In Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) 2021, St. Louis, Missouri, USA, October 10-13, 2021, IEEE.

Shortening Feedback Loops in a Live Game Development Environment

Tom Beckmann, Eva Krebs, Patrick Rein, Stefan Ramson, Robert Hirschfeld
Hasso Platter Institute, University of Potsdam
Potsdam, Germany
Email: firstname.lastname@hpi.uni-potsdam.de

***Abstract*—Game development benefits from short iterations, as it is often concerned with how a game will feel like; something that is hard to anticipate. Live programming aims at reducing the length of iterations during software development to allow for faster exploration and improve program comprehension.**

We propose a live game development environment providing a more general live programming workflow in game prototyping. The environment supports developers in two aspects: how will a code change affect the look and feel of the game and how does the behavior of the game relate to its code? The proof-of-concept environment ensures short feedback cycles by always keeping the game running while it is being developed. We propose several mechanisms to work with the running game, for example, programmers can automatically replay situations in the game by using an explicit notion of snapshots. We tentatively demonstrate the effectiveness of the live programming features in the context of the development of game prototypes of different genres.

***Index Terms*—game development, game tools, live programming, exploratory programming**

I. Introduction

When creating a game, design decisions are driven by the resulting "experience" of playing the game [1]. Depending on the environment, changing an aspect of the game and inspecting its results may require recompilation, starting the game, navigating to the state where the change is relevant, and finally checking the result against the intention. In game design, a high number of iterations is said to benefit the game's overall quality: "The more times you test and improve your design, the better your game will be" [2, p. 94]. Concerns for the developer are to maximize relevant insights within each iteration and to keep each iteration as short as possible.

To get feedback on changes to the gameplay, developers have to experience the consequences of their changes in the running game [3]. When changing the code of the game, developers inevitably face a gap between the static representation of behavior in which they applied a change and the running version of the game [4], [5]. Developers have to bridge this gap in two directions. When they applied a change and want to get feedback, they have to bring the running game into a state in which they can observe the changed behavior. In turn, developers may observe interesting behavior in the running game and may want to debug or understand it, for example when a non-player character exhibits behavior they should not have or an event is not triggered. In this situation, developers have to determine how the observed behavior relates back to the static representation.

To keep the gap between the static code and the running game small, live programming offers mechanisms that provide the impression of changing a program while it is running and getting feedback from that running instance [6], [7]. The goal of live programming is to minimize the delay between changing a program and inspecting the consequences of the change [5], [8]. In this way, dynamic behavior becomes tangible, allowing developers to assess more quickly whether their changes have the desired effects [9].

In this paper, we present a proof-of-concept game development environment supporting live programming workflows by integrating a running instance of the game to be developed[1]. We tentatively illustrate how an always running instance of a game can shorten iterations through the implementation of three game prototypes. The demos and the workflows focus on the development and refinement of specific game mechanics, rather than full games.

II. Characteristics of Game Development

Game development is a specialized software development field that differs in some regard from general software development [3], [10]. Game development involves many different activities: texture creation, 3D modeling, scripting materials, writing gameplay interactions, dialog trees, or graphics code. The line defining "programming" is blurred, as many activities involve defining dynamic behavior, for example the configuration of the behavior of a platform in a jump-and-run game.

Source code in games can be classified into script, gameplay, and engine code [10]. We focus on live programming mechanisms for gameplay code. Further, we distinguish between code and game editors [3], [11]. Code editors work on specifications of behavior, in our case gameplay code. Game editors work on the initial state of a game (for example a level) and allow developers to change the properties of elements in the game. While this separation roughly corresponds to a separation of behavior and state, the lines are again blurred, as, for example, the configuration of properties of elements in the game can entail changes to behavior. In many engines, the code and the game editor are separate tools.

[1]For more detailed, illustrated descriptions of the approach, and a more detailed walkthrough and discussion see https://doi.org/10.5281/zenodo.5082421

978-1-6654-4592-4/21/$31.00

III. A Live Programming Environment for Game Development

Our approach to enabling a live programming workflow for gameplay code involves two principles. First, the *game is running at all times* to be able to relate changes to the game directly. Second, the *programming tools are aware of the continuously running game*. As a result, our tools can both leverage the potential of run-time information and deal with challenges that arise from viewing the game while it is live. In this section, we describe the design of our proof-of-concept game environment[2], covering the underlying architecture and the implementation of the principles.

