In Proceedings of the Workshop on Live Programming Systems (LIVE) 2019
co-located with the Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH)
6 pages, Athens, Greece, October 20, 2019

An Exploratory Literature Study on Live-Tooling in the Game Industry (PREPRINT)

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

Christian Flach
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
christian.flach@hpi.uni-potsdam.de

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

Stefan Ramson
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
stefan.ramson@hpi.uni-potsdam.de

Patrick Rein
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
patrick.rein@hpi.uni-potsdam.de

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
robert.hirschfeld@acm.org

ABSTRACT

The game development industry employs customized tools which support liveness to enable fast iteration. These tools are used in sophisticated software development settings and can serve as inspiration for future directions of the field of live programming. At the same time, game development is a specialized field different from general application development. To provide starting points for exploring work on game development tools, we provide general observations and an overview over ten use cases for tools based on an exploratory literature study on articles published on Gamasutra and videos of the Game Developers Conference. Further, we illustrate how liveness is used in these use cases through example tools and environments. We discuss the examples with regard to the liveness levels according to Tanimoto. Finally, we discuss how insights from these specialized tools can be transferred to tool development for general software development.

CCS CONCEPTS

• **Software and its engineering** → Application specific development environments.

KEYWORDS

live programming, live coding, exploratory programming, game development, tools, literature study

ACM Reference Format:
Tom Beckmann, Christian Flach, Eva Krebs, Stefan Ramson, Patrick Rein, and Robert Hirschfeld. 2019. An Exploratory Literature Study on Live-Tooling in the Game Industry (PREPRINT). In *LIVE'18: Workshop on Live Programming, October 20–25, 2019, Athens, Greece.* ACM, New York, NY, USA, 6 pages. https://doi.org/XXX

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
LIVE'19, October 20–25, 2019, Athens, Greece
© 2019 Association for Computing Machinery.
ACM ISBN XXX…$15.00
https://doi.org/XXX

1 INTRODUCTION

The game development industry can provide inspirations for future tool designs and perspectives on liveness. Game development often happens under tough constraints and has to adhere to strict deadlines due to promised publishing dates. At the same time the design space for games is large and quick iteration is necessary to improve every aspect of the game, including game mechanics, 3D models, animations, artificial intelligence algorithms, or dialog design [20, p. 89]. To sustain quick iterations despite the tough constraints, game development teams often create customized tool sets. To support fast iteration even further, many of these tools incorporate some form of liveness which allows both programmers and artists to see the effects of their changes immediately in the running game.

At the same time, neither the live programming nor the live coding community has covered these tools and environments thoroughly yet [19]. In this paper, we want to provide entry points for readers wanting to explore how live programming is used in game development by providing an overview of use cases for game development tools and example uses of liveness. The overview and the examples for liveness are the result of an exploratory literature study based on articles and videos from professional game creators. We provide an overview of the use cases for game development tools, as game development is a specialized software development domain with particular requirements and constraints different from common software development settings. To understand for which ends liveness is used, we conducted the literature study with the research question: Which use cases do game creators have with regards to their tools when editing and creating games?

To illustrate how liveness is employed in these use cases, we also provide notable examples of tools and environments we encountered. We connect these to the live programming perspective on liveness by discussing their level of liveness according to Tanimoto.

Structure of this Paper. We first describe our methodology of our exploratory literature study based on the SALSA methodology in section 2. In section 3 we describe our general observations of the game development process, the use cases we identified, and examples for liveness in game development tools. Section 4 discusses the threats to validity, and future work. Section 5 concludes the paper.

2 METHODOLOGY

We conducted our exploratory literature study following the SALSA (Search, AppraisaL, Synthesis, Analysis) process [9].

2.1 Search

There are only few academic papers on live game development tools [19]. Therefore, we selected two non-academic sources for the literature study:

- Gamasutra, a blog and news website about games and game development [1].
- GDC Vault, the video archive of the Game Developers Conference (GDC) [2].

Gamasutra. We crawled the Gamasutra website https://www.gamasutra.com recursively, beginning at the startpage.

