4.1 to facilitate a comparison and designed a second, new task. The new task involves creating a dashing mechanic, a common movement option in platformer games. When the dash action is invoked, the character briefly and rapidly accelerates in a direction determined by the player. The task involves three subgoals:

- Implement a dash in a 2D side-scrolling platformer that allows crossing a gap in a prepared level.
- Next, a new level is shown with an even wider gap but with two platforms at different heights between the sides. Participants now may have to adapt their dash to also go vertically, not just horizontally.
- Lastly, a new level shows a parkour in which the character has to reach a goal position. The participant is asked to fine-tune the mechanic to ensure that the level is playable and feels sufficiently challenging.

Participants were asked to stop working on their implementation if they exceeded the given time frame by more than ten minutes. As the tasks are designed so that the second and third subtasks require iterating over the results of the first task, we were interested in ensuring that participants reached the third task. Therefore, we provided hints to participants if they ran into significant issues where we deemed their eventual resolution unlikely to generate interesting insights. We provided hints if the participant:

- was stuck for at least two minutes,
- showed an attempt to solve the problem on their own, and
- voiced the request for a hint or was prompted and agreed to receive a hint from the researcher.

Our testing setup included an external monitor, a laptop, an external keyboard, and a mouse. Participants were asked to adjust any mouse sensitivity settings prior to the study. Additional task details are in Appendix C.

5.1.2 Interview and Data Collection. After participants completed each entire task, we conducted a semi-structured interview consisting of 15 questions. We followed a similar procedure to Weninger et al., who formulated one question per cognitive dimension [25] (a mapping of used questions to cognitive dimensions can be found in Appendix A). As their questions have not been published, we formulated our own questions and refined them in multiple discussions. Based on the insights of the cognitive walkthrough, we emphasize the Viscosity dimension.

During the study, we collected a screen recording and an audio recording, as well as task completion times, task progress, and the number of hints given. To prepare the interviews for analysis, we automatically transcribed and manually corrected the audio recordings, and added notes we took while observing the participants. Finally, we imported and analyzed the resulting transcripts into MAXQDA, a program for qualitative analysis. As we were looking for qualitative statements on how using *Pronto* and Godot choose different trade-offs, we analyzed the data through a thematic analysis [5] and identified themes that concerned each tool.

Pilot Study. In preparation for the study, we recruited three participants for a pilot study. Their self-reported experience level varied from self-rated beginner to expert. In addition to running through our study script, we asked participants to complete a short feedback form regarding each task's comprehensibility and perceived difficulty. Each one of the participants attempted to complete two tasks, one with *Pronto* and one with Godot. For the pilot study, we

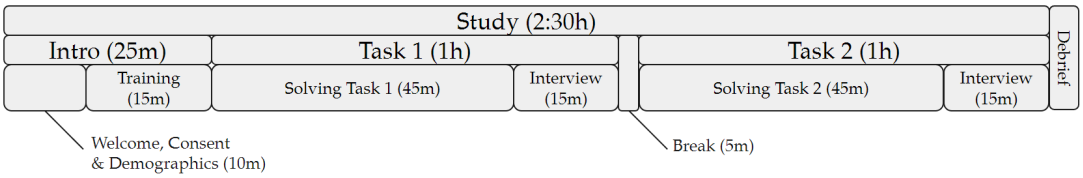

Figure 7: Timeline of the user study.

Table 1: Task-tool combinations

No.	Task	Tool
1	Dash	*Pronto*
2	Dash	Godot
3	Car	*Pronto*
4	Car	Godot

Table 2: Study conditions

Condition	First Task-Tool Combination	Second Task-Tool Combination
A	Dash-*Pronto*	Car-Godot
B	Dash-Godot	Car-*Pronto*
C	Car-*Pronto*	Dash-Godot
D	Car-Godot	Dash-*Pronto*

had initially allocated 30 minutes per task, which we raised to 45 minutes as no participant completed all sub-tasks. Of the six tested tool-task combinations, one was rated too difficult, and another was rated unsuitable for the time given, which we adapted for the final study.

Participants. Our study design did not include extensive training, so our potential participant pool was limited to students who had participated in the seminars where *Pronto* had been used. We contacted 25 possible candidates, of whom eight volunteered to participate. Seven identified as male, four were Bachelor students, and four were Master students. Participants ranged in experience from 4 to 7 years of software development, with no professional game development experience but experience with various game engines.

Five of the eight participants had taken part in a seminar designed to test and extend *Pronto*'s boundaries for game prototyping. As part of that seminar, they contributed extensions to behaviors. Notably, creating or extending behaviors is expected even of regular users of *Pronto*, as they identify use cases of their prototype they want to better support in *Pronto*. As *Pronto* was in early stages during the seminar, they also contributed quality of life and bug fixes. None of the integral design aspects of *Pronto* discussed in this paper have been co-developed by any of the participants.

5.2 Results

The average task completion rate was 89% (σ = 15.76%), with all participants reaching the third task. The completion rate was slightly higher for tasks performed with *Pronto* (91.5%, σ = 14.72%) than those attempted with Godot (87.2%, σ = 16.46%). On average, participants who completed all sub-tasks required 38 minutes (σ = 9.56). In *Pronto*, participants took 40 minutes on average (σ = 11.68) excluding two incomplete tasks, and 37 minutes on average (σ = 9.49) for Godot, excluding three incomplete tasks. We discuss the merit of our quantitative vs. our qualitative insights in subsection 6.2.

Through the thematic analysis, we identified 15 themes, of which 11 are presented in this section. The four not chosen for representation in this paper either did not pass the verification step of the thematic analysis or were not closely related enough to our research questions. Quotes have been automatically translated from German using DeepL and manually fixed. Where possible, we relate the themes to relevant cognitive dimensions of notations (CDN) [10].

