King's College London

King's Research Portal

Document Version
Peer reviewed version

Link to publication record in King's Research Portal

Citation for published version (APA):
Cook, M. (in press). The Art of Programming: Challenges in Generating Code for Creative Applications. In *AIWare 2024*

Citing this paper
Please note that where the full-text provided on King's Research Portal is the Author Accepted Manuscript or Post-Print version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version for pagination, volume/issue, and date of publication details. And where the final published version is provided on the Research Portal, if citing you are again advised to check the publisher's website for any subsequent corrections.

General rights
Copyright and moral rights for the publications made accessible in the Research Portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognize and abide by the legal requirements associated with these rights.

•Users may download and print one copy of any publication from the Research Portal for the purpose of private study or research.
•You may not further distribute the material or use it for any profit-making activity or commercial gain
•You may freely distribute the URL identifying the publication in the Research Portal

Take down policy
If you believe that this document breaches copyright please contact librarypure@kcl.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 10. Jan. 2025

The Art of Programming: Challenges in Generating Code for Creative Applications

Michael Cook
King's College London
London, UK
mike@possibilityspace.org

ABSTRACT

Programming has been a key tool for artists and other creatives for decades, and the creative use of programming presents unique challenges, opportunities and perspectives for researchers considering how AI can be used to support coding more generally. In this paper we aim to motivate researchers to look deeper into some of these areas, by highlighting some interesting uses of programming in creative practices, suggesting new research questions posed by these spaces, and briefly raising important issues that work in this area may face.

CCS CONCEPTS

• Applied computing → Arts and humanities; **• Computing methodologies → Artificial intelligence**.

KEYWORDS

computational creativity, generative systems, code generation

ACM Reference Format:
Michael Cook. 2024. The Art of Programming: Challenges in Generating Code for Creative Applications. In *Proceedings of the 1st ACM International Conference on AI-Powered Software (AIware '24), July 15–16, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3664646.3664774

1 INTRODUCTION

Creative AI is currently a popular area of study among AI researchers, with many of the best-known public-facing AI systems currently focused on creative tasks, such as image synthesis [19] or music composition [8]. Such systems raise a lot of important ethical questions about how their input data is sourced [22], what impact they have on the workers in the industries they target [2], and what the long-term impact they might have on the creative industries. They also have unexamined issues relating to how they affect the creativity of the user. Although many of these creative AI systems are promoted as making creativity 'more accessible', it's unclear whether they have *adverse* effects on their users in terms of restricting their ability to innovate, ideate and create.

Programming has an unusual position in the creative industries. While many creative media rely on programming as an integral skill (particularly game design and development, but also many forms of digital art, music production, and film), programming itself is not generally viewed as a creative pursuit. There are many contributing factors to this: the politicisation of STEM has led to backlash against technological roles in some areas of the arts, for example, and many people who do not have a background in programming may view it as a 'technical' rather than 'creative' skill and discipline. Nevertheless, many creative communities exist built on a foundation of coding as a creative activity: for example, the generative art community, NaNoGenMo [15], the livecoding music (and art) community [3], and more.

Studies looking at the usefulness of tools such as GitHub Copilot appear to largely focus on their use in industrial-scale software engineering on large projects. GitHub do not explicitly share the industries or domains that their survey participants work in, however we can intuit some things from the information they *do* release. For example, in one survey they ask 2,000 participants which languages they program in as part of a study of Copilot usage, in which the most popular languages for game development work (C++ and C#) barely feature [25]. In another survey, the test exercise given to participants is to write an HTTP server in Javascript [14]. A third survey focused only on companies with more than 1000 employees – this would exclude every creative industries studio in the UK, including the entire games sector [23].

This raises the concern that creative tasks, and creative coding as a discipline, is understudied for AIware, which means missing out on important research questions and use-cases, but also risks repeating the problems inherent in image synthesis and other creative tools: namely, that we repurpose tools not explicitly designed for creative work without giving consideration to how they are designed; what their strengths and weaknesses are; and what the needs are of the people who wish to use them. In this paper I identify some areas of potential focus, interesting research questions, and important ethical issues at play that are of particular relevance to creative workers and artists.

