Programming Studio: How Games Can Better Prepare Software Engineers

Kenneth Romero
Department of Computer Science
University of North Georgia
Dahlonega, Georgia 30597
kfrome3062@ung.edu

***Abstract*— The complexity of modern software poses a significant challenge for aspiring software engineers, as traditional computer science curricula often fall short in preparing students to effectively navigate large systems. This gap arises from the reliance on conventional teaching methods rather than engaging activity or project-based approaches that could better stimulate students' computational thinking. Games, serve as a medium that naturally cultivates computational thinking, as players grapple with the intricacies of game mechanics while playing. Drawing inspiration from game design principles, instructors can create playful learning environments that promote deeper understanding of course content and foster the development of computational thinking skills. This paper proposes a course design that leverages game design to establish such a playful environment specifically tailored for enhancing computational thinking, thereby equipping students with the necessary skills for future coursework and tackling more complex software projects.**

***Index terms*—active learning, project-based learning, activity-based learning, computational thinking, game design, play environment**

I. Introduction

I think that it's extraordinarily important that we in computer science keep fun in computing[...] Don't feel as if the key to successful computing is only in your hands. What's in your hands, I think and hope, is intelligence: the ability to see the machine as more than when you were first led up to it, that you can make it more.

— Alan J. Perlis

Current computer science curricula often fail to equip students with the skills needed to address the complexity of today's software. An emphasis on memorization hinders the development of computational thinking that is essential for success. This approach forces students to repeatedly relearn material, as they lack opportunities to apply their knowledge. Consequently, they are ill-prepared for advanced courses that require programming experience and the practical use of foundational concepts[1].

The primary obstacle is embedding computational thinking within the curriculum. "Computational thinking involves solving problems, designing systems, and understanding human behavior, by drawing on the concepts fundamental to computer science."[2] Programming, in turn, is the application of computational thinking. While curricula centered on programming have been devised, they often prioritize correctness and tend to feature relatively small-scale projects, thus failing to fully embrace computational thinking. To truly cultivate computational thinking, there must be a shift towards larger-scale projects, necessitating an educational environment conducive to project and activity based learning [3].

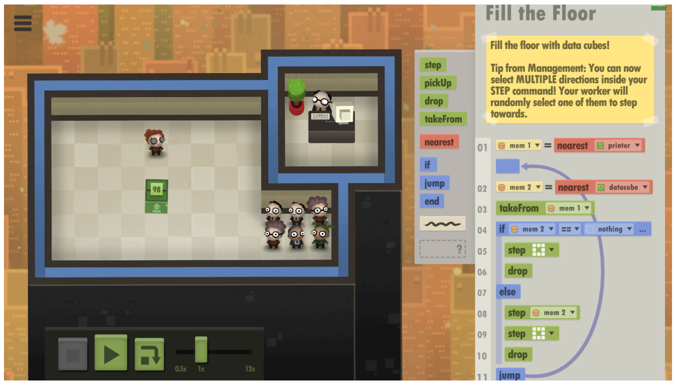
Environments that promote exploration and confidence-building are needed[4]. Classical teaching methods create environments that obscure the potential of programming and limit exploration. This stifles students' sense of agency and discourages them from exploring. Minimizing the tools needed for programming empowers students by showing them they possess the necessary resources to solve problems[3]. Empowering students to actively evaluate their understanding, iterate on their skills, and gain confidence in their abilities[5].

This paper outlines methods for selecting tools and designing courses, illustrating the creation of a curriculum that fosters computational thinking through a playful environment.

II. Creating A Play Environment

Video games are often misconstrued when it comes to their potential in education. The concept of gamification has exacerbated this misunderstanding by framing games solely as tools for enhancing engagement in the classroom. However, in reality, games serve as exemplary environments for nurturing computational thinking skills [6].

Game developers intricately design complex systems that players must comprehend to progress. Through the implementation of levels and mechanics, they construct a series of incremental challenges that players must navigate. This approach allows players to intuitively grasp the game's mechanics and continually test and refine their understanding of its systems. Moreover, certain games, such as *Kerbal Space Program* for rocket science[7], *Minecraft* for digital logic [8], or *Seven Billion Humans* for algorithms (Figure 1), offer immersive experiences that familiarize players with fundamental concepts rather than directly teaching them[9]. This immersion facilitates the development of a profound understanding of these concepts at their core.



Fill the floor with data cubes!