We selected HTML pages matching these URL patterns:

- https://www.gamasutra.com/blogs/*
- https://www.gamasutra.com/view/*

This resulted in 10 147 articles stretching over 13 709 pages. Each article consists of title, text, and possibly pictures. Sometimes an article only consists of a video and a short text, in which case we treated the video as the document.

GDC Vault. Video recordings of talks given at the Game Developers Conferences are archived and accessible in the GDC Vault. The earliest ones are from GDC 1996. Each video recording includes a title, keywords, speaker, and their company as well as an abstract of the talk. Access to some of the videos requires a paid membership. We crawled the titles and abstracts of all 716 freely available talks of the GDC main conference from 1996 to 2019. We did not crawl videos of sub conferences like VRDC and GDC Europe.

2.2 Appraisal

Both sources deal with professional game development in general, which is why we needed to drill down the amount of content. To do that, we filtered articles and videos by the word "tool".

Gamasutra. We extracted the article texts and titles from the HTML pages and only selected articles that included the word "tool" in either title or text. This process resulted in 2716 remaining pages. We read all article titles and picked articles that referred to tool usage or development. 98 articles remained.

GDC Vault. 120 videos contained the word "tool" in their title or summary. We skimmed through these videos, wrote a one-sentence summary per video, and assigned one of three categories denoting how detailed the video dealt with the tool ("high", "medium", or "low"). We removed 72 videos where either no tools were shown, or only presented very briefly. Of the 48 videos remaining, 7 were classified as "high", 9 as "medium", and 32 as "low". To determine usage of liveness, we looked into the 7 videos classified as "high" in detail.

2.3 Synthesis

We watched/read each of the selected 7 videos and 98 articles and tagged the presented tool based on the domain it applies to. Examples include "Network", "AI" and "Procedural Generation".

We evaluated what purpose the presented tool serves and which information needs it fulfills. Purposes we found include: Editing the level while the game is running, visualizing the underlying data of a game object, and debugging of AI behavior, whereas information needs were more general: "What will happen when I change this value?", "How does my game behave on a slow network or hardware?", and "Why did the dialog end in this state?". We also determined whether and how the tool changes a running system.

2.4 Analysis

To extract common use cases, we clustered similar information needs and purposes we gathered during the synthesis into groups. We then assigned a title to each group that describes its overall use case. We noted general observations that did not fit any concrete use case separately.

To select representative sample tools per liveness level, we analysed our descriptions of how the tools change a running system, as well as the assigned domain tags.

3 STUDY RESULTS

In the following, we will first describe our general observations and their influence on our results. Then we will describe each of the use cases in detail and point out example tools and relevant articles/videos. At the end, we will give examples that exhibit characteristics that fit the liveness levels.

3.1 General Observations

Aside from tools for specific use cases and information needs, we also observed general aspects of the game development process.

In general, big and complex games are usually made by large teams with experts from several domains collaborating on one game project [20]. An artist may have different abilities and requirements for a tool than a graphics programmer. Also, many tools are created throughout the development process as the need for them arises. Sometimes they are also expanded at a later point of development. Since a lot of tools are created during the development process and for different audiences, there usually is an explicit tool team that only works on tools and on improving the tool pipeline for the game.

While a lot of tools are created, not all should be active at the same time in the same editor. Instead, often a specific subset is needed for the current use case of the user. There is also a difference between tools available in external editors, and tools directly integrated into the game [8].

In this paper, we intentionally also covered tools that were not explicitly made for editing source code. Instead, we tried to present the wide variety of tools that game development teams use to create the final software artifact. We therefore included many tools targeted at non-programmers. In particular, a recurring theme was to have programmers work on the actual game for as little as possible and instead have them focus on engine and tool development for game performance improvements. At the same time, we also found many tools specifically targeted at programmers, mostly used for profiling various aspects of the game, which we decided not to report on in this survey.

Another observation is that there are high demands in terms of performance and iteration times to get the game to "feel right". Because of this demand, most designs that we observed were using a data-driven approach. A first step towards this is to simply isolate all magic numbers, such as movement speed or damage, into separate files, thereby making the tuning of these numbers a matter of changing data instead of a matter or programming. Other cases may even allow defining entire class hierarchies or complex expressions in external files that are loaded and evaluated at runtime.

