Pronto: Prototyping a Prototyping Tool for Game Mechanic Prototyping

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

Tom Beckmann
Hasso Plattner Institute
University of Potsdam
tom.beckmann@hpi.de

Leonard Geier
Hasso Plattner Institute
University of Potsdam
leonard.geier@hpi.de

Stefan Ramson
Hasso Plattner Institute
University of Potsdam
stefan.ramson@hpi.de

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
robert.hirschfeld@uni-potsdam.de

Abstract

The development of video games revolves to a large extent around the *feel* of an idea. From the very beginning, developers need to be able to quickly create and try out as many ideas as possible, as assessing the feel of an idea, and thus its viability as a game, is best done through experiencing it. An approach to prototyping is essential in this regard, as it allows developers to identify promising ideas without committing too much time or resources.

To support developers in prototyping game mechanics, we created the *Pronto* framework for the Godot game engine. This framework focuses on fast, throw-away prototypes for specific game mechanics created through visual interactions mixed with code.

Pronto consists of a set common, modular concerns in games, such as moving or colliding, called Behaviors. Developers place Behaviors visually in the game scene and connect them through code also placed in the game scene to achieve their desired effect. The first version of Pronto was itself a prototype, only suitable for a very narrow range of games. To approach a still minimal, yet flexible set of behaviors that allow developers to create any kind of game, we designed a university course where students alternate between working with the framework and extending it. In the process, they iteratively identified and addressed shortcomings and potentials of Pronto.

In this paper, we present Pronto as a tool for game developers to quickly validate game mechanics ideas, as well as the process and results of the students in the seminar to improve it.

1. Introduction

Game development benefits from fast iterations (Schell, 2014, p. 94). Developers will need to tweak and play test all parts of a game many times to ensure that they both work correctly and are fun to play (Murphy-Hill, Zimmermann, & Nagappan, 2014). This experimentation is important before main development even starts; there are many ideas for potential games, but only some of those are actually feasible or fun to play. To find these promising ideas, developers seek ways to quickly and easily try out ideas to find the few they want to polish to a full game (Kasurinen, Strandén, & Smolander, 2013).

Games are a visual medium. Developers have introduced ways to effectively prototype and obtain immediate feedback for many of the visual aspects of games via direct manipulation. For example, creating a game level is often done by means of dragging and placing assets directly at the position they will later appear at in the game.

In contrast, defining behavior is usually achieved through code. To leverage the conveniences of modern development suites, the development even tends to occur in a separate application than the game engine. This separates how developers edit concrete, visible game elements from how they edit the abstract code behind the behavior of those elements. As a result, developers have to go through multiple steps to feel how changes to the code will impact the behavior of the game. For effective prototyping, however, developers benefit from experiencing the effect of their changes as quickly as possible. Games are about the experience players have while playing them; for instance, it might be important for a mechanic

that allows players to bounce around the game world to feel fun, natural, and non-frustrating. In another situation a game might want to include a difficult chase sequence that partly feels stressful and frustrating on purpose. Game developers need access to these human experiences as often as possible during development.

To address the gap between authoring and experiencing, we designed a prototyping framework called Pronto, aimed at throw-away prototypes to validate ideas for game mechanics. For example, we want developers to be able to quickly try out whether a car racing game where the cars have a grappling hook for special movement is feasible and/or fun. Or whether a platformer where the player may invert gravity in certain parts of the game world is interesting, and so on. Pronto currently does not aim to be a tool for non-throw-away prototypes like vertical slices that are usually the basis for a later complete game after developers have already gained insights from a lot of smaller prototypes and ideas.

Pronto's main idea is to move code that is currently in source files, removed from the game, to the game itself, by scattering snippets of code into the visual representation of the game's scene, close to where they will manifest. Scattered code is anchored by means of composable Pronto *Behaviors*. These Behaviors are visual representations of a concern, such as collisions with other objects, moving an object around, or listening to player input, that are placed visibly in the game's scene and display relationships to the objects in the scene that are involved in their function.

Pronto's main benefit for prototyping is dependent on finding a comprehensive but not overwhelming set of composable Behaviors. Given too many specific Behaviors, developers might no longer be able to locate the desired Behaviors and their function may not be sufficiently flexible to allow for useful combinations with other Behaviors. Given too few, developers will need to resort back to writing code in source files to achieve their goals. A sweet spot would yield a limited few but powerful and expressive Behaviors that can be combined to achieve anything the game engine is capable of. To approach this sweet spot, we designed a seminar where students alternate between prototyping games with Pronto and extending or changing the framework to better suit the needs uncovered during their own work with the framework.

In this paper we give an overview of game prototyping in general as well as the game engine we used as a basis for our framework in section 2. We introduce the Pronto prototyping framework and its concepts in section 3. To demonstrate how and for what our prototyping framework can be used, we provide a walkthrough in section 4 followed by a description of and results from our prototyping seminar in section 5. We conclude this paper with a summary and discussion of future work in section 7.

