JOURNAL FOR MEDIA HISTORY

Adrian Demleitner

Finding the 'Magic' in Programming Games: A Critical Digital Humanities Approach to Studying Video Game Code

Abstract

The digital humanities encompass large-scale computational methods, the digitisation of historical sources but also critical approaches to all things digital. This paper takes a humanities-based approach to researching historical video game source code, examining how cultural meanings become embedded in technical structures, and how these structures then influence cultural possibilities. By studying 'ludemes', the fundamental elements of gameplay, the text explores how programmers develop theories about their software and how players engage with video games through these aesthetic-technical foundations. Drawing on media archaeology, it argues that source code operates as a layer atop multiple strata of hardware, connected to silicon and electronic signals. [This situates code within computational infrastructure and materiality, presenting older computing devices and their source code as a means of understanding the operational principles of computation and the emergence of culture from programming practice.] By synthesising technical and cultural analysis, the paper positions source code as cultural documentation worthy of humanities investigation. This approach bridges the gap between computer science and cultural studies, offering a new perspective on how interactive media participate in broader cultural transmission and create new possibilities for meaning-making, memory, and imagination.

In order to demonstrate this approach, the outlined methodology involves play-analysis, distant code reading, and close interpretation informed by media archaeological perspectives on code and materiality. The case study draws parallels between the original assembly code of the 1991 puzzle game Poizone and its 2023 Python port. This reveals that, despite the gameplay remaining unchanged, the code discloses different conceptualisations and developer perspectives that are encoded in technical architectures. A structural source code analysis demonstrates to yield insights that are not accessible through conventional media studies. This analysis shows that cultural meanings have the capacity to persist across technological changes, and that computational materiality exerts a significant influence on the process of meaning-making through the medium of algorithmic logic and player experience.

Keywords

Critical Code Studies | Ludemes | Digital Humanities | Source Code Analysis | Video Game Studies

JOURNAL FOR MEDIA HISTORY

> I finally found the magic in programming games, in being able to do whatever you want on a screen … There was this magic, which we don't have anymore: the TV was originally just a device for watching TV, and it was totally passive. And then suddenly, from one moment to the other, you plug a device into it and you can control what happens on the TV.[1]

There is a quasi-magical moment when computational instructions translate into an aesthetic experience for someone playing a video game. In this article, I concentrate on programming as a culturally significant activity, and specifically, how abstract source code is transformed into a lived human experience. My research follows this process of transmission from the programmer's theory of the software they are building[2] to the affordances that emerge on screen and enable participation in the video game. Although both ends of this path seem clear-cut, I struggled to cross the chasm between them. I have since sought a way to conceptualise the semiotic bridge which makes this moment of translation accessible to analysis and theorising.

The elements this bridge is built from must be able to confer between two fundamentally different ways of accessing and interacting with software.[3] Developers, on the one hand, abstract their intentions and encode them programmatically into source code.[4] On the other hand, there are players' phenomenological reactions to interacting with the running game.[5] This thread from developer intentions to player experiences must at one point feature elements of translation rooted in computational materiality, emerging through programming, code semantics, machine instructions, and binary signals.

Interrogating these elements is difficult because they lie at the intersection of two distinct research domains.[6] Programming has long been firmly in the hands of computer science and operations research. Player experiences, on the other hand, pertain to the social sciences and the humanities. Neither has the epistemological nor the methodological framing that allows a crossing of the gap and venturing further into the other's territory. As a promising move beyond traditional academic disciplines, digital humanities and critical code studies have established access points,[7] especially through close surface readings of source code for humanities interests.[8] Despite some favourable results, however, interest in source code from the humanities still lacks access to the structural and organisational aspects of coding. Thus, video game studies have so far only obtained limited methodological access to inquire into the computational materiality of their research objects.

The fundamental questions guiding this article concern this difficult aspect and ask whether it is worth pursuing. Can the humanities research source code on a structural level, and does video game source code offer insights that cannot be accessed by any other means? In the following, I want to offer a simple approach to researching video games that interprets programmers' theory-building in their software[9] and allows researchers to look into the structural aspects of source code, starting from a player's experience. From that first inquiry, the perspective shifts to a second set of questions regarding the very nature and form of the theories a developer builds during programming. What can be observed, looking at the developer through their source code? Together with the case worked on throughout the text, the discussion concentrates on what is to be gained from close and structural readings of code in relation to research interests within the humanities.

The video game analysed for this inquiry is *Poizone*,[10] a simple arcade puzzler developed by Paolo Baerlocher (code), Marc Andreoli (graphics), and Fabrice Hautecloque (music) in 1991 for the Acorn Archimedes, a series of personal computers developed in the UK. Of essential value to the questions raised here is the fact that Baerlocher ported the game to modern computer systems in 2023. It is exactly this difference in the codebases of the two versions that enables an investigation into the elements that translate between developer intentions and player experience. The latter is essentially the same in either version, but the source code is fundamentally different in nature and organisation. While the experience of playing the game remains unchanged, the programmer's intentions are unequally encoded, enabling a comparison and theorisation of this difference.

Backtracking the semantic transmission from source code to player experience allows for an interpretation of the programmer's intentions at the time of coding. Or, more boldly claimed, it allows the rebuilding of the programmer's theory-building for the software they were creating. Programming is not merely writing instructions for a machine, but also a cognitive act in itself.[11] In analysing critical media scholar Friedrich Kittler's approach to programming, Mark C. Marino argues that 'for Kittler, programming was a kind of theorizing, an activity of philosophical labor, involving interfacing with a machine and tracing ideas by expressing them in code'.[12]

Peter Naur explores coding as an act of thinking and theorising in his work on mental models in software creation. According to Naur, programmers develop a theory of their software through the process of writing the necessary code.[13] This process becomes evident in how the two versions of *Poizone* handle movement and collisions within the environment differently. How does the game know when the player avatar bumps into a wall? Embedded in this essential functionality is the

programmer's perspective on how they theorise their software and gameplay. These specific examples and questions will be used as a thread throughout this article.

Theoretical Considerations

This understanding of programming and a programmer's intentions assumes their cultural significance. Early works, such as the foundational *The Psychology of Computer Programming*,[14] captured the discourse on programming, shedding light on the practice from an operational perspective and incorporating an engineering viewpoint. Notably, programming has always existed as a political force,[15] a leisure activity from which homebrew video games emerged,[16] and an artistic endeavour.[17] Annette Vee's work on the history of Dartmouth BASIC,[18] a programming language developed in the 1960s, probably best outlines this significance for cultural production. The pedagogical driver behind this specific programming language was neither academic engagement with computation nor enabling a career path as a software developer. As Vee notes regarding the language's design and pedagogy, the intention was to make computing accessible:

> Kemeny and Kurtz wanted to reach the other 75% of Dartmouth undergraduates, those who were majoring in the humanities or social sciences. (…) While they generally described their pedagogical motivations along the lines of civic engagement, (…) Kurtz also told me they were invested in the creative potential of the computer.[19]

Other authors have also supported the idea that programming and coding are culturally significant. Early on, Friedrich Kittler theorised code and outlined how programming is a continuation of the practice of encoding,[20] which is necessary in all forms of communication. The emergence of critical code studies in the mid-2000s has established a practice of hermeneutical and close readings of code as a way of theorising it.[21] This approach is exemplified by Willumsen's formal analysis of *Passage*'s source code.[22] She argues that a close reading of source code can generate insights into video games as works that go beyond what their authors claim in public reflections. Critical code studies have since established themselves as an influential and valuable perspective in the digital humanities,[23] especially within the branch focused on studying born-digital works.

