Implementing Babylonian/G by Putting Examples into Game Contexts

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

Toni Mattis
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
toni.mattis@hpi.uni-potsdam.de

Marius Dörbandt
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
marius.doerbandt@student.hpi.uni-potsdam.de

Oliver Schulz
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
oliver.schulz@student.hpi.uni-potsdam.de

Martin C. Rinard
MIT CSAIL
Cambridge, MA, United States
rinard@csail.mit.edu

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

ABSTRACT

In game development, there are several ways to debug and inspect systems. These include very specialized and often visual tools, e.g. an on-demand collision box visualization. More general software engineering tools are often also available, e.g. "printf" debugging. However, default tools found in game engines are often either very specialized (like the collision box tool) or more general, but less domain-specific and spatially distant (like "printf" debugging).

Thus, we wanted to create a new tool that is as universal and easy to use as "printf" debugging but supports domain-specific representations and has the possibility to be integrated closer to the actual code parts or game elements that are involved. There are pre-existing programming environments similar to our goal: Babylonian Programming systems aim to enable developers to interact with concrete information directly in the code itself. In this paper, we introduce the resulting toolset: Babylonian/G, a Babylonian-inspired plug-in for the Godot game engine. This includes a new way of thinking about Babylonian examples in a game context, in-application probes, and the possibility of adding user input to examples.

CCS CONCEPTS

• **Software and its engineering → Development frameworks and environments**.

KEYWORDS

babylonian programming, example-based programming, video games, game programming, examples

‹Programming›Companion '24, March 11–15, 2024, Lund, Sweden

ACM ISBN 979-8-4007-0634-9/24/03
https://doi.org/10.1145/3660829.3660847

ACM Reference Format:
Eva Krebs, Toni Mattis, Marius Dörbandt, Oliver Schulz, Martin C. Rinard, and Robert Hirschfeld. 2024. Implementing Babylonian/G by Putting Examples into Game Contexts. In *Companion Proceedings of the 8th International Conference on the Art, Science, and Engineering of Programming (‹Programming›Companion '24), March 11–15, 2024, Lund, Sweden.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3660829.3660847

1 INTRODUCTION

Fast iterations are essential for game development [8]. Being able to iterate fast allows trying out as many ideas as possible and fixing problems as quickly as possible. Ideally, programmers could experiment as much as possible while interacting with a running game [9].

However, in reality many game tools such as inspector or debug windows for print statements are not part of the running game but in separate windows. These context switches are cumbersome and require additional mental effort to keep track of which values relate to which part in code and games.

There are existing programming systems and tools that aim to address this issue. This includes the Babylonian Programming system, which allow visualization and other interaction with concrete values directly in the code itself.

Thus, we combined Babylonian Programming with game programming by integrating Babylonian-style tools into an established game development environment.

In this paper, we make the following contributions:

- Babylonian/G, an extension of the Godot game engine that enables Babylonian Programming for games
- A new example type aimed at game engines and developers
- The possibility to add recorded input to examples

In the next part of this paper, we will introduce the two main foundational systems our work is based on: Babylonian Programming, including Babylonian/S, and game engines, including the open-source game engine Godot, in section 2. The new system Babylonian/G is the result of integration Babylonian Programming features into Godot. We will describe the main concepts and tools in section 3. In section 4, we will discuss related work such as other

game development tools. Finally, we will conclude the paper and describe future work in section 5.

2 BACKGROUND

To create an example-based game programming system, we mainly built upon two existing concepts and related coding environments. The first foundation is Babylonian Programming and its implementations whose concepts we integrated into the game programming context. The second essential part are game engines, programming environments commonly used in game creation and in one of which we integrated our tools.

2.1 Babylonian Programming

Babylonian Programming environments allow example-based programming directly in the code itself [5, 6]. Programmers interact and view the code itself, the examples are not in separate windows or tools. There are two main concepts that define Babylonian Programming. First, programmers need to be able to define some sort example for the code they are interested in. Second, the developers should be able make use of these examples directly in the code itself, usually by adding interactive widgets. This often includes a widget to visualize state provided by the example in-line.

Babylonian/S. Babylonian/S is a Babylonian Programming system implemented in Squeak/Smalltalk [7]. It thus allows combining Babylonian Programming with live programming and related tools already provided by Squeak/Smalltalk. A Babylonian/S code browser can be seen in Figure 1.