2. Iterating Games

Game development usually relies on fast iterations. Developers need to tweak and try ideas to see if they work as intended and are fun to play. Particularly, developers often try many ideas as prototypes to find the few promising ideas that they want to commit to. Since resources for development are usually limited, it is important to determine quickly which ideas are feasible before starting a complete development process (Schell, 2014; Kultima, 2015).

Game developers often use game engines. Game engines are development systems that streamline game creation by providing tools and solutions for tasks that often appear during game development, thus making main parts of game development reusable. Since many games are visual, game engines usually have dedicated tools for creating and editing the visual game scene. Game engines also usually include a code editor and might provide direct support for patterns that are often needed for games, such as a game loop (Nystrom, 2014). Since game engines are a core part of game development, we integrated our prototyping framework into an existing game engine.

2.1. Godot - An Open-source Game Engine

Pronto was created for the Godot game engine. We chose Godot because it is a light-weight, open-source game engine with excellent support for custom extensions(Juan Linietsky & contributors, n.d.). Godot supports both the visual creation of game scenes as well as code editing, see Figure 1.

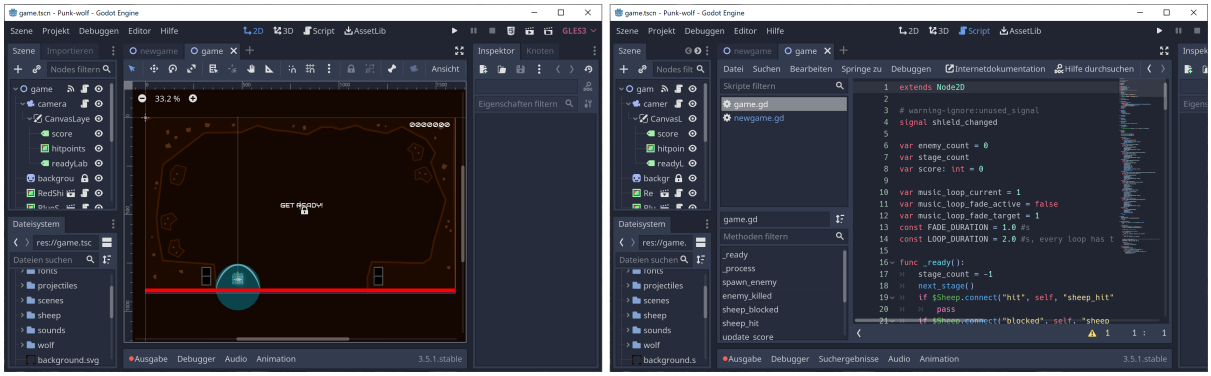

Figure 1 – The game scene editor of the Godot game engine for a 2D game can be seen on the left. The right shows the code editor of the Godot game engine for a script of the same game.

Godot includes visual scene editors for both 2D and 3D graphics. Developers can edit these scenes in an interactive, visual editor via drag and drop as well as by using specialized sidebars that provide additional editing options (e.g. setting the color of a node via a color picker). A scene consists of elements ordered in a scene tree. These tree elements are called nodes. While users can create their own nodes, Godot provides a variety of pre-defined nodes, e.g. for buttons and other visual elements. Scenes themselves can also be used as nodes. Since the scenes contain all visual information about the game, they look and feel very close to the game when it is actually run. However, behavior is defined as code in so called scripts. Scripts are attached to a specific node.

To edit a script, developers need to switch from the visual scene editor to Godot's code editor. Here, behavior of a node in the game world is defined. This editor supports many common code editing features such as syntax highlighting and autocompletion. The main language used for Godot projects is gd script, a programming language similar to Python.

Godot is extensible via an extension system. Extension developers can define custom types of node, modify editing widgets of the engine, or add additional visual cues into the game world preview. Extensions in Godot may use the same default language as scripts written for the game, allowing game developers to become tool developers with a low barrier.

3. The Pronto Prototyping Framework

The Pronto prototyping frameworks provides a way to visually and quickly create throw-away game mechanic prototypes. The framework is not designed for creating entire games; it is meant for quickly trying new, small ideas, like a rough sketch of a game mechanic.

The core idea behind Pronto is to bring code written for a game's behavior closer to the visual representation of the game. By having developers interact with visual elements in the game scene, we hope to reduce the need for context switches to a code-only view. Pronto aims to make behavior that is separate from the game scene, or "invisible", concrete and tangible directly in the scene itself. This should make it easy to create and modify parameters of game behavior.

The core concepts Pronto provides are *Behaviors* and *Connections*. Behaviors are custom Godot nodes that are a visual representation and encapsulation of a game element or aspect. Like all nodes, they can be placed and edited in the visual Godot scene editor. Even usually hidden concepts, e.g. the representation of player control keys, are now placed visually, and spatially, in the scene editor. To facilitate interaction between Behaviors, Connections can be added between them, e.g. a Connection between a *Timer* and *Spawner* could be used to make the Spawner spawn new items regularly. These Connections are also visually represented, see Figure 2.

