Cultivating Real-Life Programming Skills Through Gaming (Game-Logic)

Susan Nachawati

California State University, Long Beach, United States

To Cite This Chapter

Nachawati, S. (2024). Cultivating real-life programming skills through gaming (Game-logic). In T. A. Oliveira, & M. T. Hebebci (Eds.), *Current Academic Studies in Technology and Education 2024* (pp. 169-205). ISRES Publishing

Introduction

The importance of programming education in the modern educational landscape is crucial. Traditional learning environments, where students should sit and listen to the information provided by the teachers are unacceptable for them (Campbell, 2020). Students from various disciplines are taking programming classes, which have been shown to enhance learning and success levels and develop computational thinking skills (Rovshenov, A. & Sarsar, F, 2023), and in seeking more interesting, fun, motivating and engaging learning experiences (Anastasiadis, T., Lampropoulos, G., & Siakas, K., 2018). Scholarly publications suggest a variety of teaching methodologies that can significantly improve students' learning experiences.

With the rapid advancements in Information and Communication Technologies (ICT), educators face new demands that affect how programming should be taught in classrooms. As (Pedró, 2006) notes, today's students differ significantly from those for whom traditional educational systems were designed. Concurrently, there is a need

to bridge the gap between educational outcomes and employer needs (Voyager: direction for learning and careers community handbook set; book 4 Scarecrow education book, 2003). When teaching programming, it is crucial for students to grasp and apply abstract concepts to devise algorithms that address real-world problems. According to (Sobrel, 2020), students often have the perception that the focus is on learning the syntax of the programming language, leading them to focus on implementation activities rather than activities. (Anabela Gomes & Antonio Jose Mendes, 2007) identify multiple challenges in programming education, and I would like to combine them in two main categories: The teaching method, which often focuses on language syntax development rather than problem-solving, and the study method, where students struggle with algorithmic thinking and applying their developing problem-solving skills.

To address these challenges, educators must employ innovative teaching methods that maintain student enthusiasm and motivation, connecting programming tasks to tangible applications. This can be done using different strategies, however, I will be addressing two of them: Incorporating real-life examples in the classroom and Gamifications.

Incorporating Real-Life Examples in Programming Education

Real-life examples in programming education often involve problem-solving, which is a key skill in programming. For instance, explaining a loop through the analogy of repetitive daily routines can help students understand its purpose and functionality.

Studies have shown that incorporating real-life examples in programming education can significantly enhance students' understanding and problem-solving skills. For instance, a study published in *Education and Information Technologies* found that using visualization, interaction, and examples in the home language improves students' results (Prasad, A., Chaudhary, K. & Sharma, B., 3197–3223 (2022)). Another study in Educational Technology Research and Development explored an artificial intelligence learning platform in a visual programming environment, demonstrating the positive impact of real-world examples

on learning attitudes and performance (Chang, JH., Wang, CJ., Zhong, HX. et al, 2023).

By working through real-life examples, students can develop their logical thinking and problem-solving abilities, which enhances students' learning experience, creativity, and engagement, and provides a contextual understanding that makes abstract programming concepts more tangible. Students can begin to imagine how they can use code to solve problems or create something new, like a website or an application.

However, while using real-life examples in teaching programming has its benefits, it also presents several challenges. Real-life problems can be complex and intricate, making it difficult to simplify them into examples that beginners can understand. Finding examples that are relevant and interesting to all students can be challenging, given their diverse backgrounds and interests.

Additionally, developing and explaining real-life examples can be time-consuming. In a classroom setting with limited time, this might lead to less time for other important discussions. Some real-life examples might require resources (like specific software or hardware) that are not readily available in the classroom. There can also be a gap between theoretical concepts and their practical applications, and bridging this gap can be challenging for educators.

Despite these challenges, the benefits of using real-life examples in teaching programming often outweigh the difficulties. With careful planning and consideration, educators can overcome these obstacles and create a more engaging and effective learning environment.

Incorporating Game-Based Examples in Programming Education

Other studies suggest Implementation of a game-based learning approach (gamification) for creating engaging learning environments. While the approach of utilizing games for educational purposes holds considerable promise, it is important to recognize the complex and expensive nature of creating such games (Boyle et al, March 2016)

Crafting an educational game is a task that demands extensive planning and a diverse set of skills (Mahmood H. Hussein; et al, 2019). The used games must be developed to the right level of complexity that fits student's level of education to maintain learner interest without causing boredom or frustration (Liu, Z. Y., Shaikh, Z., & Gazizova, F., 2020). Game developers may struggle to incorporate educational objectives into their designs. At the same time, educators don't have the expertise (nor the time) required to develop games that effectively blend learning with fun (Meihua Qian & Karen R. Clark, 2016)

It is also essential to maintain a moderation between pedagogical and entertainment factors. games must be well designed and with the right level of complexity so the learners should not be bored or frustrated while playing (Zi-Yu Liu et al., 2020).

(Videnovik, M., Vold, T., Kiønig, L. et al., 2023) published a scoping literature review that provides insight into current trends and identifies research gaps and potential research topics concerning game-based learning in computer science education.

No matter what in-class strategies we use (flip class, peer programming…) we are still not being able to keep the students motivated enough to learn the concepts.

Incorporating Game-logic in Programming Education

Before writing the code for any program to solve a problem, the students must understand the problem, solve it by hand, and know how to draw a flowchart. Students often have the perception that the focus is on learning the syntax of the programming language, leading them to focus on implementation activities rather than activities such as planning, drawing, or testing (Sónia Rolland Sobra, 2020).

The Effectiveness of Using Games in Teaching Programming

Using games to teach programming concepts can be a highly effective strategy. Games are inherently engaging and fun, which can motivate students to learn and apply programming concepts. Games often have simple rules and objectives, making them easier to understand than complex real-life scenarios. Games provide immediate feedback on actions, helping students understand the consequences of their code.

Games often involve overcoming challenges or puzzles, which can encourage problem-solving skills - a key aspect of programming. Designing and coding games can stimulate students' creativity, as they can experiment with different ideas and solutions. Games provide a safe environment where students can make mistakes and learn from them without real-world repercussions.

In the context of game development, students often possess prior experience playing games, which grants them an intuitive understanding of game mechanics and logic. Their familiarity with game dynamics allows them to dissect complex problems into smaller, manageable components. Consequently, students can approach game development by breaking down the solution into logical segments, leveraging the programming language they are currently learning. This approach not only reinforces their programming skills but also encourages a systematic problem-solving mindset. By bridging the gap between gameplay experience and programming concepts, students can effectively create robust and engaging game solutions.

This paper serves as the first in a series dedicated to discovering the ideal game that can be used as a teaching tool for basic programming concepts. We delve into the representation of real-life concepts within games and explore how the same game can be adapted to different levels of learning.