A. Architecture and ECS

The environment is built around the Entity-Component-System (ECS) architectural pattern, frequently used in games [12], [13]. It separates identity, data, and code: entities are identifiers that may contain a set of components; components include all data of the game and are descriptors of the traits and state of an entity; systems query the game for all entities with a specific combination of components and then read or write the data of those components. As an example, the entity representing a game character may have a sprite component containing the image file path and a position component. A render system would then query the world for all entities that have a position and a sprite component and then draw each sprite's image at the respective positions.

The environment is built in Squeak/Smalltalk [14], [15], which provides hot code reloading infrastructure. Our environment provides several tools that users can recombine to form workspaces, similar to other software in the domain, for example Unity [16] or Blender [17]. Our environment is composed of a number of tools (see Figure 1), including tools to define the data layout of components (②), running queries against the game state also enabling the user to edit the results (①), editing the code of systems (⑥), viewing a rendered version of the game (⑨), controlling the active game state snapshot and game time (④), or profiling (⑤).

B. Editing in a Running Game

Our first principle is to keep the game running at all times, even when editing the game state. To achieve this, we formalize the idea of state snapshots. A state snapshot is a copy of the game state at a given moment in time. The game begins with an initial state snapshot Δ_0 which represents the game state that the developers prepared and will later be shipped as the game. While running, the game will be in the current state snapshot Δ_i at the time i. The environment provides several controls to advance the game state (④). If a user presses the restart button, the game's current state Δ_i will be reset to Δ_0. Further, users have fine-grained control over how the state advances via the step button, that advances the state to Δ_{i+1}. In terms of the ECS pattern, a state snapshot is a deep copy of the state of all components.

[2]The source code of the environment is publicly available on GitHub at https://github.com/hpi-swa-lab/tools-interactive-simulations/

When editing a running game, the challenge is to identify the snapshot a change to the state should be applied to. Only applying the change to Δ_i would mean that the change is not persisted and the effects only exist temporarily in the running game. In contrast, changes to Δ_0 are persisted and will exist in future runs. Applying a change to both Δ_i and Δ_0 is straightforward for state modifications that are constant between snapshots. This is, for example, the case when changing the position of a wall in a level. As its position is constant, moving it in both states is a consistent change. For derived state, however, such as a patrolling enemy's position, persisting a change becomes ambiguous. If the user moves a patrolling enemy in Δ_i with $i > 0$ and we apply that change to both Δ_i and Δ_0, the current state Δ_i will likely have become inconsistent, as restarting the game and advancing it to the same point in time i as before will likely result in the enemy being placed elsewhere, as its starting position is now different.

For changing derived state we offer users to explicitly opt-in to a persisting mode. By activating a checkbox, any change to an entity or component that exists in both Δ_0 and Δ_i will be applied to both state snapshots. Users can toggle this behavior by holding a modifier just before doing a change, enabling a quick workflow of experimentation and persisting changes.

Another problem arises with entities or components that do not exist in Δ_0, as they will only be created during gameplay. For these, a change can only be applied in Δ_i, as the correct place to persist a change would be in the entity's production rules in the system code. While more sophisticated solutions are possible [18], we approach this problem by showing the user clearly where changes to data can be persisted, and where a change would only occur in Δ_i, by coloring any fields where persisting is not possible. As such, users are made aware of the temporary nature of their changes during experimenting.

To support persistent changes to entities that are spawned at run-time, we added "entity templates". A template (③) contains a definition of an entity with its components and can be spawned at run-time. When the user changes a value in the template's definition, we apply the same change to all instance of that template in the Δ_i state. While this allows for consistent edits in Δ_i to entities that did not exist in Δ_0, it has the same draw-back described for other derived state.

Tweaking both derived and constant values during the game is a frequent process in game balancing. To support this process, users can save a snapshot Δ_r of the current game state Δ_i and use it as a point of replay (④). To enable replay, users can start recording from time r. We record all external events to the game, such as pressing keys, moving the mouse, or manually spawning an entity via the editor. When the user starts the replay, Δ_i is reset to Δ_r and as the game time advances, we trigger all recorded external events. Changes in persisting mode will be applied to all three states Δ_i, Δ_r, and Δ_0. Non-persisted changes will only be applied to Δ_i.