There are several different kinds of examples in Babylonian/S, but the examples are always defined for exactly one method and usually aim to invoke this method. Example types include but are not limited to the *Method example*, the *Script Example*, and the *World Example*. For a Method Example, the receiver and all arguments need to be defined individually via scripts or references to existing objects, called live specimen. These are then used to invoke the method when the example is run. Another type is the *Script Example* where developers can write any script that directly or indirectly calls the method. Lastly, the world example will use any method invocations already happening in the system.

To make use of these examples, developers can add widgets directly in the code. One of the most commonly used widget is the *Probe*, which allows visualizing data directly in the code itself to better understand it. Probes can display data as text but also support custom visualizations e.g. for numbers or visual elements. Other widgets include, but are not limited to, replacements (to quickly replace part of the code for example runs without removing the original piece of code) and assertions (to check certain aspects of on example run).

2.2 Game Engines

Game programmers often use game engines to create games. A game engine is a development system that provides a collection of code and tools for common game programming challenges such as developing a game loop or adding graphical components. Thus, we used a pre-existing game engine as a basis for integration Babylonian Programming features.

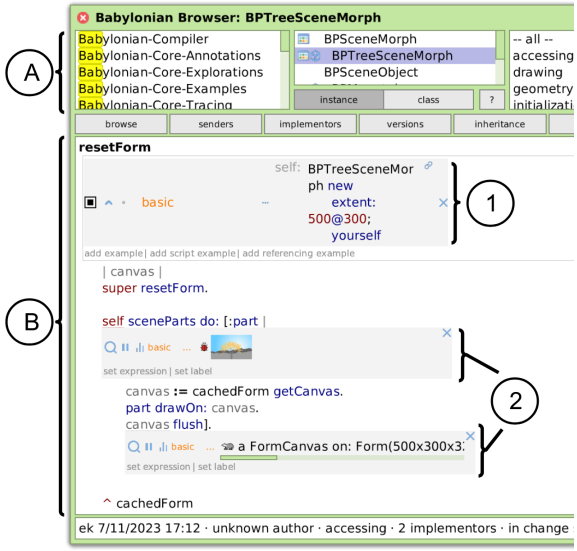

Figure 1: A screenshot of Babylonian/S. It shows a Squeak/Smalltalk code browser with an area to select a class and method to edit (A) and code edit area (B). At the top of the code area is a widget to create and manage examples (1) and two probes are placed in the code (2).

Godot. We have chosen the engine Godot as our target system, as this open-source game engine is well suited for the integration of new tools [1]. Godot has three main view: a 2D scene editor to alter graphical components, a second scene editor but for 3D objects, and a code editor. The editors as well a running game can be seen in Figure 2. If a developer chooses to run the game (or a game scene), the game will open in a separate window.

The scene editor allow arranging scene objects, called nodes, in a scene tree. These objects can be arranged using drag-n-drop in the visual editor displaying a preview of the current scene. Nodes can have attached code, so called scripts, to give them behavior.

These scripts can be written and changed in the code editor. It is a file-based code editor that in standard Godot distributions only uses text. If developers add print statement to the code, these will be displayed while the game is run in a special debug area that is separate from both the code editing view and the game window.

Plugin System. We built our tools using the plugin system of Godot which allows modifying Godot using Godot itself and its tools; the underlying engine code did not need to be altered. Thus, our tools can be used with the official engine builds distributed by Godot itself and is in theory usable together with other plugins.

3 BABYLONIAN/G

By adding Babylonian Programming features to the game engine Godot we created the Godot extension *Babylonian/G*. Our tools are focused on two aspects of Babylonian Programming: One, make it possible to use widgets directly where programmers work (thus directly in the code itself) and two, find an example concept suitable for game development.