Key Areas of Focus

1. **Basic Programming Concepts**: We will identify games that effectively illustrate fundamental programming concepts such as variables, control structures, data structures, syntax, and logic.
2. **Real-life Concept Representation**: We will analyze how games mirror real-life scenarios and principles, thereby making abstract programming concepts more tangible and relatable for learners.
3. **Game Adaptability**: We will investigate how a single game can be modified or scaled to cater to different learning stages, ensuring a progressive and level-appropriate learning experience.

Students find it hard to solve the problem by hand, so if we introduce them to problems they understand, love, and spend numerous times around the same problem while having fun. Code the problem solution using programming language and link the solution to a real-life problem.

Then we are saving the time needed to explain the problem, since most of them already understand the problem, and they are excited to know the coding behind it. In this paper, I am introducing what I am calling a "game-logic linked to real-life problem" in the classroom to teach programming concepts.

Game1

The Educational and Practical Applications of Mad Libs: A Research Perspective

Mad Libs, a popular word game that prompts players to fill in blanks with random words, has been found to have significant educational and practical applications beyond its primary function as a source of entertainment. Mad Libs can offer valuable lessons and insights applicable beyond playtime.

1. **Language Learning and Creativity**: Mad Libs can enhance language skills, vocabulary, and creativity. By selecting appropriate words to fit the context, players learn about parts of speech and sentence structure. In real-life scenarios, strong language skills are essential for effective communication, writing, and problem-solving.
2. **Algorithmic Thinking**: The process of creating a Mad Libs game involves designing an algorithm. This includes identifying placeholders, collecting input, and assembling the final story. Similarly, in programming, developers design algorithms to solve problems, such as sorting data, processing user input, or automating tasks.
3. **User Input Handling**: Mad Libs prompts users for specific types of words. This aspect of handling user input is crucial in programming applications. Real-

world software often requires user input validation, error handling, and data processing.

4. **Dynamic Content Generation**: Mad Libs dynamically generates unique stories based on user input. This concept aligns with dynamically generating content in web applications, reports, or personalized messages. For instance, e-commerce websites personalize product recommendations based on user preferences.
5. **Entertainment and Engagement**: Mad Libs entertain and engage players by turning a simple activity into an amusing experience. In real-life applications, user engagement is critical for websites, apps, and educational tools. Gamification elements, interactive interfaces, and storytelling enhance user experiences.
6. **Collaboration and Social Interaction**: Playing Mad Libs with friends or family involves collaboration, laughter, and shared creativity. In professional settings, collaboration is essential for teamwork, brainstorming, and problem-solving.
7. **Adaptability and Flexibility**: Mad Libs adapt to different themes, genres, and contexts. The same framework can generate countless variations. In programming, writing modular and flexible code allows applications to adapt to changing requirements.

Incorporating Multiple Programming Concepts into Mad Libs Game

Mad Libs game provide an excellent platform to incorporate various programming concepts. The game primarily relies on user input to fill in the blanks, utilizing the input() function to prompt users for different word types such as nouns, adjectives, and verbs. Variables are declared to store the user's input, and string manipulation techniques, such as concatenation, are used to construct the Mad Libs story by replacing placeholders in a predefined template with the user's input.

The structure of the game is based on templates containing blank spaces, which are replaced with the user's input to create a unique story each time. For more complex versions of the game, loops can be used to create multiple stories, and conditional statements can handle different scenarios based on specific user inputs.

Functions are also employed to organize the code, making it modular and easier to maintain. For instance, a function can take user input and return the completed Mad Libs story.

Beyond the basics, enhancements can be made to the game. A graphical user interface (GUI) can be created using Tkinter to make the game more user-friendly, and the pyttsx3 library can be used to convert the story to speech for added fun. Other enhancements include random story selection from a collection of templates and allowing users to select different themes for their Mad Libs.

It has been used in in "Augmenting Cross-Domain Document-Level Event Argument Data" (Joseph Gatto, Parker Seegmiller, Omar Sharif, Sarah M. Preum, 2024)

We can take the Mad Libs game to the next level by implementing some of the optional features we discussed above. Here are the steps you can follow:

1. Loops:
    - If you want to create multiple Mad Libs stories, consider using loops. You can use either a for loop or a while loop.
    - For example, you can prompt the user if they want to play again after completing one story. If they say yes, loop back to collect input for another story.
2. Conditional Statements:
    - Add conditional logic to handle different scenarios based on user input.
    - Example: If the user enters a specific word (e.g., "dragon") as a noun, you can adjust the story accordingly. Maybe switch to a fantasy theme or introduce magical elements.

3. Functions:
    - Organize your code by creating functions.
    - For instance, you can have a function that takes user input and returns the completed Mad Libs story.
    - This makes your code modular and easier to maintain.
4. Lists Usage:
    - Define lists for different parts of speech (nouns, verbs, adjectives, etc.).
    - Randomly select words from these lists to create more dynamic stories.

```
import random
# Sample lists (you can expand these)
nouns = ["cat", "tree", "ship", "wizard"]
adjectives = ["sparkling", "mysterious", "gigantic", "playful"]

# Randomly select words
selected_noun = random.choice(nouns)
selected_adjective = random.choice(adjectives)
```

5. Simple Files:
    - Include these lists in separate files (e.g., CSV or text files).
    - Read the lists from files during runtime.
    - This approach allows you to easily update or expand your word options without modifying the code.
6. Incorporating the Natural Language Toolkit (NLTK) library can enhance your Mad Libs game by automatically identifying the part of speech (POS) of user-entered words.
    - Let’s enhance the existing implementation with NLTK:

```
import nltk
from nltk.corpus import wordnet

'''import ssl

try:
    _create_unverified_https_context =
```

```
ssl._create_unverified_context
except AttributeError:
    pass
else:
    ssl._create_default_https_context =
_create_unverified_https_context
nltk.data.path.append("/users/swzyhswryh/nltk_data")

nltk.download()

# Initialize NLTK (you'll need to download NLTK data if you
haven't already)
nltk.download("wordnet")'''

# Function to get the POS tag for a word
def get_pos(word):
    synsets = wordnet.synsets(word)
    if synsets:
        return synsets[0].pos()
    else:
        return None

def get_word_with_correct_pos(requested_pos):
    while True:
        if requested_pos == 'n':
            requestedPos = 'a noun'
        elif requested_pos == 'a':
            requestedPos = 'an adjective'
        user_input = input(f"Enter {requestedPos}: ")
        pos_tag = get_pos(user_input)
        if pos_tag and pos_tag.startswith(requested_pos):
            return user_input
        else:
            print(f"Error: '{user_input}' is not
{requestedPos}. Try again.")

def main():
    # Request user input for specific parts of speech
    noun1 = get_word_with_correct_pos("n")
    adjective2 = get_word_with_correct_pos("a")
    noun3 = get_word_with_correct_pos("n")

    # Construct the Story
    mad_libs_template = f"Once upon a time, there was a
{noun1} who loved {adjective2} {noun3}."
    print("\nYour Mad Libs story:")
    print(mad_libs_template)

if __name__ == "__main__":
    main()
```

7. Enhancements (Beyond Basics):

   Once you've built the basic Mad Libs game, consider adding more features:

   - Graphical User Interface (GUI) (using Tkinter):

- Create a visually appealing interface where users can input words and see the completed story.

