Preliminary Thesis Draft

Table of Contents

I. Introduction

1.1 Background Information

1.2 CS standards

1.3 Research Problem

1.4 Research questions

1.5 Research hypothesis

1.6 Objective of the Study

1.7 Thesis Statement

II. Literature Review

2.1 Serious Games for Learning Computer Science

2.2 Gamification in Higher Education

2.3 Empirical Investigations and Systematic Reviews on Computational Thinking

2.4 Common challenges identified in introductory CS

III. Methodology

3.1 Research Design

3.2 Participants and Context

3.3 Instructional Intervention

3.3.1 Serious Game Design and Implementation

3.3.2 Mapping Game Levels to CS Conceptual Trouble Spots

3.3.3 Gamification Elements

I. Introduction

1.1) Background Information

Computational Thinking (CT) is a foundational skill in computer science education that offers students a structured and logical approach to solve problems using techniques such as pattern recognition, abstraction, and algorithm design. As undergraduate computer science programs at the higher education level continue to expand their student enrollment, many of them enter introductory Computer Science (CS) courses without having previously developed this skill. This often leads to a challenging environment where students find it difficult to understand core concepts such as algorithmic reasoning, data structures, and programming fundamentals.

Teaching introductory CS remains a persistent obstacle, particularly because many students struggle to connect abstract computational ideas with concrete problem-solving techniques. CT provides a potential bridge: by equipping students with structured approaches to understanding and breaking down problems, it may enhance their comprehension of difficult CS concepts and improve learning outcomes.

However, even when CT principles are introduced, many students continue to struggle with sustained engagement and motivation, especially when faced with abstract or technically complex topics. This has led CS educators to explore additional pedagogical strategies that make learning more interactive, intuitive, and appealing to novice learners. Among these strategies, **gamification** and **serious game development techniques** have gained increasing attention.

Gamification refers to the application of game-like elements such as goals, challenges, progression systems, and reward structures within non-game contexts in order to increase motivation and engagement. In educational settings, gamification does not turn the course into a full game. Instead, it strategically incorporates mechanisms that encourage engagement and create a sense of progression and achievement.

Serious game development techniques, on the other hand, consist of designing learning experiences that borrow from the principles of full-fledged game design. Unlike gamification, which adds isolated game elements, serious games create gaming environments with explicit educational purposes. These experiences often incorporate interactive challenges that draw on game design elements such as narrative and role-playing to mirror real-world computational or problem-solving tasks. In the context of computer science education, serious games allow students to engage with abstract concepts through immersive activities that require the use of computational thinking skills in meaningful ways.

These two approaches can potentially create a foundation that enhances student engagement and conceptual understanding when embedded within a computational thinking framework for introductory computer science education.

1.2) Computer Science Standards

Computer science education has undergone significant expansion at both the national and state level. This expansion has been formalized through the emergence of national curriculum frameworks and policy initiatives, most notably the *CS for All* movement,

which seeks to broaden access to high-quality computer science education and ensure that all students develop foundational computational skills. Central to this movement are the CSTA K–12 Computer Science Standards, which define a coherent progression of core concepts and practices that students are expected to learn throughout their academic career.

These standards explicitly emphasize many fundamental programming concepts that include: variables and data types, conditions, iteration-tracking, functions, basic data structures, as well CT practices such as abstraction, pattern recognition, and algorithmic reasoning. Many states have adopted or aligned their curricula with these national standards, formally recognizing these competencies as intended learning outcomes for all CS students. As a result, introductory computer science courses at the undergraduate level (CS1) increasingly serve as a critical point, where students are expected to master these concepts and skills.

Addressing these learning objectives are therefore not merely a matter of improving course outcomes, but a necessary step toward supporting broader educational objectives established by the *CS for All* initiative and contemporary computer science curricula.

1.3) Research Problem

Despite its known importance and recognition, computational thinking is not explicitly integrated into many introductory CS courses. Existing instructional strategies teach programming and problem-solving as their focal learning point, but without offering

students a clear or structured framework to improve their CT skills. As a result, students may learn to code, often memorizing programming syntax, but without developing a deeper conceptual understanding of the topic at hand. Many students are able to write syntactically correct programs but continue to struggle with fundamental CT concepts, limiting their ability to reason effectively about program behavior and apply their knowledge flexibly across contexts.

While prior research has explored computational thinking instruction in CS1 courses, few studies offer an approach that explicitly develops CT across multiple core programming concepts within a structured, engaging, and enjoyable learning environment. This study's approach is unique in its integration of a serious game-based platform that targets multiple CT skills simultaneously, where the framework offers students an interactive and immersive experience that goes beyond coding practice and repetition. Moreover, this study evaluates learning gains across different targeted programming concepts such as loops, conditionals, functions and data structures. By providing a comprehensive assessment of computational thinking development, it offers valuable insights for both educators and curriculum designers seeking to improve CS education.

1.4) Research Questions

In response to the research problem described above, this study seeks to investigate whether a serious game based instructional intervention grounded in computational thinking principles can improve students' conceptual understanding and reasoning in

introductory computer science. Specifically, the study is guided by the following research questions:

Q1. Does participation in a serious game–based instructional intervention lead to improved performance on assessments targeting common conceptual trouble spots in introductory computer science?

Q2. To what extent does the serious game intervention improve students' computational thinking skills compared to traditional instruction alone?

Q3. Do students demonstrate learning gains across the programming concepts targeted by the serious game (e.g., data types, loops, conditionals, functions, and data structures)?

Q4. How does repeated engagement with the game levels relate to the students' mastery of targeted programming concepts?

1.5) Research Hypotheses

Based on the research questions described in the previous section, the following hypotheses are proposed:

H1. Students who participate in the serious game intervention will demonstrate higher post-test scores on conceptual assessments than students in the control group.

H2. Students in the experimental group will show greater gains in computational thinking skills from pre-test to post-test than students receiving traditional instruction only.

H3. Students will exhibit learning gains in trouble spots targeted by game levels designed for repeated practice and mastery.

H4. There will be a positive relationship between the number of successful level completions and students' post-test performance on the corresponding conceptual assessments.

1.6) Objective of the Study

This study aims to develop and evaluate a computational thinking framework specifically designed for introductory computer science courses at the undergraduate level. The framework supported by gamification strategies and serious game development techniques seeks to strengthen students' understanding of foundational CS concepts by using CT as a guiding structure while employing game-based mechanics to boost engagement and comprehension.

1.7) Thesis Statement

The integration of a structured computational thinking framework, incorporating game-based techniques, into introductory CS courses is expected to significantly enhance students' conceptual understanding, problem-solving abilities, and overall performance relative to traditional instructional approaches.

II. Literature Review

To provide a comprehensive foundation for the proposed computational thinking framework, this literature review is organized into three thematic categories: (1) serious games for learning computer science, (2) gamification in higher education, (3) empirical investigations and systematic reviews related to computational thinking. These categories reflect the core concepts that the theoretical and practical basis of this study will be based on.