This allows artists and also programmers to make quick changes, often in a declarative manner, without the need to recompile large parts of the project. Making a change and testing that change in the game is done continuously and thus should happen quickly [4]. Fast iteration cycles are needed both when quickly iterating on prototypes in the beginning of a project and also when tuning a specific aspect of a game.

Figure 1: Path visualization of a non-player entity in "Horizon Zero Dawn". The enemy entity will move along the path in the direction indicated by the arrows.

3.2 Use Cases of Game Development Tools

The first research question is: Which use cases do game creators have with regards to their tools when editing and creating games? During the analysis of the dataset, we found ten use case clusters. Even within these use case clusters, there are differences on which temporal dimensions they focus. One of them is concerned with the future, i.e., the consequences of actions, two deal with the past or history of the game's behavior and the rest is about interacting with and altering the present.

Replay / Edit (Past). This use case is about tools that record both interactions made by the user as well as events and behavior triggered by the game. The recording allows to automatically replay the game in exactly the same way again, an example tool for which can be seen in Figure 4. In contrast to a video, this allows developers to inspect and alter the game state at any point in time, which is useful for debugging purposes [14]. The replay can also be used by game designers to adjust gameplay mechanics, by replaying the same sequence multiple times with different configurations (e.g., adjusting the gravity of a level). Recording the past can be used to create viewable example solutions. In addition it is also possible to replay only a part of the recording and then resume normal gameplay to try new interactions, which could be used by developers and players alike.

For example, a replay mechanic was created for a strategy game originally as a debugging helper for programmer and was later turned into a game mechanic for players [10].

Problem Reporting (Past). There are several ways to make reporting problems as helpful as possible for the programmers fixing the problem as well as to enable the reporting player to report as fast as possible after discovering the problem so none are forgotten. One key component to solving a problem usually is the ability to reproduce it, which can be achieved by enhancing textual reports with automatically collected data about what happened. Errors or bugs may occur only in very specific conditions (e.g., the 3D model only glitches at one part of the level at a certain angle), which means easy reproduction of the conditions is very important to solving the problems. For example, a recording made with a tool that supports replay like described in the previous paragraph allows to exactly replicate the problem.

Another way to enhance problem reports can be found in Google Stadia [21]. In the shown tool, all gameplay is recorded as a video. When a problem occurs, the reporting player sends a written description and an automatically created short video including the last few seconds before the problem, the moment of the problem itself, and a few seconds after the problem occurred.

See the Future. There were many articles contemplating tools that predict future game behavior, but nearly none about implemented future-indicating tools. While some approaches in that direction exist [26], many times such tools were just wished for.

One implemented tool, as can be seen in Figure 1, shows the future actions taken by non-player entities (e.g. the path taken by an AI).

Edit Piece of Data. Most games consist of a lot of data whose values also define the game behaviour, like the objects a game level consists of or global data like a gravity value. There are many specialized tools to edit the game's data and see the changes as fast as possible, sometimes live while the game is played, an example for which can be seen in Figure 4.

An example editor with many editing related features is the "Valve Hammer Editor" [25]. The editor shows the edited object from several different angles and thus allows to edit data throughout multiple different views.

Level Creation and Debugging. A commonly game-related use case is level creation and debugging. A game often has specific level creation tools (e.g., a terrain painting tool to change stone into grass). Game developers often want to quickly test changes made to the level by playing the level.

One example tool is the level editor made for "Horizon Zero Dawn" [5]. The tool allows seamless switching between terrain editing and gameplay. The game designer can then determine by playing whether the changes fit artistically, whether the gameplay still works and "feels" correct, and whether new bugs occur due to the change.

Figure 2: A view on electricity supply and usage in "SimCity (2013)". Houses are connected to the power net and act as electricity sinks, visualized by yellow circles.

Make the Invisible Visible. Many games work with values that are usually hidden from the the player. For debugging purposes, it may be helpful to make the hidden values visible within the running game. Showing the values in-game helps connecting them to other visible game behavior. This can be done through methods such as color highlighting or overlay text.