- Text-to-Speech (using pyttsx3 or a similar library):
  - Convert the story to speech for added fun.
- Random Story Selection:
  - Instead of a fixed template, randomly choose from a collection of templates.
  - Custom Themes: Allow users to select different themes (e.g., sci-fi, mystery, romance) for their Mad Libs.

Game 2

The Educational and Practical Applications of FizzBuzz: A Research Perspective

In the realm of computer science education, the FizzBuzz game serves as a noteworthy pedagogical tool. This simple programming task, often employed in coding interviews and introductory programming courses, provides an engaging platform for beginners to grasp fundamental concepts such as loops and conditionals. The game operates on a set of straightforward rules: a program is written to print numbers from 1 to 100, but with multiples of three replaced by “Fizz”, multiples of five by “Buzz”, and multiples of both three and five by “FizzBuzz”. Despite its simplicity, FizzBuzz encourages students to think critically about the logic and structure of their code, and its connections to real-world programming skills make it a valuable exercise for both beginners and experienced developers fostering their problem-solving skills and computational thinking.

The Significance of FizzBuzz in Programming

FizzBuzz, serves as an effective interview screening device, assessing fundamental coding skills such as loops, conditionals, and basic arithmetic. The ability to solve

FizzBuzz efficiently mirrors the real-world software development requirement of translating problem requirements into code.

FizzBuzz also enhances problem-solving skills by encouraging logical thinking and the breakdown of problems into smaller, manageable steps. This approach is crucial in software development where complex tasks need to be simplified.

The game reinforces the understanding of essential programming constructs like loops and conditional statements, which are used daily in tasks ranging from data processing to user interface development. It introduces the use of the modulo operator for checking divisibility, a concept vital for tasks like scheduling and managing cyclic data.

FizzBuzz emphasizes the importance of code readability and maintainability. Writing clean, concise code for FizzBuzz demonstrates good coding practices, which are essential for maintaining, debugging, and collaborating on real-world projects.

The game necessitates the consideration of edge cases, promoting thorough testing to ensure software correctness. It also encourages clear communication and documentation, which are vital skills in collaborative development environments.

FizzBuzz fosters algorithmic thinking which is a critical skill for optimizing performance, solving complex problems, and designing efficient systems. It serves as a teaching and learning tool, introducing beginners to programming concepts. This aspect is similar to real-world projects where mentoring junior developers or explaining code to colleagues is often required.

Lastly, FizzBuzz can be approached creatively, leading to unique solutions. This aspect of the game enhances problem-solving and fosters innovation in software development, making FizzBuzz not just a game, but a significant tool in the programming world.

1. **Interview Screening Device**:
    - **Programming Interviews**: FizzBuzz is commonly used as an interview screening task for computer programmers. It assesses

fundamental coding skills, including loops, conditionals, and basic arithmetic.

- **Real-Life Application**: Just as in interviews, real-world software development often involves solving straightforward problems efficiently. FizzBuzz reflects the ability to translate requirements into code.

2. **Problem-Solving Skills**:
   - **Programming Interviews**: FizzBuzz challenges candidates to think logically and break down a problem into smaller steps.
   - **Real-Life Application**: In software development, breaking down complex tasks into manageable components is crucial. FizzBuzz encourages this problem-solving mindset.
3. **Looping and Conditionals**:
   - **Programming Interviews**: FizzBuzz reinforces understanding of loops (e.g., for, while) and conditional statements (if, else).
   - **Real-Life Application**: Loops and conditionals are fundamental constructs used in everyday programming tasks, from data processing to user interfaces.
4. **Modulo Operator**:
   - **Programming Interviews**: FizzBuzz introduces the use of the modulo operator (%) to check divisibility.
   - **Real-Life Application**: Modulo operations are essential for tasks like scheduling, handling periodic events, and managing cyclic data.
5. **Code Readability and Maintainability**:
   - **Programming Interviews**: Writing clean, concise code for FizzBuzz demonstrates good coding practices.

- **Real-Life Application**: Well-organized, readable code is easier to maintain, debug, and collaborate on in real-world projects.

6. **Edge Cases and Testing**:
   - **Programming Interviews**: Considering edge cases (e.g., multiples of 3 and 5) is essential for FizzBuzz.
   - **Real-Life Application**: Robust software must handle various scenarios, including edge cases. Testing thoroughly ensures correctness.
7. **Communication and Documentation**:
   - **Programming Interviews**: Explaining your FizzBuzz solution demonstrates communication skills.
   - **Real-Life Application**: Clear documentation and communication are vital in collaborative development environments.
8. **Algorithmic Thinking**:
   - **Programming Interviews**: FizzBuzz encourages candidates to think algorithmically.
   - **Real-Life Application**: Algorithmic thinking is crucial for optimizing performance, solving complex problems, and designing efficient systems.
9. **Teaching and Learning Tool**:
   - **Programming Interviews**: FizzBuzz teaches beginners programming concepts.
   - **Real-Life Application**: Similar to teaching, real-world projects often involve mentoring junior developers or explaining code to colleagues.
10. **Fun and Creativity**:

- **Programming Interviews**: FizzBuzz can be approached creatively, leading to unique solutions.
- **Real-Life Application**: Creative thinking enhances problem-solving and innovation in software development.

Incorporating Multiple Programming Concepts into FizzBuzz Game

We can enhance the classic FizzBuzz game by incorporating various programming concepts.

1. Modular Design:
    - Break down the FizzBuzz logic into separate functions or methods. For example:
        - A function to check divisibility by 3.
        - A function to check divisibility by 5.
        - A function to handle the main FizzBuzz logic.
2. Object-Oriented Approach:
    - Create a FizzBuzzGame class with methods for printing Fizz, Buzz, and FizzBuzz.
    - Use encapsulation to store relevant data (e.g., range of numbers).
3. Error Handling:
    - Handle invalid input (e.g., negative numbers, non-integer values).
    - Raise custom exceptions for specific cases (e.g., "FizzBuzzError").
4. Localization and Internationalization:
    - Allow users to choose their preferred language for Fizz, Buzz, and FizzBuzz.

- Implement language-specific rules (e.g., "Fizz" in French, "Bzzz" in German).

5. Concurrency and Parallelization:
    - Use multithreading or multiprocessing to compute FizzBuzz for multiple ranges simultaneously.
    - Optimize performance by parallelizing the computation.
6. Unit Testing:
    - Write test cases to verify correctness of FizzBuzz functions.
    - Test edge cases (e.g., large numbers, zero).
7. Functional Programming:
    - Implement FizzBuzz using functional constructs (e.g., map, filter, reduce).
    - Create a list of FizzBuzz results for a given range.
8. Memorization:
    - Cache FizzBuzz results for previously computed numbers to avoid redundant calculations.
    - Improve efficiency when querying the same range multiple times.
9. Generators:
    - Create a FizzBuzz generator that yields the next FizzBuzz value on demand.
    - Use lazy evaluation to avoid precomputing the entire sequence.
10. Visualization:
    - Display FizzBuzz results graphically (e.g., bar chart, pie chart).
    - Show the distribution of Fizz, Buzz, and FizzBuzz occurrences.

