Opportunities for Approachable Game Development via Program Synthesis

Max Kreminski, Michael Mateas
University of California, Santa Cruz
{mkremins, mmateas}@ucsc.edu

Abstract

Program synthesis techniques can be used to improve the approachability of game development, but much work remains to be done in bridging the gap between program synthesis and game development support. We propose three high-level categories of research that might be undertaken in pursuit of this goal.

Introduction

Digital games are a powerful expressive medium. They can be used to communicate ideas through procedural rhetoric (Bogost 2010); to express deeply personal experiences (Anthropy 2012); and to provoke contemplation of complex phenomena (Rusch 2017).

In each of these cases, the ability of games to deal directly with *systems* is key to their expressive potential. Because games deal in systems, code plays a critical role in implementing their behavior—and games are often developed in general-purpose programming languages that take significant time and effort to learn, even in carefully constructed educational environments. As a result, the difficulty of programming represents a significant barrier to entry for many would-be creators of digital games.

Some game creation tools, such as Game-O-Matic (Treanor et al. 2012), Wevva (Powley et al. 2017), and Germinate (Kreminski et al. 2020), allow users to create certain kinds of games without having to deal with the minutiae of coding in a general-purpose programming language. In some cases, approachable tools have played a major role in making digital games available as a medium of expression to a more diverse set of creators (Harvey 2014). However, these tools achieve their approachability by restricting their users to tiny subsets of the space of all possible games. This allows these tools to present a highly simplified interface for defining game behavior, but massively limits the kinds of gameplay that users can define.

Cook has recently argued (Cook 2020) that research in automated game design (which is closely connected to the development of approachable game creation tools) should move away from narrow predefined game design spaces and instead attempt to produce computational systems that can reason over the much wider range of game behaviors implementable in general-purpose programming languages. At the same time, recent work in *program synthesis* (Gulwani et al. 2017) has shown that computers are capable of generating programs that meet user needs in a variety of contexts, from data science (Drosos et al. 2020) to graphic design (Hempel and Chugh 2016). Though they may make internal use of narrow domain-specific languages, many program synthesizers eventually output code in general-purpose programming languages like those typically used for game development. Consequently, synthesizers can generate programs that demonstrate a wide range of possible behaviors.

This raises an intriguing question: can we use program synthesis techniques to improve the approachability of game development in general-purpose programming languages? We believe the answer is yes, but that there is much work to be done to bridge the gap between program synthesis and game development support. In the remainder of this paper, we briefly propose three high-level categories of research that might be undertaken in pursuit of this goal.

Expressive Synthesis

Many existing program synthesizers aim to generate code that is purely utilitarian: for instance, code that accurately implements a user-intended transformation of spreadsheet data (Gulwani 2011). But from a software engineering perspective, game code stands out as unusual, because the code itself is often *expressive* in nature. Code that implements game mechanics must trigger visual or auditory feedback, to help the player understand what is happening in the game world when they take certain actions. Code that implements non-player character behavior typically aims to give characters a distinctive personality, which the player can discover through interaction. And code that interacts with or ties together a game's operational logics (Osborn, Wardrip-Fruin, and Mateas 2017) can be interpreted as making arguments through procedural rhetoric—for instance, assertions like "NPCs sometimes give you items when interacted with", the details of whose implementation can play a substantial role in shaping the player's attitude toward NPC interactions, the items they receive, and so on. Consequently, attempts to apply program synthesis to game development must be aware of the expressive dimension of code.

The Gemini game generator (Summerville et al. 2018) implements a limited form of expressive synthesis by allowing users to specify their rhetorical intent for a game as an answer set program, which constrains a generative space of arcade games defined by the Cygnus game description language. Generated Cygnus games are then translated directly into executable JavaScript code. However, Gemini implements a relatively limited set of operational logics; can only be used to create single-screen arcade games; uses only a small set of simple, hardcoded control schemes and entity behaviors; and takes relatively little advantage of potential programmatic channels for player feedback (such as procedural visual and audio feedback), resulting in games that are often hard for players to interpret (Osborn et al. 2019). A richer implementation of expressive synthesis could use the full capabilities of a general-purpose programming language to diverge from the specific, narrow space of games that the Cygnus GDL defines.