2.1) Serious Games for Learning CS

"*Sojourner Under Sabotage: A Serious Testing and Debugging Game*" by Philipp Straubinger et. al (2025)

Straubinger, Greller, and Fraser (2025) introduce *Sojourner Under Sabotage*, a browser-based serious game designed to teach software testing and debugging through an immersive narrative set aboard a sabotaged spaceship. The game was evaluated with 79 university students in a formal educational context, although the authors do not specify the exact course level or whether participation occurred during scheduled class

time or as an external activity. Players progressed through seven increasingly challenging levels using industry-relevant tools such as JUnit, debugging utilities, and coverage visualization within a secure browser-based environment. Between levels, students completed short mini-puzzles (e.g., wiring circuits, unlocking doors) to introduce variety and maintain engagement during gameplay. While the study does not explicitly state whether participants received concurrent general programming instruction, the intervention appears to supplement existing coursework rather than replace it. The authors selected testing and debugging as focal topics due to their recognized importance in software engineering education and their tendency to be perceived as monotonous or unengaging for students. Experimental data was collected through detailed game event logs, including code edits, test executions, coverage metrics, and error reports, which were downloadable in JSON format for analysis. Post-study feedback and behavioral measures indicated increased engagement, improved test coverage, and enhanced debugging effectiveness, suggesting that the game positively supports learning these traditionally challenging topics.

"*A Serious Game for Developing Computational Thinking and Programming Skills*" by Pedro Moreno-Ger et. al (2012)

Moreno-Ger and colleagues (2012) introduce *Program Your Robot*, a serious game designed to teach computational thinking and basic programming principles to beginners. The game frames programming tasks within a narrative, where players guide a robot through levels using sequences, conditionals, and loops to solve puzzles. Each level includes a learning phase in which new concepts are introduced, followed by an

assessment phase where players apply these concepts to gameplay challenges. Rather than writing text-based code, participants construct ordered lists of commands (e.g., move forward, turn left, repeat, if), reducing syntax-related barriers common for novices. This design was motivated by the need to lower entry barriers for beginners and address well-documented difficulties with early programming concepts. The system was evaluated in an initial usability study with 25 students from diverse backgrounds, although the authors do not specify participants' academic level, institutional context, or prior programming experience. The study does not appear to have been embedded within a formal course, focusing instead on usability and learner perceptions. Participants reported high enjoyment, ease of interaction, and perceived improvements in problem-solving skills; however, the absence of a control group and reliance on self-reported measures limit conclusions regarding objective learning gains.

***"Using Serious Games in Computer Science Education"* by Thomas Hainey et. al (2011)**

Hainey et al. (2011) contribute to the early literature on serious games by presenting a study using a digital game to teach requirements collection and analysis in software engineering education. In the game, students assume the role of a software engineer interacting with virtual stakeholders, navigating dialogue trees to gather essential system information. Players select questions or responses in simulated interviews, requiring them to probe carefully and think critically like analysts. The study involved university students, although the exact number, course level, and institution are not specified. Conducted within a formal course context, the game served as a

supplementary learning tool alongside traditional instruction. Requirements analysis was chosen as the target skill due to its conceptual complexity and the challenges learners face with conventional teaching methods. Students engaging with the game demonstrated statistically significant improvements in requirements-analysis skills compared to control groups and reported enhanced interest and enjoyment, consistent with Flow Theory, suggesting increased motivation and focus on learning tasks. The authors argue that well-designed educational games can effectively supplement, and in some cases partially replace, traditional instruction for complex, conceptual topics in computer science, providing a robust early exemplar of empirical game-based CS education.

***"A Serious Game for Programming in Higher Education"* by Thomas Hainey & Gavin Baxter (2024)**

Hainey & Baxter (2024) present the design, implementation, and evaluation of a serious game aimed at reinforcing programming skills among undergraduate students. Two empirical studies were conducted. The first involved a questionnaire with 61 students assessing openness to a programming-focused game, revealing high levels of acceptance and providing insights on preferred integration with course material. Based on this feedback, the game was developed to consolidate knowledge of rudimentary and advanced programming concepts, data structures, and algorithms. The second study integrated the game into a module as a revision aid before a formal class test, using a 2D interactive game world in which players explored a visually engaging

environment (e.g., a virtual campus) with embedded coding challenges. Each checkpoint presented programming problems, such as logic puzzles, syntax correction, or trace-the-code tasks, often using multiple-choice, code ordering, or debugging snippets rather than requiring full coding solutions. Forty-eight participants evaluated the game, reporting enjoyment and perceiving it as an effective revision tool, with 14% rating it “very effective” and 51% “effective,” and supporting both formative and summative uses. Conducted in a formal higher education context, the game supplemented existing instruction, helping students consolidate knowledge and prepare for assessments. Programming was chosen as the focus due to its centrality in CS curricula and the challenges students face in mastering key concepts. Hainey & Baxter’s work provides a replicable model for integrating gamified tools into higher education, demonstrating the potential of serious games to enhance engagement, motivation, and learning outcomes in programming education.

***“Combining Gamification and Intelligent Tutoring Systems in a Serious Game for Engineering Education”* by Ying Tang and Ryan Hare (2023)**

Tang & Hare (2023) present a serious game integrated with an intelligent tutoring system (ITS) designed to enhance engineering education through personalized and gamified learning experiences. The project, developed over multiple iterations based on student feedback and classroom observations, aims to provide an engaging and effective learning environment. In the game, students complete interactive engineering exercises structured as problem-solving challenges rather than traditional assignments. The ITS monitors students’ in-game actions and adapts guidance based on

performance and estimated emotional states captured via webcam images. The study involved students in classroom courses, although participant numbers, course levels, and institutional contexts were not reported. Conducted in a formal educational setting, the game supplemented standard instruction and provided adaptive support tailored to individual learners. Learning outcomes were evaluated using pre- and post-tests, classroom observations, and student feedback, with results indicating improvements in background knowledge, educational efficacy, and engagement compared to students who did not use the system. Engineering education was selected as the focus due to its conceptual complexity and the potential for adaptive, gamified approaches to support learning. The study demonstrates an innovative integration of ITS and gamification in a serious game, highlighting the value of personalized, interactive tools in formal instruction while noting that emotion-based data collection may have limitations regarding measurement accuracy.

***"Integrating Augmented Reality, Gamification, and Serious Games in Computer Science Education"* by Georgios Lampropoulos et al. (2023)**

Lampropoulos et al. (2023) evaluate the impact of integrating augmented reality (AR), gamification, and serious games into higher education computer science (CS) courses. The authors developed a mobile educational application for Android and iOS and conducted an experiment with 117 students, although participant course levels and institutions were not reported. Designed for multiple CS courses and difficulty levels, the application incorporated AR using Unity and functioned as a supplementary tool alongside standard instruction in a formal course context, reinforcing concepts and

promoting interaction. In the game, students completed tasks, puzzles, quizzes, and explored AR content, with embedded gamification features such as points, badges, leaderboards, and timed challenges to enhance engagement and motivation. Data collected via a 49-item questionnaire indicated improvements in student self-efficacy, engagement, and understanding of CS concepts. The authors selected CS education due to the potential for combining AR, gamification, and serious games to support both cognitive and socio-emotional learning outcomes. While the study demonstrates immediate positive effects, reliance on short-term feedback limits conclusions about long-term knowledge retention.

2.2) Gamification in Higher Education

"Swords and sorcery: a structural gamification framework for higher education using role-playing game elements" **by Konstantinos Ntokos (2019)**

Ntokos (2019) implements a structural gamification framework incorporating role-playing game (RPG) elements to enhance student engagement and motivation in a second-year "Engineering Software Systems" course. The study involved 34 higher education students and was conducted within a formal course context, supplementing standard instruction rather than replacing it. Students selected RPG classes (e.g., warrior, mage, rogue, priest), each with unique abilities that influenced classroom interactions, and participated in gamified activities such as experience points, badges, leaderboards, and in-class battles to incentivize participation. The study reports a 44% increase in student attendance and a 12% improvement in academic performance compared to a previous cohort that did not experience the framework. Qualitative

feedback indicated that while students found the gamified approach engaging, some suggested enhancements such as additional RPG classes and deeper integration of game mechanics. The framework was explicitly designed to target at-risk students with lower engagement, demonstrating the potential of RPG-based gamification to increase participation and motivation in higher education computer science courses, although the small cohort size limits the generalizability of the findings.