Types of Behaviors include but aren't limited to: Behaviors that trigger events (e.g. based on time or a collision), Behavior that cause certain actions when triggered (e.g. *Move* that moves its parent node or

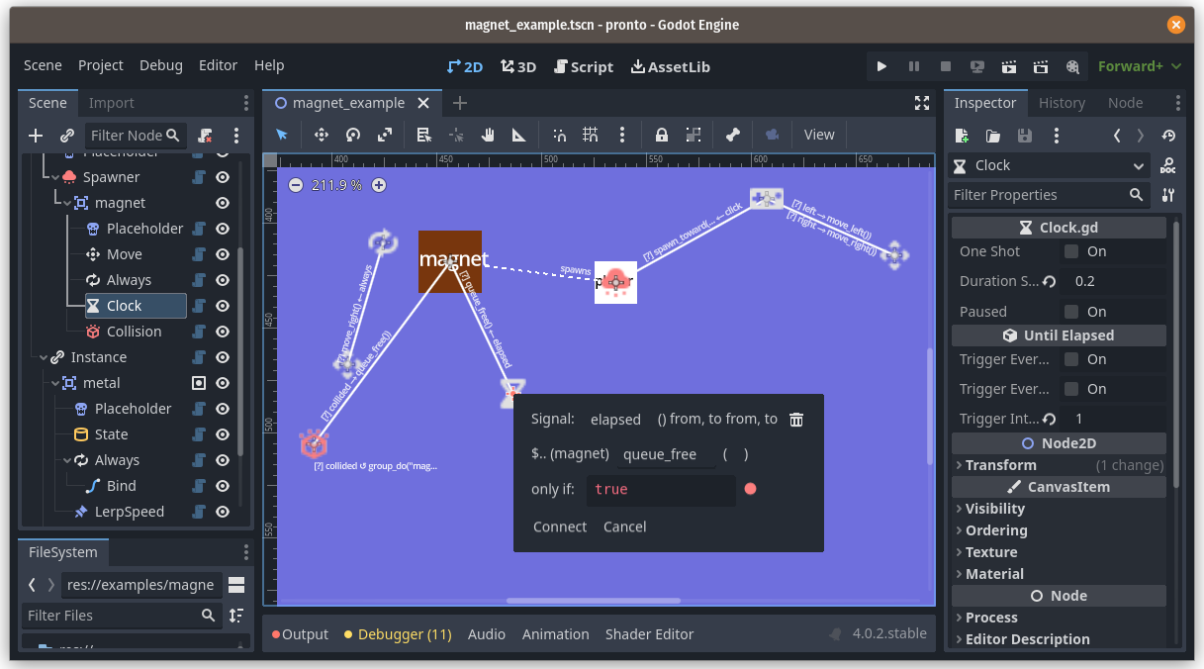

Figure 2 – The game scene editor of the Godot game engine for a 2D game including the Pronto prototyping framework. Pronto Connections are visible between game elements.

a Spawner that spawns other nodes), behaviors that contain or visualize state (e.g. Value, which can be altered via a Slider or Placeholder, a label), and Godot nodes (e.g. Area2D).

Pronto is compatible with Godot and most of its existing features. Importantly, Pronto is not designed a visual programming tool to make game programming friendlier for beginners. Instead, Connections are merely an offer to game developers to expose behavior in a visual manner, as opposed to hiding it in source code files. In particular, Connections are just standard Godot function calls, reacting to events emitted by other Nodes. Developers write code inside the Connection editor dialog that may for example transform arguments prior to the function call, as shown in Figure 2.

Of special note is that Pronto is compatible with most liveness features of Godot itself. If values are changed in the sidebar of the scene editor while the game is run, the running game will receive the updated value accordingly. Additionally, Pronto visualizes when Connections are triggered in the running game by lighting up the Connection lines in the editor to help developers isolate issues in their Connection setup quickly.

While Pronto aims to make the need for code obsolete, the compability with Godot makes possible to switch to and use the code editor if neccessary. Thus, developers are not limited in what prototypes they can create by our framework.

4. Prototyping with Pronto

The following walkthrough details how Pronto can be used to build a small game mechanic. The example we want to build is part of a racing game. The player needs to be able to control a car and use it to collect items. The finished game can be seen in Figure 4. While not detailed as part of this walkthrough, there are two labels in the game for displaying the current score and fps during testing.

We will first prepare the items that we later want to collect. An item is represented as an *Area2D*, a Godot node that reports collisions. To this main item node we add a *Placeholder*, a Pronto Behavior node that displays a rectangle with a label in the game, in this case "pickup" (top-right of Figure 4).