Also worth mentioning here is the use of program synthesis to generate boss enemy behaviors, as described by Butler, Siu, and Zook (2017). In this case, the synthesized programs are expressive in that they yield different player experiences from one boss encounter to the next—but there is no way to indicate different expressive *intents* to the synthesizer, as a game designer might often want to do (e.g., to create one boss that feels courageous and another that feels cowardly). Nevertheless, concepts like the `down-only-number` introduced as a specialized numeric type in this system may prove useful in the way that they capture relatively low-level aspects of expressive intent: a more sophisticated program synthesizer that generates boss behavior might use a `down-only-number` for boss health by default but also leave open the possibility of creating bosses that can heal themselves at certain points, indicating this to the rest of the system via the constraints placed on the health variable in a form of *type-directed* program synthesis.

Reflective Synthesis

One recurring issue in program synthesis is the difficulty that users face in precisely articulating their intent. A number of strategies for resolving ambiguous user intent have been proposed in the program synthesis literature (Zhang et al. 2020), but the problem of intent ambiguity remains open.

In a game development context, the expressive dimension of code exacerbates this difficulty. Because there may be many viable ways to write a program (e.g., an NPC behavior script or game mechanic implementation) that nominally satisfy a set of user-provided constraints, but each of these implementations may *feel* slightly different from a player experience perspective, users may have to compare many divergent implementations of their stated intent before they can adequately determine which parts of the intent are correct, which are incorrect, and which are over- or underspecified. Further, evaluating each of these complex behaviors may take a nontrivial amount of time and effort on the part of the user, especially if they have to compare them in the context of real gameplay situations to get a thorough sense of the similarities and differences between implementations.

As a result, game development applications of program synthesis may benefit from the use of casual creator design patterns (like the **Chorus Line** or **Approximating Feedback**) or reflective creator design patterns (like **Interpretive Refraction** or **Inferring Intent**) to help users more rapidly evaluate large numbers of potential intent realizations and iterate on their intent in response (Compton and Mateas 2015; Kreminski and Mateas 2021). Germinate—a graphical game creation tool built on the Gemini architecture—implements some of these design patterns; in particular, it presents users with a graphical logic programming interface for specifying a design intent and allows them to rapidly add, remove, or negate parts of their explicit design intent based on features that are present in generated games. And Synthesifter (Kreminski, Wardrip-Fruin, and Mateas 2020) aids users in building up a set of concrete positive and negative examples as they refine their synthesized program by proactively suggesting new examples, which the user can easily either accept or reject.

Additionally, it may be helpful for tools to facilitate reflection on how expressive intent relates to low-level parts of synthesized programs by providing affordances for design journaling, perhaps based on the process described by Khaled, Lessard, and Barr (2018). For instance, a tool could prompt the user to leave plaintext notes for themselves on what they liked and disliked about each synthesized program or game, allowing them to build up a searchable record of design successes and failures over time and preserve past intents even as they refine their design sense. Users might then make use of this information in conjunction with an interpretable program synthesis interface like that presented by Zhang et al. (2021) to guide the synthesizer's search, instructing it to avoid parts of the search space that have frequently yielded unwanted programs and focus its search on areas that seem especially promising instead.

Educational Synthesis

One advantage of program synthesis is that synthesized programs need not remain black boxes to their users, even if the user starts out with little knowledge of the target programming language (Crichton 2019). In fact, a program synthesizer could even be designed to model user understanding of programming constructs (as is typical in tutoring-focused applications of program synthesis, e.g. Head et al. 2017) and gradually tutorialize the language by periodically introducing new constructs and idioms. Under such conditions, the user's reliance on the synthesizer might fade away over time as they acquire greater confidence in reasoning about the meaning of code. A synthesizer that enabled this experience could be viewed as a *complementary* cognitive artifact, rather than a *competitive* one (Krakauer 2016): the new capabilities it builds in the user remain with the user even when the artifact itself is removed.