2 NOVEL DOMAINS

In this section we give some examples of domains within the creative industries that offer unique problems, opportunities or use-cases for AI research. This is not an exhaustive list - instead it is intended to highlight some interesting landmarks in the creative programming landscape, to give readers a sense of what vast new application domains, creative practices and communities exist.

2.1 Livecoding

Livecoding is the act of creating music or art using code as an act of performance. As the name suggests, it is usually done live in front of

Permission to make digital or hard copies of part or all 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 third-party components of this work must be honored. For all other uses, contact the owner/author(s).
AIware '24, July 15–16, 2024, Porto de Galinhas, Brazil
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0685-1/24/07
https://doi.org/10.1145/3664646.3664774

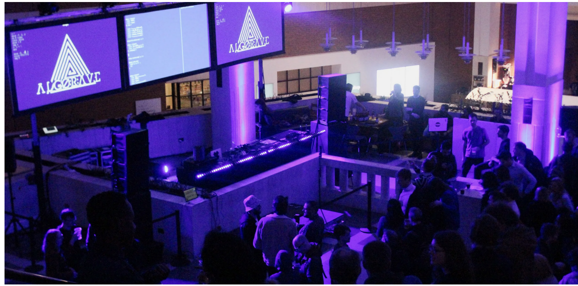

Figure 1: An algorave. Photo credit: Coral Manton.

an audience, using specialised libraries or programming languages, such as TidalCycles [17] or Gibber [20]. Livecoding tools emphasise rapid execution of code, often in a modular fashion so that segments of a codebase can be recompiled and rerun without affecting other ongoing computation. They also benefit from immediate feedback and low barriers to entry. The act of programming is often part of the performance – the code is usually visible to the audience, who enjoy the aesthetic of programming even if they may not understand what is being written.

Livecoding represents an exciting application domain for AI-synthesised code, as it is already seen as an activity that has considerable overlap with generative systems and programming languages. It is also a programming context with unique constraints on it – the live nature of the performance, the fact that the act of programming is observed by the audience, the immediate and direct experience of the output all make livecoding somewhat unique as a programming application domain. This presents novel challenges that researchers may not have encountered elsewhere. This also has a natural overlap with computer science education, or other contexts where programming is demonstrated live.

2.2 Modding

Modding in the context of videogames describes the act of creating a set of files that can be loaded on top of an existing game to change it in some way. Mods can have a variety of purposes, including adding accessibility options; making the game easier or harder; adding new content or restoring cut content; or fixing older games to run on modern hardware. In past decades mods were mostly 'homebrew' hacks that overwrote or otherwise injected themselves into pre-compiled games, however increasingly often now game developers add dedicated support for modding through the distribution of custom development tools, modding languages and APIs. Modding is a vital area of creativity and ideation for the games industry [1]. Some of the most famous games, and even entire genres, started originally as fan-made modifications of existing games [18].

Mods represent an interesting application area for AI-assisted software development. For one, modding is often seen as a route into game development, and the opportunity to build on an existing codebase means that it is popular with designers who do not have extensive programming or software engineering experience. Modding tools often use a domain-specific language or a simpler scripting language to interact with an API built onto the game, written in a more complex language such as C++ [11]. For DSLs this might mean that traditional approaches to AI-driven code synthesis do not apply, and new approaches might be required that work with different kinds of language, and smaller amounts of training examples. It might also require richer explanations, including those capable of explaining how code written in one language (such as the DSL) relates to code written in another (the game itself). Mods are often built on very large games that take a long time to run or test, which means that common approaches to testing codebases, such as writing unit tests or running the code in situ, are not applicable.

2.3 Mechanical Ideation for Games

Game mechanics are atomic systems within games that are composed into rules and game logic. The term 'mechanic' has many definitions within the literature, but broadly speaking we can define it as the smallest unit of game systems design. Jumping in *Super Mario Bros.* might be considered a mechanic; catching insects with a net in *Animal Crossing* is a mechanic; swapping two tiles to make a row of three in *Candy Crush* is a mechanic. Mechanical ideation is an appealing problem for AI research as game mechanics are often quite short and simple to define, but have wide-reaching and complex systemic effects. Past research has looked into inventing and discovering game mechanics using computational evolution [7] as well as machine learning [13].