"SimCity(2013)" has several ways to view normally hidden values [15]. Some of these were later changed from development tools into player-facing game mechanics, e.g., to view the electricity supply and usage of a town, as seen in Figure 2.

Make the Visible Invisible. Most games are a graphical medium with a lot of visuals. Sometimes normally visible elements need to be hidden to make it easier to track relevant objects. This can be combined with the previous use case of making hidden elements visible, creating context-views that only show very specific information.

The in-game views in city-building games like "SimCity(2013)" or "City Skylines" also support this by removing color from currently unimportant objects or hiding their 3D models completely [15]. While the mechanic was originally created during development for "SimCity(2013)", the release version of the games also has such views usable by players.

AI / Behavior Debugging. Games with complex behavior and AI often use specialized debugging tools. These tools enable a better understanding of the AI/behavior state by providing abstractions, such as tree view visualizations and diagrams to evaluate logs. It is often required to playtest changes after implementing them.

A very sophisticated AI debugging tool was used to implement the AI in Hitman [16]. It records a timeline for every character, including all decisions and behaviors that led to the current behavior and location of the character. It is possible to jump back and forth in the timeline of the character and see what they were looking at and interacting with at any point in time.

Hardware Emulation. Games are often made for several platforms and hardware configurations. Especially multiplayer games also highly depend on the network connection. In order to test whether the game works on all hardware configurations and whether it functions as expected, hardware can be simulated instead of physically reproduced.

An example tool for network emulation is built into Google Stadia [21]. Since it is a purely cloud-based gaming service, the network connection is a key part of the user experience. The tool allows to simulate different network speeds and errors, effectively changing the environment the game is running in, to determine its behavior when the environment changes.

Interaction between Art and Game. Games combine visual, narrative, and other artistic elements. Tools need to make externally created asset integration and adjustments of game behavior as fast and easy as possible. This encompasses a variety of use cases such as configuring 3D models, procedurally creating assets, and dialog editors.

For example, Polsinelli shows an approach to edit a dialog tree that combines art (text and graphics) with the game mechanic (the dialog) [17].

3.3 Examples of Tools Classified by Liveness Level

In this section, we will give detailed examples of tools by Tanimoto's liveness levels [22, 23]. The system that exhibits the liveness level is not always the game, but can also be an editor, acting on a specific asset or part of the game. While this makes the comparison between individual tools difficult, it will illustrate the range of applications used during the game development process more accurately.

3.3.1 Liveness Level 2. Tools that deal with program flow, but require to be manually executed are considered to have liveness level 2. We found various examples where C++ was used as the language to write both, the engine and game, mostly for performance reasons. Due to the necessity to first compile the code, we would classify all tools using this methodology as level 2. Incremental builds and even hot-swapping modules is increasingly being used in the industry, but still entails considerable delays and drawbacks, such as only being able to migrate certain types of changes [6].

In a GDC Talk, Lightbown demonstrates a tool that can be used to preview the way a 3D model will fall apart when exposed to damage [12]. Typically, a designer or developer would have to launch the game after defining the destruction pattern of a model, move to the right place in the scene and cause the right type of damage and repeat this process for each damage type and iteration. The tool instead allows choosing a type of damage and, while still in the editor, cause this type of damage to an object in the scene, to get an instant, animated preview of what the destruction pattern would look like. In this way, the tool permits designers to execute an otherwise static declaration of a destruction pattern while in the editor. Therefore, the tool is used for both the *See the Future* and *Interaction between Art and Game* use cases.

3.3.2 Liveness Level 3. If a system re-executes upon editing its description, it is considered to exhibit liveness level 3. Examples of this behavior can be found particularly in procedural generation tools. Lambe describes a plugin called DING for the 3D modelling tool Maya for generating insects [11]. It was created to speed up development and iteration times during the creation of one of their