Some real-life problems that utilize the same programming concepts as the FizzBuzz game, such as conditional logic and iteration.

1. **Smart Home Automation**:
    - Smart home devices (such as thermostats, lights, and security cameras) rely on sensors and decision-making algorithms.
    - Conditional logic determines when to adjust room temperature, turn lights on/off, or trigger security alerts.
    - State machines manage device states (e.g., idle, active, standby).
2. **Traffic Light Control Systems**:
    - Traffic lights follow predefined rules based on sensor inputs (vehicle presence, pedestrian crossings).
    - Decision-making algorithms determine when to switch lights (green, yellow, red).
    - Edge cases (emergency vehicles, power outages) require adaptive behavior.
3. **Industrial Automation**:
    - Manufacturing robots (like robotic arms) follow specific paths to assemble products.
    - Sensors detect workpieces, obstacles, and safety conditions.
    - Feedback loops adjust robot movements for precision and efficiency.
4. **Medical Devices**:
    - Infusion pumps, heart rate monitors, and ventilators use sensors and state machines.
    - Safety checks (e.g., drug dosage limits) prevent errors.
    - Real-time decisions impact patient well-being.

5. **Financial Systems**:
    - Trading algorithms execute buy/sell orders based on market conditions.
    - Risk management systems use conditional rules to prevent losses.
    - Event-driven programming responds to stock price changes.
6. **Game Development**:
    - Video games employ decision trees, finite state machines, and event handlers.
    - AI opponents follow strategies (e.g., pathfinding, combat behavior).
    - Game engines manage game states (menus, gameplay, cutscenes).
7. **Autonomous Vehicles**:
    - Self-driving cars use sensors (lidar, cameras) to perceive surroundings.
    - Decision-making algorithms navigate roads, avoid collisions, and follow traffic rules.
    - Edge cases (unpredictable pedestrians, adverse weather) require robust programming.
8. **Natural Language Processing (NLP)**:
    - Chatbots, virtual assistants, and language translation tools rely on NLP principles.
    - Conditional rules determine responses based on user input.
    - State management handles conversational context.

The Intersection of FizzBuzz and Roomba Programming: A Comparative Analysis

Roomba, a popular autonomous vacuum cleaning robot, may seem unrelated at first glance to FizzBuzz game. However, a closer examination reveals intriguing parallels and contrasts between the two.

Despite their differences in purpose and complexity, both FizzBuzz and Roomba programming rely on conditional logic and iterative processes. This comparison serves as a reminder of the diverse applications of programming, from abstract challenges to practical, real-world solutions.

On the one hand, Roomba programming is a complex process that enables the robot to autonomously navigate and clean floors. This involves the use of various sensors to detect obstacles and drop-offs, path planning to create efficient cleaning paths, decision-making based on sensor data, state machines to transition between different states (e.g., cleaning, docking, idle), feedback loops to adjust behavior, and handling of edge cases.

Interestingly, both FizzBuzz and Roomba programming involve the use of conditional logic and iterative processes. In FizzBuzz, this is seen in the checking of divisibility, while in Roomba programming, sensors trigger actions based on environmental conditions. Furthermore, Roombas repeatedly navigate and clean areas, akin to the loop in FizzBuzz. The straightforward rules in FizzBuzz (divisibility by 3 and 5) also parallel Roomba's basic rules (avoid obstacles, recharge when low).

However, significant differences exist. FizzBuzz is primarily a test of coding skills, while Roomba programming serves a practical purpose - cleaning. Moreover, Roomba programming is more intricate due to real-world challenges such as obstacle avoidance and battery management. Lastly, while FizzBuzz is abstract, Roomba programming deals with physical environments.

Game3

Rock-Paper-Scissors (RPS): A Study in Strategy and Programming

The game of Rock-Paper-Scissors (RPS), often dismissed as a simple pastime, is a complex study in strategy and decision-making that offers valuable insights applicable to real-life situations and programming concepts.

1. **Decision-Making and Logic**:
    - In RPS, players make choices based on rules (rock smashes scissors, paper covers rock, scissors cut paper). Similarly, programming involves decision-making using **conditional statements** (if-else, switch-case).
    - Understanding how to structure logic and evaluate conditions is fundamental in both RPS and programming.
2. **User Input Handling**:
    - RPS requires taking user input (rock, paper, or scissors). In programming, handling user input from forms, command-line arguments, or GUIs is crucial.
    - Learning to validate and process user input is directly applicable to building interactive software.
3. **Randomization and Simulating Opponents**:
    - In RPS, the computer randomly selects its move. This mirrors scenarios where programs generate random numbers or simulate opponents' behavior.
    - Concepts like **random modules** in Python or **pseudo-random number generators** are essential for game development and simulations.

4. **State Machines and Finite Automata**:
    - RPS can be modeled as a **finite state machine** with three states (rock, paper, scissors) and transitions between them.
    - In programming, understanding state machines helps design systems with well-defined states and transitions (e.g., game states, UI navigation).
5. **AI and Decision Trees**:
    - Enhancing RPS with AI opponents involves creating decision trees. The AI predicts the player's next move based on patterns.
    - In real-life applications, decision trees are used for recommendation systems, fraud detection, and more.
6. **Code Organization and Functions**:
    - Well-structured RPS code uses functions for readability and maintainability.
    - Similarly, modularizing code into functions or classes is crucial in larger software projects.
7. **Error Handling and Robustness**:
    - RPS code should handle unexpected inputs (invalid choices, unexpected user behavior).
    - In programming, robust error handling ensures software resilience.
8. **Scalability and Performance**:
    - Imagine expanding RPS to include more options (e.g., Rock-Paper-Scissors-Lizard-Spock). Designing scalable solutions is akin to handling larger datasets or more complex systems.

9. **Testing and Debugging**:
    - Debugging RPS helps identify issues (e.g., infinite loops, incorrect outcomes).
    - Testing strategies (unit tests, integration tests) apply universally to ensure software correctness.
10. **Algorithmic Thinking**:
    - RPS strategies involve analyzing patterns and predicting opponents' moves.
    - Algorithmic thinking—breaking down problems, devising efficient solutions—is vital in programming.

Game 4

The **Wordle** game, which gained popularity after its launch by Josh Wardle in October 2021, offers valuable insights applicable to real-life programming concepts. Let's explore how:

1. **Building Command-Line Applications**:
    - **Wordle** is typically played in the terminal. Creating your own version of Wordle as a **command-line application** helps you understand input/output handling, user interaction, and game mechanics.
2. **User Input Validation**:
    - In Wordle, players guess words. Validating user input (ensuring it adheres to rules) is crucial. Similarly, in programming, input validation prevents unexpected behavior.
3. **Data Structures (Word Lists)**:
    - Wordle relies on a **word list** (a collection of valid words). Creating, managing, and searching through word lists are fundamental programming tasks.