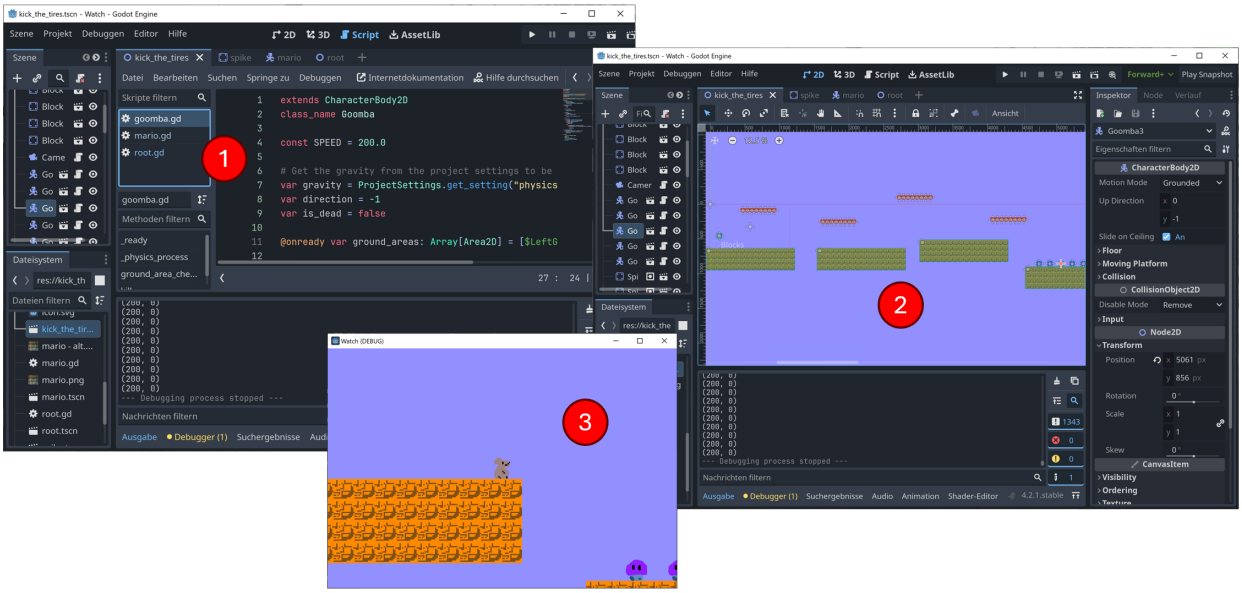

Figure 2: Screenshots of Godot. Godot has a code editor (1), a scene editor (2), and games can be run in new windows (3).

3.1 Watching game values with Probes

We added one of the interactive widgets from Babylonian Programming: The Probe. The Probes allow visualizing state directly in the code itself. Developers need to write B.probe (...) around the piece of code they are interested in. This makes probes as easy to use as "printf" debugging, with the addition that the probe call will not alter the original functionality. The probe call acts as a wrapper that executes and returns the result of the code it is wrapping, which means developers can probe expressions in existing lines of code and do not have to add new lines just to probe values. On save, this wrapper will cause the actual Probe widget to appear in the editor, three of which can be seen in Figure 3.

The Probe widget is modular configurable. Developers can thus switch between different kinds of visualization: a vector for instance could either be displayed as text of the numeric values or as an arrow pointing in the same direction as the vector. Since the Probes work with running games and thus may get many values depending on the framerate, these values can be plotted over time (e.g. numbers as a graph) or the Probe can be set to only display the current value. Probes can be paused and will continue displaying the last values received before pausing.

As we currently cannot adjust line height of the Godot editor, the Probes have limited space. To allow programmers more freedom, the Probes' position can be changed via drag-n-drop. A line will connect the Probe to the line it originates from while it can be placed in more convenient position.

These Probes work with any game opened through the engine, using the running game as "an example", which is similar to World Example found in Babylonian/S, and the newly introduced example concepts. Because of this connection, we also added the possibility to display the Probes in-game. In-game Probes can be set to either always be on the screen, e.g. to display an essential value related

Figure 3: A screenshot of in-line Probes in the code editor in Babylonian/G

to the player character always in the top left. But in-game Probes can also be displayed for and attached to each node that uses the script, e.g. to see one Probe attached to each individual enemy in the game in order to easily to mentally connect the values to the specific enemies, which can be seen in Figure 4.

3.2 What is an example in a game engine?

In Babylonian/S, examples are currently aimed at one specific method. However, in game development programmers often think about the entire game scene and its interactions in order to understand code behavior. Because of this, we shifted the focus from examples that run one specific method to examples that are interconnected with existing tools for running and altering game scenes.

