GAME DEVELOPMENT ENVIRONMENT FOR LEARNING WITH LIVE-PROGRAMMING AND TIME-TRAVEL

Katsarakis Alexandros

Thesis submitted in partial fulfillment of the requirements for the Masters’ of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes, Heraklion, GR-70013, Greece

Thesis Advisor: Prof. Anthony Savidis
UNIVERSITY OF CRETE
COMPUTER SCIENCE DEPARTMENT

GAME DEVELOPMENT ENVIRONMENT FOR LEARNING WITH LIVE-PROGRAMMING AND TIME-TRAVEL

Thesis submitted by
Katsarakis Alexandros
in partial fulfillment of the requirements for the Masters' of Science degree in Computer Science

THESIS APPROVAL

Author:
Katsarakis Alexandros

Committee approvals:
Anthony Savidis
Professor, Thesis Supervisor
George Papagiannakis
Associate Professor, Committee Member
Dimitrios Grammenos
Principal Researcher, Committee Member

Department approval:
Polyvios Pratikakis
Assistant Professor, Director of Graduate Studies

Heraklion, September 2021
Abstract

Serious games are video games primarily designed for learning purposes, with related applications and positive influence in various fields including primary education, health and engineering. Serious games for programming encourage players to use mainly algorithmic mechanics to carry out game tasks and complete missions. Such mechanics may be formally algorithmic, or not theoretically linked to strict programming principles in some circumstances, but they are nevertheless classified as programming-related games. We present a novel serious game in which the main goal is to develop a typical 2D game.

In this context, live programming refers to the capability to alter the program while it is still running by reprogramming on-the-fly itself. Technically, this is not related to self-modifying programs, as the modification logic is externally originated, either by the programmer or even by another third-party system. In terms of development life-cycle, the traditionally separated editing, compilation, linking and execution loops become a single running mode, thus providing an interactive experience with immediate feedback and effect on program changes.

The development of games is overall a very demanding process even for simple cases, generally organized as a simulated universe (game terrain) with animated objects (game characters) that interact with each other, while the simulation pace is regulated with successive timed game loops. Both the behavior of game characters and the reaction of the game terrain to various events are programmed, making instant change and testing a powerful tool, especially to newcomers and junior programmers.

In this thesis, we present a novel learning-oriented interactive development environment supporting rapid visual programming of typical 2D games. We put particular emphasis in providing an integrated environment where typical game asset editing, such as character animation and game terrain properties, are effectively interleaved with more demanding aspects like game scripting, immediate programming and iterative testing. In particular, we enable instant scripting of any pickable game object with a live visual programming editor, with all
actions taking place during gameplay, essentially removing any boundaries between editing, implementation and testing phases. Even while the game is played, new game objects may be created and modified, by attaching new attributes and events, and adding custom event handlers. Object attributes may be updated on-the-fly by the user either interactively, or by adding new visual code causing such modifications.

Finally, in our developing environment, time may be freely rewinded and replayed, undoing or redoing any relevant user actions and game state modifications internally, providing a unique live debugging facility. It is possible to drop the entire history onwards, from any point in time, and continue from there throughout such sessions. With a video player inspired user interface, time-travel is more familiar and debugging with it becomes far easier to manage. We have created as a case study a remake of the classic Super Mario video game, to demonstrate the process of developing a game from scratch while utilizing all new features.
ΠΕΡΙΒΑΛΛΟΝ ΑΝΑΠΤΥΞΗΣ ΠΑΙΧΝΙΔΙΩΝ ΓΙΑ ΜΑΘΗΣΗ ΜΕ ΜΗΧΑΝΙΣΜΟΥΣ ΑΜΕΣΟΥ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ ΚΑΙ ΚΙΝΗΣΗΣ ΣΤΟ ΧΡΟΝΟ
Περίληψη

Τα σοβαρά παιχνίδια είναι ηλεκτρονικά παιχνίδια όπου έχουν κυρίως σχεδιαστεί για διδακτικούς σκοπούς και είναι εφαρμοσμένα με θετικές επιδράσεις σε διάφορους τομείς, συμπεριλαμβανομένου τον τομέα της βασικής εκπαίδευσης, τον τομέα της υγείας και των μηχανικών. Τα σοβαρά παιχνίδια για τον προγραμματισμό ενθαρρύνουν τους παίκτες να χρησιμοποιήσουν κυρίως αλγοριθμικούς μηχανισμούς για να φέρουν εις πέρας τους στόχους του παιχνιδιού και να ολοκληρώσουν τις αποστολές. Τέτοιοι μηχανισμοί μπορεί να είναι αυστηρά αλγοριθμικοί ή ακόμα και όχι άμεσα συνδεδεμένοι με θεμελιώδεις προγραμματιστικές αρχές σε αρκετές περιπτώσεις, αλλά εντούτοις να είναι κατηγοριοποιημένα ως σχετικά με τον προγραμματισμό παιχνίδια. Εμείς θα παρουσιάσουμε ένα νέο σοβαρό παιχνίδι όπου θα έχει ως βασικό στόχο την ανάπτυξη τυπικών παιχνιδιών δύο διαστάσεων.