JOURNAL FOR MEDIA HISTORY

However, this discourse lacks a formal analysis of what David M. Berry calls the 'computational image', a material perspective on the encoding and transmission of meaning in source code:

> Where Heidegger contrasts universe and world, and for Sellars this indicates the *scientific* and the *manifest* image, here I want to think through the possibility of a third image, that of the closed 'world' of the computer, the computational image. I want to understand how one knows one's way around with respect to things in a computational image, and conversely, the computational way of making sense of the world and how it gives expression to that sensibility.[24]

Sonja Fizek outlines the initial inability to cross disciplines or domains of knowledge, as discussed in the introduction, through a media-ontological perspective.[25] She points out that video game images are made up of more than just light; they are also constructed of mathematics, calculated on the fly by a machine and in relation to its operations. This process creates the need for a different kind of visual literacy among spectators (from players to researchers). Thus, Fizek positions video games as aesthetic forms that rely on the tension between representation and computation. Born-digital images that emerge from running games are no longer only representations – they display software operations, but are also operational.[26] Thus, in order to understand the thread of translation from machine execution to human experience, there is a need to study the game's computational materiality, and vice versa. Doing so requires, among other approaches, deep readings of the source code's different aspects, such as semantics, pragmatics, and structures.[27] Without bridging these perspectives, research inquiries might miss out on essential insights regarding how technical constraints shape creative decisions, how programming practices encode visions of play, and how gaming cultures emerge.

Building on semiotics, I approach this problem space through the study of a game's ludemes, guided by how programmers conceptualise their code and how players access a video game. Ludemes are affordances with the particular function of making games playable. They have various definitions, most often as fundamental units of designing or researching video games.[28] Interestingly, Pierre-Yves Hurel and Damien Hansen approach ludemes from the perspectives of sociology and semiotics,[29] and Hansen outlines ludemes as fundamental meaningful units of a quasi-video game language, consisting of a mechanical element or rule of interaction (*mechanème*) and an audiovisual

representation (*graphème*, *acoustème*).[30] These ludemes are then organised into syntactic and recursive sequences, creating the larger structures of the game. Unlike some previous approaches that focused primarily on rules and mechanics, Hurel's and Hansen's conceptions of ludemes incorporate both the concrete elements of games and aspects of game makers' intentions and players' experiences. Finally, these ludemes are encoded in the game's computational materiality and highlight how rules and game mechanics can have rhetorical effects.[31]

Like Hurel and Hansen, among other authors, I do not necessarily agree on all the defining aspects of what makes ludemes. Moreover, the lax definition of ludemes may pose a significant risk of becoming a conceptual trick that allows a justification of findings without further inquiry. There is, nonetheless, a valid argument for the productive powers of this concept in positioning it as a boundary object. As outlined by Susan Leigh Star, boundary objects are concepts that are flexible enough to be adapted and understood differently by multiple communities or disciplines while still maintaining enough structure to facilitate collaboration and sharing across those boundaries.[32] Bridging insights and approaches in game studies, semiotics, and sociology, these humble units then continue to bear fruit here through critical code studies.

Consequently, I propose a working definition of these small units of gameplay: ludemes are the affordances games offer to become playable, organised into semantic sequences and of a memetic nature. They cover phenomenological ground through their aesthetic representations,[33] are structurally inscribed in source code by their mechanical aspects, and open up semiotic investigations through their semantics and memetics. Ludemes can be learned and known through gameplay, but need to be conceptualised and encoded through programming.[34]

Contextualisation

Baerlocher's path to serious game development accelerated when he acquired an Acorn Archimedes computer in his late teenage years. The machine's capabilities, demonstrated by David Braben's 3D game *Zarch*,[35] convinced him that a career in video games was possible. This computer system was mainly marketed in the UK and had only limited success in Switzerland, where the game was developed in 1991. Following this acquisition, he joined the demo group Arc Angels during his undergraduate studies and connected with like-minded programmers across Europe as well as the demo scene community surrounding the Archimedes. Baerlocher repeatedly mentioned his

fascination with the Acorn's ARM processor capabilities, demonstrating his knowledge of its architecture and showcasing his technical prowess:

> I can show exactly the piece of code where I try to exploit the capabilities of the ARM instructions to the maximum. To do it in the minimum number of cycles. (…) With the ARM processor, each instruction can be conditional, meaning that it will only be executed if the last test gave a positive or negative result. (…) So, we could do fairly subtle things with few instructions. That was one of the tricks, among others, that allowed us to optimize these drawing routines.[36]

The emergence of the ARM processor family is, in itself, a significant chapter in computer history. ARM is shorthand for 'Acorn RISC Machine', where RISC stands for 'Reduced Instruction Set Computer', indicating a simpler processor architecture. ARM was developed in the 1980s by Acorn Computers, with Sophie Wilson as the principal architect of its instruction set. The processor architecture was designed to be efficient and simple, and it has since become one of the most widely adopted processor designs globally, powering everything from smartphones to servers. Whereas the first version of *Poizone* was written in ARM Assembly, the port was realised in Python. There is a fundamental difference between these two programming languages that warrants consideration: their categorisation as low- and high-level, respectively. Low-level languages like Assembly are close to machine code and ask the programmer to explicitly manage hardware resources and memory, giving control but making code harder to read and write. High-level languages like Python, on the other hand, abstract away those hardware details, allowing for a focus on logic and more easily readable syntax, although at the cost of control and efficiency.

In light of my interests in programming as culturally significant, it would be easy to dismiss Friedrich Kittler's provocative statement that 'there is no software'.[37] His argument need not be read as a dismissal of the cultural agency of programming or programming languages, but rather as an observation on the nature of computing and code that challenges common assumptions about how computers actually work. Contrary to the promises of the software industry, Kittler argues that software does not exist as an independent, immaterial abstraction. All code ultimately depends on, and resolves into, hardware, physics, and electrical processes.

> Or rather, it would not exist if computer systems did not need – at least until now – to coexist with an environment of everyday languages. (…) The so-called philosophy of the so-called computing community places all its stock in hiding hardware behind software, and electronic signifiers behind human/machine interfaces. In a duly philanthropic spirit, handbooks for high-level programming languages warn of the mental breakdown that would result from writing trigonometric functions in assembler code.[38]

Media archaeologist Wolfgang Ernst continues Kittler's line of thought, arguing that computation is not just symbolic processing but a material, physical operation carried out by devices, circuits, timing, and energy flows.[39] Hardware is not incidental, but shapes what computation can do, how fast it can do it, and how it ages or fails. Ernst's inquiries connect computing to media archaeology. In this context, computer code does not ephemerally reside in the cloud, but is a concise layer on top of several strata of hardware, connected all the way down to silicon and electronic signals. Building on the archaeological perspectives of Jussi Parikka and John Durham Peters on the material substrates of media,[40] the digital humanities can profit from deepening their engagement with born-digital objects by considering their situatedness in computational infrastructure and hardware.

At the same time, media archaeology can benefit from embracing the cognitive and operational structures that emerge from code, treating computational materiality as inseparable from algorithmic logic. Seen like that, older computing devices and their source code are useful not just as historical artefacts but also as ways to understand the operational principles of computation and how culture emerged from the practice of programming. Kittler's and Ernst's theoretical perspectives on the relationships among ludemic conceptualisation, code, and materiality will inform the following analysis.