***"An Empirical Investigation of the Different Levels of Gamification in an Introductory Programming Course"* by Hazra Imran (2023)**

Imran (2023) investigates the effects of varying levels of gamification on motivation, engagement, and performance in an introductory online programming course. The study involved 450 undergraduate students who were assigned to course versions featuring zero, low, or high levels of gamification, differentiated by the number and complexity of elements such as points, badges, leaderboards, and narrative features. Conducted in a formal course context, all participants received the same programming instruction, with gamification serving as an overlay rather than a replacement for instructional content. Data were collected at pre-course, mid-course, and post-course stages using validated measures, including the Intrinsic Motivation Inventory (IMI) and the Student Course Engagement Questionnaire (SCEQ). While no significant differences were observed during the early stages of the course, post-course results showed that students exposed to high-level gamification demonstrated significantly greater motivation and academic

performance. The study highlights the importance of gamification intensity in introductory programming education, suggesting that richer gamified experiences can improve knowledge retention and sustain student motivation. The findings are strengthened by the use of validated metrics, providing reliable empirical evidence for designing effective gamified learning environments.

***"Gamification Improves Learning: Experience in a Training Activity of Computer Programming in Higher Education"* by Rafael Mellado and Claudio Cubillos (2024)**

Mellado and Cubillos (2024) investigate the impact of gamification on learning outcomes in a higher education computer programming course focused on data structures. The study involved over 200 students across multiple semesters, and was conducted in a formal course context. Participants were divided into experimental and control groups, with the experimental group completing Moodle-based exercises enhanced with gamification elements such as points, leaderboards, badges, and time-based challenges. All students received standard programming instruction, with gamification serving as a supplementary tool to reinforce learning. Pre- and post-tests, analyzed statistically, indicated that the gamified group achieved significantly higher post-test scores, demonstrating increased engagement, motivation, and learning outcomes. The authors selected a data structures course due to its conceptual difficulty and the need for enhanced student motivation, and the research is grounded in a solid methodological design using comparative metrics and statistical analysis. While the

findings provide strong evidence for the short-term effectiveness of gamification in higher education programming courses, the study does not assess long-term knowledge retention.

***"Supporting computational thinking through gamification"* by J. Isaac and S.V. Babu (2016)**

Isaac and Babu (2016) investigate the use of gamification to support computational thinking through a virtual reality programming environment called VENVI. The study addresses the challenge of making abstract computational concepts more accessible to novice learners. Forty students from a local school were randomly assigned to either a gamified or non-gamified version of the software. Conducted in a formal learning context, the gamified environment incorporated points, levels, and trophies to reward progress, signal status, and indicate achievement. In this way, fostering motivation, engagement, and active experimentation. VENVI allowed students to program a virtual character to perform different move sets in a 3D environment, reinforcing concepts such as sequences, loops, conditions, parallelization, and functions. Gamification served as a supplementary tool, reinforcing understanding without replacing instruction. Assessment after the activity indicated that the gamified group demonstrated higher engagement and improved computational thinking performance compared to the control group. The study highlights the potential of combining VR and gamification to provide an interactive,

hands-on platform that makes abstract programming concepts more accessible to novice learners.

***"Encouraging Early Mastery of Computational Concepts Through Play"* by Hannah M. Dee et al. (2020)**

Dee et al. (2020) explore the integration of play-based learning and gamification to support computational thinking in young learners. The study involved reusable workshops conducted in formal and semi-formal educational settings, designed to be replicable across classrooms and coding clubs. Students engaged in playful activities categorized by type—exploratory, creative, role play, symbolic—each targeting specific computational thinking concepts such as sequencing, abstraction, and problem decomposition. Advanced topics, including robotics and artificial intelligence, were also incorporated into gamified exercises. The workshops supplemented standard instruction, providing an interactive platform for hands-on exploration and experimentation. Qualitative feedback from educators and students indicated high engagement, motivation, and feasibility, although the study lacked extensive quantitative assessment of learning outcomes. The research demonstrates the potential of structured play to make abstract computational concepts accessible, engaging, and adaptable for young learners across diverse educational contexts with activities that are easy to adapt and replicate.

2.3) Empirical Investigations and Systematic Reviews on CT

"*Gamification and computational thinking in education: a systematic literature review"* by T. Sapounidis (2024)

Sapounidis (2024) presents a systematic literature review examining gamification and computational thinking (CT) in educational settings. The review synthesizes findings from 37 studies, analyzing methodologies, gamification elements, and reported learning outcomes. From these findings, the author emphasizes that gamification—through elements such as points, badges, leaderboards, and challenges—can enhance student engagement, motivation, and the development of computational thinking skills. While initial evidence is promising, most studies focus on short-term effects, and research on long-term learning outcomes and knowledge retention remains limited. The review also highlights that gamification provides opportunities for experimental learning approaches that make CT more enjoyable and accessible. Additionally, it identifies gaps in standardization, as educational contexts, participant ages, and gamification implementations vary widely across all studies. Sapounidis concludes that further empirical studies are needed to establish best practices and a solid theoretical

foundation for integrating gamification with CT education, particularly in formal school and higher education contexts.

"*A Systematic Review of Computational Thinking Approach for Programming Education in Higher Education Institutions"* by F.J. Agbo et al. (2020).

Agbo et al. (2020) present a comprehensive review of the application of computational thinking (CT) in higher education institutions. The authors investigate the emerging importance of CT as a fundamental skill in computer science education and its potential to help students overcome challenges in learning computer programming at introductory and intermediate levels. They distinguish CT as a problem-solving and abstraction process, while programming is framed as the concrete implementation of solutions through writing, testing, and debugging code—activities that often pose significant difficulties for novice learners. To address these challenges, the reviewed studies employ approaches such as visualization, games, puzzles, and CT-driven activities to motivate students and improve learning outcomes. The review analyzes 57 studies published between 2006 and 2019, examining CT definitions, instructional strategies, assessment techniques, and educational outcomes. Findings indicate a noticeable increase in CT-focused programming education research since 2017, reflecting growing interest and reported effectiveness of CT-based approaches. Overall, the review

highlights the transformative potential of thoughtfully integrating computational thinking into higher education programming curricula to improve students' conceptual understanding and problem-solving skills.

***"Integration of Computational Thinking Skills in STEM-Driven Computer Science Education"* by Renata Burbaitė, Vaidotas Drąsutė, and Vyacheslavas Štuikys (2018)**

The authors explore the integration of computational skills (CT) within STEM-oriented computer science education at the high school level. Recognizing CT as an essential and foundational skill in the digital age, the authors propose a model that embeds CT into STEM-oriented curriculum in order to enhance students' problem-solving skills and critical thinking. They argue that traditional CS education often falls short in properly developing essential CT skills such as algorithmic thinking, abstraction, and decomposition. Therefore, the proposed model addresses this gap by contextualizing CS concepts within real-world STEM applications, making the learning process more meaningful and relevant for students. The approach emphasizes project-based learning and hands-on activities that bridge theory and practice, supporting deeper conceptual understanding. A key contribution of this paper is the introduction of an assessment framework aligned with the revised Bloom's taxonomy, which evaluates CT skills across multiple cognitive levels, from basic knowledge acquisition to higher-order thinking. This model offers a more structured framework for educators to assess students'

development in CT, ranging from very basic knowledge, up to higher order thinking skills. Overall, the paper provides valuable insights in curriculum design and assessment strategies for developing computational and critical thinking in STEM-focused computer science education. However, the authors note that further work is needed to examine how the model can scale across more diverse educational contexts.