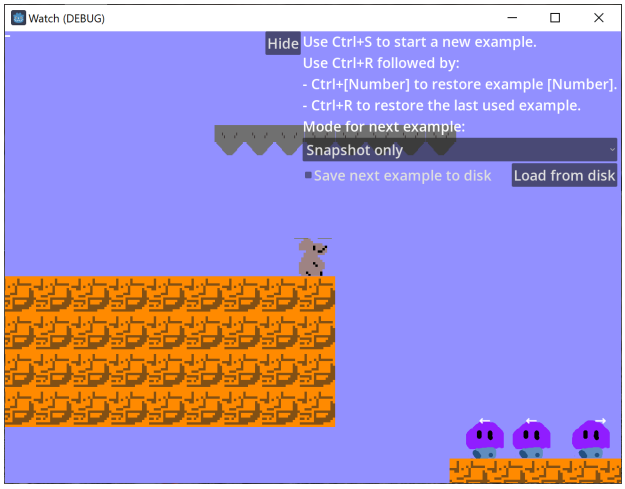

Figure 4: A screenshot of in-game Probes attached to each enemy node in Babylonian/G

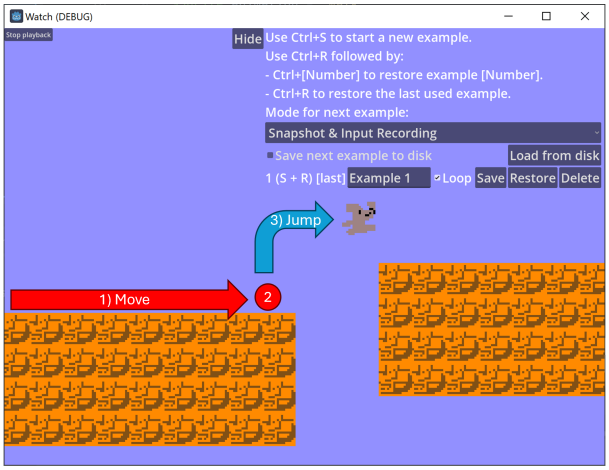

Figure 5: A screenshot of an Example in Babylonian/G. The player is moved through the level to a fitting point in front of a cliff (1). Then, example recording is started with a shortcut (2). The player moves the character across the cliff and ends the recording (3). The recorded example can now be replayed any time: The game will reset to point (2) and perform the recorded jump afterwards.

Thus, the Watches display values recorded from a currently running game. This makes the game scene file together with events that occur during runtime an example that can be used for Babylonian Programming. However, except for the initial state, these examples are hard to reproduce.

In order to have examples that are easier to reproduce and fit to a specific game situation, we added a new kind of example: the *Game Scene State Example*. Developers can run the game and save an example at any point, which cause a serialized version of the current game state to be saved. These examples can then be reloaded in the game window at any point. This makes it possible to have examples at any point in the scene and also to preserve state that might otherwise change (e.g. if enemy attack power is initialized with a random value at game start, this value would be preserved in the Game Scene State Example). The UI and workflow of recording and loading examples can be seen in Figure 5.

3.3 Using User Input in Examples

Additionally to saving examples based on game state alone, we made it possible to also record user input such as keyboard events. This makes it possible for examples to also include common actions that a player would take. We currently experimented with two main use cases for this feature. Firstly, game developers want to use this in combination with Watches to illustrate how certain core gameplay loops work. Secondly, by playing the example with input in an endless loop it is possible to experiment with game behavior (e.g. jump height of the player character) without having to manually recreate the situation each time.

This concept could potentially be reapplied to other Babylonian Programming environments such as Babylonian/S. This would require to either add a way for users to record input for already defined examples whose graphical objects etc are currently usually not visible to the user. It could also be integrated into the Example Mining toolset, which enables users to record method invocations while interaction with the system [3].

4 RELATED WORK

There are game development tools that address similar concerns to the ones our tools address. For instance, if the code values a programmer wants to view are variables of a scene object, many game engines like Godot have tools to view these values while running the game. However, this view is usually detached from code and running game

The examples used for Babylonian/G are similar to general concept of a save game. Many games have a feature to save progress in the game. While some games limit when and what is saved, games that enough information to recreate the state of an entire scene might inform future versions of our examples.

There are also other game development tools that save game state and user input: replays tools can be used by developers both to provide tutorials for users as well as to easily receive problem situation from users to debug [2].