Analysis

To follow the semiotic thread between developers and players as it manifests in video games' computational materiality, researchers need to adopt both perspectives. This approach is attempted through a process outlined in Table 1, with the first phase concentrating on identifying relevant ludemes through a play analysis.[41] The focus is on the game's affordances for play.[42] This allows for the generation of a vocabulary of concepts through which the source code can be approached as a catalogue of questions and computational inquiries. The second phase is guided by a distant reading

Phase	Step	Description
1	*Play Analysis*	*Playing the game and noting what affordances are offered to the player in order to be able to play the game.*
	Formation of Ludemes	*Reviewing notes and clustering of mechanical and aesthetic aspects into small units of gameplay.*
2	*Distant Reading Code*	*Exploring the entire codebase with the help of text manipulation tools according to the ludemic vocabulary.*
	Mapping Ludemes in Code	*Sorting the findings into a format which can be reviewed and analysed towards its structures.*
3	*Close Reading Code*	*Reading through findings regarding stylistic and semantic values.*
	Analysis of Findings	*Contextualising findings and relating them within the given structures and their paratextual material. Critical analysis in relation to the research questions and semiotic transfers.*

Table 1. The phases and steps for analysing source code using a ludemic approach.

strategy, in which basic text and pattern-searching tools help gain a fundamental understanding of the codebase's structures and provide an overview of them.[43] Completing the analysis is a scalable reading,[44] in which breakdowns of gameplay and source code help transition into critical close readings and the analysis of findings. All of these will be discussed in detail in the following sections.

When Paolo Baerlocher developed the puzzle game *Poizone* in 1991, the use of the keyboard's arrow keys for gaming was already an established method of play. These humble directional inputs were originally developed to move a cursor around a text-based screen. Early computers did not feature graphical user interfaces or the ability to interact with a mouse; instead, they displayed a terminal and walls of text. Consequently, programmers of early arcade game adaptations for home computers had to figure out how to play these games without a joystick, since the hardware was not present in every household. Suddenly, a fundamental aspect from the perspective of play had to be negotiated, and developers had to invest considerable effort in making this kind of interaction work, making it an interesting case to study. In the following, the analysis is restricted to the discussion of this specific aspect of gameplay and its technical implementation.

Play-Analysis and Formation of Ludemes

The analysis begins with setting the stage for *Poizone*, a block-pushing puzzle game that draws inspiration from the genre-defining *Soko-Ban*[45] and especially the arcade game *Pengo*.[46] The protagonist finds itself in various landscapes, from frozen plateaus to the insides of an electronic circuit. Where *Pengo* was rather minimal and *Pacman*-esque in its approach and aesthetics, concentrating on pushing blocks and evading monsters, *Poizone* has a refined narrative centred on toxic waste and environmental pollution – an intentional choice by Baerlocher. The challenge of the game increases through each level, asking the player to remove a number of toxic blocks within a given time. These toxic blocks can be of different types, each with its own rules for removal. Some need to be pushed and slide freely for a moment before they crush. Others can only be removed by pushing from the sides, not from the top or bottom.

Figure 1 shows the protagonist, Zozo, pushing the 'Green Chemical Substance' block in order to destroy it. Several elements stand out as important to gameplay. Through the penguin avatar, the player can interact with the game world and different kinds of blocks, while a heads-up display informs us of our progress within the game. All of these elements, along with their interactions, can be defined as ludemes. The figure also highlights another critical aspect of ludemes: their memetic properties. Taking inspiration from *Soko-Ban* and *Pengo*, the ludic elements and visual language of both these games can be recognised in *Poizone*, and figure as genetic material from which the latter drew and evolved.

What follows is a thick description of the first few moments of playing *Poizone*. Figure 2 shows what we see when we start playing. Our protagonist, Zozo, is centred on the screen and looks straight at us. We are surrounded by blank space and various kinds of blocks. From the tutorial, we know that we can – and must – interact with these blocks. On the right side, an interface displays essential information related to the game's challenges, namely, removing a certain number of toxic blocks within a given time. There are further visual elements, such as small snow-covered bumps and flags. The only action left to do at this moment is to press the arrow keys on the keyboard and move around. We quickly learn that the blocks stop our movement, but not the decorative elements, such as the little blue flag. Walking up to a block allows us to commence the game's second core action: pushing, which is essential to remove blocks from the game. The tutorial taught us that certain blocks have certain rules that we need to adhere to.

Figure 1. Zozo crushing the green chemicals block into a non-functional block in order to clean up the environment and win this level. Screenshot by the author.

Pushing the blocks the right way destroys them, as shown in Figure 1. Meanwhile, the percentage in the heads-up interface increases, giving us a rough idea of how many blocks need to be removed in order to reach the goal. Destroying the decorative blocks does not affect the counter, but they still play a crucial role. The tutorial taught us that the green blocks need to slide for at least a bit before they can crash into another block to remove it. Not adhering to this rule knocks us out for a brief moment. At this point, it becomes apparent that there are no lives or energy in the game, unlike other games. Instead, lying unconscious, time is ticking away while we are being forced to lie still.[47] Pressing the arrow keys at this moment has no effect; we are reduced to waiting to awake again before we can continue playing.

Figure 2. Zozo looking at us, awaiting further instructions. Screenshot by the author.

This brief excerpt of a longer playthrough of the first level leaves behind a rich vocabulary of ludemes and their affordances. These concepts had to be extracted from visual and written notes made during the play analysis through a process that clusters the observations into small aesthetic-mechanical units of gameplay. Play analysis can be quite bountiful, and the need to focus on specific aspects and details in building a ludemic vocabulary arises. Reducing the scope of research data, the analysis concentrates on characters, objects, and actions that, in one way or another, concern themselves with the game mechanics of moving the player avatar within the game. Table 2 collects those ludemes, allowing the articulation of a set of questions through which the source code is approached in the next phase.

Ludeme	Description
Avatar	*The character that we control during gameplay: Zozo.*
Moving	*Moving our character through keyboard or other hardware input.*
Exploring	*A game loop in which we move our character around to learn more about the environment and how we relate to it.*
Blocks	*The main object with which we interact in order to reach the game's given goals is by removing some of them.*
Pushing	*The main way of interacting with blocks.*
Removing	*Removing is a core mechanic of the game, but it depends on moving and pushing the right way. In more complex arrangements, removing a block involves several steps, akin to a puzzle, to successfully close this game loop.*
Monsters	*These non-player characters create additional challenges, since a collision with them knocks the avatar out, and make the moving and pushing mechanics more difficult. They can be removed by pushing blocks into them.*
Evading	*The act of trying to accomplish the core goal of removing enough blocks while not being hindered by the monsters.*
Heads-up Display	*This user interface on the right side tracks vital metrics about our progress toward our goal of removing enough blocks within the given time. It is an extradiegetic ludeme that is nonetheless crucial for playing the game.*

Table 2. List of the constructed ludemes and their descriptions.

- Player control: How is the player's wish to move taken into account – where does the program capture and react to player input?
- Avatar representation: How is the internal representation of the avatar created, and how does the program know where Zozo is and in which direction it is moving?
- Internal state management: How does the program enable interaction with the environment, and how does it know where the avatar can or cannot walk?
- Interaction feedback: How does the program communicate its internal state? For example, how does it keep track if a move was successful, or if Zozo collided with anything, and let the player know the result of the initial input?