2.4) Common challenges identified in introductory CS

***ProgMiscon – "A Curated Inventory of Programming Language Misconceptions"* by Chiodini, L et. al (2021)**

Chiodini et al. (2021) introduce *ProgMiscon*, a curated inventory of programming language misconceptions, presented at the ITiCSE conference. The primary objective of the work is to consolidate and categorize misconceptions that have been repeatedly reported across decades of introductory computer science education research. The authors aim to provide reusable references from multiple sources that support instructional design and support further empirical investigation. The inventory organizes misconceptions by core programming concepts such as variables, expressions, and data representation. Several of the documented misconceptions relate to how values are handled in expressions, reflecting persistent difficulties with **data types** and **type** behavior across programming languages. These misunderstandings often stem from students relying on informal or mathematical intuition rather than an accurate understanding of programming language semantics. For example, learners may incorrectly assume that numeric operations behave identically to arithmetic ones

performed outside of programming contexts, which causes faulty expectations about numerical results that deal with division, precision,and truncation. These types of programs often compile and execute without errors, misleading students by reinforcing inaccurate mental models of computational behavior. The study highlights that these misconceptions are not tied to a specific language or context, but instead recur across different educational settings. From a CT perspective, these issues are closely tied to challenges in abstraction and algorithmic reasoning, as students must reason about how data is represented and then predict how those representations behave during execution. Overall, ProgMiscon provides valuable evidence about different misconceptions with introductory CS concepts, especially those related to numerical expressions and data handling. Its contribution lies in validating the recurrence of these misconceptions and reinforcing the need to address students' mental models, rather than focusing solely on syntax-level correctness in programming.

***"Students' Misconceptions and Other Difficulties in Introductory Programming"* by Yizhou Qian and James Lehman (2017)**

Qian and Lehman (2017) present an empirical study of students' misconceptions and learning difficulties in introductory programming courses, published in *ACM Transactions on Computing Educatio*n. The paper serves both as a systematic review and an analytical framework for understanding why novice programmers struggle with core programming concepts. A key contribution of the study is the identification of recurring misconceptions related to numerical expressions and variable behavior during

program execution. The authors report that students often misunderstand how values are manipulated when variables are reassigned or when expressions involve multiple operations, leading to incorrect predictions of program output even when the code is syntactically correct. The study further shows that misconceptions related to variable expressions and data handling occur consistently across different programming languages, indicating that these difficulties are conceptual rather than language specific. This finding supports the classification of **data type** confusion and **numerical expressions** as fundamental problem areas in introductory programming education. In summary, Qian and Lehman provide strong evidence that misunderstandings of numeric computation and program execution remain a major barrier for novice programmers, where their findings reinforce the need for instructional interventions that explicitly target students' mental models and computational thinking skills.

***"Misconceptions about Control Flow in Introductory Programming Courses and Their Remedy"* by Dimitri Eckert (2025)**

Eckert (2025) investigates frequent misconceptions related to control flow and conditional logic in introductory programming courses, in a study published at the ITiCSE conference. The work focuses on how students reason about conditionals, logical operators, and branching behavior in program execution. The study points out that despite explicit instruction, many students struggle to correctly predict program behavior when conditional statements and logical expressions are involved.

One of the most recurrent misconceptions concerns the evaluation of boolean expressions, particularly when multiple conditions are combined using logical operators

such as AND and OR. Students often misinterpret operator precedence and boolean operator semantics, leading to incorrect assumptions about which branches of a program will execute and their respective outcomes. These issues are further compounded in cases involving nested conditionals or complex logical expressions that control program flow.

The study draws on empirical evidence from classroom observations and targeted assessments designed to evaluate students' reasoning about conditional logic. Findings indicate that control flow misconceptions persist across different programming languages and instructional contexts. Eckert shows that these misunderstandings often remain hidden because programs involving flawed conditional logic may still compile and execute, producing output that appears acceptable to novice learners. As a result, students may create incorrect mental models of branching behavior and control flow.

Overall, Eckert provides strong empirical evidence that misconceptions about conditional logic and control flow represent a major learning challenge on introductory programming education. The study reinforces the classification of **conditional and logical operators** as key trouble spots in introductory CS courses and offers a clear rationale for developing new approaches that promote accurate control-flow reasoning.

***"Scope Rule Comprehension by Novice Python Programmers"* by Mark Holiday (2025)**

Holiday (2025) examines novice programmers' understanding of variable scope and lifetime in Python, a topic widely recognized as conceptually challenging in introductory computer science education. Published at SIGCSE 2025, the study focuses on how

beginners reason about variable behavior across different program contexts, including functions, conditional blocks, and nested scopes.

One of the most common misconceptions identified is the belief that variables exist globally by default, or that once a variable is defined, it remains accessible regardless of the context in which it appears. This misunderstanding leads students to incorrectly predict program behavior when variables are shadowed, redefined, or accessed outside their valid scope. Another recurring issue is confusion between variable naming and identity, where students assume that variables sharing the same name refer to the same memory location across different scopes. This misconception is further exposed in programs that deal with functions, where students incorrectly expect changes to local variables to affect global state, or vice versa. These errors suggest a lack of understanding of variable lifetime and scope logic rather than syntax-based confusion.

In summary, the paper provides recent empirical evidence that misunderstandings of **variable scope** and **lifetime** remain prevalent among novice programmers, thereby justifying their classification as a core trouble spot in introductory programming education. From a computational thinking perspective, these difficulties are closely associated with challenges in abstraction and reasoning about program state and execution.

***"The Fear of Loops: Analyzing Novice Students' Understanding of Iteration"* by Rogerson & Scott (2010)**

Rogerson and Scott (2010) investigate novice students' understanding of iteration and loop constructs in introductory programming courses, in a study published at the ITiCSE conference. Through classroom assessments and analysis of student feedback, the

authors identify loops as a persistent cognitive obstacle for beginning programmers. Learners frequently display difficulty predicting the number of iterations a loop will execute, tracking changes to variable values within loop executions, and interpreting correctly the relation between loop conditions and the evolving program state. These difficulties often manifest even when students recognize the correct syntax of loop structures, in this way indicating that the challenge lies in conceptual interpretation rather than syntax alone. The authors report that many students treat loops as single instances rather than as repeated processes, which often leads to incorrect predictions about how the loop behaves. For example, students may incorrectly anticipate the final value of a loop counter or may fail to recognize when a loop will terminate, most particularly in *while* loops with dynamic exit conditions. In summary, Rogerson and Scott conclude that instruction on loops needs to focus beyond syntax and involve techniques that can help students mentally simulate repeated execution of loop iterations, tracking variable changes, and evaluating loop exit conditions. The findings strongly indicate that difficulties with **loop reasoning** and **iteration tracking** represent a fundamental trouble spot in introductory CS education, in which they are closely related to challenges in algorithmic reasoning and computational thinking.