Tip from Management: You can now select MULTIPLE directions inside your STEP command! Your worker will randomly select one of them to step towards.

```
mem 1 = nearest printer

mem 2 = nearest datacube
takeFrom mem 1
if mem 2 == nothing ...
    step
    drop
else
    step mem 2
    step
    drop
jump
```

Figure 1: A solution to a problem in Seven Billion Human using the least amount of code

Drawing inspiration from the structure of games, educators can design learning environments that promote deeper comprehension of essential concepts. By integrating game-like elements, such as incremental challenges and immersive experiences, alongside traditional teaching methods, educators can foster intuitive and active learning while addressing any misconceptions students may have[3]. Through careful selection of tools and the incorporation of playful projects and problems, play environments can emulate the engaging nature of games, enticing students to enjoy learning for its intrinsic value rather than extrinsic rewards [10], [11], [12].

A. *For the Student*

Through the creation of playful learning environments, computational thinking can flourish. Establishing a playful environment requires careful selection of tools and assignments that empower students, fostering their agency and confidence. This approach encourages exploration, experimentation, and iteration of knowledge and skills. The choice of appropriate tools is dictated by the curriculum's objectives. Just as game developers must adhere to the constraints of their genre while innovating with mechanics and levels, instructors must similarly curate and deploy tools and assignments that reinforce the core concepts of the curriculum. This approach not only cultivates a deeper understanding of the subject-matter but also nurtures an appreciation for it by giving rise to an intrinsic want to learn [5], [6].

Tools serve as vehicles for students to actively engage with the fundamental concepts of the curriculum. Simplifying the complexity of these tools enables students to direct their focus towards understanding the curriculum itself, rather than grappling with the intricacies of the tools[6], [9]. When tools are easier to grasp, students are more inclined to believe in their ability to master them, fostering a sense of agency and confidence. This empowerment fosters student autonomy in practicing with the tools, thereby providing greater opportunities for students to develop a personal connection with the content and deepen their understanding of the concept or computer science as a whole.

As students develop a personal connection with the material, they become motivated to explore the concepts more deeply, leading to a heightened appreciation for them[5]. This intimate familiarity with the concepts allows for the integration and interweaving of ideas, fostering a holistic understanding of how the discipline operates as a cohesive system. Over time, this empowers students to construct a cohesive framework of concepts in computer science, enabling them to consistently broaden their perspective on the possibilities within the field[3].

Recognizing the diversity among students' abilities is key to nurturing computational thinking[3]. Assignments should encourage exploration and iteration, fostering a systemic perspective on concepts. By building upon prior knowledge and skills iteratively, students gain confidence and agency in applying these concepts practically. This approach encourages experimentation, deepening comprehension and fostering intuitive understanding[6]. Activity and project-based assignments exemplify this approach, providing practical engagement within simulated environments. Open-ended tasks allow students to demonstrate creativity and understanding, instilling an intrinsic want to learn. This iterative process promotes growth in confidence and skills, facilitating direct application and fostering a social dimension as students share discoveries, ideas, or designs with peers.

The social dimension of play cultivates community, empowering students and fostering a culture within the environment. Just as games have clans and sports have clubs, programming has its hacker communities[13], [14]. Belonging to such communities provides extrinsic motivation, as students find value in being part of a larger group. Community membership also instills a sense of responsibility and sets expectations, motivating students to explore and improve their skills[15].

Games developers craft an environment that instill an intrinsic want from the player to continually better their skills and understanding of the game. Instructors can do the same with play environments by utilizing activity and project-based assignments and careful choice of tools. This intrinsic want cultivates a social aspect as students want to share their ideas and skills to others.

B. *For the Instructor*

Play environments offer significant benefits to instructors akin to how game developers iterate on their games for improvement. Instructors can dynamically adjust courses based on student skills, participating alongside them to set benchmarks or assist those facing challenges. This flexibility allows instructors to creatively engage students, enhancing their learning experience and deepening understanding. Moreover, instructors can learn from students, much like game developers gather feedback from players, leading to continuous improvement in course delivery and effectiveness[3], [16].

An activity and project based approach provides instructors with valuable free time [4], [17], which can be utilized to enhance the course in various ways. This time can be spent completing assignments alongside students, potentially turning it into a competition or an opportunity for instructors to demonstrate their expertise and build credibility[3]. Additionally, instructors can use this time to experiment with new technologies or tools to determine their suitability for the curriculum. This exploration can inspire instructors to gain fresh perspectives on teaching certain topics and better engage students. Moreover, instructors can directly assist students who are struggling with activities or projects by providing guidance, diagrams, or demonstrations.