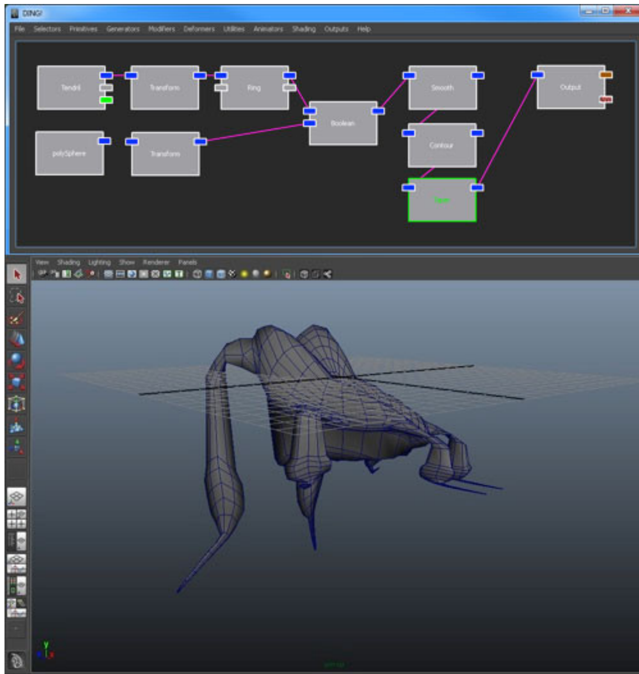

Figure 3: Screenshot of Lambe's system showing the node network for generating an insect above and the result on the bottom. The last node in the network is an output node that takes the transformed stream and displays it as a 3D mesh.

games. The tool is able to create geometry and transform it using a variety of operators. The operations are described in a graph via nodes, each node having a set of parameters that modify its exact behavior. Changing any parameter or the structure (*Edit Piece of Data* use case) of the graph triggers an immediate rebuild of the model, allowing the designer to quickly explore different options.

3.3.3 Liveness Level 4. For liveness level 4, any change that is triggered will be streamed to a running system, instantly rendering effects of the change visible.

An example of this can be found in the dialog system created for "The Witcher 3" [24]. The dialog system in this game is driven by recorded audio files of the dialog, thus handling the *Interaction between Art and Game*. An editor resembling video editing software (seen in Figure 4) allows the designer to create cuts, move objects around, blend animations, trigger events, or even conditionally cause different actions to play, based on the state of the environment, for example depending on the time of day. At all times, every change is instantly rendered visible in a preview that can either be in a dedicated preview window or directly inside the game (*Edit Piece of Data* use case).

3.3.4 Liveness Level 5 and 6. In liveness level 5, the tool strategically suggests changes to adapt the program for what it guesses its user intent is. In liveness level 6, the tool goes beyond this and can generate whole artifacts upon request.

One recent example of liveness level 5 and 6 can be found in the work-in-progress tool "Promethean AI" [13]. The tool allows

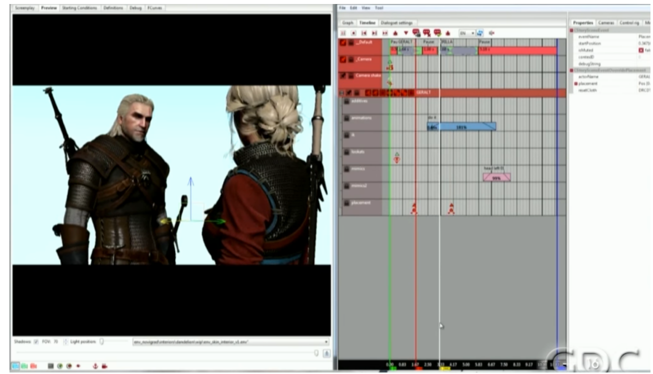

Figure 4: The dialog editor from "The Witcher 3". On the left a preview window plays the animation continuously and allows to jump back to a relevant in the timeline (*Replay / Edit (Past)* use case). On the right a sequence editor allows scheduling events or camera cuts. Parameters can be adjusted in the adjacent properties panel and take effect immediately.

its users to describe a setting in natural language, for example "children's room with 80's furniture". The tool then generates a scene matching this description by sampling from a collection of 3D models, which corresponds to the *Level Creation and Debugging* use case. W consider this to correspond to a system with liveness level 6. Users can then issue abstract commands, such as "swap the bed and the cupboard", and the system takes care of rearranging the adjacent bed table or the general room layout to accommodate the request. Further, recordings of the system demonstrate its ability to collaborate closely with a human author. Asking the system to show green alternatives for the door will prompt the user to select a door from the system's catalogue that will then be automatically inserted and the walls adjusted to fit the door. Editing the room in this way allows to *Edit Piece of Data*. The strategic adaptions that maintain the semantics of the furniture in this case we consider as liveness level 5.