Pronto *Theme 1: Pre-defined behaviors ease implementation and reduce complexity.* Five participants explicitly expressed how pre-defined Behaviors would simplify expression of constructs they usually perceive as cumbersome or complex to express. *Pronto* appears to thus better match the level of *Abstraction* that users seek and reduce *Hard Mental Operations* (CDN) for common tasks, where users have to take note of extra steps in the notation to achieve their desired goal.

> "The Platform-Controller [...] and the Move-Behavior are two nodes that reduce the amount of work required greatly and simplify small things."
>
> "The Move node is great, simply say 'move up' and it just works."

Multiple statements also mentioned the ease of gathering player input.

> "Gathering input events and taking action, that was easy."

Similarly, working with state through the *Store* Behavior was praised. Wrapping state in a Behavior, too, appears to match users' intuition and thus contribute to *Consistency* (CDN).

> "The 'at()' method was really handy."
>
> "I found it pleasant, accessing Values and the Store Behavior directly. I think that makes a lot of sense."

Pronto *Theme 2: Distinction and interoperability of pre-defined behaviors.* We were able to confirm Insight 1 of the cognitive walkthrough, indicating that perceived overlap between roles of Behaviors would present a challenge. Three participants explicitly criticized the *PlatformerController* and *Move* Behaviors, either being unsure of which Behavior they should use or expecting them to work together to achieve their desired goal. These Behaviors thus appear to fail to meet the right level of *Abstraction* that users are seeking and are lacking in their *Role-expressiveness* (CDN).

Table 3: Overview of participants' task order and progress

Participant	Condition	First Task					Second Task				
		Tool	Task	Progress (%)	Time (min)	Hints	Tool	Task	Progress (%)	Time (min)	Hints
1	A	*Pronto*	Dash	100	54	1	Godot	Car	66	55	2
2	C	*Pronto*	Car	100	35	0	Godot	Dash	66	55	2
3	D	Godot	Car	100	46	3	*Pronto*	Dash	100	42	3
4	B	Godot	Dash	100	38	2	*Pronto*	Car	100	49	3
5	C	*Pronto*	Car	66	55	2	Godot	Dash	100	35	1
6	A	*Pronto*	Dash	66	55	5	Godot	Car	66	55	0
7	B	Godot	Dash	100	40	1	*Pronto*	Car	100	40	1
8	D	Godot	Car	100	25	0	*Pronto*	Dash	100	19	1

> "The Platform-Controller is the one thing that confuses me the most when dealing with *Pronto*. The relation between Move- and Platform-Controller, especially with the addition of gravity."

> "I had the feeling [the Platform Controller] should work with the Move-Behavior, but it didn't synergize."

Pronto *Theme 3: Runtime visualizations fulfill concrete need.* Six out of eight participants explicitly highlighted the usefulness of runtime visualizations. The runtime visualizations allow users to rapidly map between cause and effect for all interactions in the game, supporting debugging and comprehension and thus *Visibility* (CDN).

> "It was obviously handy to immediately see if something was being triggered."

> "I find it useful to see what method is being called."

> "I think the most useful thing is the flashing of connections during run-time because it is unambiguous, and they jump out of the scene."

Pronto *Theme 4: Readability and expressiveness of connections.* Six of the participants said that *Pronto* provided a concise overview of their implementation progress through the visual connections in the scene view, supporting *Progressive Evaluation* (CDN), where the notation provides feedback for evaluating incomplete solutions.

However, five participants criticized the readability of the labels along connection arrows. Label font size is tied to the zoom level to ensure that users can still see scene objects, even when zoomed out, and not just large text. Further, while the arrows are labeled with details on what function the connection performs, its text is truncated to a fixed character limit, sometimes omitting relevant information.

> "I need to zoom in quite a lot to be able to read this at all."

> "[Not making the text larger when zoomed out keeps the scene] more organized, but not quite readable, pixelated."

> "I was just about to say it would be great to see the current values [of *Store* behavior variables], but then I zoomed in and saw that that already exists."

Also related to connections, the floating window concept for editing connections generally worked well. When opening multiple connections simultaneously, one participant criticized that it is unclear which connection the dialog belongs to.

Pronto *Theme 5: Insufficient documentation.* A major limitation of our cognitive walkthrough was that we assumed an expert user. During the user study, the relevance of helping users recall the functionality of Behaviors became apparent. Participants generally wished for more extensive documentation. For instance, similar to Behavior roles above, some signal names communicated ambiguous roles:

> "What the signals mean [...] I don't know what the difference between 'pressed' and 'down' is, whether or not they are identical, no idea."

> "It would help me to know what each of these does when I hover over them, as currently, I can not see that without looking at the documentation."

Pronto *Theme 6: Invitation to (Rapid) Prototyping.* During the interviews, three participants decidedly stated that they suspect the prototyping with *Pronto* to be faster than the alternative with Godot.

> "It was meaningfully faster than using just Godot."

> "I considered myself to be rather quick, especially when building things."

One participant noted that they spent almost the entirety of the last sub-task in the running game tweaking values using the *Live Value HUD* and did not have to switch back to the editor. Another participant remarked:

> "You could test everything relatively quickly, without everything crashing."

Other participants commented that they felt encouraged to try out different options and iterate more quickly, reducing *Premature Commitment* (CDN), where decisions must occur in a certain order or are not easily reversible. In particular, they pointed out that the visual programming aspect of *Pronto* supported discovery and sparked their creativity.

> "You are more likely to see what possibilities you have and what methods can be called. You can experiment faster."

> "I was less hesitant throwing things away, I don't know why."

> "You get inspired when you see the available possibilities. And you are encouraged to try out many things when you can just simply theoretically duplicate things or add them with a click and so on."

Godot Theme 1: Documentation. Through the feedback for Godot, the participants highlighted how important good documentation is for them. Half of the participants explicitly and positively commented on the availability or level of detail of the Godot documentation:

> "I know that within Godot, I can access the docs quickly. I find this comfortable, getting there with a single click and looking at the definition of functions to understand how to use them."