***"Misconceptions about Loops in Introductory Programming Courses"* by Eckert, Timmermann & Kautz (2022)**

The authors investigate the specific difficulties that novice programmers exhibit when reasoning about loops and iterative control structures in C++ programming language. The study focuses on undergraduate learners enrolled in a first-year programming

course and examines their responses to tasks requiring correct interpretation of program execution, specifically for loop constructs such as *for* and *while* loops. The most common misconceptions identified include skipping loop initialization, failing to account for the loop update step, and incorrect expectations about how many times a loop's body will execute. These challenges reveal that novice learners often lack accurate mental models of how iterative control structures operate during execution, particularly in reasoning about the interaction between loop initialization, conditions, and updates across multiple iterations. From a computational thinking perspective, the documented difficulties are directly tied to core CT skills such as algorithmic reasoning and tracking changing variable state over time. The authors suggest that instructional strategies which make execution sequences, loop initialization, and iterative updates explicit — such as program visualization or interactive simulation — may be effective in addressing these conceptual gaps. Overall, Eckert et al. provide contemporary empirical evidence that misunderstandings of loop behavior and iteration tracking among novice programmers remain prevalent. The study not only reinforces the importance of loop reasoning as a critical conceptual challenge, but also provides guidance for designing new instructional techniques that can clarify loop behavior and iterative computation.

***"Eliciting a Novice Programmer's Mental Model of Arrays"* by Fatema and Pérez-Quiñones (2021)**

Fatema and Pérez-Quiñones (2021) investigate how novice programmers conceptualize array data structures in introductory programming courses. Presented at SIGCSE 2021, the study provides empirical evidence on the extent to which beginners can accurately

reason about the behavior and semantics of arrays, which are among the first foundational data structures introduced in CS1. Students were presented with program snippets involving array declaration, indexing, iteration, and assignment, and were asked to explain their expectations about program behavior after execution. The findings reveal that many novice learners hold inaccurate or incomplete mental models of array structure and semantics. For example, students often confuse arrays with scalar variables, assuming that an array name refers to a single value rather than a sequence of indexed elements. Other students incorrectly believe that array indices can begin at arbitrary values or that array elements are stored non-contiguously, reflecting a lack of understanding of how memory and indexing mechanisms operate. Additional misconceptions emerge in contexts involving loops and array updates. Learners frequently predict incorrect outcomes for code that modify array contents within loops, indicating mental model issues with tracking changes to array values across iterations. These errors indicate that novices struggle not only with abstract representation of arrays as a collection of values, but also have trouble reasoning about how loops can affect those values — an issue that overlaps with loop reasoning challenges as well. In summary, this work provides empirical support for classifying data structures, specifically **arrays** and **lists** mental models, as a core problem area in introductory CS. The results demonstrate that novices' array misconceptions are not just isolated errors but are also a part of a broader challenge in developing accurate mental models for data structures, an issue that is likely to persist without targeted pedagogical intervention.

***"Introductory programming: a systematic literature review"* by Luxton-Reilly et al.**

(2018)

Luxton-Reilly et al. (2018) present a systematic literature review of introductory programming research, synthesizing approximately 15 years of studies to identify dominant themes, persistent challenges, and research gaps in CS1 education. Across the reviewed literature, the authors report that novice learners consistently demonstrate incomplete or inaccurate mental models of several fundamental programming concepts, particularly function parameter passing, and the distinction between value and reference semantics. These misconceptions persist even after students complete their introductory courses, suggesting that traditional instructional approaches often fail to foster an accurate conceptual understanding. For example, students may misunderstand how values are passed and returned from function parameters, and how data passed to a function relates to original variables in the calling context. They may not distinguish between *passing by value* –where a function receives a copy of the argument's value– and *passing by reference* –where a function can modify the caller's variable state–, leading to incorrect output expectations during program execution. The authors further argue that these misconceptions are reinforced by the fact that programming languages often do not make the mechanics of parameter handling explicit within typical source code. As a result, students try to understand their behaviors based on limited feedback, which can then reinforce incorrect computational thinking patterns. These findings highlight the need for instructional strategies that can make hidden execution mechanics and memory changes in programs more visible and understandable to new learners. In summary, the systematic review emphasizes the fact that misconceptions related to **parameter passing** and the distinction between **passing**

by value and **reference** are persistent challenges in introductory CS. The findings suggest that a more explicit focus on how data moves and changes during program execution could benefit novice learners and help strengthen their computational thinking skills.

III. Methodology

3.1 Research Design

This study employs a quasi-experimental research design to investigate the effectiveness of a serious game intervention grounded in computational thinking (CT) principles for addressing common conceptual trouble spots in introductory computer science (CS) education. The primary objective is to determine whether integrating a serious game, augmented with gamification techniques and aligned with CT principles, can lead to measurable improvements in CS1 students' conceptual understanding and problem-solving performance when compared to traditional instructional approaches.

The research design involves the comparison of two groups of undergraduate students enrolled in an introductory programming (CS1) course: a **control group**, which receives

conventional traditional instruction, and an **experimental group**, which receives the same instruction supplemented by the serious game intervention. Both groups are exposed to identical course content, learning objectives, and instructional duration, with the serious game serving as the only difference between conditions.

To evaluate learning outcomes, a **pre-test/post-test design** is employed. Prior to the intervention, all participants complete a diagnostic assessment designed to measure the understanding of selected introductory CS concepts associated with common CS trouble spots, which include: loop reasoning, conditional logic, variable scope, data structures (arrays and lists), and function parameter passing. This structure allows an objective analysis of the learning gains between both groups.

Overall, this research design is intended to provide a controlled and valid evaluation of whether a CT-driven serious game intervention can meaningfully improve learning outcomes in introductory computer science education. By grounding the intervention in empirically established trouble spots and comparing it against traditional instruction within a real classroom setting, the study aims to contribute substantial evidence to the study of research on serious games, gamification, and computational thinking in higher education.

3.2 Participants and Context

The participants in this study are undergraduate students enrolled in an introductory computer science (CS1) programming course at a higher education institution (possibly AUM). The course is a required component of the CS curriculum for all the students involved and will typically be taken by first-year students with varying levels of prior

programming experience. As it is common with introductory CS courses, the group of students will consist of diverse academic backgrounds, including those with no formal exposure to programming prior to enrollment

The study is also conducted within the context of a standard academic semester, in which the course content covers fundamental programming concepts such as variables, data types, conditional statements, loops, functions, and basic data structures. These topics align directly with the conceptual trouble spots identified in the literature review and targeted by the serious game intervention developed for this research.

Demographic data for students, including prior programming experience and self-reported familiarity with programming concepts, are collected at the beginning of the study to contextualize the sample and support subsequent analysis. Moreover, participation in the study is voluntary, and all students are informed about the research objectives and procedures prior to data collection.

3.3 Instructional Intervention

This section introduces the serious game as the core instructional framework used in the study and explains how it implements computational thinking principles along with gamification elements to address the known CS1 trouble spots.

3.3.1 Serious Game Design and Implementation

The instructional intervention developed for this study consists of a serious game designed to reinforce computational thinking (CT) skills in introductory computer science education. The game is intended to function as a supplementary learning tool, used

jointly with traditional instructional teaching such as lectures and laboratory sessions. Its primary goal is to provide students with an interactive environment in which they can engage with fundamental programming concepts through problem-solving activities within the educational game.

The serious game is implemented using Unreal Engine 5 software, a modern game development platform that supports the creation of immersive, interactive 3D environments and flexible gameplay mechanics. The game is designed as a single-player experience, allowing students to progress independently at their own pace. Moreover, in order to promote and sustain student engagement, the game draws inspiration from multiple game design genres such as third-person platformers, first-person shooters, and racing-style mechanics. These genre elements are not employed for entertainment alone but are adapted to serve pedagogical objectives by framing computational problems within interactive scenarios during gameplay.