In the context of game development, educational applications of synthesis might prove especially useful in helping users discover new game engine API features. Additionally, a programming tutor-like system for game development could be integrated with game *design* knowledge to assist users in linking code to design concepts and vice versa.

References

Anthropy, A. 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.

Bogost, I. 2010. *Persuasive Games: The Expressive Power of Videogames*. MIT Press.

Butler, E.; Siu, K.; and Zook, A. 2017. Program synthesis as a generative method. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*.

Compton, K.; and Mateas, M. 2015. Casual creators. In *International Conference on Computational Creativity*, 228–235.

Cook, M. 2020. Software engineering for automated game design. In *2020 IEEE Conference on Games (CoG)*, 487–494. IEEE.

Crichton, W. 2019. Human-centric program synthesis. In *PLATEAU Workshop @ UIST*.

Drosos, I.; Barik, T.; Guo, P. J.; DeLine, R.; and Gulwani, S. 2020. Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*.

Gulwani, S. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46(1): 317–330.

Gulwani, S.; Polozov, O.; Singh, R.; et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4(1-2): 1–119.

Harvey, A. 2014. Twine's revolution: Democratization, depoliticization, and the queering of game design. *G|A|M|E – Games as Art, Media, Entertainment* 1(3).

Head, A.; Glassman, E.; Soares, G.; Suzuki, R.; Figueredo, L.; D'Antoni, L.; and Hartmann, B. 2017. Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, 89–98.

Hempel, B.; and Chugh, R. 2016. Semi-automated SVG programming via direct manipulation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, 379–390.

Khaled, R.; Lessard, J.; and Barr, P. 2018. Documenting trajectories in design space: a methodology for applied game design research. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*.

Krakauer, D. 2016. Will A.I. harm us? Better to ask how we'll reckon with our hybrid nature. https://nautil.us/blog/will-ai-harm-us-better-to-ask-how-well-reckon-with-our-hybrid-nature. Accessed on 2021-07-01.

Kreminski, M.; Dickinson, M.; Osborn, J.; Summerville, A.; Mateas, M.; and Wardrip-Fruin, N. 2020. Germinate: a mixed-initiative casual creator for rhetorical games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 16, 102–108.

Kreminski, M.; and Mateas, M. 2021. Reflective creators. In *International Conference on Computational Creativity*.

Kreminski, M.; Wardrip-Fruin, N.; and Mateas, M. 2020. Toward example-driven program synthesis of story sifting patterns. In *Joint Proceedings of the AIIDE 2020 Workshops (AIIDE-WS-2020)*.

Osborn, J. C.; Dickinson, M.; Anderson, B.; Summerville, A.; Denner, J.; Torres, D.; Wardrip-Fruin, N.; and Mateas, M. 2019. Is your game generator working? Evaluating Gemini, an intentional generator. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 15, 59–65.

Osborn, J. C.; Wardrip-Fruin, N.; and Mateas, M. 2017. Refining operational logics. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*.

Powley, E. J.; Nelson, M. J.; Gaudl, S. E.; Colton, S.; Ferrer, B. P.; Saunders, R.; Ivey, P.; and Cook, M. 2017. Wevva: Democratising game design. In *Thirteenth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Rusch, D. C. 2017. *Making Deep Games: Designing Games with Meaning and Purpose*. CRC Press.

Summerville, A.; Martens, C.; Samuel, B.; Osborn, J.; Wardrip-Fruin, N.; and Mateas, M. 2018. Gemini: Bidirectional generation and analysis of games via ASP. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 14.

Treanor, M.; Blackford, B.; Mateas, M.; and Bogost, I. 2012. Game-O-Matic: Generating videogames that represent ideas. In *Proceedings of the Third Workshop on Procedural Content Generation in Games*.

Zhang, T.; Chen, Z.; Zhu, Y.; Vaithilingam, P.; Wang, X.; and Glassman, E. L. 2021. Interpretable program synthesis. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*.

Zhang, T.; Lowmanstone, L.; Wang, X.; and Glassman, E. L. 2020. Interactive program synthesis by augmented examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, 627–648.