A key challenge for automated game design research in this space is creating mechanics with broad potential and novelty. Small changes to a well-known design can be enough to build an entire game upon, but identifying which mechanics have potential and which are likely to be shallow is very difficult. Different types of game also have different needs from new mechanics. Action-oriented or very dynamic games often favour mechanics that are playful, expressive or that have a high skill ceiling. By contrast, so-called 'systems-driven' games such as roguelikes prefer mechanics which tightly couple themselves with existing game systems, extending the possibilities much like a new keyword in a programming language. Understanding, and more important *evaluating*, new mechanics with this in mind is a challenging task.

2.4 Generative Art

The use of computers to create art is many decades old, with the Victoria and Albert museum documenting collaborations between artists and scientists in the 1960s to create pioneering works of what we now call generative or algorithmic art [24]. Nowadays there are many, many tools for creating art through programming, and many unconventional and homebrewed ways to do it too. Processing and p5.js are two examples of popular digital art tools that use common programming languages (like Java, Javascript and Python), have accessible APIs, run on the web and are extremely lightweight [9]. The *demoscene* can also be seen as a kind of generative art, where programmers write extremely compressed programs which, when executed, create elaborate and beautiful artworks, pieces of music or animations [21]. Generative art has strong parallels to work in computational creativity [5] where AI techniques were leveraged to create visual art (among other things), although computational creativity practitioners often de-emphasised themselves as creators or artists in the process.

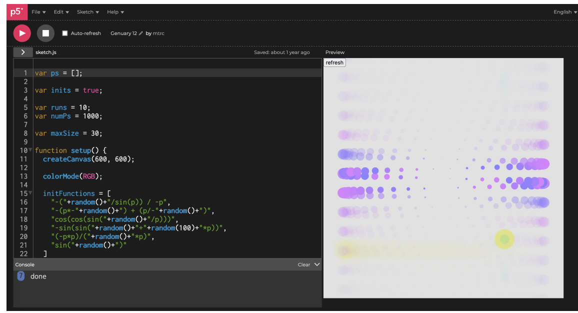

Figure 2: A screenshot of the p5.js interface, showing the code (left) and output (right).

Generative art is an interesting domain for coding assistance precisely because it is so broad, and its users work in ways that are often intuitive and exploratory. Some artists describe their programming IDE as a sketchbook, and tools like Processing emphasise this feeling by always displaying the output of a script side-by-side as new code is being written (Processing also refers to files as 'sketches'). This more rapid, iterative and fluid approach to programming might enable new perspectives on how AI code synthesis is deployed. For example, rather than waiting to be called upon by the user, or focusing code completion on areas the user is currently writing, an AI code generator might simply add new code to a completely different section of the sketch, without asking or informing the user at all. While this might sound chaotic or completely undesirable in a more ordinary commercial context, this might open up new forms of creative coding for people, emphasising serendipitous connections and surprising discoveries.

3 RESEARCH PROBLEMS

3.1 Evaluation and Visualisation

AI-generated code is usually examined visually before being accepted, at which point it is subject to whatever testing or quality control process the programmer might normally employ for other code. In some of the creative use-cases we have outlined above this is possible, however other use-cases require a considerably different approach to both understanding the impact of a piece of code, and for evaluating its appropriateness. In livecoding, for example, code is usually not tested before being run directly in front of the audience. Livecoders and other performance artists may have different needs to feel confident that a piece of code will not do something harmful or damaging to the performance – this might be as simple as having a second sandbox where they can preview the code, in the same way a mixing deck allows sounds to be previewed on one channel before being output to another.

Evaluation in general is a difficult task, however, because the qualities a creative coder is looking for in a piece of code are difficult to automatically test or objectively measure. Livecoders will often execute a small segment of new code briefly before stopping it and going back to change it. For another example, suppose we are trying to generate a new mechanic for a videogame. We might be able to validate a game mechanic to show it does not violate any constraints or that it passes unit tests; however in order to properly understand the proposed code snippet we would need to play the game ourselves, and ideally would need to adversarially playtest the game with several other people. This may mean that AI assistance for some creative application areas benefits from live or hot-swappable coding environments where new snippets can be immediately compiled and integrated into a running build.