> "I am a big fan of the immediately accessible complete documentation of an object [...] and not just some sort of pop-up [...]."

Godot Theme 2: Context switches slow down prototyping. For the development with Godot, participants criticized context switches.

> "This split also splits the train of thought into 'this is the scene' and 'this is the script belonging to it' which is not very intuitive."

> "There is a clear separation between [scene] and [scripts], which can be annoying, [...] and there is an additional separation to the actually running game. Which meant I was pretty busy with context switches."

Participants also criticized switches between editor and game, in particular for purposes of tweaking values. Notably, Godot does support live reloading of values in some cases without restarting the game by switching to the editor and performing regular edits, which these participants did not make use of.

> "It is very annoying because I need to close the game, look at the code, tweak the number by some sort of factor, relaunch the game, and test again. This takes time. This switching between Editor and Game and Editor and Game back and forth."

> "Like I said, integrating this and then changing parameters, playing [...] has probably cost me the most time apart from debugging, or maybe even more than debugging."

Godot Theme 3: Time and effort intensive debugging. Seven out of the eight participants criticized the debugging tools or processes in Godot, wishing for feedback that is better integrated with the running game.

> "I find it a bit annoying to [log variables to the console] if I want to know their value. It would be great to be able to say 'put yourself in the game window, so I don't have to look [in another window]."

> "If I didn't need print statements to see all the values I am changing, [...] I could save two to three steps of debugging."

> "I would have liked for the values that I am changing to be visible at all times."

In more general terms, the feedback Godot provides was critiqued by five of the eight participants.

> "You don't get, I think, anything visual unless you build it yourself."

> "I needed to figure out what was happening in the method and that is not particularly easy in the visualization [...]."

One participant requested explicit visualizations for built-in methods that they perceived as complicated within the game view, such as `is_on_floor()`, relating also to the following theme of complex interfaces.

Godot Theme 4: Complex interfaces. Seven of the eight participants stated they struggled with one or more aspects of specialized knowledge related to implementing mechanics in Godot. Five participants mentioned challenges with implementing physics or physics-related aspects, especially simulating velocity in Godot's movement interface. This manifests as issues of *Hard Mental Operations* (CDN).

> "The `move_and_slide` [interface] was confusing for me, where you have to set the velocity [first]."

> "Velocity did not work as I expected, but I couldn't even say what I expected."

Available Features and Change Requests. Participants were asked to wish for a change or addition in each tool during the interviews. For *Pronto*, four participants (P1, P2, P4, P6) requested filters or views that would allow them to hide certain parts of their implementation in the scene to improve visual clarity for complex prototypes. Two participants (P4, P5) asked for automatic arranging of behaviors in the scene.

Several feature requests did already exist in *Pronto*, without the participants realizing it, such as the `toggled` signal on the *Key* behavior (P6). P7 requested to change the velocity of the *Move* behavior dynamically, which can be accomplished through a connection to the *set(prop, value)* method. Maybe surprisingly, only one participant wished for more keyboard-centric interactions. All other users reported or demonstrated no issues with the drag-and-drop interactions used heavily in *Pronto*.

For Godot, P5 wished for the ability to change individual properties without relaunching the prototype. Another participant, P2, requested an interface that provides the input vector for two axes. Both features are already present in Godot.

5.3 Threats to Validity

In the ideal case, the user study would have been conducted with enough participants to reach a saturation effect during the analysis [22], which we did not observe in our study. However, the recruited eight participants covered every study condition twice. As we focused on identifying usability problems through qualitative analysis, the eight participants are within the range of five to ten participants recommended by prior usability testing research [23, 39]. As outlined in subsubsection 5.1.2, five participants of the seminar contributed new or modified Behaviors to *Pronto*'s codebase but not to the core of *Pronto* or any of the design considerations discussed here, leading to potential bias. If the study were repeated, it could

include training, which would allow the potential candidates for recruiting to include more diversity in gender identity, age, and level of education.

The 15 minutes of training we included to refresh the participants' memories were not enough to bring them to the same level of expertise they had while participating in the seminars. Multiple participants identified a lack of experience with either tool as a potential issue during the interviews. Two participants directly mentioned a steeper reactivation curve for existing *Pronto* knowledge.

To ensure that our insights were comparable and that we could observe various common usage patterns across relevant features, we gave participants fine-grained tasks designed to force specific actions. This naturally constrained participants' degrees of freedom and may thus have prevented them from pursuing more creative solutions.

6 Discussion and Future Work

We designed our cognitive walkthrough and user study to find the impact of combining game logic and scene objects in a single view through connections. In the following, we discuss our insights to determine the merit of the design choice as opposed to a split between scene and game logic, as found in Godot.

6.1 Bridge Between Visual and Textual Programming

Choosing the right representation is an essential factor for designing user interfaces that support creative tasks [41]. *Pronto* mixes the direct manipulation and command language interaction styles: for the connections, it employs a direct manipulation drag-and-drop interface [34, p.231], which tends to be well-suited for exploration, while the effects of a signal triggering are expressed through GDScript code as a command language [34, p.329]. Based on our evaluation, this mix appears to form an effective bridge between visual and textual representations of code that matches the users' desired abstraction level well, as evidenced by *Pronto* themes 1 and 6 in subsection 5.2, "Pre-defined Behaviors ease Implementation" and "Invitation to Prototyping". Below we described how *Pronto* supports aspects that visual tools are often lauded for, while also exposing textual code where it appears to support expert users' needs better.