In the context of programming, we can consider this system equivalent to a code autocompletion that understands abstract commands and is able to generate code matching a certain request. Commands such as "open and parse the csv file passed to this method" or "generate a room" respectively would be level 6. Commands to automatically refactor source code while being fully aware of context and constraints, or moving furniture while staying aware of dependencies and semantics of the room, would be level 5.

4 DISCUSSION

In the following, we will the describe threats of validity to the results of the study. After that, we will discuss possible future work.

4.1 Threats to validity

A bias was introduced through our sampling method: first, we only considered articles and videos containing the word "tool", potentially missing articles that describe tools without explicitly mentioning the word. Second, our method of filtering this list of articles and videos was for some parts based on just the title or

short sections in the video. Further, each article or video was only coded by one author.

We only considered the two sources we described that the authors, not coming from a game development background, considered the richest in information. This assessment has not been confirmed by anyone from the game industry.

Lastly, game studios do not necessarily share all their tools or workflows openly. Even though a culture of sharing and building on top of another studio's ideas was a practice we often observed in our sources, we have found for example not a single article or video in which a studio presented an exhaustive overview of the tools it uses.

4.2 Future Work

This paper should only be regarded as a basis for further investigation. In particular, a quantitative analysis, potentially involving even more sources, could be interesting. It may allow to get a more accurate feeling on the level of exchange inside the game development community and ascertain how prevalent the aspects we identified as essential for game development tools actually are across different tools.

Further, it could be interesting to exchange with people from the game industry on the findings. This could both act as verification of the findings, as well as a means to determine new sources for literature.

It could also be interesting to specifically analyse the various visual programming languages currently in use by the game development industry. These tools act on various data formats, such as source code, dialog trees, or world generator settings.

Inspecting existing, state of the art game development tools and identifying opportunities where insights from the live-programming community could benefit the tooling could also be considered. This might both benefit the visibility of these insights as well as benefit the game development community.

5 CONCLUSION

We examined a variety of game development tools, some of which are aimed at programmers, most aimed at artists and (game) designers. We identified information needs, clustered them into common use cases, and gathered general observations on the game development process.

We believe that the strongly visual, interactive, and highly domain-specific nature of games and their tools could be of interest to the live programming community. The various tools we presented may serve as inspiration on how to make programming interfaces accessible to non-programmers. Even though some of the observed tools rely on the fact that there is a simulation constantly running in the background, many systems could be automated to exhibit similar behavior, even if they are supposedly static, for example by means of providing examples to functions [7, 18].

While some of the use cases we found are relatively game-specific (e.g., *Level Creation and Debugging*), some are also applicable to general software development. For example, many debuggers show the values of variables while the program is halted (*Make the Invisible Visible*). Further, some tools allow to emulate a bad network connection, e.g., to check how a website loads under these conditions (*Hardware Emulation*) [3].