3.2 Interventions and Clarity

The relationship between a programmer and the program is also different in creative contexts. Systems like copilot are called on-demand when a programmer wants to instruct the system to produce some code, while other assistants act like autocomplete, guessing at the code that might be written next. Some creative workflows may not admit this type of intervention however - for example, if the user is in a creative flow they may not want to stop to query a system, or they may find autocompletion works against their own feeling of expression or independence. Creative programmers might prefer to see automated code generate and execute in altenrative windows, providing 'parallel workspaces' they can jump into if they see one which appeals to them [16]. This might provide more of a feeling of curation or exploration which might connect to the creative context better. A turn-taking approach, which is employed by many mixed-initiative creative AI systems, might also support artistic exploration better [13].

In [4] Colton et al emphasise the need for automated code generators working in creative spaces to produce code that is 'human-understandable... [and written] in human-like ways'. Although this paper was written before the advent of LLMs, a recurring point made about LLM coding assistants is that the code they generate can be accurate while also being hard to understand. In commercial, secure or safety-critical environments this is a concern for robustness and reliability reasons. In creative spaces this is not necessarily a downside – a generative artist might only care about the visual effect of the code, not what its internals do, and the mystery could actually enhance the act of creative experimentation in some cases. However, it is also worth recognising that many subcommunities within creative spaces are self-taught programmers who may not be used to reading and understanding complex code written by other people (or systems). As a result, for those that *do* wish to understand the code they are looking at, the need for that code to be easily understandable and clearly documented may actually be higher than for other industry software domains.

3.3 Innovation and Risk

When solving more traditional everyday problems in software engineering, one might argue that we desire the *least* interesting solution possible. If we ask an AI assistant to produce code to sort a list of database entries, or create a dictionary from some table data, we presumably want an algorithm that is straightforward, efficient and boring. In this regard, having foundation models reproduce structures and approaches they have already seen is arguably a desirable feature. For creative applications, however, we are often looking for the opposite. Livecoders and generative artists might be exploring areas of an expressive space that have never been encountered before, and a lot of game design work is predicated on

building systems which are novel. Depending on whether coding assistants are deployed for ideation, code completion or co-creativity, we may want them to be more or less innovative with the kinds of code they are capable of generating. As we mentioned earlier, it will be critical to understand whether AI code synthesis has a negative effect on creativity and expressivity for artists, something which may be difficult to understand without large long-term studies.

Related to this is the issue of risk. Most AI code generation is focused on commercial or scientific contexts where the potential for harm is high. Security flaws, memory leaks, harmful side-effects and other concerns could have catastrophic effects for a company or product if not caught in time. While these concerns certainly exist in some creative domains (videogame development, for example) some creative coders may want to explore ideas which are more dangerous or even approach some of the red lines that LLMs may have internally. For example, famous digital art projects include games which delete files off the player's hard drive whenever they kill an enemy [10], websites which upload and download random files from the internet, or that connect to random open socket connections on the internet. Creative coding projects may well have apparent similarities in structure or function to worms, viruses or other harmful software. This raises issues in whether coding assistance can be easily provided for those users, or whether they will be flagged as false positives and refused support.

4 BROADER CHALLENGES

4.1 Ethics

Code generation via LLMs has come under criticism for many of the reasons LLMs have more widely, both in general terms (such as the large economic and energy cost of training the models) and more specific ones (such as the claims that Copilot may be recreating licensed code or breaching the licenses of some of the code it was trained on). While these issues are not new, they are particularly applicable for the creative industries where there is already considerable criticism of and resistance to the use of AI in other areas, such as art. The AIware community should take these concerns seriously and look at ways these concerns can be addressed before trying to engage deeply with these communities.