Σε αυτό το γενικό πλαίσιο, ο ζωντανός προγραμματισμός αναφέρεται στην δυνατότητα αλλαγής του προγράμματος ακόμα και όταν αυτό το πρόγραμμα εκτελείται επανα-προγραμματίζοντας τον εαυτό του κατά την εκτέλεση. Τεχνικά, αυτός δεν θα πρέπει να συγχέεται με τα αυτο-τροποποιητικά προγράμματα, καθώς οι αλλαγές στην λογική προέρχονται εκτός του προγράμματος ειτέ από τον προγραμματιστή ή ακόμα και από ανεξάρτητο εξωτερικό σύστημα. Όσον αφορά τον κύκλο ζωής της ανάπτυξης, η παραδοσιακά χωρισμένη σε διόρθωση, συγκέντρωση, σύνδεση και εκτέλεση επανάληψη γίνεται μια μοναδική εμπειρία με άμεση ανατροφοδότηση και αποτέλεσμα σε αλλαγές του προγράμματος.

Η ανάπτυξη των παιχνιδιών είναι συνολικά μια πολύ απαιτητική διαδικασία ακόμα και για απλές περιπτώσεις, οργανωμένα σαν ένα τεχνητό σύμπαν (έδαφος παιχνιδιού) και κινούμενα αντικείμενα (χαρακτήρες του παιχνιδιού) που αλληλεπιδρούν μεταξύ τους, και όπου ο ρυθμός ελέγχεται από επαναλαμβανόμενες χρονομετρημένες επαναλήψεις του παιχνιδιού. Αμφότερες, η συμπεριφορά των χαρακτήρων και η αντίδραση του περιβάλλοντος σε διάφορα γεγονότα
είναι προγραμματισμένα, κάνοντας άμεσες αλλαγές, βοηθώντας ειδικότερα νεοφερμένους και αρχάριους προγραμματιστές.

Σε αυτή την εργασία, θα παρουσιάσουμε ένα νέο διδακτικό διαδραστικό περιβάλλον ανάπτυξης που υποστηρίζει γρήγορο οπτικό προγραμματισμό για τυπικά διαδιάστατα παιχνίδια. Δόθηκε ιδιαίτερη έμφαση στο να παρέχουμε ένα περιβάλλον στο οποίο τυπικά στοιχεία των παιχνιδιών όπως κινήσεις χαρακτήρων και ιδιότητες του περιβάλλοντος συνυπάρχουν με πιο απαιτητικά στοιχεία όπως άμεσο προγραμματισμό αποφυγόντας επιλεγόμενου αντικειμένου του παιχνιδιού με έναν ζωντανό οπτικό τροποποιητή, με όλες τις ενέργειες να λαμβάνουν χώρα κατά την διάρκεια του παιχνιδιού, και ουσιαστικά αναιρώντας τις διαφορές μεταξύ των φάσεων τροποποίησης, υλοποίησης και του ελέγχου. Ακόμα και όταν το παιχνίδι παίζεται, νέα αντικείμενα μπορούν να δημιουργηθούν και να αλλάξουν, προσθέτοντας νέα χαρακτηριστικά, γεγονότα, και ειδικούς χειριστές. Τα χαρακτηριστικά των αντικειμένων μπορούν να ανανεωθούν κατά την διάρκεια της χρήσης είτε διαδραστικά είτε προσθέτοντας έναν νέο οπτικό κώδικα που θα προκαλέσει αυτές τις αλλαγές.

Τέλος, στο περιβάλλον ανάπτυξης παιχνιδιών που προσφέρουμε, ο χρόνος μπορεί ελεύθερα να γυρίσει πίσω και να προχωρήσει μπροστά, να ανακληθούν και να επαναληφθούν σχετικά γεγονότα όπως και η εσωτερική κατάσταση του παιχνιδιού, δίνοντας μια διαφορετική λειτουργία αποσφαλμάτωσης. Προσφέρεται επίσης η δυνατότητα να διαγράψουμε το ιστορικό εγγραφών, από οποιαδήποτε στιγμή και να συνεχίσουμε από εκείνο το σημείο. Με διεπαφή εμπνευσμένη από την συσκευή αναπαραγωγής βίντεο, το ταξίδι στον χρόνο έρχεται πιο κοντά στον χρήστη και κάνει πιο εύκολη την αποσφαλμάτωση. Δημιουργήσαμε μια περίπτωση μελέτης ανα-κατασκευάσμα το κλασικό παιχνίδι Super Mario, για να δείξουμε την διαδικασία ανάπτυξης ενός παιχνιδιού από την αρχή με την χρήση όλων των νέων χαρακτηριστικών.
Acknowledgements

First and foremost, I would want to express my gratitude to Prof. Anthony Savidis, my supervisor, for his assistance and support throughout both my Bachelor and Master's studies. Providing useful knowledge on how to conduct research on a topic as well as how to put a concept into action, which was significant in the completion of my thesis work. Also, I would want to express my gratitude to each member of the PLATO laboratory for creating a welcoming environment in which I was able to overcome any problems that arose.