Engaging in activities where students take on the role of the instructor enables a dynamic exchange of knowledge[3], [16]. This allows instructors to assess students' understanding of the course material and gain insights into their perspectives on the concepts being taught. Understanding the effectiveness of teaching methods is essential for refining course delivery. Moreover, this approach empowers instructors to learn new teaching methods from students, enhancing the instructor's ability to convey curriculum concepts effectively and fostering stronger connections with students. By facilitating this student-as-instructor dynamic, instructors can better tailor the course to meet students' learning needs, ensuring a deeper understanding of the material.

Just as game developers engage players to assess game enjoyment and may also play themselves to enhance skills and build community, instructors can adopt similar approaches utilizing activity and project-based assignments. This enables instructors to tailor the curriculum to students' skills and understanding, fostering deeper comprehension of advanced coursework while nurturing a supportive learning community where students can collaborate and develop their understanding alongside peers and instructors.

III. Course Design

This course draws inspiration from several sources: Daniel Zingaro's *Algorithmic Thinking*[18], John Ousterhout's *CS 190: Software Design Studio*[19], Jae Woo Lee's course[1], hackathons[14], and game jams[13]. The assignments follow an activity and project-based approach.

The course is divided into three sections. The first section, inspired by Zingaro's work [18], focuses on competitive programming problems to assess students' programming proficiency or teach a new language. The second section, inspired by hackathons[14], aims to build students' confidence and provide practice with libraries. The final section draws inspiration from Ousterhout's course[19] and game jams[13], particularly Jonathan Blow's and Chris Hecker's *Depth Jam*[15], incorporating thematic project assignments for iterative work and peer feedback. The third section encourages exploration of ideas and creativity.

A. *Choosing a Programming Language and Library: Why Odin and Raylib?*

In truth, any language or library can be chosen as long as it aligns with the course curriculum or the institution's preferred language. However, one may wish to adapt this course to teach introductory programming or transition to a different language [1]. This course focuses on utilizing play to foster computational thinking, thus requiring a simple, comprehensive language. *Python*, *Go*, and *C* are excellent choices due to their rich ecosystems, ubiquity, and simple, familar syntax, providing an "easy to learn, hard to master" experience that allows students to accomplish more with less and concentrate on implementation design.

Odin[20] is a *C* alternative inspired by *Pascal*, *Go* and *Python* [21], featuring a simple build system, easy interoperability with *C*, and a rich standard and vendor library. The base installation of *Odin* provides a battery-included experience for new students, with a simple syntax for quick learning. This enables a joyful programming experience with the power of *C*.

Raylib[22] is simple library for putting graphics on the screen. It was originally created for art students to understand the basics of game programming [23]. Written in *C99*, portable to different platforms and languages, *Raylib* is a great library to know for student's or instructor's own future tools, projects or hobbies. The simplicity enables students to focus on creating their ideas, whereas the portability allow students to explicitly take their skills to other languages or platforms.

B. *Learning Odin through Advent of Code: 2 Weeks*

This section utilizes programming problems from *Advent of Code* [24], focusing on problem-solving rather than algorithms. These problems are an excellent resource for developing computational thinking and familiarity with a new programming language. Additionally, they offer accompanying questions that force students to iterate on their designs to meet requirements.

The class structure comprises three sections: independent work time, collaborative thinking, and lecture. During independent work time, students practice computational thinking, research skills, and proficiency with the programming language. Collaborative thinking allows students to review each other's code and explain their reasoning, fostering self-review. Lecture time is reserved for the instructor to present preferred solutions, emphasizing language features, recommended libraries, or efficient problem-solving approaches. Independent work should consume $\frac{3}{5}$ of class time, with collaborative thinking encouraged for $\frac{1}{5}$. However, lecture time may overlap with collaborative thinking to emphasize why the problem was selected. The duration of this course section is dependent on students' language proficiency, maximum 2 weeks.

```
package aoc
import "core:strings"
...
bad_words :: []string{"ab", "cd", "pq", "xy"}

part_1 :: proc(file: ^string) -> (total: int) {

  line_loop:
  for line in strings.split_lines_iterator(file){
    for word in bad_words {
      if strings.contains(line, word) {
        continue line_loop
      }
    }
...
```

Listing 1: A snippet of *Odin* code solving *Advent of Code* problem: Day 5 of 2015[1]

C. *Learning Raylib with Conway's Game of Life: 4 Weeks*

Students will learn *Raylib* by creating a simple game, such as *Conway's Game of Life*. This approach enables them to experiment with rendering to the screen, capturing user input, and implementing UI elements. By focusing on continually adding new features to the game rather than figuring out how to make it, students have the freedom to explore *Raylib* and personalize their project. This fosters a personal connection to the game and enhances their understanding of *Raylib*'s capabilities.