Distant Reading Source Code and Mapping Ludemes

Rather than attempting to read code line by line as one might read a text, Till A. Heilmann argues for a surface-level, computational method that treats the source code as raw material to be processed and reduced through operational transformations.[48] His approach begins with basic observations about file organisation, sizes, and types, then progressively breaks down the code into smaller, analysable components using standard command-line tools like *grep, sed,* and *awk*. These tools can be found on Unix-like systems and are generally applied for searching, editing, and processing text files. This method is based on the assumption that source code can go beyond manual observation and contains meaningful patterns and structures that can be studied.

This approach might seem underwhelming compared to other computational approaches in the digital humanities, especially cultural analytics[49] and software-assisted distant reading,[50] which use machine learning and advanced algorithms to process millions of pages of text or image files. The issue at hand links back to Peter Naur's *Programming as Theory Building*,[51] in accordance with Janna Omena's reflections on technicity.[52] If the original conceptualisation of the software coming into being is held by the programmers themselves, the digital research toolset needs to reflect that. A distant reading will necessarily inject layers of abstraction between a researcher and the code they are studying, the intentions of which they are attempting to rebuild and interpret. Applying machine learning to source code can provide overviews, but such distant reading methods will only allow for a limited interpretation of the meaning of direct experience.

Heilmann's methodology involves systematically extracting different elements from the codebase, such as separating comments from actual code, creating comprehensive word lists, and mapping object hierarchies.[53] This first step helps in constructing patterns, frequencies, and structural relationships that would become invisible when reading individual sections of code. This methodology aligns with digital humanities approaches to computational text analysis, where systematic extraction and pattern recognition serve as interpretive tools rather than ends in themselves, similar to how query design in database studies treats the interrogation of data structures as a form of critical analysis.[54] This approach serves as preparation rather than a replacement for traditional code analysis, providing the entry points and structural understanding necessary for the meaningful interpretation of how a software reflects broader cultural and technological contexts.

The *grep* command in Listing 1 illustrates this approach by searching for where the avatar is referenced in the code. This inquiry for patterns searches for 'pengo', 'pengi', and 'penguin' at the same time, regardless of whether it was written with or without capital letters. In a brief skim of the code, different ways of addressing Zozo – the in-game name of the protagonist – can be observed. Within the code, the penguin is generally labelled 'pengo', honouring *Poizone*'s inspiration and referencing the *Pengo* arcade game. Since *Poizone* has a two-player feature, the code also sometimes speaks of 'pengi', the plural of 'pengo'. Zozo can then be found called 'pengo', 'pengo1', or 'pengi1', in different variants with and without capital letters. This variation can be captured comfortably with a *grep* query, of which an abbreviated excerpt is shown in Listing 2:

```
grep -iEn "peng([oi]|uin)" POIZONE_10.txt
```

Listing 1. Example of the grep command to search the code for the avatar.

```
5542:CMP R7,#0:MOVGT R1,R1,LSR#2:ADDGT R1,R1,#32    ; pengi pousse le bloc
5548:.noPengo2
5640:       ;**************** IDEM but for pengo 2 ****************;
5896:.Assassin          ; for 2 players mode > ctrl if Pengo killed by
partner
5898:; Cas Pengo1
5901:CMP R0,#0:BMI casPengo2       ; Pengo1 deja mort! plus la peine
d'enqueter
5903:CMN R0,#1:BEQ casPengo2                          ; pas de crime
possible
[...]
6066:CMP R0,#0         ; Pengi est-il arrete'?
6294:CMP R6,#12        ; Pengo mort si CFC pas casse' depuis le haut
6312:CMP R6,#8        ; m'indique direction du pengo R6<8 = left&right
6466:.ArbitreStartBlk2  ; idem for pengo2
6587:BL MovePengi1
6588:BL MovePengi2
```

Listing 2. Abbreviated output of the grep search command. The number on the left is the command output, which indicates the line number in the code.

By reducing thousands of lines of code to a manageable set of inquiries, the approach enables the identification of sections of code related to the ludemic vocabulary for closer examination. Further, the process allows for a better understanding of the game's structural foundations and

prepares the groundwork for deeper cultural and technical analysis. Tables 3 and 4 index relevant places in the source code related to relevant ludemes and list them for the two *Poizone* versions.

The table breakdown of ludemic elements provides a basic inventory of the distribution of game mechanics and objects across the codebase. It enables a high-level perspective on how the code is

Ludeme	Lines	Commentary
Avatar	*254*	*Coordinates variables*
	689	*'dessine un sprite sous controle du Controller'*
	1349	*'Sprite Controller (interface pour l'utilisateur) (max. 20 sprites)'*
Moving	*272*	*Moving and pushing variables*
	652	*Collision control*
	4224	*Joystick routines*
	4353	*Input flags that keep the player's choice of direction*
	4411	*Reading input keys or controller*
Transporter	*6060*	*'Transporter', which is a teleporter*
Teleporting	*6155*	*Teleportation*
Blocks	*885*	*'Dessine tout l'ecran de jeu (Background+Blocks)'*
	6238	*Block rules*
Pushing	*272*	*Moving and pushing variables*
	4430	*Reading push and checking if pushable*
	4905	*Initiate destroying block '.ArbitreCrash'*

Table 3. List of relevant pieces of the assembly code regarding the ludemes. Commentaries in quotes are verbatim taken as found in code, whereas the rest are notes made by the author.

Ludeme	File	Lines	Commentary
Avatar	*poizone.py*	*1175*	*Rendering Zozo*
Moving	*poizone.py*	*799*	*Reading direction input*
Transporter	*globals.py*	*151*	*Teleporters are initiated via land data*
Teleporting	*penguin.py*	*390*	*Check if on the teleporter and react accordingly*
Block(s)	*constants.py*	*115*	*Block class and types*
	penguin.py	*102*	*Block rules*
	poizone.py	*623*	*Rendering background and blocks*
Pushing	*poizone.py*	*842*	*Reading controls*
	penguin.py	*61*	*Push block*

Table 4. Analogous to Table 3 for the Python source code.

organised and how much of it had to be written for which ludemes. This is a first step towards a structural reading of source code. At this moment, there is no focus on the specific details of the technical implementation, but rather on understanding the mental model the programmer has built during development. This mental model is akin to the theory of software coming into being and relates at once to the developer's intention and to the technical context, both of which shape how the code is written and structured.

Source code is most often pragmatically optimised. A fundamental aspect of this practice is the modularisation and reuse of pieces of code. This fragmentation occurs through different structures that programming languages offer, such as subroutines or functions, or by splitting the codebase into several files. This process is already evident when tracking the ludemes and listing the relevant code in Tables 3 and 4. In subroutines, for example, the developer simply tells the program to jump to a different place in the code, execute that part, and then return. Since this can be done in various places throughout the program, the subroutine can be reused for tasks that need to be performed many times, such as rendering a block on the screen. This referentiality leads to a dense network of

pointers, links, and information flows. If source code is text, it leans heavily toward hypertext, as conceptually outlined by George Landow,[55] or toward cybertext, as outlined by Espen Aarseth.[56]

To fully grasp the structural relationships among them, a visual mapping approach can be valuable.[57] While the tables index the presence and location of specific ludemes, they cannot easily communicate the interconnections among them. A relational visualisation transforms this tabular data into a network that highlights the entanglement and distribution of the ludemes in code. Figure 3 illustrates this approach, clarifying how hierarchical dependencies and fundamental structures are expressed. This visual representation makes it possible to identify clusters of related functionality, trace the flow of game logic across different programming languages and files, and understand how seemingly disparate ludemes, such as teleportation and avatar movement, might share underlying computational foundations.