I am grateful to my family for their support throughout my studies; without them, I would not have been able to complete this work. I especially want to thank my brother Antonis for his valuable input throughout my studies. I would also like to thank Giota for her constant support throughout this period. Finally, I would like to express my gratitude to all of my friends for their support and being there for me in every situation.
στην οικογένεια μου
Contents

Contents ................................................................................................................................. 13
List of Figures ........................................................................................................................ 16
List of Tables .......................................................................................................................... 18
Introduction ............................................................................................................................ 19
  1.1 Background ...................................................................................................................... 19
    1.1.1 Teaching Programming with Tools ........................................................................ 19
    1.1.2 Visual Programming Languages ........................................................................... 20
    1.1.3 Live Programming .................................................................................................. 20
    1.1.4 Serious Games ......................................................................................................... 21
    1.1.5 Reverse Debugging ................................................................................................ 21
  1.2 Problem Definition ......................................................................................................... 22
  1.3 Contributions ................................................................................................................ 23
Related Work .......................................................................................................................... 25
  2.1 Serious Games for Programming .................................................................................... 25
  2.2 Live Editing .................................................................................................................... 27
  2.3 Visual Programming for Learning .................................................................................. 28
  2.4 Time Travel in Games .................................................................................................... 29
System Overview and Architecture ....................................................................................... 31
  3.1 System Macro-Architecture ............................................................................................ 34
    3.1.1 Engine ..................................................................................................................... 35
    3.1.2 Actions ................................................................................................................... 36
    3.1.3 User Interface ......................................................................................................... 37
  3.2 Game Object System ....................................................................................................... 38
    3.2.1 Properties ............................................................................................................... 42
    3.2.2 Methods ................................................................................................................ 45
    3.2.3 Built-in Objects ...................................................................................................... 46
3.3 Game Logic Scripts

3.3.1 Hierarchical Organization

3.3.2 Visual and Textual Scripts

3.4 Game State System

3.4.1 State Capture

3.4.2 State Restore

3.5 User Interface

3.5.1 Live Property Sheets

3.5.2 Visual Scripts Workspace

3.5.3 Time Travel Controller

3.5.4 Live Game Assets

3.5.5 Live Options Menu

Live Development

4.1 Live Programming

4.2 Live Editing

4.3 Live Collections

4.4 Live Animation Creation and Editing

Time Travel

5.1 State Deltas

5.2 State Snapshots

5.3 Snapshots vs Deltas

5.4 Dealing with Commands

5.5 Chained Recordings

5.6 Time Travel in Testing

Case Study

6.1 Incremental Development Process

6.2 Basic Terrain and Game UI

6.3 Player and Characters

6.4 Game Logic

Conclusion and Future Work
List of Figures

Figure 1: Example of single coding area with many scripts .......................................................... 22
Figure 2: System’s overview ........................................................................................................... 31
Figure 3: Macro-architecture overview .......................................................................................... 34
Figure 4: Architecture composure of Engine component and the main managers ....................... 35
Figure 5: Object Class Hierarchy .................................................................................................. 39
Figure 6: Renderer’s Architecture ................................................................................................. 40
Figure 7: ObjectManager’s Architecture ....................................................................................... 40
Figure 8: States of an object and scripting transitions between states ......................................... 42
Figure 9: Attributes of an object and script reaction on attribute change ..................................... 43
Figure 10: Flags of an object and scripting on flag change behavior ............................................. 44
Figure 11: Events of an object and scripting behavior when triggered .......................................... 45
Figure 12: Collisions of an object and scripting behavior when collide with given object .......... 45
Figure 13: Scripting area with visual programming on the left and JavaScript on the right ......... 47
Figure 14: ScriptingManager Architecture .................................................................................... 48
Figure 15: Visual blocks sample to interact with our system’s API ............................................... 49
Figure 16: Live property sheet user interface ................................................................................. 53
Figure 17: Visual scripts workspace .............................................................................................. 54
Figure 18: Time-Travel Controller User Interface ............................................................................ 55
Figure 19: Live game asset user interface ...................................................................................... 56
Figure 20: Live options menu user interface .................................................................................. 57
Figure 21: Preview of currently registered (created) animations .................................................... 61
Figure 22: Edit current values of already registered(created) animation ....................................... 62
Figure 23: Time travel example with 487 frames and currently previewing frame 361 extra information ........................................................................................................................................ 63
Figure 24: Chained recording timeline and usage with the stack properties .................................. 67
Figure 25: Time travel usage pipeline and intended use ................................................................. 68
Figure 26: JSON with information regarding the bounding box of the films .................................. 70
Figure 27: Game are after the terrain and a simple game UI was introduced ............................... 71
Figure 28: Preview of all the loaded films from the configuration file ............................................ 82
Figure 29: Animation creation pane that the user can choose and create new animations ............ 83
Figure 30: On the left side we can see the User Interface used for preview and editing purposes. On the right side we can see two ways to take an object snapshot ........................................... 84
Figure 31: Scene snapshot User Interface for scene swap ............................................................... 85
Figure 32: Two ways to copy an object .......................................................................................... 85
Figure 33: Renaming process using provided UI. .......................................................... 86
Figure 34: Before a new flag data is added there is no “on extra change” after it is added a new script showed up. ........................................................................................................ 87
Figure 35: Live grid preview (red overlay over the solid objects) top without active preview bottom with. ........................................................................................................ 88
Figure 36: Search result that shows objects and objects that contain attribute. ................. 89
Figure 37: Search within a popup for faster indexing. ..................................................... 89
Figure 38: Demonstration of why moving User Interfaces is important............................ 90
Figure 39: Enabled and disabled User Interfaces that the user is able to toggle. .............. 91
Figure 40: Guidance prompt to help the user achieve a certain goal. ............................... 92
Figure 41: Examples of editor translation. ...................................................................... 92
Figure 42: Scaling example with a reduced screen width. ............................................... 93
Figure 43: HTML Iframe injected as game object (with all the properties and methods) ...... 94
List of Tables

*Table 1:* Important information to get from game state according to the usage. .................. 50
*Table 2:* Deltas vs Snapshots on important metrics. .............................................................. 65
Chapter 1

Introduction

1.1 Background

We would like to share some background on a few key subjects so that the reader may better comprehend the thesis. This background is intended to provide a short perspective for the reader who is not familiar with a topic that we believe is important to our work.