To spawn items, we add *Spawner* and a *Clock* Behavior to the scene. We add the item is as a child of the

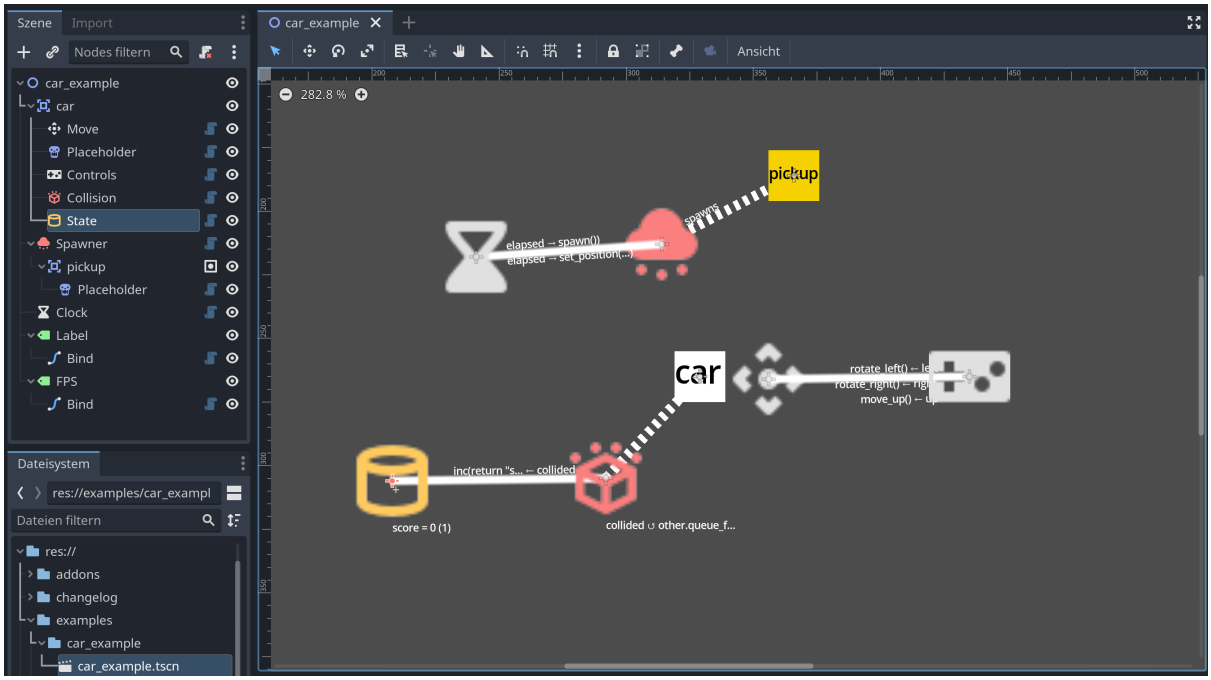

Figure 3 – The example game opened in the scene editor. The Clock *timer is connected to the* Spawner *which in turn is connected to the "pickup" item it spawns. The car node is connected to a* Collision*, while the* Move *child node of the car is connected to the* Controls*.*

Spawner in the node tree, which instructs the *Spawner* to hide the item on game start and create a visible duplicate of it when called. We then add two connections between the *Clock*, acting as a timer, and the *Spawner*. The first connections triggers the *Spawner* and the second moves it to a random position on the screen. We configure the *Clock* to trigger every five seconds.

The second game element we need is the car itself. The car is another *Area2D*. We again add a *Placeholder*, displaying the label "car", as a child (bottom-center of 4). We then add several more children; Move, Controls, Collision, and State.

To make our car controllable by the player, we add connections between Move and the Controls for the *up*, *left*, and *right* triggers. The *Move* behavior moves its parent node when triggered.

The car and *Collision* are then connected. When triggered, we remove the collidee, which in this case is the item. We also connect the *Collision* node with our Pronto *State* node that in this case saves the score of our player. Whenever a collision is triggered, the score is incremented (bottom-left of Figure 4).

Now that the basic setup for a simple racing game is complete, developers may start experimenting with parameters of their setup. For example, they may change the frequency in which coins spawn or the speed of the car.

5. Prototyping Pronto

The following example use cases are from a prototyping seminar we designed with Pronto in mind. The seminar is aimed at graduate computer science students and teaches both game mechanics prototyping as well as working and modifying the tools and frameworks that participants use for development. With Pronto as its subject, the students in the course iteratively and collaboratively extended the scope of games that could be created.

Students worked in small teams; two students per team were encouraged, but they could work on their own if preferred. Nine graduate students participated in the seminar, in five teams. In order for students to create many prototypes, we structured the seminar into several short iterations. Each iterations has

Figure 4 – The example game while running. As some time has passed, some "pickup" items have been spawned. The player can steer the car with their keyboard.