The overall structure of the game is organized into a series of discrete levels, each representing a different learning experience. This level-based organization allows students to focus on specific conceptual challenges without being overwhelmed by multiple topics simultaneously. Furthermore, the progression through the game is non-linear where appropriate, allowing learners to revisit previously completed levels and reinforce their understanding as needed. Objectively, each level is designed to align with one or more conceptual trouble spots commonly encountered in introductory programming courses, as identified through the literature review. These trouble spots

also correspond directly to competencies defined in national and state CS standards, so reinforcing their concepts through each level is a learning priority for the game.

Moreover, from a pedagogical perspective, the serious game emphasizes computational thinking over programming syntax for learning goals. Instead of requiring students to write code, the game tasks are designed to help learners reason about program behavior, identify patterns, predict outcomes, and apply computational concepts in interactive contexts. This approach encourages students to develop accurate mental models of computation that can later be transferred to formal programming tasks encountered in the course lectures and laboratory exercises.

Interaction within the game is centered on problem-solving activities that require students to make decisions, interpret feedback, and reflect on the consequences of their actions. In this way, the game provides immediate feedback in response to player decisions, allowing students to identify misconceptions and adjust their reasoning accordingly during gameplay.

Additionally, to motivate students to master their understanding of each of the topics at hand, they are allowed to re-attempt game levels an unlimited number of times. This design choice encourages experimentation and reduces the fear of failure, enabling them to learn from mistakes and refine their understanding through repeated practice. Performance within each level is evaluated based on the accuracy and efficiency of the player’s decisions, with improved performance indicating a deeper conceptual understanding of the underlying programming concepts. This iterative engagement

model supports self-paced learning and accommodates students with varying levels of prior experience.

3.3.2 Mapping Game Levels to CS Trouble Spots

The serious game intervention is structured around a set of interactive levels, each explicitly designed to address one or more conceptual trouble spots commonly encountered by students in introductory computer science courses which were identified through the literature review. The game aims to promote conceptual understanding through experiential learning by mapping individual gameplay mechanics from each level to specific computational thinking challenges that address these trouble spots.

Upon launching the game, students are placed in a central lobby environment that functions as a navigational hub. Within this lobby, a series of portals are presented, each corresponding to a different level of the game. Then, the player can navigate to the entrance of each portal, where a brief description of the associated level is displayed including the targeted programming concepts and learning objectives. This design allows students to make informed decisions about which concepts to engage with and supports self-directed learning. Once a portal is selected, the corresponding level is loaded as a distinct game map.

First, **Level 1: Data Types and Type Recognition**, addresses misconceptions related to data types, type identification, and type mixing. It is implemented using a first-person shooter (FPS) game mechanic, where the player is placed in an arena populated with multiple targets representing different data types (e.g., `boolean`, `character`,

`integer`, `double`, and `string`). At any given moment, the player is assigned a current objective indicating which data type should be selected or shot at. The player must correctly identify and shoot only the targets that correspond to the active data type objective. However, objectives change periodically (e.g., every 30 seconds), requiring players to repeatedly distinguish between data types under time constraints. To successfully complete the level, the player must correctly identify and shoot a minimum number of targets for each data type. Also, incorrect selections are recorded as errors through an in-game error counter, which serves as a performance metric for the level. This design encourages accurate type recognition and reinforces the importance of distinguishing between different data representations, directly addressing common misconceptions related to implicit casting and type confusion.

Second, **Level 2: Loop Reasoning and Iteration Tracking**, focuses on loop reasoning, iteration control, and loop termination conditions. It is implemented as a racing-style game set on a circular track, simulating the repetitive nature of loop execution. A `while` loop condition is continuously displayed in the game's user interface (UI), along with the current values of the variables involved in that condition.

As the player navigates the track, they interact with collectibles that modify the values of the displayed variables. The player's objective is to strategically alter these values so that the loop condition displayed in the UI evaluates to `false`, allowing the race to terminate and the finish line to be crossed. If the condition remains `true`, the player continues looping indefinitely around the track. This mechanic makes loop semantics explicit and tangible, helping students visualize the relationship between variable

updates, condition evaluation, and loop termination which is an area that novice programmers frequently misunderstand.

Third, **Level 3: Variables and Conditional Logic,** targets difficulties related to variable state tracking and conditional reasoning. It is designed as a third-person platformer in which players traverse a series of platforms, encountering code snippets displayed within the environment where each snippet presents a short program segment involving variables and conditional statements.

Then, at each checkpoint, players must select the correct outcome or evaluation of the code from a set of multiple-choice options. Incorrect selections result in in-game penalties, such as damage inflicted by enemy drones which reduces the player's health. The objective is to progress through the level while minimizing damage, thereby inducing careful reasoning about variable values and conditional execution outcomes. This level emphasizes mental simulation of code execution, a key computational thinking skill often underdeveloped in novice learners.

Finally, **Level 4: Functions, Parameter Passing, and Lists**, builds upon the game mechanics introduced in Level 3 while addressing more advanced conceptual trouble spots, including function parameter passing, and list or array manipulation. Using the same third-person platforming framework, players encounter more complex code snippets that incorporate function calls, arguments passed by value or reference, and operations with list-based data structures such as arrays.

Players are again required to reason about program behavior and select correct outcomes from the multiple-choice options given. This level is designed to strengthen students' mental models of function execution and data structure manipulation, which are frequently cited as persistent sources of misconception in introductory programming courses.

In conclusion, these levels together form a structured progression of learning experiences that align gameplay mechanics with specific computational thinking challenges. The serious game aims to provide targeted practice on known trouble spots while maintaining high levels of engagement by combining conceptual reasoning tasks with interactive game environments.

Finally, to provide a concise overview of how each game level aligns with the conceptual trouble spots identified in the literature, Table X summarizes the mapping between targeted CS concepts, the common challenges associated with them, and the corresponding gameplay mechanics used in the serious game.

Game Level	Trouble Spot	Challenge explained

Level 1: Data Types	**Data type confusion and type mixing**	-Students mis-predict results of expressions that mix types or misunderstand implicit conversion/coercion.
Level 2 : Loops	**Loop termination and iteration tracking**	-Students struggle not only with off-by-one errors, and mentally tracing loop executions, predicting variable changes across iterations, and understanding termination conditions
Level 3: Variables and Conditionals	**Variable scope and lifetime, conditional evaluation**	-Confusing local vs global variables; assuming variables persist outside their block; misunderstanding variable lifetime. -Misunderstanding logical operator precedence; misunderstanding short-circuiting (AND/OR) and conditional branching logic (IF/ELSE).
Level 4: Functions and Lists	**Function parameter passing, array/list mental models**	-Believing function parameters always modify originals; misunderstanding call-by-value vs call-by-reference. -Students form incorrect mental models of arrays/lists leading to indexing errors

3.3.3 Gamification Elements

The serious game integrates targeted gamification elements designed to support mastery-oriented learning rather than competitive or speed based performance. The primary goal of these elements is to encourage repetition and conceptual accuracy when engaging with the core programming concepts at hand and with computational thinking skills.

A central gamification mechanism employed in the game is a badge-based mastery system, where each game level is associated with a unique achievement badge that

represents mastery of the conceptual trouble spot addressed in that level. A badge is awarded only when a player completes a level with perfect performance, defined as successfully completing all required objectives without committing conceptual errors during gameplay. This design choice places emphasis on accuracy and understanding, rather than speed or completion time, reinforcing careful reasoning and deliberate problem solving.