Close Reading the Source Code

This third and final phase of the analysis finally tackles the question asked at the beginning of this section regarding how the game knows where and when the player is allowed to walk. This is just one of the many possible questions, but *moving around Zozo* in *Poizone* will guide this discussion as a case study. What is missing in this process is a close reading, comparison, and analysis of the relevant bits of source code in both versions. The inquiry starts with looking into the original *Poizone* version, programmed in ARM Assembly.

The game is written in a little over 8090 lines of code, contained in one single file, and Baerlocher did not use any video game development framework. Although such frameworks existed in the early 1990s for other programming languages, Assembly-based libraries were rare due to the language's complexity and low-level nature, making Assembly-based video game frameworks uncommon. The organisation of the code in this single text file mostly happens through a rich application of comments, some of which are quite skilfully laid out. Since source code is essentially plain text without the possibility to change the typography of individual parts, like titles or marking keywords in bold, programmers often rely on special characters and create typographic layouts through whitespace in order to structure code for easier reading. These kinds of layouts can become quite elaborate and decorative, beyond their initial function for readability. The following snippet in Listing 3 illustrates the variables that help the program remember the kind of relationship the avatar

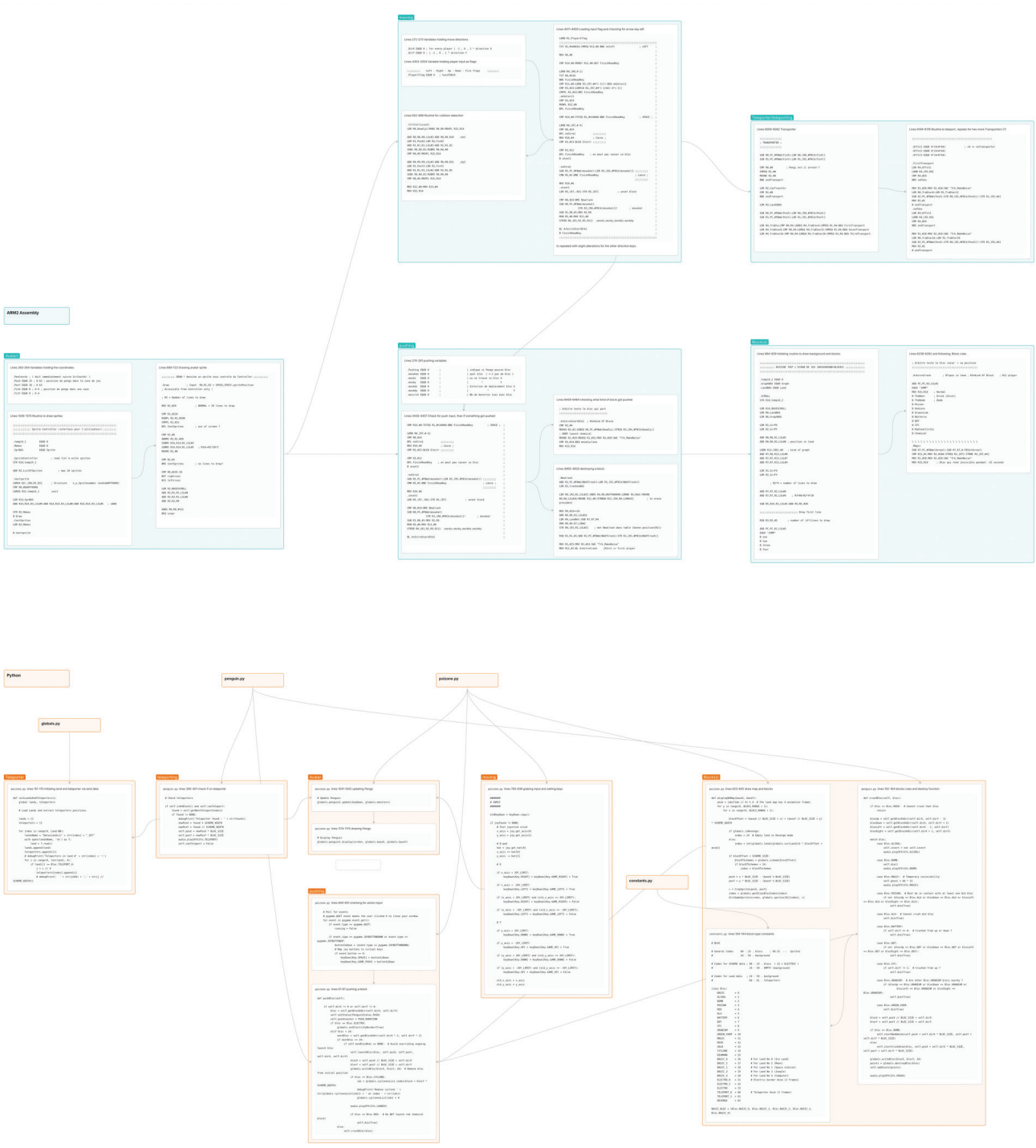

Figure 3. An example of a visual mapping. Above are code snippets and their connections to the original game's codebase; below are the corresponding samples from the ported version. The ludemes visualised here are avatar, moving, blocks, and teleportation. A large-scale version of this mapping is accessible in the accompanying open-access analysis data repository.

```
.DirX    EQUD 0       ; for every player | -1 , 0 , 1    * direction X
.DirY    EQUD 0       ;                  | -1 , 0 , 1    * direction Y
.PengDir EQUD 0       ;                  | deltaSprite du Pengo
.Pushing EQUD 0       ;                  | indique si Pengo pousse bloc
.movwhat EQUD 0       ;                  | quel bloc  ( =-1 pas de bloc )
.movbx   EQUD 0       ;                  | ou se trouve ce bloc X
.movby   EQUD 0       ;                  |         "            Y
.movbdx  EQUD 0       ;                  | direction de deplacement bloc X
.movbdy  EQUD 0       ;                  |             " Y
.movcrsh EQUD 0       ;                  | Nb de monstres tues avec bloc
.Deadly  EQUD 0       ;                  | phase d'agonie de Pengo
```

Listing 3. Example of comments and variables in the original game's assembly code.

has with a given block. Did Zozo push it, and from which direction? These relationships, defined by the programmer, save values in memory and help the program understand what the player intended to do.

Given the amount of code devoted to handling player input and moving Zozo accordingly, compared to other functionality Baerlocher needed to program, this simple task seems rather complex. The difficulty of handling such a common thing arises from the programming language's need to directly interact with the input and computing hardware, namely, joysticks, keyboards, and the processor. Assembly has no supportive layer in between to address this aspect, resulting in a lot of code that does not align with the direct player experience or the developer's intentions. Instead, this simple interaction functionality adds lines of code and complexity for managing hardware states. Baerlocher had to go to great lengths to correctly infer what the player intended to do and then convey that directly to hardware memory to remember and act on those choices.

The excerpt in Table 5 demonstrates how moving the *avatar* on screen in response to player input emerges from the interplay of multiple code components. These are not organised into one single block of code that corresponds neatly to the moving ludeme. Although they appear close to each other in relation to the complete code file, this particular ludeme is scattered throughout the codebase, with relevant parts in different sections and scopes. Some of this code relates directly to how the developer intended the game to be played – how they designed it. Another large portion of the code had to be written to manage the game state and communicate with the hardware.