Visual connections for event-driven expressions. The connections appear as an intuitive way to formulate event-based "when ... then ..." statements to model the game's control flow. This aligns with findings that people tend to naturally formulate game rules in a constraint or event-based manner [24] and would thus better support users in working creatively [29]. On the fine-granular level of the effect or "then ...", users of *Pronto* formulate textual code instead. On this level, users change the game state by calculating changed values, for example through mathematical expressions. The mix of visual expression for events and textual expression for effects worked well for users with one exception: users criticized that they not only want control-flow dependencies (events) visualized but also some data dependencies (state accesses). Based on our own use, we hypothesize that users would prefer not to manually draw logical connections that can be inferred from text, but to have an automatic connection appear when they reference a variable. For example, when a value important to the game is read, such as a score, they wished for connections between the *State* behavior storing the score and the connections that read or write the value. We derive that users appear to conceptualize both temporal and logical dependencies visually, whereas they conceptualize seemingly all other expressions textually, akin to mathematical formulae. *Pronto*'s mix of visual and textual would thus match the natural way users prefer to express different forms of expressions. Concerning interactions with the visual code as connections, we had hypothesized in our cognitive walkthrough that heavy reliance on drag-and-drop may pose an issue (Insight 4: Drag-and-drop) but the user study did not confirm this hypothesis.

Palettes for authoring expressions. Another prominent feature of visual programming environments is palettes: collections of all elements of the visual notation. In Scratch [28] or Snap [11] these appear as sidebars, filled with blocks that users can drag onto their canvas. In other visual programming environments, similar concepts to palettes are used to present collections of all elements of the visual notation. For example, in the node-and-wire programming environments of Unreal Blueprints [38] or Blender [4], palettes instead appear as searchable menus that instantiate new nodes. Such a "self-revealing" design, where the user interface clearly informs users what can be done, appears to support discovery and experimentation [29] and is thus especially desirable in a prototyping context. In *Pronto*, we have two dialogs that act akin to palettes: first, the list of behaviors informs users of what categories of actions or events are exposed. Second, when formulating a connection, we present the full lists of signals and methods that the involved nodes expose to users. Users can thus draw a connection between two objects they want to have interact with each other without having to know their API. The system then presents the user how the two connected objects can interact with one another through the list of signals and methods.

6.2 The Single View Supports Rapid Prototyping

The insights of our evaluation point at multiple benefits of the combination of game logic and scene in one view for rapid prototyping. We describe our insights in the following.

Co-location of elements supports experimentation. Parallel exploration of alternatives has been pointed out as an important aspect of creative work [36]. The combination of logic and scene elements enables users to select the full unit of a game object at once. This form of co-location not only supports overview but users also perceived it as useful for experimentation: through one select and one duplicate action, they could create a full copy of their design and begin changing aspects without losing their prior state. For comparison, in Godot, users need to find and duplicate the right subscene file, a script file, and then instantiate the new subscene next to the original. Users pointed out the benefits of co-located elements in *Pronto* through the statements collected in *Pronto* Theme 6 "Invitation to (Rapid) Prototyping" that express more willingness to throw elements away and experiment.

Exposed magic numbers support experiments. Once a basic implementation of a prototype had been completed, users began tweaking

parameters they introduced. Often, these were represented as simple "magic numbers" in their Godot scripts. Thus, even though Godot supports live-reloading of scripts and values, tweaking parameters of multiple scripts that depend on one another requires frequent switches between panels, scripts, or objects. In contrast, as *Pronto* consolidates all game objects, their logic, and thus also their magic numbers in one screen, participants were generally able to access any relevant number without requiring switching panels. A common pattern to circumvent this issue is to collect all relevant parameters in a single global script in Godot. Users of *Pronto* benefit even without this pattern, and thus can also experiment with values they had not previously promoted to be in the global script.

Interfaces of behaviors matched users' needs but may not generalize. Our cognitive walkthrough indicated in "Insight 1: Behavior granularity" that the behaviors should match the tasks well but may lead to some confusion due to overlapping responsibilities. Indeed, our user study showed that behaviors matched the level of granularity that participants were looking for while creating their prototypes as documented in *Pronto* Theme 1: they neither exposed too much nor too little detail. This is in contrast to Godot, as described in Godot Theme 4 in subsection 5.2 ("Complex interfaces"). However, in the general case, two concerns with our approach to behaviors became apparent through the study. First, behaviors for a specific functionality may be missing and would have to be created by the designer as part of or in preparation for the prototyping session. Second, behaviors may not be at the adequate level of abstraction: as of right now, behaviors act as primitives in the system, such that users must either use the behavior as-is or recreate its functionality from scratch using lower-level behaviors. As an example, consider the high-level PlatformController behavior that encapsulates input handling and causing movement. While the behavior works well for basic platforming, it does not support double-jumping, i.e., triggering a second jump while in the air. Consequently, users would have to recreate its functionality using input handling and movement behaviors instead. In many situations, this limitation is likely even desirable: if the mechanic the user wants to test is concerned with movement, the extra control gained from a custom implementation could help, whereas the default movement behavior can be helpful if the prototype is concerned with other aspects. Still, in its current implementation, a mix of abstraction levels manifested as an issue when users perceived an overlap, as documented in *Pronto* theme 2, "Distinction and interoperability of pre-defined behaviors". As part of future work, it would be interesting to allow gradually lowering the level of abstraction of behaviors. Designers could thus gain more control when they need it by asking *Pronto* to display a single higher-level behavior as a modifiable composition of multiple lower-level behaviors.

Feedback concerning prototyping speed. To determine quantitatively whether users were able to prototype faster is not possible through the study we performed, as we only collected qualitative insights. At the same time, assessing time to completion of a prototype may be an undesirable or misleading metric [29], as much of the design of *Pronto* aims to get users to experiment with their design. In terms of a qualitative comparison between the *Pronto* and Godot, participants expressed frustration about switching contexts in Godot (Godot theme 2, "Context switches slow down prototyping"). They lauded *Pronto* for encouraging experimentation while hinting at fast access to relevant parts as a possible reason, as documented in *Pronto* theme 6, "Invitation to Prototyping".