However, the nature of creative code tasks may also make it easier to address these issues. Because of the desire for more exploratory, high-temperature solutions that propose more unusual suggestions, it may be possible to train much smaller models on much leaner datasets. This opens up the possibility that larger game developers could train focused models entirely on their own code archives, for example, or that synthetic datasets of code could be generated for training livecoding assistants. While lower-quality and less reliable code completion is unacceptable for safety-critical scenarios or production branches, it is much more acceptable in playful or sketchbook-like scenarios with experienced users.

This still leaves us with the question of who this technology is for. GitHub and Microsoft have gone to great lengths to dissect the notion of productivity, and how Copilot affects this. They express productivity in terms of programmer happiness, and state that their aim is not to replace developers but to support them and let them focus on interesting work. However even they observe that some jobs are at risk [12]. It's important for everyone in this emerging field to consider what their research is for, whom it benefits, and what negative externalities this may bring. I would argue this is crucial whether you are targeting creative domains or otherwise.

4.2 Adapting To Existing Practice

In [6] Cook suggests that game developers might need to adopt different approaches to the design and engineering of game codebases in order to write code that is more amenable to comprehension and extension by AI systems. Software engineering has a certain amount of consistency across industries, yet there exist individual subcultures and traditions in engineering large projects that develop in a particular application area, decade or part of the world. The creative industries, and in particular applications where code is being used as a sketchbook or an experimental medium, rely on quick hacks, rapid evaluation and testing, and fast iteration. Code is often messy and poorly structured, fixes and changes are hacked in to meet deadlines, and mistakes creep in as programmers crunch to meet release dates or other milestones.

A lot of the discourse around coding assistants focuses on one of two scenarios: the first, a casual assistant to a complete novice working on a solo project; the second, a large company with clear pipelines for production, testing, review and planning. A different set of approaches and considerations may be required to properly support the creative industries, who often do not neatly fall into either of these categories. One of the primary reasons that new technology often fails to take hold in the games industry, for example, is that researchers do not properly understand and adapt to the hard requirements that companies working in this sector have, and the way the tool fits into an existing engineering culture.

5 CONCLUSIONS

Programming is a creative endeavour. I would argue this is as true for those writing backend code for a web application as it is for those writing python live on stage at a rave. Yet the needs of some creative coding communities may be very different to those of more traditional commercial engineers, either in technology, workflow, ergonomics or attitude. This poses problems if we develop coding assistants that only have one particular kind of programming in mind; but it also provides exciting opportunities for new research, new ideas and new challenges if we embrace these creative spaces and the people who work within them – and take those lessons and inventions back into other programming cultures, in return.

Speaking from experience, however, it is only possible to work with these communities if we respect them, understand them, and crucially take their concerns and questions seriously. The use of LLMs is not uncontroversial, and while it may be seen as a productivity tool in many industries, it is important to recognise that creative spaces are much more critical of their implementation and use. However this, too, should be viewed as an opportunity rather than a roadblock. These communities are asking us to find different ways forward, new tools, techniques and approaches that not only integrate with their way of programming, but that respect their ideals, their needs and their hopes for future technology. I hope that the AIware community can embrace this exciting and interesting space and apply their ingenuity and passion to it, to find ways forward that everyone is excited by.

REFERENCES

[1] Anna Anthropy. 2012. *Rise of the Videogame Zinesters: How Freaks, Normals, Amateurs, Artists, Dreamers, Drop-outs, Queers, Housewives, and People Like You Are Taking Back an Art Form.* Seven Stories Press.

[2] Aleena Chia. 2022. The artist and the automaton in digital game production. *Convergence* 28, 2 (2022), 389–412. https://doi.org/10.1177/13548565221076434

[3] Nick Collins and Alex McLean. 2014. Algorave: Live Performance of Algorithmic Electronic Dance Music. In *Proceedings of the International Conference on New Interfaces for Musical Expression*. Goldsmiths, University of London, London, United Kingdom, 355–358. https://doi.org/10.5281/zenodo.1178734

[4] Simon Colton, Ed Powley, and Michael Cook. 2018. Investigating and Automating the Creative Act of Software Engineering. In *Proceedings of the International Conference on Computational Creativity*.