1.1.1 Teaching Programming with Tools

Since the beginning of computer science (CS), many attempts have been made to make the transition of someone with no prior knowledge into the CS world go smoothly. The introduction of teaching tools was one of the most influential attempts to enrich the student's experience. These tools are designed to help students improve their programming skills in the general field of computer science, but they can also be tailored to a specific topic.

There have been several studies that look at the current tools for teaching and introducing people to programming [27][28][29]. With narrative tools [30], flow-model tools [31], specialized output realizations [32], tiered language tools, and visual programming tools [21] being some of the most prevalent categories [29].

Since technology has become such an integral element of the educational system, it was inevitable that these tools would be used to introduce children to technology from an early age. Schools worldwide are adopting relevant tools to teach programming rather than the old-school methods because it is more exciting and engaging for students regardless if they want to pursue a career in CS.
1.1.2 Visual Programming Languages

In the last decade, many attempts have been made to teach programming languages by removing the barrier of learning a specific language's syntax and structure. One of the most successful of these attempts is called Visual Programming Languages (VPLs) [34] which consists of provided blocks that the user interacts with by dragging and dropping them in an area to simulate writing code; since every block corresponds to a text-code equivalent.

The pre-defined blocks usually comprise simple operations such as loops, logic, and procedures that allow the user to accomplish particular tasks quickly. Furthermore, these VPLs are occasionally accompanied by more sophisticated blocks that serve a specific purpose and are built for particular applications or domains.

Many students were drawn to VPLs because they offered an enjoyable experience and in a fulfilling setting. Moreover, they made VPLs so famous among non-programmers that they are now being utilized for a range of purposes such as programming Internet of Things devices [35] other than just learning how to program.

1.1.3 Live Programming

The capability to change the program while it is still executing is termed as liveness [41]. It is a well-studied topic and it is frequently used to make modifications on the fly and test new ways without having to change the source code and rebuild the project.

Live programming and editing adds a layer of programmability and customizability to the system that may be used in various ways to improve the user's experience. For example, it is frequently used in conjunction with teaching tools (as referred in 1.1.1), as it allows for easy integration of user changes.

As programmers, we have a pipeline that starts with writing source code, compilation, and execution. However, we think that, for beginners, it is not as straightforward, and the distinction between those elements is hazy. That is why providing a system that does precisely what the user expects is important (single coding mode that encompasses all of the processes mentioned).
1.1.4 Serious Games

Serious games are video games created only to serve as learning tools rather than for pure entertainment [1]. The concept of utilizing video games to demonstrate a scientific study or to train a professional to be able to accomplish a specific work has piqued the interest of every sector imaginable since the beginning of the video game era. These video games were created to instruct individuals on various (and most of the time unusual) events, ranging from the healthcare industry [2] to introducing people into new concepts such as Intelligent Environments [3].

There is no surprise that, especially in computer science, there is an abundance of serious games with goals that range from learning a new programming language [3] to simplifying the learning of a new algorithm [5].

1.1.5 Reverse Debugging

In the recent decade, a new method of debugging [36] named reverse-debuggers [37] has evolved, allowing programmers to go "back in time" in order to inspect the program's state at a specific point in time. Compared to standard debuggers, these debuggers provide a lot more information about the runtime.

Because time travel does not exist and must be simulated, reverse debuggers face a big problem. In order to simulate it, the system must preserve its state after each state change. This is inefficient and can quickly spiral out of control because it can slow down the runtime and requires significantly more memory than a standard debugger.

Reverse debuggers are incredibly intriguing and helpful tools that have gone under the radar of most programmers for the better part of the previous two decades mainly because of the significant limitations we mentioned.
1.2 Problem Definition

Current programming teaching tools that use visual programming have significant flaws that need to be addressed, especially as the majority of their users have no prior programming knowledge. One of the most significant problems lies in the editor itself; the way it is organized can be completely changed, since most programs provide a big area for users to use and manage on their own. Even for the most experienced users, creating a game that is a little bit larger than typical single object mini-games, becomes onerous to manage and handle.

Figure 1 shows an example of a Scratch project with a large number of micro coding scripts on the same area and in close proximity to one another. An experienced developer can see how and why this approach is prone to errors right away.

The problem described above is just one of several that can arise as a result of the complexity of an object/sprite mechanism supplied by a tool. Other significant difficulties that an experienced programmer would note include the lack of ways to link properties and functions to specific objects, the lack of object reproducibility, and, of course, the lack of state and state manipulation.

The lack of debuggers and debugging features in most teaching tools is another issue. It is critical to provide a mechanism to debug and help the user to figure out what went wrong in a situation, as this can save much time going through the source code. Unfortunately, despite
recent efforts to incorporate debuggers into VPLs [38] and in the tools themselves (Tynker [39]), debugging capabilities and means to figure out what went wrong in a specific context is limited.