6.3 Runtime Visualizations for Debugging

The need to debug brings designers into a different mindset compared to prototype creation or tweaking, where they either opportunistically attempt to find the defect in their game or follow a structured approach to narrow down possible causes [42]. Godot has excellent support for synchronizing changes in the editor to the running game, which *Pronto* inherits. Using live-synchronization, designers can quickly modify state or test hypotheses to narrow down the problem space.

Still, participants of our studies pointed out how they preferred debugging in *Pronto*, as expressed in Godot theme 3, "Time and effort intensive debugging" and *Pronto* theme 3, "Runtime visualizations fulfills concrete need". Based on the observations we made, we believe *Pronto* appears to better support debugging because of its single scene view. Users appeared to make use of two advantages: first, the entirety of their program's logic can be seen on one screen. As connections blink when activated, they have an efficient way to find entry points for potential failures due to missing or too frequent activations. Second, when users wanted to live-edit or inspect properties in Godot, they still needed to switch scenes or scripts, whereas in *Pronto*, the same live-editing and inspection facilities are present but all objects are directly accessible.

Depending on the type of bug, the event-based connections notation may make it harder to trace the issue, as it may sometimes introduce indirections or additional dependencies. Further, using Godot's built-in step-wise debugger is impractical to trace connection activations, as users have to first traverse *Pronto*'s stackframes for every activation. Semantic stepping [37] may present a solution to this issue. During our study, no bugs that were obscured due to the connections occurred, however.

6.4 Limitations of The Design

Pronto is explicitly designed for prototyping of a single or very few mechanics in an isolated setting. This constraint allows it to fit all concerns in the single scene view, which, as discussed, appears to bring many of the benefits that users experienced during prototyping.

Readability of complex scenes. Issues manifested in particular when users were trying to work with scenes that spread out far on the screen or that contained many connections. When zoomed out to see all connections, users were unable to read labels of connections that communicate important aspects of the connections' functionality, as documented in *Pronto* theme 4, "Readability of Connections". The limit of what fits well on a single screen in *Pronto* forms a ceiling of practical project complexity. We believe this ceiling is higher in Godot because users are encouraged to introduce their own abstractions: users will routinely create subscenes for the player, enemies, or other relevant objects. Consequently, when other users want to comprehend a game setup, they can benefit from these game-specific abstractions. In contrast, as per *Pronto*'s

design, abstraction is not supported in the same way: all connections are visible at all times. This issue was hinted at in our cognitive walkthrough as "Insight 2: Refactoring connections". While spatial proximity and the connections themselves should help users realize whether a connection is relevant or not for the aspect of the game they want to understand, designers cannot just encapsulate and thus hide a complex piece of game logic in a subscene.

Going beyond prototypes. As *Pronto* has access to all functionality of Godot, there are no games that could only be expressed in Godot but not in *Pronto*. Still, the design constraint of containing all game logic in a single scene puts a natural limit to complexity of scenes. We have observed users create games whose complexity surprised us but the benefits that come with *Pronto* also diminish as users exceed the screen space: connections were cutting across game objects, making them difficult to duplicate, or many connections fired after every action, making the visualization less useful.

Integrating with Godot. Pronto's functionality, in theory, integrates without issue into a regular Godot game development workflow: users can add behaviors and connections in subscenes, e.g., just for the player character, and reuse those scenes as part of a larger game. While this would preserve many of the benefits users appreciated when creating scenes, the debugging experience and overview would necessarily suffer, as concerns get spread across multiple files. It would be interesting future work to investigate how users could spread behaviors across scenes and still gain the benefits of *Pronto* reported in our study, for example by selecting subscenes of interest to them to get runtime feedback for.

Generating textual code. To further integrate with common game development workflows, the logic expressed through *Pronto*'s connections could even be inlined in readable, generated textual code. Connections are a combination of user-authored expressions and Godot signals that cooperate with Godot nodes and behaviors. Generated code for the example in Figure 3 would thus take a form akin to the following GDScript, where behaviors are instantiated off-screen and connections are instead expressed through Godot code that combines the user-authored snippets with a `connect` call:

```
extends Node2D

def _ready():
    $Button.connect("pressed", $door, func():
        $door.queue_free())

    var press = KeyBehavior.new()
    press.key = "a"
    press.connect("pressed", $door, func():
        $door.queue_free())
```

7 Conclusion

In this paper, we investigated the effect of combining scene objects and game logic in a single view to create throwaway prototypes for game mechanics rapidly. We evaluated our implementation to test this idea, *Pronto*, in a cognitive walkthrough and a user study that compared the prototyping workflow in *Pronto* to Godot. Our findings indicate that participants felt well supported by *Pronto*: they appeared to experiment more than in Godot and reported that they felt more efficient in creating and debugging prototypes. *Pronto*'s mix of visual and textual code worked well in our study. Issues occurred when users began to exceed the natural boundary of *Pronto*'s design, the single screen. While *Pronto* remains usable, the advantages begin to diminish when users construct complex scenes with many connections or spread scenes out, such that they no longer fit on one screen.

Based on these insights, we hope that more game prototyping tools consider choosing a similar trade-off: tools may experiment with designs that constrain users to a single screen to bring them the benefits of immediate overview and access. They may also investigate the use of hybrid visual and textual notations, where visual notations are used for those parts that users appear to conceptualize visually and text otherwise.