Classes serve as dedicated workspaces for students to collaborate on their project. They are encouraged to attend to share ideas, showcase their progress or simply work on their game. While the project has requirements, they are intentionally vague to foster student creativity. For example, in the case of the *Game of Life* project, requirements might include controlling the size of cells and the window, implementing save states for player creations, adding a UI, and providing game speed control. Depending on students' programming proficiency, the instructor may adjust requirements to optimize difficulty to promote learning. Additionally, programming time allows the instructor to have free time to pursue their own hobby projects, catch up on work, or work on the same project alongside students.

```
package game_of_life
import rl "vendor/raylib"
...
game_draw :: proc(size: int) {
  rl.BeginDrawing()
  defer rl.EndDrawing()
  rl.ClearBackground(rl.YELLOW)
  for &row, y in game_space {
    if y >= size {
      break
    }
    for &space, x in row {
      if x >= size {
        break
      } else if space {
        rl.DrawRectangle(
          i32(x * cell.width),
          i32(y * cell.height),
          i32(cell.width),
          i32(cell.height),
          rl.PINK,
        )
      }
    }
  }
}
...
main :: proc() {
  size := SMALL
  cellsize := game_creator(size)
  game_init(cellsize, cellsize, size)

  rl.InitWindow(i32(size * cell.width), i32(size *
cell.height), "Game of Life")
  defer rl.CloseWindow()
  rl.SetTargetFPS(30)

  for !rl.WindowShouldClose() {
    game_draw(size)
    game_play(size)
  }
}
```

Listing 2: Another snippet of *Odin* code, but utilizing *Raylib* to create an early implementation of *Conway's Game of Life*

This section consists of two parts: the first and second iterations. During the first iteration, students have two weeks to implement the Game of Life. Afterwards, students read their peer's code and provide feedback. With this feedback, students

[1]https://adventofcode.com/2015/day/5

undergo their second iteration, which will serve as the final product graded by both the instructor and their peers.

D. *Group Project and Code Reviews: 8 Weeks*

This section of the course adopts the structure of Ousterhout's course, with a focus on fostering students' creativity in programming rather than software design. Drawing inspiration from game jams, students are provided with a theme, mechanic, genre, or application to incorporate into their project. This approach gives them a main goal while allowing for exploration, experimentation, and iteration. It serves as an opportunity for students to enhance their proficiency with the language, deepen their knowledge of the library, develop computational thinking skills, and build confidence in their abilities.

Students have four weeks to develop their initial concept, aiming to produce a finished level that utilizes the core program feature for the first code review. Subsequently, the following four weeks are dedicated to iterating on their program based on feedback, similar to the previous project. However, students are now paired up, allowing them to tackle more complex projects or achieve a higher level of polish. Each project should ideally consist of around 1000 - 1500 lines of code per student[16], although this is not a strict requirement as the number may vary depending on how efficiently students express their ideas.

This section largely mirrors the previous one, providing students with dedicated time to work and explore while giving instructors flexibility for personal pursuits. The main distinction lies in affording students more time to develop a more complex system compared to the previous project.

E. *Drawbacks of the Course*

The primary drawback of the course is the substantial amount of reading required by the instructor, which limits its scalability[16]. While the first section can be automated relatively easily, the heart of the course lies in the projects. Therefore, small class sizes are necessary to conduct the course effectively, as the instructor aims to avoid overwhelming themselves with the task of reviewing large volumes of code.

IV. Conclusion

By incorporating game design principles into course development, educators can craft a playful learning environment by carefully choosing tools and assignments. This approach benefits both instructors and students. Students not only deepen their understanding of the curriculum but also have increased opportunities to apply fundamental computer science concepts and skills. Activity-based learning establishes the groundwork by fostering computational thinking through problem-solving and intentional tool practice, while project-based learning increases difficulty and complexity, and integrates concepts into a cohesive system. This allows students to work with larger codebases, collaborate, review code, and manipulate systems, better preparing them for future endeavors in a playful learning environment. Allowing students' ingenuity and mastery to be rewarded through personal goals and desires.

Instructors also reap advantages by having free time to enhance the course. They can observe students, actively participate in activities/projects, or experiment with new tools and environments. This dynamic mirrors the relationship between game developers and players: developers strive to impart their ideas to players, while players seek enjoyment and engagement with the tools, environments, and problems (games) made by developers.

References

[1] H. S. Jae Woo Lee Michael S. Kester, "Follow the River and You Will Find the C," Mar. 2011.