To get continuous feedback on their changes, users can let the editor automatically replay the recording. For example, users can modify the system code to tweak a one-off interaction, such as entering a trigger zone and see how

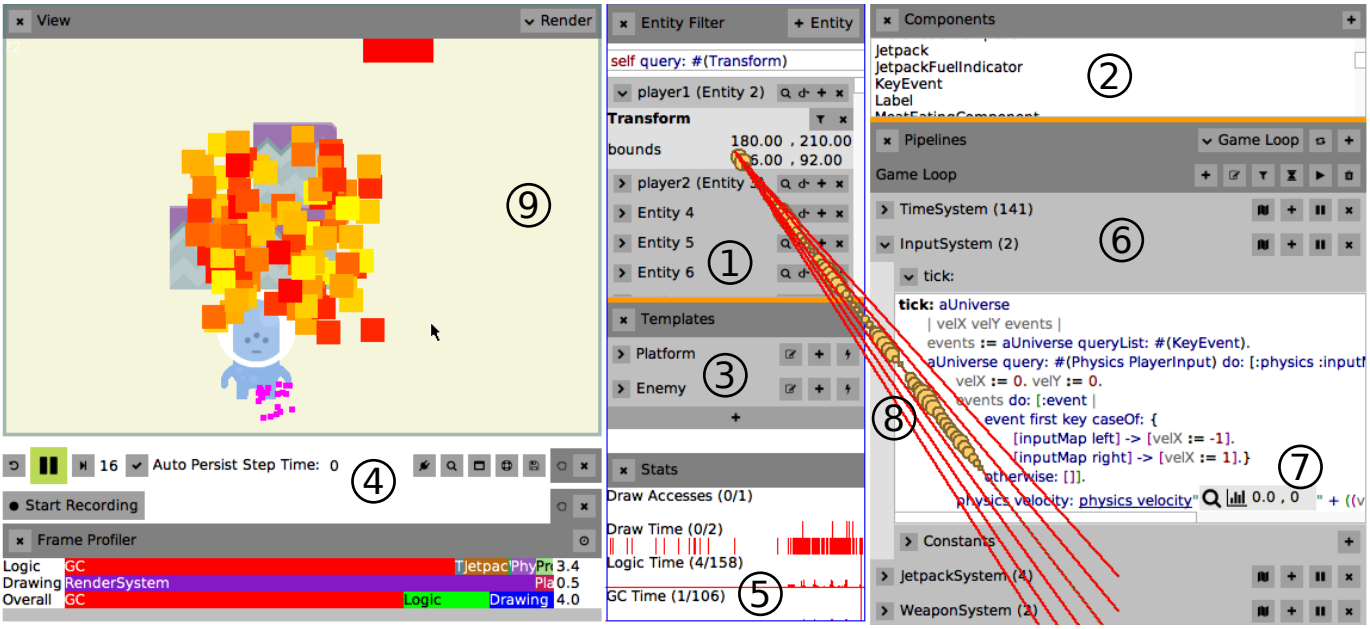

Fig. 1. A screenshot of our environment. We list the elements that will later be mentioned here for reference: ① entity filter, ② component structure definitions, ③ template definitions, ④ pause/play/step, auto-persist, and replay controls, ⑤ profiling and frame breakdown, ⑥ system code editors, ⑦ a code watch, ⑧ query connection bubbles, ⑨ a game view

the recorded situation evolves given this new system code. While the repeated execution of the recording makes feedback available quickly, it may result in inconsistent game state as changes to the code are directly reflected in the game behavior, rather than from Δ_0 or Δ_r. To specifically support continuous and consistent feedback on code that triggers an effect once users can configure the editor to auto-restart. After each save, the game will then reset to Δ_r if it exists, or Δ_0 otherwise.

C. Integrating with the Editor

Our second principle for a live programming experience with a game development environment is a tight integration of the running game with the game and the code editors. In our prototype, the code is directly integrated within the game development environment in the form of ECS systems that appear in a separate panel (⑥).