References

[1] Tom Beckmann, Eva Krebs, Leonard Geier, Lukas Böhme, Stefan Ramson, and Robert Hirschfeld. 2024. Ghost in The Paper: Player Reflex Testing with Computational Paper Prototypes. *PPIG 2024 Proceedings* 35th Annual Workshop (2024), 8–19. https://www.ppig.org/files/2024-PPIG-35th--proceedings.pdf

[2] Paul Beynon-Davies, C Carne, Hugh Mackay, and D Tudhope. 1999. Rapid application development (RAD): An empirical review. *European Journal of Information Systems* 8 (09 1999). doi:10.1057/palgrave.ejis.3000325

[3] Paul Beynon-Davies, Hugh Mackay, and Douglas Tudhope. 2000. 'It's lots of bits of paper and ticks and post-it notes and things . . .': a case study of a rapid application development project. *Information Systems Journal* 10, 3 (2000), 195–216. doi:10.1046/j.1365-2575.2000.00080.x arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1046/j.1365-2575.2000.00080.x

[4] Blender Online Community. 1994. *Blender - a 3D modelling and rendering package.* Blender Foundation, Blender Institute, Amsterdam. http://www.blender.org

[5] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative Research in Psychology* 3, 2 (01 2006), 77–101. doi:10.1191/1478088706qp063oa

[6] Andrea A. diSessa. 1985. A principled design for an integrated computational environment. *Hum.-Comput. Interact.* 1, 1 (March 1985), 1–47. doi:10.1207/s15327051hci0101_1

[7] Henrik Engström, Jenny Brusk, and Patrik Erlandsson. 2018. Prototyping Tools for Game Writers. *The Computer Games Journal* 7, 3 (June 2018), 153–172. doi:10.1007/s40869-018-0062-y

[8] Vanessa Freudenberg, Yoshiki Ohshima, and Scott Wallace. 2009. Etoys for One Laptop Per Child. In *2009 Seventh International Conference on Creating, Connecting and Collaborating through Computing*. IEEE Computer Society, USA, 57–64. doi:10.1109/C5.2009.9

[9] Tracy Fullerton and Chris Swain. 2004. *Game Design Workshop.* CMP Books, London, England.

[10] T. R. G. Green and M. Petre. 1996. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages & Computing* 7, 2 (01 06 1996), 131–174. doi:10.1006/jvlc.1996.0009

[11] Brian Harvey and Jens Mönig. 2015. Lambda in blocks languages: Lessons learned. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond).* IEEE, USA, 35–38. doi:10.1109/BLOCKS.2015.7368997

[12] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the future: the story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Atlanta, Georgia, USA) *(OOPSLA '97).* Association for Computing Machinery, New York, NY, USA, 318–326. doi:10.1145/263698.263754

[13] Alan Kay. 2005. Squeak etoys, children & learning. *online article* 2006 (2005), 1–8. http://www.squeakland.org/resources/articles

[14] Amy J. Ko, Thomas D. LaToza, and Margaret M. Burnett. 2015. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering* 20, 1 (01 02 2015), 110–141. doi:10.1007/s10664-013-9279-3

[15] Eva Krebs, Tom Beckmann, Leonard Geier, Stefan Ramson, and Robert Hirschfeld. 2023. Pronto: Prototyping a Prototyping Tool for Game Mechanic Prototyping. *PPIG 2023 Proceedings* 34th Annual Workshop (2023), 157–168. https://www.ppig.org/files/2023-PPIG-34th--proceedings.pdf

[16] Jens Lincke, Robert Krahn, Dan Ingalls, and Robert Hirschfeld. 2009. Lively Fabrik A Web-based End-user Programming Environment. In *2009 Seventh International Conference on Creating, Connecting and Collaborating through Computing*, Vol. 36.