- You'll learn about **lists**, **sets**, or even more advanced data structures like **tries** for efficient word storage.

4. **Randomization and Pseudo-Randomness**:
   - Wordle selects a secret word randomly. Understanding **random number generation** is essential for games, simulations, and cryptography.
   - You'll encounter concepts like **seeded randomness** and **pseudo-random number generators**.
5. **Algorithmic Thinking (Feedback Mechanism)**:
   - Wordle provides feedback on guessed letters (correctly placed, misplaced, or wrong). Designing this feedback involves **algorithmic thinking**.
   - Similar thinking applies to algorithms for sorting, searching, and optimization.
6. **Text Formatting and User Interface**:
   - Using libraries like **Rich** (as in the Python Wordle clone tutorial), you'll enhance the game's appearance.
   - In real-life applications, creating an attractive user interface (UI) is crucial for user experience.
7. **Code Organization (Functions)**:
   - Wordle can be modularized into functions (e.g., checking word validity, providing feedback). Organizing code improves readability and maintainability.
   - In larger projects, well-structured functions are essential.
8. **Error Handling and Robustness**:

- Wordle handles invalid guesses gracefully. Learning to handle exceptions, edge cases, and unexpected scenarios is vital.
- Robust code ensures software reliability.

9. **Iterative Development**:
   - Wordle starts as a prototype and evolves into a polished game. This mirrors the iterative development process in software engineering.
   - You'll learn to incrementally improve your codebase.
10. **User Experience (UX)**:
    - Wordle's success lies in its engaging UX. Applying UX principles—simplicity, feedback, and responsiveness—translates to better software.
11.
```
'''The code is short, but I included three methods to
select the computer word
I also showed how to use the dictionary and how to
translate a word. In this code I am translating to
Arabic.
'''
from nltk.corpus import words
uWord = input ("enter English word ")
allWords = words.words()
if uWord in allWords:
    print("this is an English")
else:
    print("this is NOT an English")
import random
#selecting the cWord Randomley from nltk words
cWord =random.choice(allWords)
print(cWord) #!=5
while len(cWord) !=5:
    cWord = random.choice(allWords)
print(cWord)

#Having fun with dictionary
from PyDictionary import PyDictionary
dic1 =PyDictionary("fun", "name")
dic =PyDictionary()
dic1.printMeanings()
print("***************")
if dic.meaning(uWord):
    print(uWord, "is in the dictionary")
    print(dic.meaning(uWord))
```

```
    #You can also translate it :-)
    print(dic.translate(uWord, "ar"))
else:
    print(uWord, "is in not the dictionary")

#getting word list from file
fileR = open('wordleList.txt', 'r')
dataC= fileR.read()
words = dataC.split("\n")
print(words)
#end of file

#using my own list
words = ["clone", "marks", "floor"]
cWord = random.choice(words).lower()
flag = True
attempt = 1
print("The secret word is", cWord)
print("Add game rules, use functions, enter your word at
the prompt ->")

for i in range(6):
    print()

    uWord = input("->")
    #validating word length
    while len(uWord) != 5:
        print("enter a 5 letter word ")
        uWord = input("->")
    if uWord.lower() == cWord.lower():
        print("\033[2;32;48m", uWord)
        print("\033[0;30;48m,:-)")
        break
    else:
        for c in uWord:
            if c in cWord:
                if uWord.index(c)== cWord.index(c):
                   #if uWord[i]==cWord[i]:
                        print("\033[2;32;48m", c,
sep="", end="")

                else:
                        print ("\033[3;33;48m", c,
sep="", end="")

            else:
                print ("\033[0;30;48m", c, sep="",
end="")
print("\033[0;30;48m")
```

ChatBot

Simple chatbots may seem basic, but they offer valuable insights into real-life programming concepts. Let's explore how:

1. **Building Command-Line Applications**:
    - Creating a simple chatbot involves handling user input and generating responses. This mirrors building **command-line applications** where input/output handling is crucial.
2. **User Input Validation**:
    - Chatbots validate user input (e.g., understanding commands, handling unexpected messages). In programming, input validation prevents errors and ensures robustness.
3. **Data Structures (Rule-Based Responses)**:
    - Simple chatbots often use **rule-based** responses. Understanding data structures (like dictionaries or lists) for mapping inputs to predefined outputs is essential.
4. **State Management**:
    - Chatbots maintain conversation state. Similarly, real-world applications (e.g., e-commerce carts, game progress) require effective state management.
5. **Error Handling and Resilience**:
    - Chatbots handle unexpected queries gracefully. Learning to handle exceptions and edge cases applies universally in programming.
6. **Text Processing and NLP**:
    - Even basic chatbots process text (splitting sentences, identifying keywords). This aligns with natural language processing (NLP) techniques used in more advanced systems.

7. **Modularity and Functions**:
    - Organizing chatbot code into functions improves readability and maintainability. In larger projects, modular design is crucial.
8. **Feedback Mechanisms**:
    - Chatbots provide responses based on user input. Similarly, feedback loops are essential for algorithms, simulations, and AI models.
9. **Scalability and Performance**:
    - Scaling chatbots (adding more responses, handling concurrent users) parallels scaling software systems.
10. **User Experience (UX)**:
    - Chatbot interactions impact UX. Designing intuitive prompts, clear responses, and error messages applies universally.

```
11. def simple_chatbot(input_text):
        # Define rules and responses
        rules = {
            r'hello|hi|hey': 'Hello! How can I help you?',
            r'bye|goodbye': 'Goodbye! Have a great day!',
            r'how are you': 'I am a chatbot. I don\'t have
    feelings, but thanks for asking!',
            r'your name|who are you': 'I am a simple
    chatbot.',
            r'weather': 'I am sorry, I do not have
    information about the weather.',
            r'what is up': 'The sky :-)',
            r'my name': 'Susan',
            r'isa': 'Andre'
            # You can add more rules and responses as needed
        }

        # Check input against rules and provide a response
        for pattern, response in rules.items():
            if re.search(pattern, input_text,
    re.IGNORECASE):
                return response

        # Default response if no matching rule is found
        return 'I am sorry. I did not understand that.'
```

```
# Main loop for the chatbot
again = True
while again:       #True:
    user_input = input('You: ')
    if user_input.lower() == 'exit':
        print('Chatbot: Goodbye!')
        again =False          #break
    response = simple_chatbot(user_input)
    print('Chatbot:', response)
```

The previous games were introduced in an introductory programming class level CS1, and with every game, we simplified the code to the minimum to match the students' programming level. We also made sure to link it to real-life applications.

We surveyed the students about this approach, and we got the following feedback.