[2] J. M. Wing, "Computational thinking," *Commun. ACM*, vol. 49, no. 3, pp. 33–35, Mar. 2006, doi: 10.1145/1118178.1118215. Available: https://doi.org/10.1145/1118178.1118215

[3] T. L. Orit Hazzan Noa Ragonis, *Guide to Teaching Computer Science*, 3rd ed. Springer Cham, 2020. doi: 10.1007/978-3-030-39360-1

[4] B. K. Hunnicutt, "Leisure and play in Plato's teaching and philosophy of learning," *Leisure Sciences*, vol. 12, no. 2, pp. 211–227, 1990, doi: 10.1080/01490409009513101. Available: https://doi.org/10.1080/01490409009513101

[5] M. Resnick, "Mitch Resnick: Let's teach kids to code," Jan. 2013. Available: https://youtu.be/Ok6LbV6bqaE?si=Dcke52GbHQSVSMPr

[6] J. Blow, "Video Games and the Future of Education," Jul. 2020. Available: https://youtu.be/qWFScmtiC44?si=TXFZcDcP4Jc1fjd4

[7] C. Hall, "TO THE MUN AND BACK: KERBAL SPACE PROGRAM," Jan. 2014. Available: https://www.polygon.com/features/2014/1/27/5338438/kerbal-space-program

[8] L. Prayaga, J. Davis, A. Whiteside, and A. Riffle, "An Exploration In The Use Of Minecraft To Teach Digital Logic To Secondary School Students," vol. 2, p. , 2016.

[9] Z. Barth, "Zachtronics: 10 Years of Terrible Games," Mar. 2017. Available: https://youtu.be/Df9pz_EmKhA?si=NsKBJnryr0br4K8T

[10] M. T. Morazán, "Using Video Game Development to Motivate Program Design and Algebra Among Inner-City High School Students," *Electronic Proceedings in Theoretical Computer Science*, vol. 321, pp. 78–99, Aug. 2020, doi: 10.4204/eptcs.321.5. Available: http://dx.doi.org/10.4204/EPTCS.321.5

[11] N. E. Cagiltay, "Teaching software engineering by means of computer-game development: Challenges and opportunities," *British Journal of Educational Technology*, vol. 38, no. 3, pp. 405–415, 2007, doi: https://doi.org/10.1111/j.1467-8535.2007.00705.x. Available: https://bera-journals.onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8535.2007.00705.x

[12] L. Cao and A. Rorrer, "An Active and Collaborative Approach to Teaching Discrete Structures," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, in SIGCSE '18. Baltimore, Maryland, USA: Association for Computing Machinery, 2018, pp. 822–827. doi: 10.1145/3159450.3159582. Available: https://doi.org/10.1145/3159450.3159582

[13] G. Lai *et al.*, “Two Decades of Game Jams,” in *Proceedings of the 6th Annual International Conference on Game Jams, Hackathons, and Game Creation Events*, in ICGJ '21. Montreal, Canada: Association for Computing Machinery, 2021, pp. 1–11. doi: 10.1145/3472688.3472689. Available: https://doi.org/10.1145/3472688.3472689

[14] K. Jenkins, “Creating Hackathons that Work (with Jon Gottfried),” Mar. 2024. Available: https://youtu.be/Q9t8oxD5Bs8?si=5DpZwkfioEFjS9jQ

[15] J. Blow, “The Depth Jam,” May 2024. Available: http://the-witness.net/news/2012/05/the-depth-jam/

[16] J. Ousterhout, “Can Great Programmers Be Taught? - John Ousterhout - Agile LnL,” 2022. Available: https://youtu.be/LtRWu9DErgU?si=tX2K_tBY8jM0-kuD

[17] S.-G. Chappell, “Plato on Knowledge in the Theaetetus,” *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, 2023.

[18] D. Zingaro, *Algorithmic Thinking: A Problem-Based Introduction*, 2nd ed. San Francisco: No Starch Press, 2023.

[19] J. Ousterhout, “CS 190: Software Design Studio,” 2016. Available: https://web.stanford.edu/~ouster/cgi-bin/cs190-spring16/index.php

[20] B. Hall, “Odin,” 2024. Available: https://odin-lang.org/

[21] K. Jenkins, “Is Odin "Programming done right"? (with Bill Hall),” Jan. 2024. Available: https://youtu.be/aKYdj0f1iQI?si=yFBnWqCrMAznFcLw

[22] R. Santamaria, “Raylib,” 2024. Available: https://www.raylib.com/

[23] T. DeVries, “Interview with 'raysan5' (author of Raylib),” Apr. 2024. Available: https://youtu.be/sksueAxjCzo?si=45-BdKbYLpbOhc2z

[24] E. Wastl, “Advent of Code,” 2023. Available: https://adventofcode.com/2016/about