5 CONCLUSION AND FUTURE WORK

In this paper, we described an extension for the Godot game engine called Babylonian/G. The new tools make it possible to use core Babylonian Programming features for game development. Developers can now visualize example data directly in the code editor and in the game scene using Watches. At any point while running the game, programmers can create examples based on game state, and user input if wanted, in order to explore and experiment with specific game situations.

The current state of these tools is prototypical; in the future, many quality of life adjustments could be made such as spacing widgets to be well readable while also addressing the issue of limited screen space. The Watches could also support more custom visualizations, e.g. for enumerations, arrays, and trees. It should

also be possible to use in-game Watches to jump back to the related parts in the code. Since the example currently rely on a running game, it would also be interesting to experiment with saving specific Watch values permanently; we initially did not see a use case for this, but for e.g. an educational purpose it might prove useful. Also, we currently were focussed on the 2D scene editor. Exploring the 3D editor and making adjustments (e.g. to correctly display 3D vectors) is also of interest.

Babylonian/G was currently only tested internally and with a few pilot testers. In the future, conducting user study on the usability of our tools and their effect an certain game programming tasks would give us more insights on the strength and challenges of the tools. Initially these studies could be conducted using computer science students, but later studies might also recruit industry experts.

The Game Scene State Examples are currently limited because of the serialization process. If a developer changes the game scene, an example based on serialization will often not reflect these changes. For instance, we could add a new node to the original scene that displays hat on top of the player character. If we load the example using our current mechanism, the hat would be gone after loading the example because it was not part of the original serialization. Because of this, we are looking into saving examples as patches that can preserve such changes.

Lastly, we might also integrate features from other system. This might include more Babylonian Programming widgets such as replacements or assertions. However, might also include other game development tools that have possibilities for synergy with Babylonian Programming. For instance, being able to easily create in-game sliders to experiment with values like in the Pronto prototyping framework could work well in tandem with input-based examples [4].

ACKNOWLEDGMENTS

This work is supported by the HPI–MIT "Designing for Sustainability" research program[1].

REFERENCES

[1] Ariel Manzur Juan Linietsky and contributors. [n. d.]. *Godot Engine.* https://godotengine.org/

[2] Stas Korotaev. [n. d.]. Time Manipulation in Unity - Recorded Solutions. https://www.gamedeveloper.com/design/time-manipulation-in-unity---recorded-solutions. Accessed: 2024-02-08.

[3] Eva Krebs, Patrick Rein, and Robert Hirschfeld. 2022. Example Mining: Assisting Example Creation to Enhance Code Comprehension. In *Companion Proceedings of the 6th International Conference on the Art, Science, and Engineering of Programming* (Porto, Portugal) *(Programming '22)*. Association for Computing Machinery, New York, NY, USA, 60–66. https://doi.org/10.1145/3532512.3535226

[4] Eva Krebs, Beckmann Tom, Leonard Geier, Stefan Ramson, and Robert Hirschfeld. [n. d.]. Pronto: Prototyping a Prototyping Tool for Game Mechanic Prototyping *(PPIG 2023)*.

[5] Fabio Niephaus, Patrick Rein, Jakob Edding, Jonas Hering, Bastian König, Kolya Opahle, Nico Scordialo, and Robert Hirschfeld. 2020. Example-based live programming for everyone: building language-agnostic tools for live programming with LSP and GraalVM. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2020, Virtual, November, 2020.* ACM, 1–17. https://doi.org/10.1145/3426428.3426919

[6] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming - Design and Implementation of an Integration of Live Examples Into General-purpose Source Code. *Art Sci. Eng. Program.* 3, 3 (2019), 9. https://doi.org/10.22152/programming-journal.org/2019/3/9

[7] Patrick Rein, Jens Lincke, Stefan Ramson, Toni Mattis, Fabio Niephaus, and Robert Hirschfeld. 2019. Implementing Babylonian/S by Putting Examples Into Contexts. In *Proceedings of the Workshop on Context-oriented Programming - COP '19.* ACM Press. https://doi.org/10.1145/3340671.3343358

[8] Jesse Schell. 2014. *The Art of Game Design - A Book of Lenses, Second Edition.* CRC Press, Boca Raton, Fla.

[9] Bret Victor. 2012. Inventing on Principle. http://vimeo.com/36579366

Received 2024-02-08; accepted 2024-02-26

[1] https://hpi.de/en/research/cooperations-partners/research-program-designing-for-sustainability.html