Figure 1. Age average

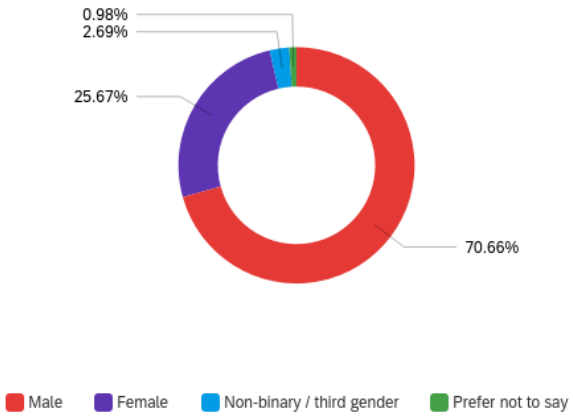

Figure 2. Gender

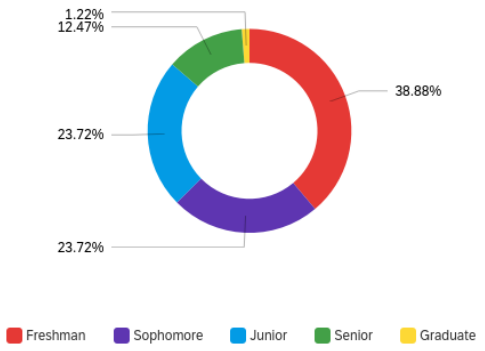

Figure 3. Academic Year

#	Field	Minimum	Maximum	Mean	Std Deviation	Variance	Count
1	Academic Year:	1.00	5.00	2.13	1.10	1.22	409

Gender	Choice Count	Choice Ratio
Male	287	70.66%
Female	103	25.67%
Non-Binary/Third Gender	11	2.69%
Prefer not to say	4	0.98%
Academic Year	**Choice Count**	**Choice Ratio**
Freshman	159	38.88
Sophomore	97	23.72
Junior	97	23.72
Senior	51	12.47
Graduate	5	1.22

Rate of confidence in programming:

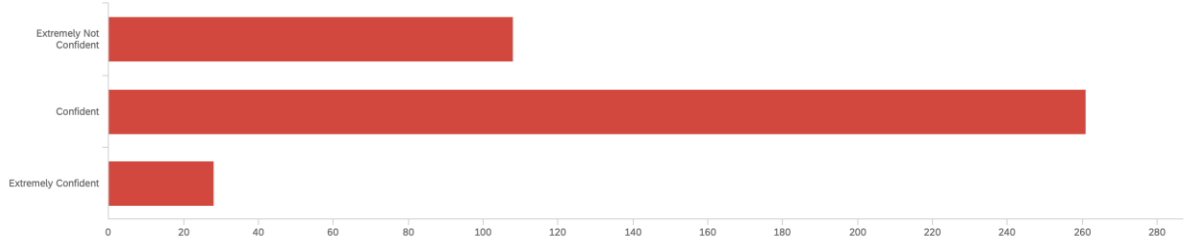

Figure 4. Programming Confidence level

Only those who answered yes to the previous question answered the next two questions.

Have your instructor(s) used game-logic to explain programming concepts?

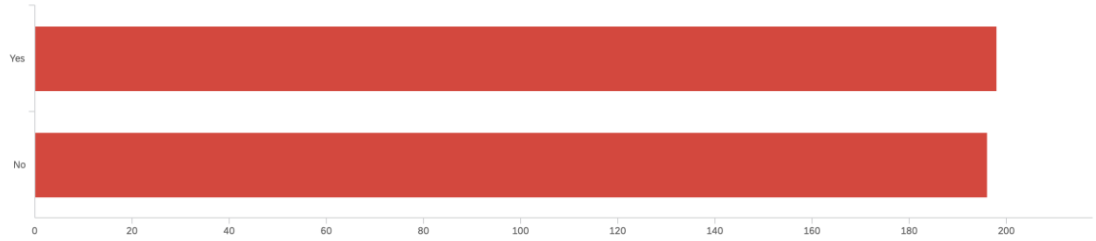

Figure 5. In-class game-logic usage.

How satisfied are you with using game-logic examples to learn programming concepts?

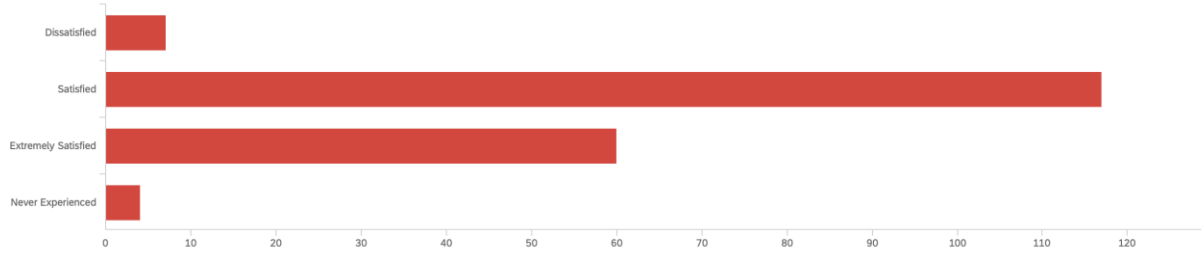

Figure 6. Game-logic satisfaction

How effective do you think using game-logic examples is in helping you understand programming concepts?

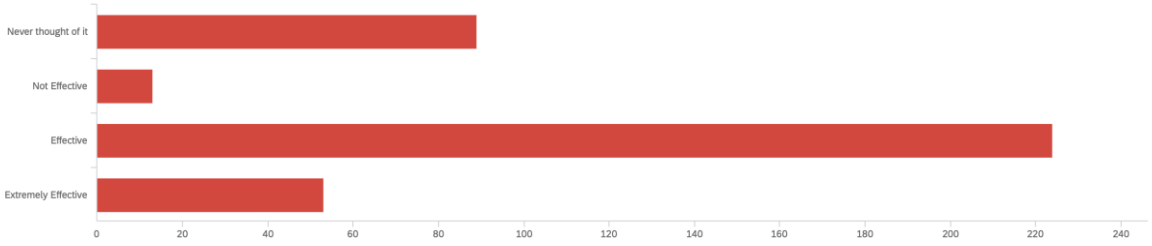

Figure 7. Effectiveness of game-logic

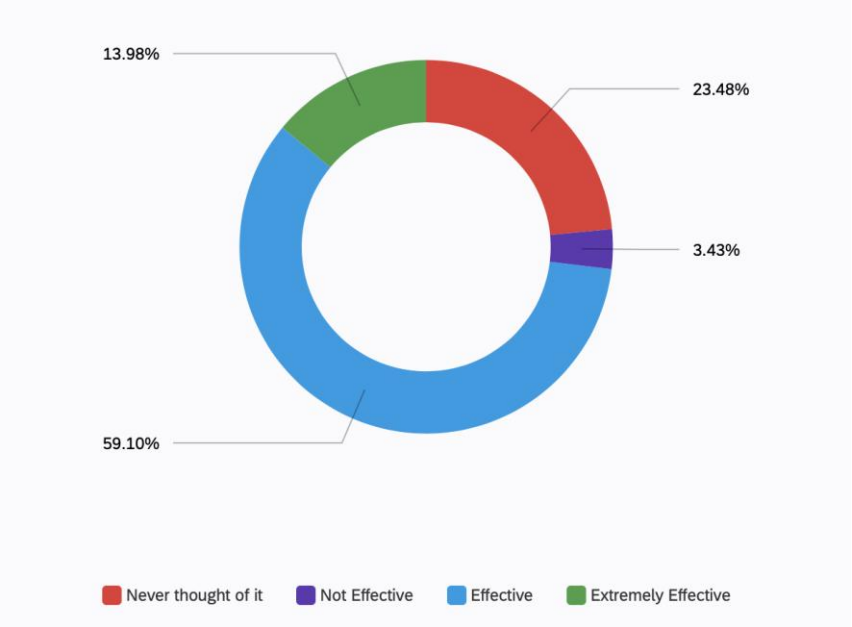

Figure 8- Another graph showing game-logic effectiveness

We asked the students if they have thought about the programming logic behind a game you are playing?