The final issue we want to address, is that modern teaching technologies often overlook user-driven live editing and live programming in general. Live editing in teaching tools was popularized by Xerox developers with Smalltalk [40] in the early days of computer science, giving the ability to handle memory interactively for data storage and manipulation. We bring it up as a problem because developers have long emphasized the importance of the system’s live property, particularly as an introduction tool, but since then many teaching tools have moved away from that concept.

### 1.3 Contributions

Our contributions focus on establishing a development system that will allow end-users to create games while also learning necessary programming and game development concepts.

We will go over the research and development of such a system and how we put it together to give a non-expert user a complete programming experience. We are not forcing our users to learn concepts and ideas; instead, we are attempting to effectively keep them engaged with the topic by providing a pleasurable experience while also having a low-effort-high-reward system.

When it came to our research objectives, we tried to push the live element of our system to its limits and provide a live experience for almost every part of it, from manipulating objects, properties, and methods to debugging and serving the ever-expanding management system while in a single running mode.

Our second research goal was to introduce a new simple-to-use testing tool for our system that would deliver a plethora of data to the end-user to determine what went wrong in the programming of the game logic. We realized the potential uses of the time travel feature after implementing and testing it, both as a testing/debugging tool and as a gameplay component if an end-user wishes to explore it further. We present two alternative approaches to implementing this functionality, along with an assessment of their possible benefits and drawbacks.
Finally, we used our system to re-create the old-time classic game Super Mario in a case study to verify the feature usability, user experience, and, of course, the two key research topics (live programming and time travel).
Chapter 2

Related Work

In this chapter, we will look at some of the most prestigious works in the fields where we believe our work fits. We will concentrate on the new ideas, tools, and features that these works introduced, as well as how we tried to learn from and occasionally even improve on them. Of course, some concepts are more revolutionary than others, which is why we are focused on works that introduced us to something new. We will compare our system to those works in the following chapters, explain why we made certain decisions, and incorporate some of the great concepts introduced by these works.

2.1 Serious Games for Programming

CodinGame

CodinGame [8] is a web-based platform for programmers that allows the users to solve challenges in any language they want (currently supporting 25 programming languages). As the challenges evolve into something more difficult, the problem-solving level rises, and the test cases become more demanding.

Due to its game-like user interface, engaging nature, and presentation of the topic in an easier for the student to understand approach, CodinGame has managed to shift students' perceptions about game-based learning [9].

It is worth noting that one of CodinGame's biggest achievements, is that it is turned into a recruitment tool for a lot of companies all around the world by hosting global contests [10] and hackathons. Which shows companies are trusting serious games as a good indicator of someone’s skills and knowledge.
**CodeCombat**

CodeCombat [7] is a serious game aiming to teach young people about software development. The student can learn JavaScript, Python, HTML, and CoffeeScript, among other languages.

CodeCombat is a closed-system game that allows users to manage a character by giving them specific instructions. Each level introduces a new challenge for the user to solve. The difficulty is enhanced by including more instructions as well as new barriers to overcome.

**7 Billion Humans**

As a puzzle-solving game, 7 Billion Humans [51] requires the user to move around and interact with human workers by using an in-game programming language. In comparison to other similar serious games, this one differs in that the same program is used to control all humans at the same time. To test the code further, after completing a puzzle, it will replay that stage with different starting values for each subsequent puzzle to test the code once more. Final ranking is based on the number of steps taken by the code to complete the task.

**Shenzhen I/O**

Shenzhen I/O’s [52] uniqueness lies in simulating and teaching assembly language along with circuit elements to simulate real-world electronics and embedded systems.

**Rocky’s Boots**

Rocky’s Boots [53] was one of the first educational software for PC that successfully used an interactive GUI as a learning environment.
2.2 Live Editing

**Sketchpad**

People recognized the relevance of User Interface as early as 1963, and that the user should be able to interact with the computer in a variety of ways. Sketchpad did exactly that, giving users a new way to draw and create shapes on the computer and see immediate feedback on their computer screen.

Ivan Sutherland built Sketchpad for his PhD thesis [50] (which led him to obtain the Turing Award in 1988). Sketchpad offered several new ideas that would later be regarded important to the development of computer science. It is for this reason, that he is usually regarded as a computer graphics pioneer.

Ideas like, “master” and “occurrences” paved the way for Object Oriented Programming (with Smalltalk). Geometric constraints were also a major invention that Sketchpad provided, like pinning a line or a certain length of an object. Finally, Sketchpad provided new ways to represent 3D objects that could be live created and modified from the user.

**Dynabook / Smalltalk**

Alan Kay envisioned Dynabook as an educational tool and a repository for all types of knowledge, characterizing it as "a personal computer for children of all ages." Dynabook was planned to be a notebook-sized device that everyone would possess and that could store any information imaginable while responding to the user's demands [49]. Many decades later, Apple introduced the iPad, which was based on the same concept.

Dynabook also allowed the user to edit any visible object on the screen by allowing the user to delete, transpose, and structure the objects in a live setting. It also had a Smalltalk API for connecting different components of the system.

With Dynabook came Smalltalk, the language that introduced Object Oriented Programming and was pioneer in live editing since you could add new properties to objects and create instances totally independent from one another.
2.3 Visual Programming for Learning

**Scratch**

Scratch [6] is the most popular block-based visual programming language for students and young individuals. It is primarily aimed towards children aged 8 to 16. Scratch's major objective is to be used as a coding educational tool, and it serves as a gateway for young people to enter the world of programming and programming thinking.