Upon returning to the game's central lobby or navigational hub, players are able to visualize which level (and by extension, which conceptual trouble spots) have been mastered based on the badges earned. This provides learners with immediate feedback on their progress and helps them identify levels that may require further practice or attempts. In this way, the badge system encourages students to reflect on their understanding and make informed decisions about which levels to revisit. Furthermore, players may re-attempt any level an unlimited number of times with no penalties imposed for incorrect decisions or failed attempts. This design choice reduces fear of failure and motivates students through persistence, allowing them to learn from mistakes and refine their mental models through repeated engagement. In other words, the game fosters a low-stakes environment for learning and conceptual growth, by removing pressure associated with grades or time constraints.

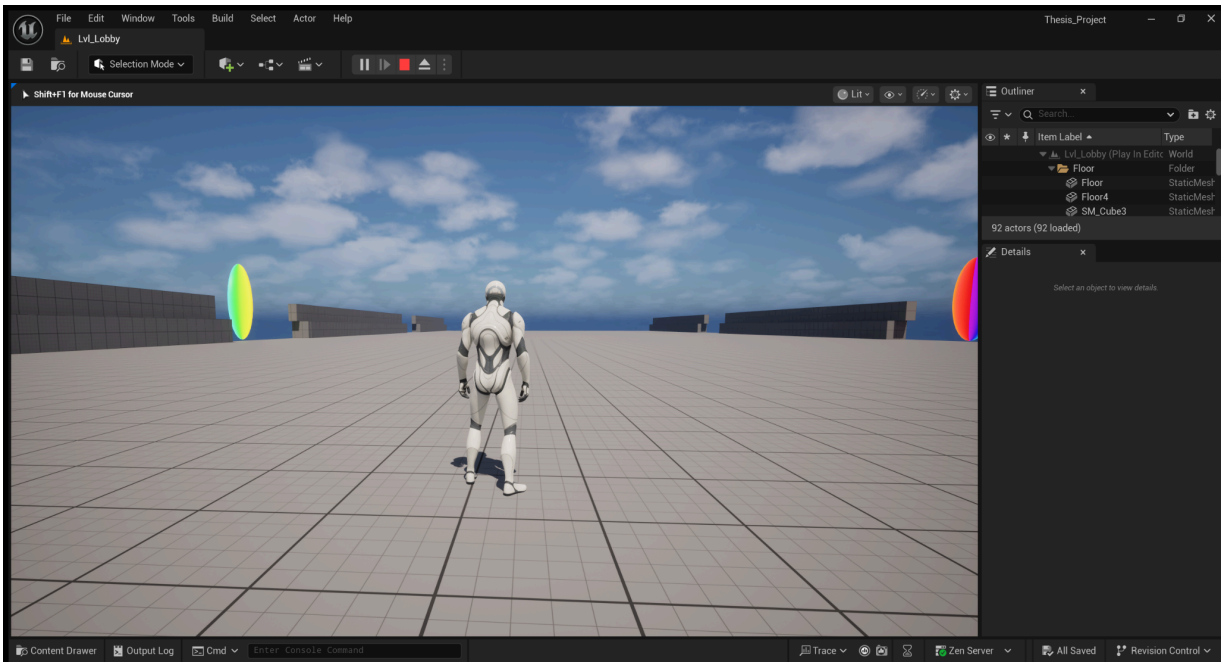
Overall, the gamification elements implemented in the serious game are aligned with principles of mastery learning and motivation. Instead of relying on leaderboards or competitive ranking, the design focuses on personal progress, conceptual accuracy, and persistence. This approach ensures that gamification serves as a pedagogical

support mechanism that reinforces computational thinking development and deeper understanding of introductory computer science concepts rather than distracting away from them.

Development Snapshots

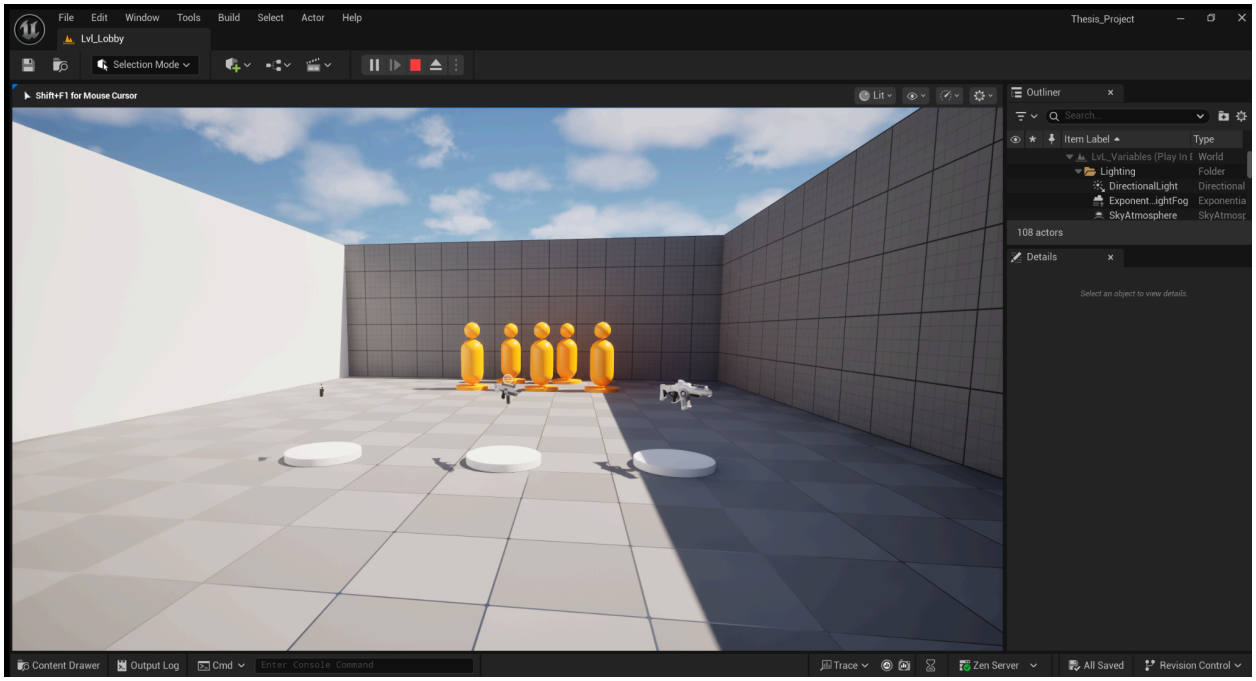
Lobby)

The player starts in the lobby area where they must choose a portal/level to travel to