The editor visualizes some of the underlying semantics directly. The arrangement of ECS system editors (⑥) corresponds to the execution order of the ECS systems. Similarly, dataflow can be visualized by filtering for usage of a specific component. Systems that use the selected component will be colored to allow for faster navigation. Further, users can enable drawing connections from panels (⑧), which act as source of specific components, to the code systems that use them. For example, using the entity filter tool (①) to search for entities that have a Transform component and then enabling the connections, will draw a line to all systems that query for Transform components. To help users to find usage patterns of components, bubbles sized to the number of entities that matched each query travel along the connections. For example, spawning particles will likely cause a high increase in bubble sizes for certain components.

The entity filter tool (①) is well suited to quickly create ad-hoc editors for various use cases. The data inside the query tool is fully live and editable. To ensure convenient editing even when the values change continuously, such as the acceleration of a falling body, we freeze the value when an edit interaction is started. The updating in-game value is then shown next to the editing field.

D. Integrating with the Game

To foster aspects of direct manipulation and reduce the distance between the tools and the game, the environment supports users in integrating custom tooling directly within their game. The working hypothesis is that the line between tooling and game needs to blur to allow writing tools with as little cognitive overhead as possible. As such, while users could change values through the entity filter directly, they can instead create or use specialized tooling systems to accommodate a certain use case more adequately.

For example, we can create the typical gizmo user interface (UI) element for changing an object's transformation through the use of a custom system. This system runs after the main rendering system to draw on top of the game's entities and waits for mouse events to transform objects accordingly. To separate tooling from game code, we allow grouping systems into sets and toggling them in their entirety.

Further, we allow users to open multiple views (⑨) on the game and configure which systems run to render that view. This allows users to realize various common tools, such as the typical four-split view in 3d game editors, where one view displays the perspective view on the game and the other three are locked to the x, y, and z-axes respectively.

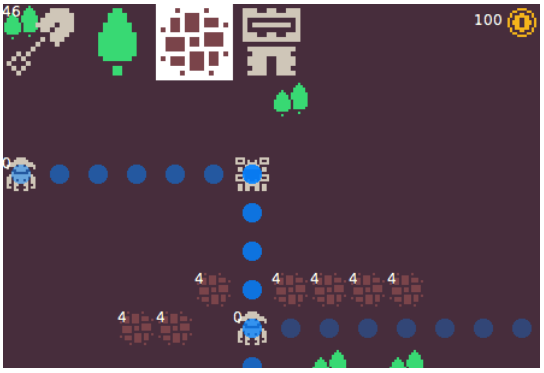

Fig. 2. Demonstration of the pathfinding debug overlay. This optional system will draw the next steps of each enemy actor as blue circles and draw a number over each tile with a movement cost of more than zero, such as the greyish boulders that have a movement cost of four.

IV. Discussion

In this section, we describe our experience of working with the proposed environment and discuss how our implementations of the two principles to provide faster iterations worked out in practice. Overall, we implemented three simple games of different genres: an "Anno 1602"-like isometric map renderer to benchmark the drawing debug capabilities of the environment, a top-down tower defense game on a 2D grid, and a real-time side-scrolling platformer. We will briefly describe notable tool usage for the latter two.

The tower defense game explores complex interactions between various types of actors: towers can have effects, terrain elements can be placed by the player and need to be taken into account by the pathfinding of walking and flying enemies, and menus for buying upgrades and placing towers. The game makes heavy use of templates (③). Towers and enemies were balanced by tweaking template definitions and automatically replaying (④) a scenario that should be balanced. A custom debug drawing overlay was used to debug the A* pathfinding (see figure 2). By also placing watches in the code, and then stepping the pathfinding forward using the time controls (④), we could investigate the implementation, both in terms of runtime values and through visualizing the path and the fields' traversal costs visualized on top of the game.

The platformer game demonstrates the use of a simple physics system. Live values are changing constantly as actors are subjected to gravity and other forces. Again, the replay feature (④) was of help. For example, for level design, we did a jump of maximum distance, then replayed that jump, paused, and placed level tiles accordingly. Additionally, the framework allowed us to implement a level builder directly within the game using ECS systems.

Overall, the UI supported us in various ways to quickly assemble tooling for game-specific debugging and testing. Especially the ability to define custom sets of systems to run in a view proved helpful for a variety of use cases. The custom views encouraged a mindset of creating small systems for game-specific purposes that made the development and debugging workflow easier. An observed challenge is how tools may be shared between games, as the data layout of components differs. To tackle this, we experimented with higher-order systems that allow for projections of components to an expected form.