IEEE, USA, 11–19. doi:10.1109/c5.2009.8
[17] Colleen Macklin and John Sharp. 2016. *Games, Design and Play.* Addison-Wesley Educational, Boston, MA.
[18] John Maloney. 1995. Morphic: The self user interface framework. *Self 4.0 Release Documentation* 1995 (1995), 1–27.
[19] John H. Maloney and Randall B. Smith. 1995. Directness and Liveness in the Morphic User Interface Construction Environment. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology* (Pittsburgh, Pennsylvania, USA) *(UIST '95)*. Association for Computing Machinery, New York, NY, USA, 21–28. doi:10.1145/215585.215636
[20] Jon Manker and Mattias Arvola. 2011. Prototyping in game design: externalization and internalization of game ideas. In *Proceedings of the 2011 British Computer Society Conference on Human-Computer Interaction, BCS-HCI 2011, Newcastle-upon-Tyne, UK, July 4-8, 2011*, Linda Little and Lynne M. Coventry (Eds.). ACM, New York, NY, USA, 279–288. http://dl.acm.org/citation.cfm?id=2305366
[21] Javier Marco, Eva Cerezo, and Sandra Baldassarri. 2012. ToyVision: a toolkit for prototyping tabletop tangible games. In *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems* (Copenhagen, Denmark) *(EICS '12)*. Association for Computing Machinery, New York, NY, USA, 71–80. doi:10.1145/2305484.2305498
[22] Janice M. Morse. 1995. The Significance of Saturation. *Qualitative Health Research* 5, 2 (01 05 1995), 147–149. doi:10.1177/104973239500500201 Publisher: SAGE Publications Inc.
[23] Jakob Nielsen and Thomas K. Landauer. 1993. A mathematical model of the finding of usability problems. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems* (New York, NY, USA) *(CHI '93)*. Association for Computing Machinery, New York, NY, USA, 206–213. doi:10.1145/169059.169166
[24] John Pane, Chotirat Ratanamahatana, and Brad Myers. 2000. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human Computer Studies* 54 (10 2000), 237–264. doi:10.1006/ijhc.2000.0410
[25] Marian Petre. 2006. Cognitive dimensions 'beyond the notation'. *Journal of Visual Languages & Computing* 17, 4 (01 08 2006), 292–301. doi:10.1016/j.jvlc.2006.04.003
[26] 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. *The Art, Science, and Engineering of Programming* 3, 1 (2019), 1. doi:10.22152/programming-journal.org/2019/3/1
[27] Alex Repenning. 1993. Agentsheets: a tool for building domain-oriented visual programming environments. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems* (Amsterdam, The Netherlands) *(CHI '93)*. Association for Computing Machinery, New York, NY, USA, 142–143. doi:10.1145/169059.169119
[28] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (nov 2009), 60–67. doi:10.1145/1592761.1592779
[29] Mitchel Resnick, Brad Myers, Kumiyo Nakakoji, Ben Shneiderman, Randy Pausch, and Mike Eisenberg. 2005. Design Principles for Tools to Support Creative Thinking. *Report of Workshop on Creativity Support Tools* 20 (01 01 2005).
[30] Emanuel Montero Reyno and José Á Carsí Cubel. 2009. Automatic prototyping in model-driven game development. *Comput. Entertain.* 7, 2, Article 29 (jun 2009), 9 pages. doi:10.1145/1541895.1541909
[31] Emanuel Montero Reyno and José Á. Carsí Cubel. 2008. Model Driven Game Development: 2D Platform Game Prototyping. In *GAMEON'2008, (Covers Game Methodology, Game Graphics, AI Behaviour, Game AI Analysis, AI Programming, Neural Networks and Agent Based Simulation, Team Building, Education and Social Networks), November 17-19, 2008, UPV, Valencia, Spain*, Vicente J. Botti, Antonio Barella, and Carlos Carrascosa (Eds.). EUROSIS, Belgium, 5–7.
[32] José Rouillard, Audrey Serna, Bertrand David, and René Chalon. 2014. *Rapid Prototyping for Mobile Serious Games.* Springer International Publishing, USA, 194–205. doi:10.1007/978-3-319-07485-6_20
[33] Jesse Schell. 2015. *The art of game design: a book of lenses.* CRC Press, USA.
[34] Ben Shneiderman, Catherine Plaisant, Maxine Cohen, Steven Jacobs, Niklas Elmqvist, and Nicholas Diakopoulos. 2016. *Designing the User Interface: Strategies for Effective Human-Computer Interaction* (6th ed.). Pearson, London, UK.
[35] Iris Soute, Tudor Vacaretu, Jan De Wit, and Panos Markopoulos. 2017. Design and Evaluation of RaPIDO, A Platform for Rapid Prototyping of Interactive Outdoor Games. *ACM Trans. Comput.-Hum. Interact.* 24, 4, Article 28 (aug 2017), 30 pages. doi:10.1145/3105704
[36] Michael Terry, Elizabeth D. Mynatt, Kumiyo Nakakoji, and Yasuhiro Yamamoto. 2004. Variation in element and action: supporting simultaneous development of alternative solutions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vienna, Austria) *(CHI '04)*. Association for Computing Machinery, New York, NY, USA, 711–718. doi:10.1145/985692.985782
[37] David Ungar, Henry Lieberman, and Christopher Fry. 1997. Debugging and the experience of immediacy. *Commun. ACM* 40, 4 (April 1997), 38–43. doi:10.1145/248448.248457
[38] Nicola Valcasara. 2015. *Unreal Engine Game Development Blueprints.* Packt Publishing Ltd, Birmingham, UK.
[39] Robert A. Virzi. 1992. Refining the Test Phase of Usability Evaluation: How Many Subjects Is Enough? *Human Factors* 34, 4 (01 08 1992), 457–468. doi:10.1177/001872089203400407 Publisher: SAGE Publications Inc.
[40] Markus Weninger, Paul Grünbacher, Elias Gander, and Andreas Schörgenhumer. 2020. Evaluating an Interactive Memory Analysis Tool: Findings from a Cognitive Walkthrough and a User Study. *Proceedings of the ACM on Human-Computer Interaction* 4 (18 06 2020), 1–37. Issue EICS. doi:10.1145/3394977
[41] Yasuhiro Yamamoto and Kumiyo Nakakoji. 2005. Interaction design of tools for fostering creativity in the early stages of information design. *International Journal of Human-Computer Studies* 63, 4 (2005), 513–535. doi:10.1016/j.ijhcs.2005.04.023 Computer support for creativity.
[42] Andreas Zeller. 2005. *Why Programs Fail: A Guide to Systematic Debugging.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

A Interview Questions and Related Cognitive Dimensions of Notation

Table 4 presents the set of questions used in our interviews. 13 of these questions can be directly attributed to at least one Cognitive Dimension. This follows a similar structure in prior work [40].

B Actions in the Cognitive Walkthrough

A visualization of the actions in the cognitive walkthrough is shown in Figure 8.

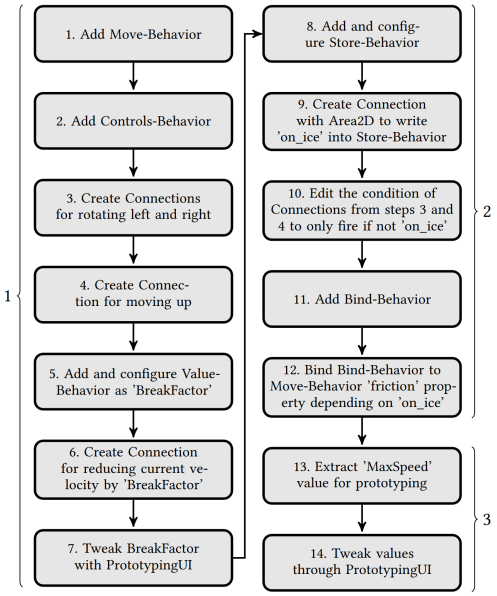

Figure 8: Sequence of actions required to solve the cognitive walkthrough's task, labeled by sub-goal 1 through 3: creating a top-down car with a sliding mechanic and special behavior on ice terrain.