Scratch's straightforward design allows even youngsters with little prior expertise to develop and share a basic game with their peers, allowing them to be creative regardless to gender, race, color, or anything else that would be considered discriminatory [11].

It allows users to make sprites that include a scripting area. The user can drag some pre-defined blocks into this programming area, which must be precisely positioned inside the field in order to code. The animation and sound editor, which allows the user to create or import new films and audio to add into the game, is another wonderful feature that was required for such a tool.

It is worth noting that Scratch has extensions that can be used to create more complex or specialized games. Both physical and digital libraries can benefit from these extensions [12].

**Tynker**

*Tynker* [39] is an educational programming platform that teaches children and teens how to learn and gain coding abilities through the use of Visual Programming Languages (VPLs). Tynker, in addition to Scratch, caters to many levels of competence and knowledge, as well as multiple programming languages, to prepare users for every coding circumstance.

Tynker has been used in over 90,000 schools all around the world over the last few years, owing to their sophisticated programming system and the curriculum that their platforms provide.

Tynker also brought other features that were well-received and well-liked such as an integrated debugger, a backpack, and a new stage/actor model resulting in multiple awards and publications in recent years.
2.4 Time Travel in Games

There are plenty of games that lets the user manipulate time, although most of the times, the mechanic revolves around pausing or slowing down the current time or retrying a certain part of the stage, but they rarely let you actually go back in time and rewind the process that you had made, since that is something much more difficult and requires a lot of space and time complexity to create. That is why in this related work chapter we will only include games that simulate the whole world going back in time.

**Braid**

*Braid* [46] is an indie puzzle game released in 2008 by Number One. A defining game element is the player's unlimited ability to reverse time and "rewind" actions, even after dying. This feature was innovative and enabled the user to replay something over and over again without the need to start the whole stage from the beginning.

The innovation of this feature was recognized by both players and critics which provided Braid with multiple awards and became the highest rated title on Xbox Live.

**Blinx: The Time Sweeper**

*Blinx* [48] was published by Microsoft Game Studios in 2002 and included a “video-like” feature in the form of power-up that the user could utilize to slow down, speed up, pause and most importantly rewind time. Rewinding the whole stage in order to fix previous mistakes and progress in the game.

The game received mixed reviews due to bad game’s execution and more notably the control methods.
Chapter 3

System Overview and Architecture

The primary purpose of our work is to allow a user with little to no experience to develop a new universe that was on their minds just moments before. At the same time, we aim to guide them in a road of learning in order to express themselves in ways that they previously could not.

We chose the path of games, primarily platformer games, because most young people who are interested in pursuing a career in computer science will likely want to create their own game even at a young age. That is why we provide a simple and intuitive User Interface (UI) to assist
people in making links between real life and the virtual world, while also providing them with powerful tools to easily bring their ideas to life.

One of the primary features we intended to include, was that the system should be entirely hosted on a static website, allowing it to be accessed by most browsers and without the need for any specific hardware or software other than a web browser.

Another important consideration for us during the system's development, was to provide tools that also cater to users with prior experience with large-scale game engines (such as Unity [13], Unreal Engine [14], CryEngine [15]). We think experienced users should feel comfortable while using our system and be able to efficiently utilize the system's capabilities, and eventually contribute and expand the system itself according to their needs.

Because the concept of build and run is difficult to grasp for someone with little to no programming knowledge, we decided to provide the user with one main state to work with. In that regard, the state will always remain as running, the user will be able to make changes to every element of the game and the result will take place on the spot. Making a live system, of course, means that we had to come up with solutions to difficulties that arose; in the following chapters, we will go into the issues and the solutions we came up with in detail.

When developing a teaching tool, the developer must maintain a balance between efficiency, accessibility, and user comprehension. That is why the features we provide are well-thought-out, in order to allow the user to develop more entertaining and interactive games while also avoiding confusing the user with unnecessary (and usually undesirable) information.

Educating them on how to construct objects, how to assign values to each object, and how to write code is an essential aspect of our work. There are User Interfaces dedicated to each of the above, as seen in Figure 2. Of course, more advanced features for creating animations, object states, clipboard, and so on are included. As a result, the more time a user spends on the platform, the more familiar the user becomes with the system’s features in order to include them into the game, both as game features and as a method to improve the coding experience.

We will go over how our game object system works in greater depth in this chapter, because we believe the game object system we provide is critical to the user's development and learning experience. Similarly, we'll discuss how scripting the game logic works, as well as our strategy in
solving the problem of a single code region for each item that we mentioned in paragraph 1.2. We will also see how we used game state to our advantage, how our approach to game state differs in order to give the user a lot more freedom on adding game features and handling what happens in the game, and how we used state capture and restore to create object and scene snapshots, time travel, and other features. Finally, we will have a look at the user interface, which is one of the most important aspects of our system because it serves as the user's gateway to the optimal editing experience possible.

We will also demonstrate our architecture design for a better understanding of the features and our engineering methodology on how we designed and developed the system. The macro-architecture and how each key component is integrated and communicates with one another will be our main focus. In addition, we will present some key architecture designs for each feature in the following paragraphs for a better knowledge of how these features are implemented and a better sense of how they work.
3.1 System Macro-Architecture

When creating a system, we have to consider the architecture, such as how the system will be structured, what components will be used, how these components will communicate with one another, who will be responsible for each individual task, and how our system will be structured to be de-coupled and provide foundations for extensibility.