We believe that the environment enabled us to find suitable, working designs faster than in traditional game engines. To enable larger changes such as refactorings, while maintaining live programming benefits, a concept such as edit transactions [19] may be worth exploring.

V. Related Work

Live feedback in game development has seen little explicit coverage up to 2017 [7]. In later work, live programming for game development has been considered, but only hypothetically [20]. This paper in turn presents a first prototype of a feasible game development environment. In general, mechanisms for live and exploratory programming, such as probes [21], [22], can also be applied in game development whenever code is involved. However, many of them focus on general-purpose software development and are not tailored to a specific domain. The application of replay in the context of games has been shown in a demo by Victor [23], where he manipulates the jump height of a character to match a specific jump distance in a level. The general method of replay takes inspiration from the ideas of omniscient debuggers [24]. Further, a variety of mechanisms have been proposed for deterministic replay [25]–[27]. The Godot Game Engine [28] supports some form of a live programming approach by synchronizing changes while the game is running, although requiring switching between tools. Currently, changes to initial state override any existing run-time state. Changes to game state can not be persisted. In the Unreal Engine [29], the running game needs to be interrupted and a separate edit mode entered. A notable game programming tool set that provides live feedback is the tooling at Guerrilla Games [30]. The tool set features profiling and debugging overlays that run within the game's context, as well as tools that paint directly within the running game, both features similar to our proposed environment. However, users have to interrupt the running game to enter an edit mode.

VI. Conclusion

We presented the prototype of a live programming environment for game development. By keeping the game running at all times, adapting the tools to be aware of a running game, and offering affordances to integrate custom tooling directly within the game, we hope to enable users to answer what a change will look and feel like in the game, and relate the effects of a change back to code faster. By allowing for shorter, and therefore more, iterations, the overall quality of developed features should increase. We evaluated the prototype by developing three proof-of-concept games. We found the new mindset around keeping the game running to be beneficial during development and hope that our prototype can help to better assess how live programming approaches can shorten iterations in game development.

Acknowledgment

We gratefully acknowledge the financial support of HPI's Research School[3] and the Hasso Plattner Design Thinking Research Program[4].

References

[1] J. Kasurinen, J.-P. Strandén, and K. Smolander, "What do game developers expect from development and design tools?" in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 36–41. [Online]. Available: https://doi.org/10.1145/2460999.2461004

[2] J. Schell, *The Art of Game Design: A Book of Lenses*, 2nd ed. Natick, MA, USA: A. K. Peters, Ltd., 2014.

[3] E. R. Murphy-Hill, T. Zimmermann, and N. Nagappan, "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*, 2014, pp. 1–11. [Online]. Available: https://doi.org/10.1145/2568225.2568226

[4] D. Norman and S. Draper, *User Centered System Design*. Lawrence Erlbaum Associates, Inc., Publishers, 1986.

[5] D. Ungar, H. Lieberman, and C. Fry, "Debugging and the experience of immediacy," *Commun. ACM*, vol. 40, pp. 38–43, 04 1997.

[6] S. L. Tanimoto, "A perspective on the evolution of live programming," in *Proceedings of the 1st International Workshop on Live Programming, LIVE 2013, San Francisco, California, USA, May 19, 2013*, B. Burg, A. Kuhn, and C. Parnin, Eds. IEEE Computer Society, 2013, pp. 31–34.

[7] P. Rein, S. Ramson, J. Lincke, R. Hirschfeld, and T. Pape, "Exploratory and live, programming and coding - A literature study comparing perspectives on liveness," *Programming Journal*, vol. 3, no. 1, p. 1, 2019. [Online]. Available: https://doi.org/10.22152/programming-journal.org/2019/3/1

[8] P. Rein, S. Lehmann, T. Mattis, and R. Hirschfeld, "How live are live programming systems? benchmarking the response times of live programming environments," in *Proceedings of the Programming Experience 2016 (PX/16) Workshop*. ACM, 2016, pp. 1–8.

[9] C. M. Hancock, "Real-time programming and the big ideas of computational literacy," Ph.D. dissertation, Massachusetts Institute of Technology, 2003.