These ludemes of *moving Zozo* can be compared to the code in the 2023 port of *Poizone*. This version of the game is written in Python, a general-purpose language designed to be easier for

Lines	Commentary
4224–4250	*Checking if the player has a joystick plugged into the computer and is using it.*
4284–4305	*Reading the input of the joystick.*
4325–4338	*Defining which keyboard keys would be used for which of Zozo's actions.*
4353–4356	*Initiating the player flag. This variable is used to track which keys the player pressed.*
4379–4419	*The '.MovePengi1' routine prepares the game to move Zozo by putting all the variables in place.*
4411–4412	*Preparing the player flag before checking which keys were pressed.*
4412–4459	*These roughly 50 lines of code in the '.LEFT" routine were copied and minimally modified for all four directions Zozo can move. They check whether Zozo can move in that direction as well as whether the penguin is colliding with a monster. There is also a check for whether there is a block in that direction and how we interact with it.*

Table 5. List of the lines of code that relate to the moving around ludeme.

humans to read, and incorporates modern programming concepts. One of these is better organisation of the codebase through object classes, which is the code equivalent of real-world expressions, as well as moving code that belongs together into separate files. For the ported version of the game, Baerlocher built on *pygame*, a Python framework that helps in developing video games. The framework does most of the heavy lifting for tracking the game state and communicating with the hardware. Counting the entirety of lines of code in all of *Poizone*'s Python files adds up to roughly 3200 lines. The code also includes comments, as shown in Listing 4, but this time they are rather brief and used only where the code does not speak for itself.

```
def initPenguin():
 global penguin1

 # Create player's penguin
 penguin1 = penguin.Penguin()
```

Listing 4. Example of the commenting style in the ported game's Python code.

The fundamental idea behind modularity resides in the idea that certain software operations do not need to be written several times. Instead, they can be implemented once and then reused in other parts of the software. Different programming languages offer various structures to organise the code in a modular fashion. In the ported version's code, this aspect is evidenced by its file structure. The file *poizone.py* contains the main game code, for example, while *globals.py* hosts snippets of code that need to be used in other files, and files named like *penguin.py* and *monsters.py* handle everything related to these player and non-player characters. This results in game-related functionality involving fewer lines of code, but it is often spread across several files. Splitting and organising the codebase into separate routines and files helps the developer create more reusable, easier-to-maintain code. It also highlights different ways of conceptualising ludemes and interactions. Compared to the game's original code, reading player input is far less complicated, and Baerlocher was able to lean on *pygame*'s offerings.

Following the outline in Table 6, the penguin object class's update function on lines 279–479 handles all the logic related to moving, including checks for whether we are walking or pushing, whether we are blocked by an obstacle or by the level border, and whether we have run into one of the monsters, among other things. These 203 lines of code are among the largest routines in the ported version, indicating that much of the game logic could not be handled by the framework and was instead tied to the developer's game design. Having broken down the video game's source code through play analysis, building a ludemic vocabulary, distant reading and mapping the territory, and finally a close reading, this finding offers a valuable insight. Having the answers to the question of how the program knows whether Zozo can move in a specific direction, as implemented in the penguin class, enables a discussion of the programmer's perspective and his theory of the software he was writing. Three positions are of relevance to this discussion: the difference in ludeme implementation, the developer's perspective on game and play, and, to some extent, the player's experience.

Discussion

Although the two games appear functionally identical to players, they represent completely different computational materialities and technical origins. The original 1991 version exists as raw Assembly code speaking directly to hardware, while the contemporary port operates through layers of

File	Lines	Commentary
poizone.py	*759–772*	*Checking if a joystick or joypad is connected and remembering that relation in the 'joy' variable.*
poizone.py	*799–807*	*Asking for the current state of the input device. This simply saves the directions the player pressed in 'keyDown'.*
poizone.py	*1041*	*Player intention is fed into the avatar's 'update' function.*
penguin.py	*279–479*	*Handling all the logic that is related to moving.*
poizone.py	*1175*	*Re-rendering Zozo on screen.*

Table 6. List of the lines of code that relate to moving Zozo, but this time in the game's ported version.

abstraction provided by Python and the *pygame* framework. Every aspect has been fundamentally reconceptualised, from low-level hardware manipulations to high-level, object-oriented structures. The very substance of the game's existence has transformed. Where once it lived as precise processor instructions that managed memory and registers, it now exists as interpreted scripts that leverage modern libraries. This situation raises profound questions about the identities of born-digital objects. Akin to the 'Ship of Theseus' paradox, one might question whether this is still the same game after exchanging all its constituent parts. The ludemes persist in identical form for players, yet their technical substrate has been completely reconstituted, revealing the complex relationship between a game's experiential essence and its material computational reality.

The code not only differs in programming language or hardware platform, but also in the experience the developer brings to the table, which is especially visible in how the code is organised. In the original game, the entire codebase is in a single file, adhering to the imperative programming paradigm. The ported game has its code neatly arranged across different files in accordance with mental models or functionality. There is also the ported game's dependence on the *pygame* framework. Abstracting some core functionality of the game lays bare which code manages game state and supportive functions, and which code is of a ludemic nature. While implementing *moving Zozo around* in the 1991 version, Assembly code involved talking to and managing hardware, especially memory; the same task was now largely handled by the framework. This shift highlights how certain ludemes are so fundamental that they are naturalised into common knowledge bases like the *pygame* framework.

Deepening this specific thread enables a first foray into what otherwise difficult-to-access insights the source code can offer us. Implementing *moving Zozo around* includes checking whether the penguin will collide with another object or character. The game needs to include an authoritative institution capable of determining whether such an event has occurred and directing the involved parties to act accordingly. The program needs to ask, for example, whether the penguin collided with a monster, and if so, knock out Zozo. Interestingly, the code in the ported game was not organised around this question. The Python code is built around mental concepts, such as the main game code, monsters, and the penguin, and it is organised into files and programmatic structures to mirror these concepts. In the ported game's code, the specific functionality to check if Zozo is colliding is not handled by the main game code, but is attached to the penguin class, which represents the mental model of the avatar. The question the program hence asks is: 'Do I, the penguin, collide with a monster?'

This aspect is a valuable insight into how the developer assumes the player's perspective during development and how this is, in turn, unintentionally encoded in the video game's structures. Pierre-Yves Hurel outlines this process in his research on how amateur video game developers hone their tastes by oscillating between playing games and then recreating the elements they like.[58] Following Naur and regarding the study of the source code, this observation indicates that the programmer is developing several theories in parallel. The primary theory concerns the knowledge about the software coming into being,[59] and entangled with it, a secondary theoretical system emerges, which concerns itself with ideas and intentions about gameplay.

The questions raised by the 'Ship of Theseus' in regard to a plurality of theories at play force a confrontation with what authenticity means for born-digital objects. The paradox traditionally offers no singular answer but reveals that *sameness* itself is philosophically underdetermined. Does the game's identity reside in its ludic function and the experience of play, its materiality of code and hardware, or some irreducible combination of both? Further, it can be argued that authenticity in born-digital objects is fundamentally plural. The original 1991 version maintains authenticity as a historical artefact that embodies the material constraints and possibilities of its era, while the contemporary port achieves authenticity through the faithful preservation of the ludic system and design intentions. In other words, born-digital authenticity is not about material fidelity, but about what aspects of the original we deem essential to preserve. The answers depend on the epistemological commitments regarding what games fundamentally are and how their preservation is assembled.[60]

To finalise the discussion, there is a recognition of code working as a translation mechanism. It is not merely converting human intention into machine instruction but mediating between different technological epochs, programming paradigms, and cultural moments.[61] The memetic persistence of *moving around* or *pushing* ludemes across entirely different technical implementations demonstrates how code serves as a vessel for cultural transmission, carrying forward play patterns and interaction vocabularies that transcend their material substrates.