As shown in Figure 3, the Macro-architecture layer consists of four main parts. First, we have the engine and engine extensions, which hold all of the system's functionality and provides an API [42] that, in combination with the blackboard [43], gives full access to the system either through API calls or through system-events triggered by the blackboard using the observer pattern [44]. The two remaining components are using the given API and blackboard that we briefly mentioned in order to achieve a higher level of functionality as we will analyze in the following paragraphs.
3.1.1 Engine

The engine is the most important component in the overall system's engineering; the engine is responsible for the system's functionality. The API that the engine will eventually export is composed of carefully selected APIs from sub-components that we refer to as "Managers" and are in charge of the necessary functionality.

![Architecture composition of Engine component and the main managers.](image)

Figure 4 shows how the engine component is made up of Managers; each of these managers has its own architecture design, which are discussed more extensively together with the APIs in the appropriate paragraphs. The eight managers displayed in Figure 4 are our system's basic and most important managers, who are aptly dubbed "core managers". Ultimately, those managers should be sufficient to create a game.

The game loop exists in the engine, and it is responsible for keeping the game running at a given frame rate while also dictating the order in which the core managers will perform their tasks. For example, before calling the animations and AI of the objects, we want to check for input on every frame.
**Extensibility**

We supply extra managers in addition to the core managers, which will be installed into the engine and will provide additional functionality. As a result, the exported API will expand in proportion to the number of additional managers installed.

Engine extension is a layer above the engine which gives a means for anyone to create a manager which is in charge of completing a specific task. Of course, there are several guidelines to follow when creating a manager, including: The manager’s sole dependent API calls should be on the core managers, and no other engine extensions API should be accessed. Second, the managers must be subclasses of the Manager class, which provide certain useful methods to the manager. Lastly, the newly appointed manager should only use the blackboard to trigger or monitor any event in order to take appropriate actions.

Following the development of the basic engine managers, we could now utilize the engine as a "black box" and design engine extensions for our own needs, following the engine extensions guidelines. The following are some of the significant features that came as a result of engine extensions: time travel, live grid, geometry computations (between objects), live clipboard, and snapshotting are some of the features that were completely developed as engine extensions.

### 3.1.2 Actions

Actions is the smallest part of our macro-architecture but a very important one when created and utilized properly. Actions are JavaScript scripts totally independent from the engine that utilize the Engine API and the blackboard (Figure 3) to automate certain actions (where the user is also able to attach a keybind) and provide a higher-level functionality for both the developer of the system and the end-user (if the developer wants to include it into the game).

We used actions to automate repeated scripting for activities ranging from saving the current game to replaying certain keystrokes at precise moments which we thoroughly used during the development of the system and especially during the preparation of the case study.
Extensibility

The key benefit of providing actions is that anyone with some JavaScript understanding can easily automate tasks and then utilize these scripts as macros throughout the editing process and then in the game as mechanics. It was critical for us to create a mean for developers to automate things within the system on their own since we believe it provides the developer of the action a new level of abstraction to explore.

3.1.3 User Interface

The user interface is the last component we'll discuss. It is mostly employed to introduce a new gateway for the user into system's features. The user interface architecture is kept basic, but it is quite powerful. It is in charge of installing and then keeping track of which UIs are currently displayed and hidden in the system.

One of the most useful features of the UI manager is that the users are able to select the UIs they want throughout the editing process, as well as install new UIs from an online database\(^1\) for a better experience.

Extensibility

Even from the architecture of the UI component itself it is easy for someone to point out that it was built with the idea of providing a way for extensibility. Someone must provide a folder with HTML, CSS and JavaScript to integrate a new user interface into our system. Our UI component will then obtain the HTML file, parse it, and display it to the system. After that, we will invoke the JavaScript callback that the user specified when the HTML file was loaded. From that point forward, the developer who provided the JavaScript code has control over the specific UI and can communicate with the system through the API and the blackboard.

\(^1\) This database does not exist yet.
In order to properly include a new UI, someone must export the following object from the JavaScript file that will enable the system to import the new UI into the system:

```javascript
export default {
  name: 'the name of the UI',
  link: './src/UI/{folder}/{htmlFile}.html',
  cb: onLoadCallback,
  removable: true, // if the UI can be removed after installation
  loadOnInstall: true // if we want to load the UI right away
};
```

During the creation of the system, we built 20 UIs to provide support to the user. Each of these UIs were built independently without any interference between them, which is important when creating an extensible UI system.

Last but not least, the UI component introduces the "objectification" of HTML elements, which is a very powerful and valuable feature. An element will become a game object by adding a "category" property to the HTML element during parsing and will be part of the game as a typically defined object from then on. These objects retain all of their properties, including location, width, height, and content.

### 3.2 Game Object System

We define every aspect with which the user can interact as game objects. There are two types of game objects on an abstract level. System Objects are unique objects the user can interact with, but certain operations such as removing and copying are restricted. The second type of object is User Objects, which are unrestricted by default. It is vital to note that most users will not be able to distinguish between the two kinds of game objects since system objects are created automatically by the system and exist throughout the game's lifespan.