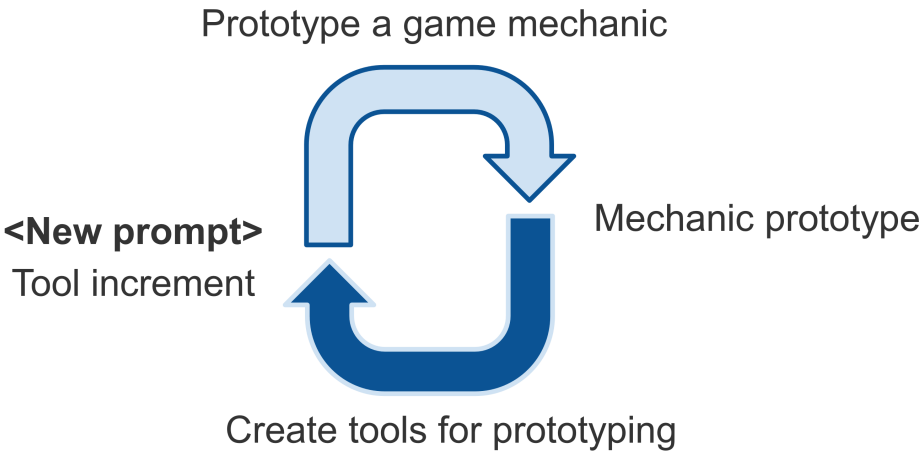

Figure 5 – The two phases of the seminar. In the first phase, students build a game mechanic based on the prompt. In the second phase, students work on Pronto itself.

two phases, each a week long, which is visualized in Figure 5. In the first week, students prototype a new game mechanic. In the second week, students modify Pronto based on insights gained in the previous week. After each phase, students hand in a changelog containing a short video and text detailing what they build and their insights, which is also used to inform the other teams of changes in the use of the framework.

The first phase starts with the students receiving a prompt, e.g. "Build a mechanic for a Terraria-style game"[1] during the weekly meeting. Teams then get to discuss briefly what mechanic they would find interesting. In particular, we encouraged students to consider whether their chosen mechanic would work well in a paper prototype. If so, they are encouraged to look for a different idea that would rely more strongly on the possibilities provided by Pronto. Once chosen, the team presents their plan and the group discusses its applicability and further directions.

Each team then implements their own game mechanic. In this phase each team works on separate branches of Pronto. Students may deliberately break functionality of Pronto or extend it in non-idiomatic ways in this phase. The only goal is to build a prototype of a mechanic as quickly as possible and to obtain feedback on it. The changelog for this phase includes a demo of the mechanic prototype, which parameters or variations were tweaked and considered to be especially interesting during testing, and changes to the Pronto framework that they had to hack in or would like to see to better support their workflow in the previous week.

The goals for the second phase are based on the feedback and change requests to the framework from the previous phase. During a meeting with all teams and the course instructors, changes are discussed. Desired changes are prioritized and feedback from all teams concerning its rough design is gathered. Each team then picks framework modifications from the discussed list and implements the modifications directly on Pronto's main branch. Teams communicate breaking changes through the seminar's group text channel. The changelog at the end of this phase contains a demonstration of how to use the new framework modification as well a textual description of API changes.

The seminar took place during the summer of 2023.

5.1. Phase 1: Building Grappling Hooks with Pronto

For the prompt "Build a mechanic for a Terraria-style game", one team implemented a grappling hook mechanic. The player shoots out a grappling hook that collides with walls and allows the player to move along the hook's line once attached. The team tested three variants of grappling hooks.

The first grappling hook variant is a teleportation-based grappling hook. The player can teleport directly to the hook after it landed, which can be seen in Figure 6. The second variant was considered a "static" grappling, the player can move on a straight line between their original position and the hook. The third and final variant can be considered a "classic" grappling hook; the player can swing around as if connected by rope to the hook. The team experimented with velocity and reach of these variants.

Based on their implementation, the team proposed several changes or new features for Pronto. Two proposals were especially positively received by other teams: the first would extend the list of triggers supported by the *Controls* behavior for mouse drag inputs. Teams had encountered multiple times that they wanted to react to the mouse cursor moving only while a mouse button was held. The proposal would allow to do achieve this without having to keep track of mouse button states manually.