Conclusion

The digital humanities involve not only large-scale computational methods or the digitisation of old sources, but also a fundamental critical approach to all things digital. What is often lost in this discipline's perspective is the digital technologies' relationship to media and materiality. The analysis presented in this article demonstrates how digital humanities can move beyond algorithm-based approaches toward more fundamental engagement with computational materiality.[62] Rather than treating code as a black box or reducing it to surface-level effects, this methodological approach enables researchers to examine how cultural meanings become embedded in technical structures and how these structures in turn shape cultural possibilities. Baerlocher's inhabiting of the player perspective within the penguin's object class regarding collision detection exemplifies how cultural imagination becomes encoded in technical architectures. Such findings emerge only through the synthesis of cultural interpretation and technical investigation, demonstrating the inadequacy of purely formalist or purely cultural approaches to understanding computational media. Digital humanities must develop methodologies capable of reading these technical layers as cultural documents that deserve preservation and interpretation.[63]

Going further, this case outlines how the encoded structures in source code are worthy objects of investigation. In programming, meaning and message do not exist solely at the level of words or lines of code but emerge from deep-rooted structures dispersed throughout large codebases. The presented framework opens a pathway to meaningful readings of these structures by showing their relationality to developer intentions and player phenomenology. Further, the discussed case outlines how historical source code studies must attend to more than surface readings. Source code emerges from this analysis as a unique form of cultural memory that preserves not only functional specifications but also the cognitive and social contexts of its creation. The comparison between

Poizone's two versions underlined how code carries forward ludemic concepts across technological discontinuities while simultaneously documenting the evolution of programming habits. The original assembly code's direct hardware management and the port's framework dependencies mark distinct moments in the historical relationship between programmer and machine, between creative vision and technical constraint.

Finally, this study opens a path for further inquiries and validations of structural deep readings in critical code studies. The magical moment of translation between developer intention and player experience, mediated through the complex technical and cultural work of programming, emerges as a fundamental site for understanding how meaning is made and transmitted in computational media. This moment, preserved in source code and reconstructed through ludemic analysis, offers media archaeology access to the cultural work performed by technical systems and the technical work performed by cultural imagination. Through this synthesis of technical and cultural analysis, interactive media can be understood as a participant in broader cultural transmission while simultaneously creating new possibilities for meaning-making, memory, and imagination. The source code of games like *Poizone* thus becomes not merely a functional specification but a cultural document, preserving, in technical form, the visions, constraints, and possibilities that shaped particular historical moments in human–computer interaction.

Acknowledgments

The case presented within this article is contextualised by work I was fortunate to accomplish with my dear colleague Pierre-Yves Hurel. In addition to distant and close readings of the source code and analyses of the paratextual material in Baerlocher's personal archive, we conducted oral history interviews with all the actors involved in developing and publishing the game. This work was done within the framework of Confoederatio Ludens, a research project investigating the history of video games in Switzerland. We are still able to study *Poizone* thanks to Robin François and his colleagues from the Swiss Video Game Archivists,[64] who helped to preserve the original video game and its source code. This process, in turn, inspired the creation of a digital archive by Paolo Baerlocher, of equal importance to my research.

JOURNAL FOR MEDIA HISTORY

Declaration

The author declares no competing interests and was solely responsible for conceiving, designing, analysing, and writing this manuscript. A polished excerpt of the presented analysis, including a high-resolution version of the ludeme-in-code mapping, is publicly available.[65]

Endnotes

1. Pierre-Yves Hurel and Adrian Demleitner, "Paolo Baerlocher, from Ticino to Paris: The Demoscene as a Game Engine," *ROMchip* 7, no. 2 (2025), https://romchip.org/index.php/romchip-journal/article/view/225.
2. Peter Naur, "Programming as Theory Building," *Microprocessing and Microprogramming* 15, no. 5 (1985): 253–61, https://doi.org/10.1016/0165-6074(85)90032-8.
3. David M. Berry, *The Philosophy of Software: Code and Mediation in the Digital Age* (Palgrave Macmillan, 2015).
4. Mark C. Marino, *Critical Code Studies* (MIT Press, 2020).
5. Daniel M. Feige, *Computerspiele: eine Ästhetik* (Suhrkamp, 2015).
6. Moritz Feichtinger, "From Source-Criticism to System-Criticism, Born Digital Objects, Forensic Methods, and Digital Literacy for All," *Zenodo*, September 13, 2024, https://doi.org/10.5281/zenodo.13907816.
7. Marino, *Critical Code Studies*; Mark C. Marino and Jeremy Douglass, "Situating Critical Code Studies in the Digital Humanities," *Digital Humanities Quarterly* 17, no. 2 (2023), https://www.digitalhumanities.org/dhq/vol/17/2/000713/000713.html.
8. Markus Krajewski, "Kulturtechnik Programmieren. Quellcode kritisieren. Drei Beispielszenarien," in *Quellcodekritik: zur Philologie von Algorithmen*, ed. Hannes Bajohr and Markus Krajewski (August Verlag, 2024).
9. Naur, "Programming as Theory Building," 253.
10. Paolo Baerlocher, *Poizone*, Eterna, released 1991.
11. See "Kittler's Code" in Marino, *Critical Code Studies*, 161.
12. Marino, *Critical Code Studies*, 166.
13. Naur, "Programming as Theory Building," 255.
14. Gerald M. Weinberg, *The Psychology of Computer Programming* (Van Nostrand Reinhold, 1971).
15. Jaroslav Švelch, *Gaming the Iron Curtain – How Teenagers and Amateurs in Communist Czechoslovakia Claimed the Medium of Computer Games*, Game Histories (MIT Press, 2018).
16. Melanie Swalwell, *Homebrew Gaming and the Beginnings of Vernacular Digitality* (MIT Press, 2021).
17. Daniel Botz, *Kunst, Code und Maschine: Die Ästhetik der Computer-Demoszene* (transcript Verlag, 2014).
18. Annette Vee, "BASIC FTBALL and Computer Programming for All," *Digital Humanities Quarterly* 17, no. 2 (2023), https://www.digitalhumanities.org/dhq/vol/17/2/000696/000696.html.
19. Vee, "BASIC FTBALL and Computer Programming for All."