ACKNOWLEDGMENTS

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

REFERENCES

[1] 2019. *Gamasutra - The Art & Business of Making Games.* https://www.gamasutra.com/
[2] 2019. *GDC Vault.* https://www.gdcvault.com/
[3] Kayce Basques. 2019. *Network Analysis Reference.* https://developers.google.com/web/tools/chrome-devtools/network/reference#throttling
[4] Stephen Broadley. 2016. *Empowering Content Creators.* https://www.gdcvault.com/play/1023274/Empowering-Content
[5] Sander van der Steen Dan Sumaili. 2017. *Creating a Tools Pipeline for 'Horizon: Zero Dawn'.* https://www.gdcvault.com/play/1024124/Creating-a-Tools-Pipeline-for
[6] Mark DeLoura. 2019. *Unreal Engine 4.22 released.* https://www.unrealengine.com/en-US/blog/unreal-engine-4-22-released
[7] Jonathan Edwards. 2004. Example Centric Programming. *SIGPLAN Not.* 39, 12 (Dec. 2004), 84–91. https://doi.org/10.1145/1052883.1052894
[8] David "Rez" Graham. 2012. *In-Game Debugging and Visualization Tools.* In-GameDebuggingandVisualizationTools
[9] Maria Grant and Andrew Booth. 2009. A Typology of Reviews: An Analysis of 14 Review Types and Associated Methodologies. *Health Information & Libraries Journal* 26, 2 (2009), 91–108. https://doi.org/10.1111/j.1471-1842.2009.00848.x
[10] Stas Korotaev. 2018. *Time Manipulation in Unity - Recorded Solutions.* https://web.archive.org/web/20190801140838/https://gamasutra.com/blogs/StasKorotaev/20180715/322091/Time_Manipulation_in_Unity__Recorded_Solutions.php
[11] Ichiro Lambe. 2012. *Procedural Content Generation: Thinking With Modules.* https://web.archive.org/web/20190801115617/https://gamasutra.com/view/feature/174311/procedural_content_generation_.php?page=5
[12] David Lightbown. 2017. *Getting Productivity from Play: How Ubisoft Is Making Better Tools by Using a Familiar Resource.* https://www.gdcvault.com/play/1023953/Getting-Productivity-from-Play-How
[13] Alissa McAloon. 2018. *Promethean AI aims to take the grunt work out of worldbuilding through AI.* https://web.archive.org/web/20190801114902/https://gamasutra.com/view/news/322318/Promethean_AI_aims_to_take_the_grunt_work_out_of_worldbuilding_through_AI.php
[14] Sean McDirmid. 2013. Usable live programming. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld (Eds.). ACM, 53–62. https://doi.org/10.1145/2509578.2509585
[15] Dan Moskowitz. 2013. *Exploring SimCity: A Conscious Process of Discovery.* https://www.gdcvault.com/browse/gdc-13/play/1017948
[16] Christian Nutt. 2011. *An Engine For Assassination: IO's Tech Director Speaks.* https://web.archive.org/web/20190801141745/https://gamasutra.com/view/feature/134933/an_engine_for_assassination_ios_.php
[17] Pietro Polsinelli. 2019. *Direction Tools For Your Game's Dialogues.* https://web.archive.org/web/20190801141934/https://gamasutra.com/blogs/PietroPolsinelli/20190108/333894/Direction_Tools_For_Your_Games_Dialogues.php
[18] 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. *CoRR* abs/1902.00549 (2019). arXiv:1902.00549 http://arxiv.org/abs/1902.00549
[19] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2018. Exploratory and Live, Programming and Coding: A Literature Study Comparing Perspectives on Liveness. *The Art, Science, and Engineering of Programming* 3, 1 (2018). https://doi.org/10.22152/programming-journal.org/2019/3/1
[20] Jesse Schell. 2014. *The Art of Game Design: A Book of Lenses* (2nd ed.). A. K. Peters, Ltd., Natick, MA, USA.
[21] Pawel Siarkiewicz. 2019. *A Guide to Developing on Stadia.* https://www.gdcvault.com/play/1026499
[22] Steven Tanimoto. 2013. A Perspective on the Evolution of Live Programming. In *1st International Workshop on Live Programming (LIVE 2013) (LIVE '13).* IEEE Press, Piscataway, NJ, USA, 31–34. https://doi.org/10.1109/LIVE.2013.6617346
[23] Steven L. Tanimoto. 1990. VIVA: A Visual Language for Image Processing. *Journal of Visual Languages and Computing* 1, 2 (1990), 127–139. https://doi.org/10.1016/S1045-926X(05)80012-6
[24] Piotr Tomsinski. 2016. *Behind the Scenes of Cinematic Dialogues in 'The Witcher 3: Wild Hunt'.* https://www.gdcvault.com/play/1023285/Behind-the-Scenes-of-Cinematic
[25] Valve. [n.d.]. *Valve Hammer Editor.* https://web.archive.org/web/20190801141246/https://developer.valvesoftware.com/wiki/Valve_Hammer_Editor
[26] Bret Victor. 2012. *Inventing on Principle.* http://worrydream.com/#!/InventingOnPrinciple