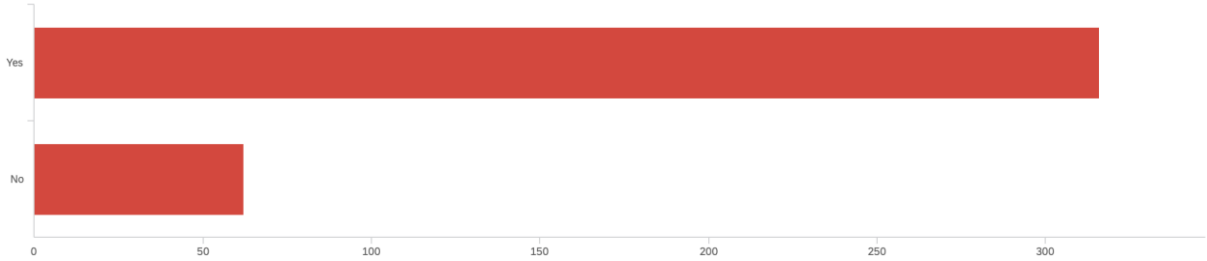

Figure 9. 83.60% of the participating students thought about programming logic behind a game playing

We asked them if they think learning how to program a game enhances their motivation to continue learning programming?

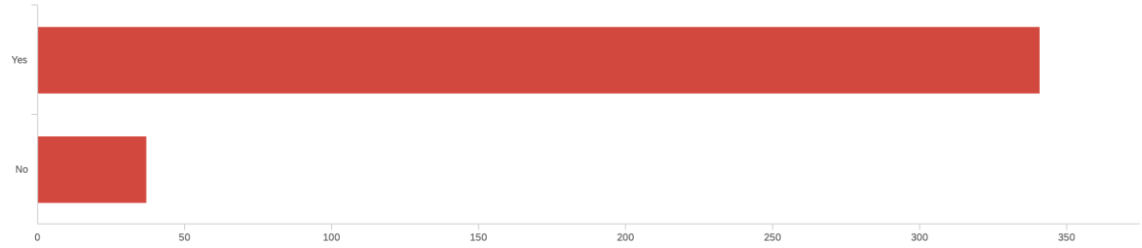

Figure 10. Learning how to program a game motivates students

It shows that 90% of the participants think that learning how to program a game enhances their motivation to continue to learn programming.

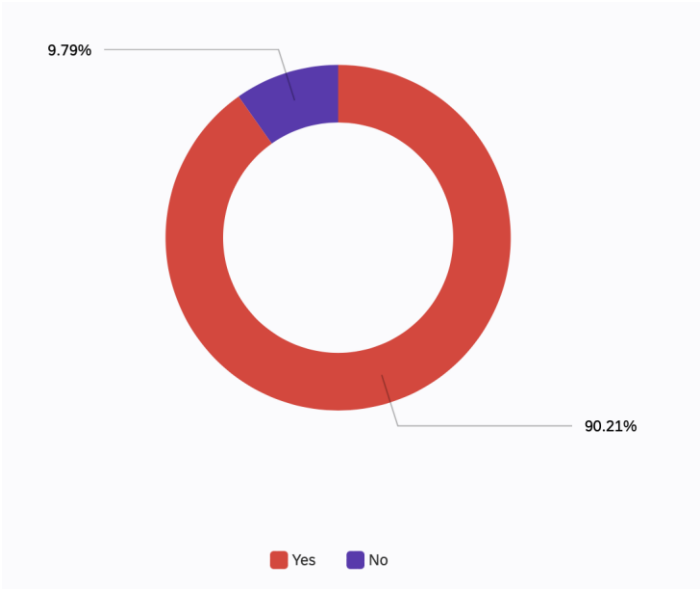

Figure 11. Pie chart showing motivation.

We asked if using game-logic examples used in class made learning programming more enjoyable?

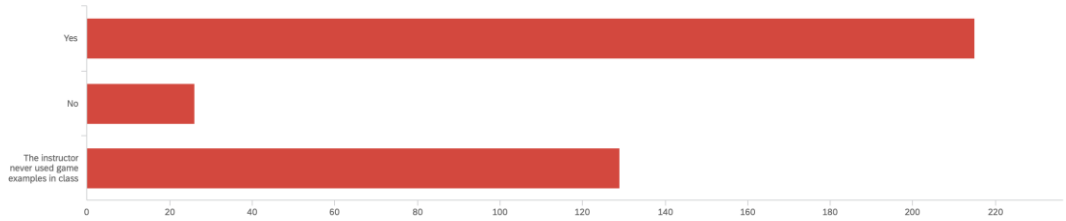

Figure 12. Game-logic examples made learning enjoyable

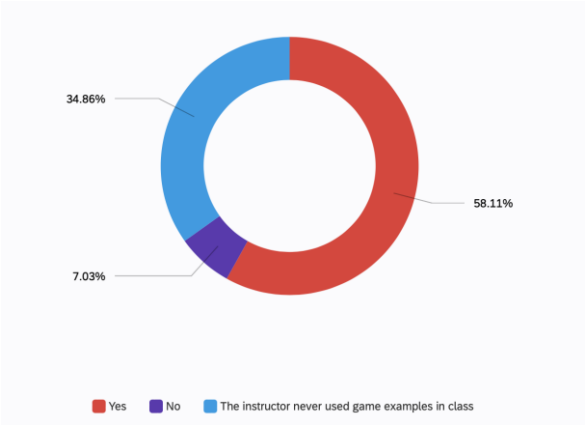

Figure 13. Pie chart representation for figure 13

We asked those whose instructor's used game-logic in class, if the instructor relates the logic behind game programming with real-life examples:

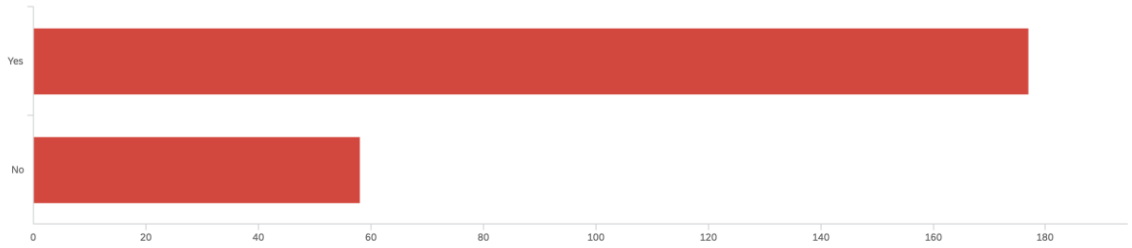

Figure 14. Game-logic & Real-life examples

We asked the students if they recommend using game-Logic to teach programming concepts to other students, and 93.97 said yes.