The second proposed change would introduce a new Behavior to be added to Pronto to let developers render a line between two elements in the game, which otherwise required referencing two nodes in code and continuously repositioning and rotating a thin rectangle according to the nodes' positions. The new Behavior instead introduced a new basic component that would realign automatically based on other nodes' positions, for example to easily visualize a grappling hook, laser, or the next target of an enemy during testing. The implementation proposed by the team would only fulfill the basic needs so

[1] `hhttps://terraria.org/`, accessed: 2023-05-31

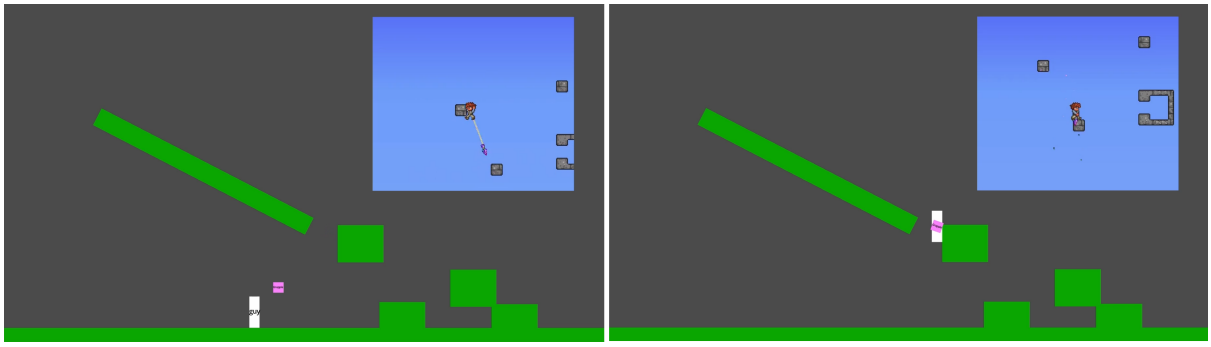

Figure 6 – Phase 1: The grappling hook mechanic that was built by students, in this case the teleportation grappling hook. The left side shows the player while still standing on the ground and shooting the grappling hook. On the right side, the hook has landed and the player has teleported to the hook.

far uncovered by the teams and could be extended further once further potential has become clear.

5.2. Phase 2: Adding Mouse Dragging and Line Visualizations to Pronto

The student team worked on both an extension for mouse dragging and the new line rendering Behavior.

The new *VisualLine* Behavior must be configured to take references to two Nodes in the game scene. When the game is run, a line will be drawn between these two node and update according to their position. This enables building grappling hook lines, laser beams, and other mechanics without leaving the scene editor to reach for code.

To support better mouse handling, the students extended the *Controls* Behavior. Connections to and from *Controls* can now also use a drag trigger, which is fired when a mouse button is held and provides the mouse cursor's position as a parameter. Additionally, the mouse-up event now also provides a duration as a paramater than Connections may choose to use. Both changes can be seen in Figure 7.

5.3. Breaking Changes and Throw-Away Prototypes

Initially, there were issues as we asked students to work in parallel on the same, relatively small Pronto code base without creating separate branches during the framework extension phase. Breaking changes could mean that code written by other teams may no longer work from one commit to the next.

To mitigate the issue, we asked students to consider two factors. First, as mentioned before, breaking changes are to be announced in the seminar's text channel. Second, the prototypes created during the mechanics phase should be kept small and fast to re-create. Not only does this emphasize their nature as throw-away prototypes, it also means that teams do not have to be extra careful when performing more fundamental changes to the framework. If a team needs to explore further directions from a prior prototype, they should be able to re-create the prototype within minutes. As we asked teams to create branches during the prototype phase, there was also no risk of prototypes that teams were still working on to spontaneously break due to other teams' changes.

5.4. After the Seminar

At the end of the seminar, the student teams had successfully created prototypes for six different prompts and made additions as well as changes to Pronto. During the in seminar discussions, students were both able to discuss their prototypes and what the experienced during playtests as well as what they want from and how they want interact with Pronto based on working with it.

Prompts focused on one game aspect, or type of game, that mechanic prototypes should be created for. To give students a direction for their own prototypes, prompts included at least one already existing from that pool as an example together with a list of fitting mechanics from that game. To test Pronto's capabilities, the prompts aimed at covering a variety of possible game mechanics. The final prompt was testing even the pupose of Pronto itself: Should it stay focused on individual mechanics prototypes or