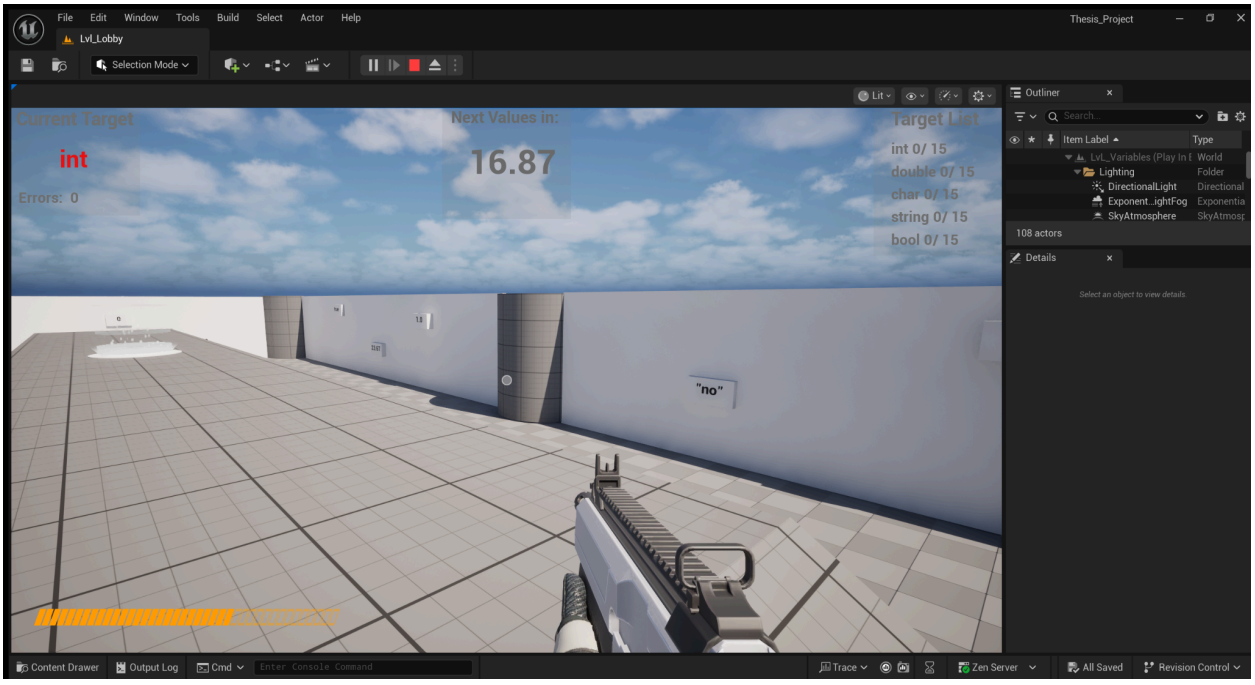


Level 1) Datatypes

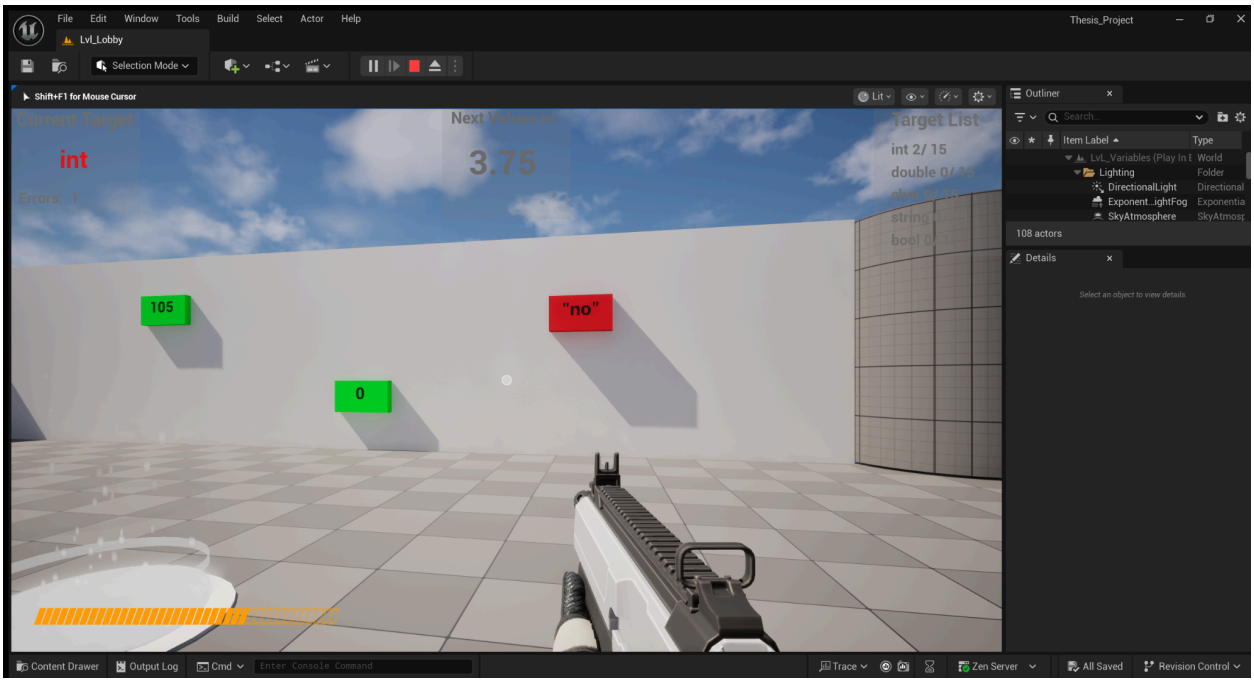
The player enters the level and picks up a weapon of choice



They explore the arena where the target values are located within the walls

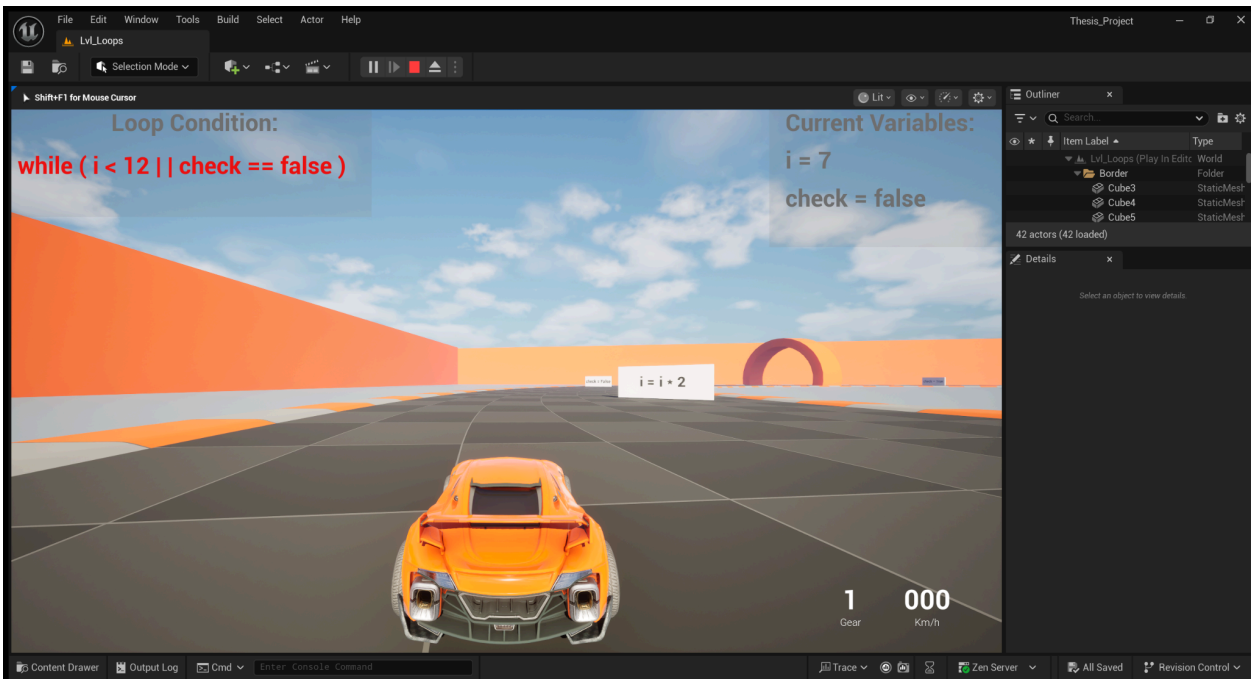


The player shoots at the target values, and depending if the datatype target was correct or not, it will display on green or red respectively



Level 2) Loops

The player must race around the circular track while interacting with variable collectibles



After changing the current variables correctly and evaluating the loop condition to FALSE, they are able to end the race