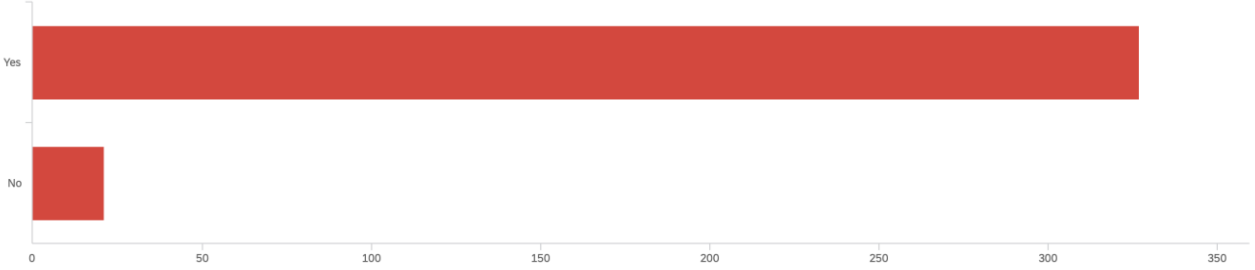

Figure 15. The answer to recommending the use of game-logic

The survey, inclusive of questions and responses, reflects the students' responses using diverse pedagogical approaches by multiple instructors within the university's CS1 course. My implementation of game logic over the past three semesters has yielded observable enhancements in student motivation, satisfaction, and class attendance. The research will be extended to evaluate the impact on student retention rates. Towards the end, I included the survey I used, and the default report.

To conclude, integrating game-logic into programming education significantly elevates the levels of student engagement, overall satisfaction, and the intrinsic motivation to delve deeper into the subject matter, thereby infusing a sense of enjoyment into the educational environment. This innovative approach has garnered increased interest among students from diverse academic disciplines, leading to a growing consideration for computer science as a minor specialization. It is anticipated that this trend will be reflected quantitatively in the form of reduced attrition rates and bolstered student retention—details of which will be presented subsequently. However, it is crucial to anchor this approach with solid real-life examples. Failing to do so may inadvertently convey the notion that programming is solely for the gaming industry, thereby overlooking the true essence of programming education: to foster critical thinking and problem-solving skills.

References

Anabela Gomes & Antonio Jose Mendes. (2007). Learning to program-difficulties and solutions. *In International Conference on Engineering Education–ICEE (Vol. 7).*

Anastasiadis, T., Lampropoulos, G., & Siakas, K. (2018). Digital game-based learning and serious games in education. *International Journal of Advances in Scientific Research and Engineering*, *4*(12), 139–144.

Boyle et al. (March 2016). An update to the systematic literature review of empirical evidence of the impacts and outcomes of computer games and serious games. In *Computers & Education* (Vol. 94, pp. 178-192). Elsevier.

Campbell, L. (2020). Teaching in an inspiring learning space: An investigation of the extent to which one school's innovative learning environment has impacted on teachers' pedagogy and practice. *Research Papers in Education* 35(2), 185–204.

Chang, JH., Wang, CJ., Zhong, HX. et al. (2023). Artificial intelligence learning platform in a visual programming environment: exploring an artificial intelligence learning model. *Education Tech Research Dev*, https://doi.org/10.1007/s11423-023-10323-z.

Joseph Gatto, Parker Seegmiller, Omar Sharif, Sarah M. Preum. (2024). https://doi.org/10.48550/arXiv.2403.03304.

Liu, Z. Y., Shaikh, Z., & Gazizova, F. (2020). Using the concept of game-based learning in education. *International Journal of Emerging Technologies in Learning (iJET), 15(14),* 53–64.

Mahmood H. Hussein; et al. (2019, july). A Digital Game-Based Learning Method to Improve Students' Critical Thinking Skills in Elementary Science. *7*(DOI: 10.1109/ACCESS.2019.2929089), 96309 - 96318.

Meihua Qian & Karen R. Clark. (2016, October). Game-based Learning and 21st century skills: A review of recent research. *Computers in Human Behavior, 63*, 50-58.

Pedró, F. (2006). THE NEW MILLENNIUM LEARNERS: Challenging our Views on ICT and Learning. *OECD-CERI*.

Prasad, A., Chaudhary, K. & Sharma, B. (3197–3223 (2022)). Programming skills: Visualization, interaction, home language and problem solving. *Educ Inf Technol 27*(https://doi.org/10.1007/s10639-021-10692-z).

Rovshenov, A. & Sarsar, F. (2023). Research Trends in Programming Education: A Systematic Review of the Articles Published Between 2012-2020. *Journal of Educational Technology & Online Learning, 6(1),* 48-81.

Sobrel, S. (2020). The first programming language and freshman year in computer science: characterization and tips for better decision making. In *Advances in Intelligent Systems and Computing* (Vol. 116, p. vol. 116). Springer.

Sónia Rolland Sobra. (2020). *The first programming language and freshman year in computer science: characterization and tips for better decision making.*

Videnovik, M., Vold, T., Kiønig, L. et al. (2023). Game-based learning in computer science education: a scoping literature review. *International Journal of STEM Education, 10*(54).

Voyager: direction for learning and careers community handbook set; book 4 Scarecrow education book. (2003). Lanham, Maryland, Scarecrow Press.

Zi-Yu Liu et al. (2020, 7 31). Using the Concept of Game-Based Learning in Education. *International Journal of Emerging Technology in Learning, 15*(14), 53–64.

About the Author

Susan Nachawati is a wife, mother of three, and a lecturer at California State University, Long Beach. She received her B.S. from the University of Damascus in Electrical Engineering. She was the first and only one to graduate from a two-year program in Biomedical Engineering in the Middle East, which was the highest degree possible in the field at that time in Syria. Coming to the United States gave her the opportunity to pursue her career in education, and she earned her M.S. in Computer Engineering from California State University, Fullerton. She also earned her Ph.D. in Applied Mathematics in Engineering from Claremont Graduate Universities. Susan is a volunteered Genexcel mentor on campus, helping first generation students discover how to unlock and achieve their potentials. She is also an educational volunteer at the Aquarium of Pacific to help support restoring the ecosystem with the best of her ability and to give back to her community.

E-mail: susan.nachawati@csulb.edu
ORCID: 0009-0002-6262-4479