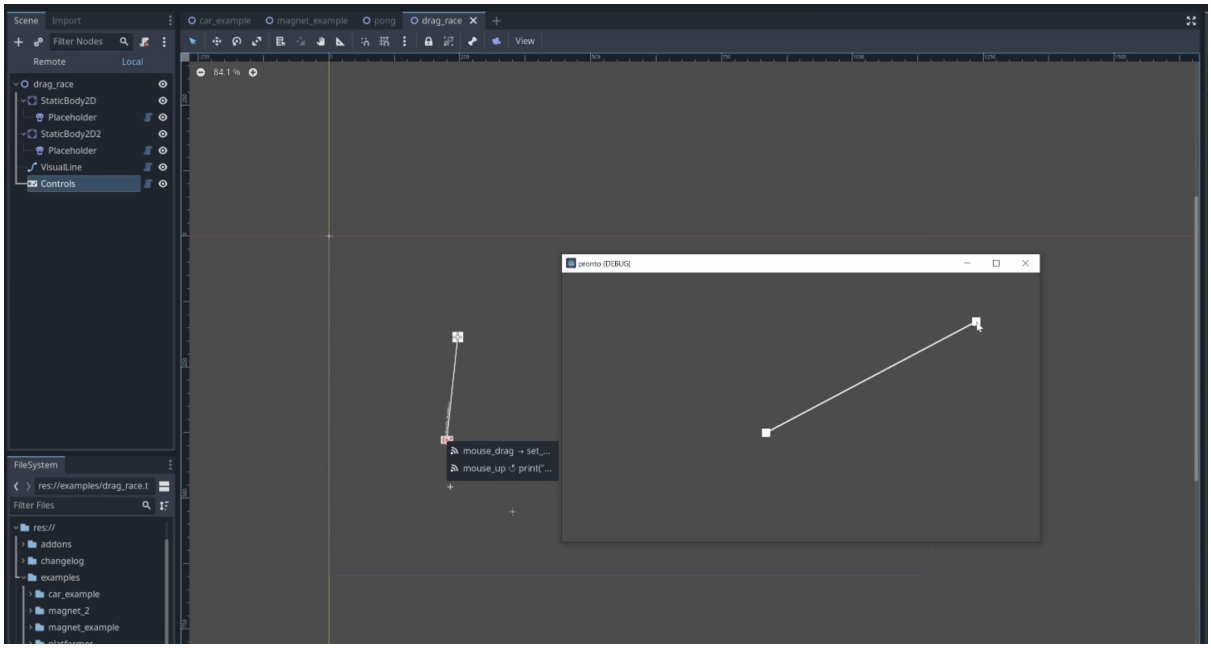

Figure 7 – Phase 2: A screenshot of two Pronto framework modifications that were built by students, visible in both scene editor and the running game. The new Behavior can render lines between two nodes. The Controls *node has extended support for mouse interactions.*

could it even be used to make an "entire" game? Since this prompt was given the same iteration time as all other prompts, one week, this tested of course the feasability for very small games, not larger projects,

The following six prompts, and example games, were used during the seminar:

- Mechanics for a top-down car racing game (e.g. Mario Kart)
- Mechanics that are triggered by player actions in a platformer game (e.g. Terraria)
- Mechanics that focus on the game environment in a platformer game (e.g. Celeste, Mario)
- Mechanics in simulation games (e.g. SimCity)
- Mechanics in turn-based games (e.g. Pokémon)
- An entire game (e.g. Asteroids, Doodle Jump)

While discussing with the students and looking at their created prototypes, it became apparent that Pronto is currently not equally well suited for all kind of mechanics. For mechanics with one or more clear player characters (e.g. the top-down racing game mechanics and the platformer mechanics), students usually reported Pronto as well suitable. For the other prompts (simulation game mechanics, turn-based game mechanics, and "an entire game"), students reported some trouble; most often creating all neccesary connections created a lot of visual clutter that made it difficult to create new connections, edit existing connections, and to what is influencing what at a glance.

6. Related Work

Most modern game engines feature low-code programming interfaces, either aimed at designers who do not have professional programming training, or beginners. As an example, Unreal Engine features a Blueprints system (Epic Games, 2012) that allows developers to use node-and-wires visual programming to define game behavior. Targeted at beginners, Construct 3 (Scirra, n.d.) offers a visual interface akin to

a table that defines connections between triggers and actions. Its mental model is thus similar to Pronto as presented here; however, the table is removed from the game's visuals and developers are led through a wizard with multiple forms to set the connections up, whereas in Pronto script code is used what would be specified in forms in Construct 3.

Game engines tend to advertise prototyping primarily when it comes to level design. For example, most engines feature tools to quickly "block out" levels, by combining primitive shapes that will later be replaced by specifically designed assets. Another common practice to try out novel ideas is to "mod" existing games: a shooter such as Half-Life 2 is taken and parts of its mechanics are modified. This may result in a new objective for the game, potentially not even involving elements integral to the original game such as shooting. Starting from a finished game allows developers to benefit from a well-defined set of basic functionalities, such as character movement, artificial agents, or menus, on which to add their own spin.

Visual and Low-Code Programming Visual programming environments for graphical programs, such as Scratch (Resnick et al., 2009) or Snap (Harvey & Mönig, 2015), are also commonly used to develop prototypical or even fully realized games. Their strengths for prototyping typically stems from a highly domain-specific set of primitive programming units, such as character movement, and great feedback loop.

While there are visual programming environments that utilize connection-based interactions and metaphors, they are usually designed for more general purposes than game prototyping. For instance, Lively Kernel features the lightweight Lively Connections (Lincke, Krahn, Ingalls, & Hirschfeld, 2009) as well as the full dataflow system Fabrik (Ingalls et al., 2016). Both systems allow users to connect general UI elements, which makes it possibly to create dataflow-based programs in a visual manner.