C Detailed Task Design

Dashing Mechanic Prototype. With *Pronto*'s strong focus on supporting 2D game mechanic prototyping, the goal was to select two typical mechanics one might see in a standard video game. The first task chosen was the implementation of a dashing mechanic for

Table 4: Overview of our interview questions and their relationship to the Cognitive Dimensions of Notations

Question	Dimension													
	Viscosity	Visibility	Role-expressiveness	Abstraction Gradient	Closeness of Mapping	Consistency	Diffuseness	Hard Mental Operations	Error-Proneness	Hidden Dependencies	Premature Commitment	Progressive Evaluation	Secondary Notation	General
What is your opinion about the steps you had to take for creating your initial implementation (sub-task-1) with the tool?	x													
What was your process for adapting the mechanic to the new environment (sub-task-2 & 3)?	x													
What was it like to make multiple changes to your implementation (sub-task-2 & 3)?	x													
Could you please describe the tools expressiveness in the context of this task?		x	x											
Briefly describe how you interacted with existing abstractions (predefined components, interfaces, lifecycle methods) and what, if anything, you were missing?				x										
Was there anything that surprised you about the tool you were using?					x	x								
If anything, was there something that was unnecessarily extensive or complex to do?							x	x						
If anything, what was especially easy to accomplish?							x	x						
Did you encounter any errors? And if so, do you believe the tool could have helped to prevent them?									x					
What, if anything, would you want the tool to communicate more clearly?										x				
Did the tool support the order of your creative process or did you have to adapt your process to the tool's?											x			
How well were you able to track your implementation progress for each subtask?												x		
Would you, and in what way, have benefited from additional ways to annotate, organize or structure your implementation?													x	
What was your overall impression of the experience?														x
If you could change exactly one thing about the tool you were using, what would you change?														x

an existing player character. Dashing, or the temporary and rapid acceleration of a player's character in the direction of choice, is a common feature in many video games. Some even make this their primary means of traversing levels. A select few games that have a significant focus on dashing are the indie hit *Celeste*[11], where the player guides a young woman on her quest to climb an enormous mountain, *Ori and the Blind Forest*[12], where the player controls a guardian spirit on a journey to restore a dying forest, and *Enter the Gungeon*[13], a top-down rougelike where dashing is one of the only defensive moves available for escaping the hoard of bullets that are shot the player's way.

Sub-Task 1. During the study, participants must initially implement a dash to allow them to cross the gap between the two sides from left to right. No further requirement is given. The player's jump alone does not allow for crossing the gap.

Sub-Task 2. Having implemented the initial dash, the participants are now confronted with a much wider gap, including two platforms of different heights that they must utilize to cross the gap. As this would not be possible with a purely horizontal dash, the task also required participants to implement dashing in at least eight directions.

Sub-Task 3. Finally, the level changes again, and participants must fine-tune their mechanic to traverse platforms of various sizes and in different positions to reach a marked goal on the other side of the screen. This level progression is visualized in Figure 9.

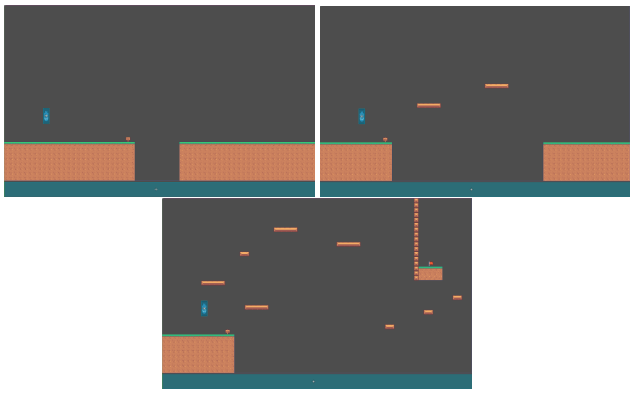

Figure 9: Overview of dash task level changes from sub-task to sub-task

Driving and Sliding Mechanic Prototype. Another mechanic often found in games is the movement of vehicles. Driving can be an integral part of various games and is often modified based on the desired game effect, ranging from *Mario Kart's*[14] arcade-like driving and drifting all the way to *SnowRunner's*[15] hyper-realistic traction simulation.

Sub-Task 1. Presented with an empty level, participants must implement driving for an existing car object. They are tasked with ensuring the car can accelerate, brake, and steer. Additionally, the car must have a capped top speed and be affected by friction.

[11] https://www.celestegame.com/, last accessed: 2025-02-11
[12] https://www.orithegame.com/blind-forest/, last accessed: 2025-02-11
[13] https://enterthegungeon.com/, last accessed: 2025-02-11
[14] https://store.nintendo.de/de/mario-kart, last accessed: 2025-02-11
[15] https://www.focus-entmt.com/en/games/snowrunner%20, last accessed: 2025-02-11

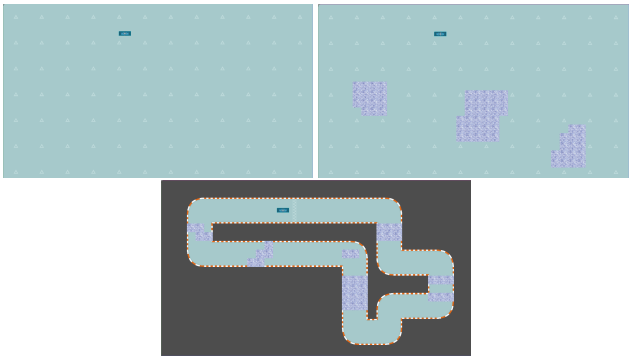

Figure 10: Overview of car task level changes from sub-task to sub-task

Sub-Task 2. Having done so, participants are presented with a level that contains icy patches. When the car drives over the ice, all controls need to be disabled, and the car should continue sliding on the ice until it once again reaches a non-ice surface. Once this is implemented and demonstrated, participants may move on to the final sub-task.

Sub-Task 3. The level changes once more. Now, it contains a race track as well as icy patches, and participants must continuously tweak their driving mechanic to complete at least one lap without leaving the track. This level progression is visualized in Figure 10.