[5] Simon Colton and Geraint A. Wiggins. 2012. Computational creativity: the final frontier?. In *Proceedings of the 20th European Conference on Artificial Intelligence*. IOS Press.

[6] Michael Cook. 2020. Software Engineering for Automated Game Design. In *Proceedings of the IEEE Conference on Games*.

[7] Michael Cook, Simon Colton, Azalea Raad, and Jeremy Gow. 2013. Mechanic Miner: Reflection-Driven Game Mechanic Discovery and Level Design. In *Proceedings of the Evo* Conference*.

[8] Prafulla Dhariwal, Heewoo Jun, Christine Payne, Jong Wook Kim, Alec Radford, and Ilya Sutskever. 2020. Jukebox: A Generative Model for Music. arXiv:2005.00341 [eess.AS]

[9] The Processing Foundation. 2001. Processing. https://processing.org/.

[10] Zach Gage. 2009. lose/lose. http://www.stfj.net.

[11] Epic Games. 2023. Verse Language Reference. https://tinyurl.com/verse-lang.

[12] GitHub. 2023. FAQ: Is GitHub Copilot intended to fully automate code generation and replace developers? https://github.com/features/copilot.

[13] Matthew Guzdial, Nicholas Liao, Jonathan Chen, Shao-Yu Chen, Shukan Shah, Vishwa Shah, Joshua Reno, Gillian Smith, and Mark O. Riedl. 2019. Friend, Collaborator, Student, Manager: How Design of an AI-Driven Game Level Editor Affects Creators. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*.

[14] Eirini Kalliamvakou. 2022. Research: quantifying GitHub Copilot's impact on developer productivity and happiness (GitHub blog). https://tinyurl.com/github-happiness.

[15] Darius Kazemi and Hugo van Kemenade. 2013. National Novel Generating Month. https://nanogenmo.github.io/.

[16] Antonios Liapis, Georgios N. Yannakakis, and Julian Togelius. 2013. Sentient Sketchbook: Computer-Aided Game Level Authoring. In *Proceedings of the 8th Conference on the Foundations of Digital Games*.

[17] Alex McLean. 2014. Making programming languages to dance to: live coding with tidal. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling & Design* (Gothenburg, Sweden) *(FARM '14)*. Association for Computing Machinery.

[18] Mike Minotti. 2014. The history of MOBAs: From mod to sensation. https://venturebeat.com/games/the-history-of-mobas-from-mod-to-sensation/.

[19] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. 2021. Zero-Shot Text-to-Image Generation. *CoRR* abs/2102.12092 (2021). arXiv:2102.12092 https://arxiv.org/abs/2102.12092

[20] Charlie Roberts. 2021. gibber.cc. https://gibber.cc/.

[21] Vincent Scheib, Theo Engell-Nielsen, Saku Lehtinen, Eric Haines, and Phil Taylor. 2002. The demo scene. In *ACM SIGGRAPH 2002 Conference Abstracts and Applications*. Association for Computing Machinery.

[22] Christoph Schuhmann, Romain Beaumont, Richard Vencu, Cade W Gordon, Ross Wightman, Mehdi Cherti, Theo Coombes, Aarush Katta, Clayton Mullis, Mitchell Wortsman, Patrick Schramowski, Srivatsa R Kundurthy, Katherine Crowson, Ludwig Schmidt, Robert Kaczmarczyk, and Jenia Jitsev. 2023. LAION-5B: An open large-scale dataset for training next generation image-text models (OpenReview Thread). https://openreview.net/forum?id=M3Y74vmsMcY.

[23] Inbal Shani. 2023. Survey reveals AI's impact on the developer experience (GitHub blog). https://tinyurl.com/github-impact.

[24] Victoria and Albert Museum. 2024. Digital art. https://www.vam.ac.uk/articles/digital-art.

[25] Albert Ziegler, Eirini Kalliamvakou, Shawn Simister, Ganesh Sittampalam, Alice Li, Andrew Rice, Devon Rifkin, and Edward Aftandilian. 2022. Productivity Assessment of Neural Code Completion. In *Proceedings of the 6th Annual Symposium on Machine Programming*.

Received 2024-04-05; accepted 2024-05-04