Video Games and Education There are many ways in which video games are used in educational context. As video games are a large and influental medium, there are also many students that play video games, making them a promosing and potentially enganging part of education (Squire, 2003). For instance, games may be used as part of edutainment that uses games to teach general topics such as mathematics or biology through a video game.

As video games themselves are programs and part of computer science, there are also used in computer science education. Several programming environments intend to introduce programming by allowing students to program games or solve game-like puzzles. For example, the aforementioned environment Scratch was conceived for use in educational context. Another example is the programming language Logo, and other languages based on or inspired by it, that was meant to be easily learnt by programming beginners and can be used to solve game-like puzzles(Abelson, Goodman, & Rudolph, 2004).

Additionally, there are also courses on general programming aspects that may use video games as examples in a lecture or as a topic for a lecture accompanying project. Some of these courses teach or study also aspects of game development itself, with results being published at venues such as the Game Developers Conference (more focused on industry participation) or the Conference on Games (more focused on participation from scientific communities).

7. Conclusion and Future Work

In this paper we introduced Pronto, a visual and interactive prototyping framework for game mechanics in the Godot game engine. Pronto focuses on quickly creating prototypes by letting programmers define behavior directly between the visual elements of their game. To demonstrate how our framework can be used, we described both a walkthrough as well as a seminar we designed and held.

There are several ways that the Pronto framework could be extended in the future. Both small quality of life improvements as well as bigger extension, e.g. new behavior nodes or even completely new

interactions, are possible.

Currently, Pronto was only used in one seminar. To further evaluate its uses, it would be possible to both use it in future seminars as well as other venues. For example, it would be possible to explore how useful Pronto is at prototyping ideas if used in a game jam. Studies like these would provide insights on Pronto's usefullness in a complete, non-restricted prototyping phase.

Additionally, further studies on using Pronto in an educational context seem promosing. This could evaluate how the topic of game development influences motivation and results of students as well as what students learn about tool and framework creation and modification.

It would also be interesting if ideas of Pronto could not only be used for throw-away prototypes, but integrated into a finished game. The inherent nature of prototypes as quick, throw-away may make this impossible, but maybe some of the concepts behind Pronto might also be of use outside of prototyping.

8. References

Abelson, H., Goodman, N., & Rudolph, L. (2004, 10). Logo manual.

Epic Games. (2012). *Unreal engine blueprint.* Retrieved from https://docs.unrealengine.com/4.26/en-US/BlueprintAPI/

Harvey, B., & Mönig, J. (2015, October). Lambda in blocks languages: Lessons learned. In *2015 IEEE blocks and beyond workshop (blocks and beyond)* (p. 35-38). USA: IEEE. doi: 10.1109/BLOCKS.2015.7368997

Ingalls, D., Felgentreff, T., Hirschfeld, R., Krahn, R., Lincke, J., Röder, M., ... Mikkonen, T. (2016, October). *A world of active objects for work and play - the first ten years of lively.* ACM. Retrieved from https://doi.org/10.1145/2986012 doi: 10.1145/2986012

Juan Linietsky, A. M., & contributors. (n.d.). *Godot engine.* Retrieved 2021-07-08, from https://godotengine.org/

Kasurinen, J., Strandén, J.-P., & Smolander, K. (2013). What do game developers expect from development and design tools? In *Proceedings of the 17th international conference on evaluation and assessment in software engineering* (p. 36–41). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2460999.2461004 doi: 10.1145/2460999.2461004

Kultima, A. (2015). Developers' perspectives on iteration in game development. In *Proceedings of the 19th international academic mindtrek conference* (p. 26–32). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2818187.2818298 doi: 10.1145/2818187.2818298

Lincke, J., Krahn, R., Ingalls, D., & Hirschfeld, R. (2009, January). Lively fabrik a web-based end-user programming environment. In *2009 seventh international conference on creating, connecting and collaborating through computing.* IEEE. Retrieved from https://doi.org/10.1109/c5.2009.8 doi: 10.1109/c5.2009.8

Murphy-Hill, E. R., Zimmermann, T., & Nagappan, N. (2014). Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development? In *36th international conference on software engineering, ICSE '14, hyderabad, india - may 31 - june 07, 2014* (pp. 1–11). Retrieved from https://doi.org/10.1145/2568225.2568226 doi: 10.1145/2568225.2568226

Nystrom, R. (2014). *Game programming patterns*. Genever Benning.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., ... Kafai, Y. B. (2009). Scratch: Programming for all. *Commun. ACM*, *52*(11), 60–67. Retrieved from https://doi.org/10.1145/1592761.1592779 doi: 10.1145/1592761.1592779

Schell, J. (2014). *The art of game design - a book of lenses, second edition*. Boca Raton, Fla: CRC Press.

Scirra. (n.d.). *Construct 3.* Retrieved 2021-07-08, from https://www.construct.net

Squire, K. (2003, 10). Video games in education. *International Journal of Intelligent Simulations and*

Gaming, *2*, 49-62. doi: 10.1145/950566.950583