20. Friedrich Kittler, "Code oder wie sich etwas anders schreiben lässt," in *Cultural Studies*, vol. 18, ed. Karin Bruns and Ramón Reichert (transcript Verlag, 2007), https://doi.org/10.14361/9783839403396-008.
21. Mark C. Marino, "Critical Code Studies," *Electronic Book Review*, December 4, 2006, https://electronicbookreview.com/essay/critical-code-studies/.
22. Ea Christina Willumsen, "Source Code and Formal Analysis: A Hermeneutic Reading of Passage," *Proceedings of DiGRA/FDG 2016 Conference*, January 1, 2016, https://doi.org/10.26503/dl.v2016i1.764.
23. Marino and Douglass, "Situating Critical Code Studies."
24. Berry, *The Philosophy of Software*, 131.
25. Sonia Fizek, "Through the Ludic Glass: Making Sense of Video Games as Algorithmic Spectacles," *Game Studies* 22, no. 2 (2022), https://gamestudies.org/2202/articles/gap_fizek.
26. Jussi Parikka, *Operational Images: From the Visual to the Invisual* (University of Minnesota Press, 2023).
27. Stefan Höltgen, "Humanities of the Digital: Philologische Perspektiven auf Source Codes als Beitrag einer computerarchäologischen Knowledge Preservation," in *Digitale Schriftlichkeit: Programmieren, Prozessieren und Codieren von Schrift*, ed. Martin Bartelmus and Alexander Nebrig (transcript Verlag, 2024), 207–30, https://doi.org/10.14361/9783839468135.
28. Nis Bojin, "Ludemes and the Linguistic Turn," *Proceedings of the International Academic Conference on the Future of Game Design and Technology* (New York, NY), Futureplay '10, May 6, 2010, 25–32, https://doi.org/10.1145/1920778.1920783; Cameron Browne, "Everything's a Ludeme Well, Almost Everything," *XXIII BOARD GAME STUDIES COLLOQUIUM – The Evolutions of Board Games*, April 13, 2021, https://sorbonne-paris-nord.hal.science/hal-03737317.
29. Pierre-Yves Hurel, "L'expérience de création de jeux vidéo en amateur - Travailler son goût pour l'incertitude" (Université de Liège, 2020), https://orbi.uliege.be/handle/2268/247377.
30. Damien Hansen, *Parler le jeu vidéo: Le ludème comme unité minimale d'une grammaire vidéoludique ?* (Presses universitaires de Liège, 2023), https://books.openedition.org/pulg/18941.
31. Ian Bogost, *Persuasive Games: The Expressive Power of Videogames* (MIT Press, 2007), https://doi.org/10.7551/mitpress/5334.001.0001.
32. Susan Leigh Star, "This Is Not a Boundary Object: Reflections on the Origin of a Concept," *Science, Technology, & Human Values* 35, no. 5 (2010): 601–17, https://doi.org/10.1177/0162243910377624.
33. Feige, *Computerspiele*, 81–132.
34. Hurel, "L'expérience de création de jeux vidéo."
35. Braben David, *Zarch*, Superior Software Ltd., released 1987.

36. Hurel and Demleitner, "Paolo Baerlocher, from Ticino to Paris."
37. Friedrich A. Kittler, "There Is No Software," in *The Truth of the Technological World* (Stanford University Press, 2014), 219–29, https://doi.org/10.1515/9780804792622-016.
38. Kittler, "There Is No Software," 223.
39. Wolfgang Ernst, "Time, Temperature and Its Informational Turn," *Culture Machine* 17, no. Thermal Objects (2018), https://culturemachine.net/vol-17-thermal-objects/time-temperature/.
40. Jussi Parikka, *What Is Media Archaeology?* (Polity Press, 2012); John Durham Peters, *The Marvelous Clouds: Toward a Philosophy of Elemental Media* (University of Chicago Press, 2015).
41. Hansen, *Parler le jeu vidéo*; Rudolf Thomas Inderst, "Close Playing," in *Handbuch Digitale Medien und Methoden*, ed. Sven Stollfuß et al. (Springer Fachmedien Wiesbaden, 2023), 1–15, https://doi.org/10.1007/978-3-658-36629-2_33-1.
42. Jonas Linderoth, "Beyond the Digital Divide: An Ecological Approach to Gameplay," *Proceedings of DiGRA 2011 Conference: Think Design Play*, January 1, 2011, https://doi.org/10.26503/dl.v2011i1.543.
43. Till A. Heilmann, "Wie liest man 100'000 Zeilen Code?" in *Quellcodekritik: zur Philologie von Algorithmen*, Erste Auflage, ed. Hannes Bajohr and Markus Krajewski (August Verlag, 2024).
44. Johannes Pause and Niels-Oliver Walkowski, "Scalable Viewing," *Zeitschrift für Medienwissenschaft, ZfM Online, Open-Media-Studies-Blog*, March 11, 2019, https://zfmedienwissenschaft.de/online/scalable-viewing.
45. *Soko-Ban*, Thinking Rabbit Inc., released 1984.
46. *Pengo*, SEGA Enterprises Ltd., released 1982.
47. Time as resource and mechanic, mirroring the material operations of the processor, a silent nod to Ernst Wolfgang.
48. Heilmann, "Wie liest man 100'000 Zeilen Code?"
49. Lev Manovitch, *Cultural Analytics* (MIT Press, 2020).
50. Franco Moretti, *Distant Reading* (Verso, 2013); Taylor Arnold and Lauren Tilton, *Distant Viewing: Computational Exploration of Digital Images* (MIT Press, 2023).
51. Naur, "Programming as Theory Building."
52. Janna Joceli Cavalcanti de Omena, "Digital Methods and Technicity-of-the-Mediums. From Regimes of Functioning to Digital Research" (PhD thesis, Universidade Nova de Lisboa, 2021), https://run.unl.pt/handle/10362/127961.
53. Heilmann, "Wie liest man 100'000 Zeilen Code?"
54. Stephen Ramsay, ed., *Reading Machines: Toward an Algorithmic Criticism* (University of Illinois Press, 2011); Moretti, *Distant Reading*; Arnold and Tilton, *Distant Viewing*.

55. George P. Landow, *Hypertext 3.0* (Johns Hopkins University Press, 2006), https://doi.org/10.56021/9780801882562.
56. Espen Aarseth, *Cybertext: Perspectives on Ergodic Literature* (Johns Hopkins University Press, 1997).
57. Adrian Demleitner et al., "Choose Your Own Adventures in Source Code," in *.D64: Scientific Explorations of Computer Game History Within the Digital Humanities – A C64 Disk Book* (Weimar, 2024).
58. Hurel, "L'expérience de création de jeux vidéo."
59. Naur, "Programming as Theory Building," 253.
60. Dany Guay-Bélanger, "Assembling Auras: Towards a Methodology for the Preservation and Study of Video Games as Cultural Heritage Artefacts," *Games and Culture* 17, no. 5 (2022): 659–78, https://doi.org/10.1177/15554120211020381.
61. Marino, "Critical Code Studies," 47.
62. Marino and Douglass, "Situating Critical Code Studies."
63. Adrian Demleitner, "Preserving Situated Practices," *Born-Digital Collections, Archives and Memory (University of London),* April 3, 2025, https://doi.org/10.5281/zenodo.15282539.
64. Éléonore Bernard et al., *Pixelvetica - Sauvegarder le jeu vidéo suisse: État des lieux de la préservation du jeu vidéo en Suisse et dans le monde*, Memoriav and Pixelvetica, August 1, 2022.
65. Adrian Demleitner, "Poizone Analysis Material," *Zenodo*, May 7, 2026, https://doi.org/10.5281/zenodo.20067738.

Biography

Adrian Demleitner is a researcher at the HKB Institute of Design Research in Switzerland, specialising in the intersection of critical code analysis and video game studies. As part of Confoederatio Ludens, he is writing his Digital Humanities dissertation at the University of Bern in Switzerland. His research explores the intricate relationship between code and semiotics in video games, emphasising programming as a meaningful human activity.

He obtained a BA in Process Design from Hyperwerk in Basel and an MA in Design Research from the MA Design programme at HKB in Bern. His recent positions include working as a scientific software programmer on a participatory image archive research project at the University of Basel (Digital Humanities), and as a junior researcher on an electronic waste project at the Institute of Experimental Design and Media Cultures (IXDM). ORCID: 0000-0001-9918-7300

TMG Journal for Media History

Volume 29 No (1)/2026

DOI

https://doi.org/10.18146/tmg.943

PUBLISHER

Netherlands Institute for Sound & Vision