[10] J. Blow, "Game development: Harder than you think," *Queue*, vol. 1, no. 10, p. 28–37, Feb. 2004. [Online]. Available: https://doi.org/10.1145/971564.971590

[11] S. Broadley. (2016) Empowering content creators. Presentation at 2016 Game Developer Conf. (GDC 16). [Online]. Available: https://www.gdcvault.com/play/1023274/Empowering-Content

[12] S. Bilas. (2002) A data-driven game object system. Presentation at 2002 Game Developer Conf. (GDC 02). [Online]. Available: gamedevs.org/uploads/data-driven-game-object-system.pdf

[13] T. Johansson. (2018) Job system & entity component system. Presentation at 2018 Game Developer Conf. (GDC 18). [Online]. Available: gdcvault.com/play/1024839/Job-System-Entity-Component-System

[14] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*. USA: Addison-Wesley Longman Publishing Co., Inc., 1983.

[15] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay, "Back to the future: the story of squeak, a practical smalltalk written in itself," in *ACM SIGPLAN Notices*, vol. 32, no. 10. ACM, 1997, pp. 318–326.

[16] U. Technologies. Unity engine. [Online]. Available: https://unity.com

[17] B. Foundation. Blender. [Online]. Available: https://www.blender.org/

[18] S. Burckhardt, M. Fahndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato, "It's alive! continuous feedback in ui programming," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 95–104. [Online]. Available: https://doi.org/10.1145/2491956.2462170

[19] T. Mattis, P. Rein, and R. Hirschfeld, "Edit transactions: Dynamically scoped change sets for controlled updates in live programming," *Art Sci. Eng. Program.*, vol. 1, no. 2, p. 13, 2017. [Online]. Available: https://doi.org/10.22152/programming-journal.org/2017/1/13

[20] A. R. Martin and S. Colton, "Towards liveness in game development," in *IEEE Conference on Games, CoG 2019, London, United Kingdom, August 20-23, 2019*. IEEE, 2019, pp. 1–4. [Online]. Available: https://doi.org/10.1109/CIG.2019.8848092

[21] S. McDirmid, "Usable live programming," in *78e50801-b344-5b44-ad68-fff3a6b3b16a*, ser. Onward! 2013. New York, NY, USA: ACM, 2013, pp. 53–62. [Online]. Available: http://doi.acm.org/10.1145/2509578.2509585

[22] D. Rauch, P. Rein, S. Ramson, J. Lincke, and R. Hirschfeld, "Babylonian-style programming - design and implementation of an integration of live examples into general-purpose source code," *Programming Journal*, vol. 3, no. 3, p. 9, 2019.

[23] B. Victor. (2012) Inventing on principle. [Online]. Available: http://vimeo.com/36579366

[24] B. Lewis, "Debugging backwards in time," 2003.

[25] H. Thane and H. Hansson, "Using deterministic replay for debugging of distributed real-time systems," in *Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000*. IEEE Comput. Soc, 01 2000. [Online]. Available: https://doi.org/10.1109/emrts.2000.854015

[26] D. Aumayr, S. Marr, S. Kaleba, E. G. Boix, and H. Mössenböck, "Capturing high-level nondeterminism in concurrent programs for practical concurrency model agnostic record & replay," *Art Sci. Eng. Program.*, vol. 5, no. 3, p. 14, 2021. [Online]. Available: https://doi.org/10.22152/programming-journal.org/2021/5/14

[27] Y. Chen, S. Zhang, Q. Guo, L. Li, R. Wu, and T. Chen, "Deterministic replay: A survey," *ACM Comput. Surv.*, vol. 48, no. 2, pp. 17:1–17:47, 2015. [Online]. Available: https://doi.org/10.1145/2790077

[28] A. M. Juan Linietsky and contributors. Godot engine. [Online]. Available: https://godotengine.org/

[29] J. Wilson. (2019) Unreal engine 4.22 released. [Online]. Available: https://www.unrealengine.com/en-US/blog/unreal-engine-4-22-released

[30] D. Sumaili and S. van der Steen. (2017) Creating a tools pipeline for 'horizon: Zero dawn'. [Online]. Available: https://www.gdcvault.com/play/1024124/Creating-a-Tools-Pipeline-for

[3] www.hpi.uni-potsdam.de/research_school

[4] www.hpi.de/en/research/design-thinking-research-program