Each object comprises properties as well as methods, as inspired by Object-Oriented Programming[19][20]. Depending on what the property is used for and how it impacts the objects, we provide a more descriptive name in the system (such as State for possible states of an object, Attributes for values stored in the object, and Flags for Boolean values). We refer to methods as events to ensure that the user understands that every event has a matching code that reacts whenever a specific event occurs.
We employ classes that we have named categories to provide default properties and methods to the object as another nod to Object-Oriented Programming. On construction, each category will add some characteristics to the object (for example, if it is a rectangle, it will add Width and Height; if it is a circle, it will add Radius instead).

**Architecture of the Game Object System**

As we discussed in section 3.1, every function is handled by an appropriate Manager, therefore the ObjectManager is exclusively responsible for handling and modifying the objects in our game object system.

To present a comprehensive understanding of how the ObjectManager is organized, we must first discuss the architecture surrounding the object. The object class contains critical data for the object, as well as Handlers that act as a middleman between the game-specific properties and methods on each object instance.

For our system, Object is an abstract class from which Object<Renderer> classes inherit (as shown in Figure 5). These classes are responsible for overriding several critical methods of the Object class, such as render, add, and remove, to meet the demands of the renderer.
Object<Renderer> classes are also in charge of providing default values for properties and methods.

From the classes we mentioned above, new classes that will be used as the "construction" classes for our game objects later are derived. These classes are generally small and most of the times dictate the shape that will be drawn with some exceptions such as the Text.

![Figure 6: Renderer’s Architecture.](image)

![Figure 7: ObjectManager’s Architecture.](image)

At this point, it should be clear that game objects, depending on their category are being treated differently and maybe even rendered by a different mechanism. In order to handle all the objects in the same way, each renderer must be responsible for rendering the objects created from categories which belong to it. In order to achieve that, every renderer supplies a scene that whenever an object is being created is being installed and rendered on each frame until it gets removed. For example, for our Dom renderer our scene consists of a div where every object is being drawn on. Lastly, each renderer exports the constructor classes so that the object manager can access them and create new instances as shown in Figure 6 and Figure 7.
ObjectManager, is the class that contains the renderers that our system is using, the Object Constructors which is a collection of all the Constructor Classes of every individual renderer and of course, it keeps maps with all the user game objects and the system game objects that are currently in the game. ObjectManager is the only part of our architecture able to access the objects and it is responsible to serve the API related to the objects to the engine.

**Implementation of Object’s Properties and Methods**

States consist of a tag and three other variables: TransitionFrom, TransitionTo, WhileInState. The tag is a string defining the visible name for the user and the three other variables are codes that the user defined and will get executed at the appropriate times.

As briefly mentioned earlier in this paragraph during the architecture analysis, each game object consists of a stateHandler. In stateHandler we are able to add new states and switch between them through API calls. Whenever a switch happens, we execute the TransitionFrom code from the old state then we execute the TransitionTo from the new state and of course on each frame we execute the current state’s WhileInState. Also, it is worth noting that, after the transition happens an event named "setCurrentState" is fired through the blackboard.

In the same line, there are four more implementations for Object Attributes, Object Flags, Object Events, and Object Collisions, each with its own set of code variables that can be populated by the system or by the user. Each handler is responsible for holding every attribute or method for that specific object and acting in accordance with the expected behavior.

We keep our architecture consistent by implementing it the same way for all the different possible properties, and with that we also manage to make every possible attach-point to the object entirely dynamic and configurable on-the-fly.
3.2.1 Properties

Object States

Every object is assigned a default state called idle when it is created, and this is considered the object's default state. There are three possible tabs to add a script for each state. When the object enters a new state, when it exits a state, and finally, on each frame while the object is in that state. Finally, the user can click on each tab, as shown in Figure 8, and then be able to add code to the script in order to run on the actual occurrence specified by the name. We will explain in detail the scripting mechanism in paragraph 3.3.
Object Attributes

Figure 9: Attributes of an object and script reaction on attribute change.

There are two types of attributes: system attributes and user attributes. System attributes are those that the system has already decided what will happen when the value changes and they cannot be removed. There are user defined attributes in addition to system-defined attributes, which are added by the user and contain the value as well as the code to be triggered when the value changes. The code within the tab provided to the user will be triggered right after the value change, which means that the user can just access the value in order to react to the value change. As shown in the example in Figure 9, the user defined an attribute "score" and then directly there was a tab created named "on score change" in which the user decided and scripted the code to react by changing the text of the object to show the new score to the user.

Now that we have looked at what occurs when the value of an attribute changes, we can answer the question, "How can someone modify the value of an attribute?" There are a few options for changing a value. This was a design decision made to help the user in selecting the best fit for them. The User Interface, as depicted in Figure 6, is the most common approach for a novice user to manipulate values. However, code and other unique ways can also be used to change the value of an attribute. For example, dragging an object through the game area can modify its x and y values.
Object Flags

The system specified flags, and the user-defined flags are available and treated on the same page as the object attributes when an object is created. The system defined flags are behaving similarly to the system defined attributes which means the user is not able to remove them, and also, they have predefined reactions. The user-defined flags are treated similar to the user-defined attributes and have a tab for the user to add the code to be executed upon change. The flags' primary function is to supply the object with unique attributes in a Boolean format. isMovable, isRemovable, moveThroughGrid, isSolid, isCollidable, and many more system-defined flags are some examples.
3.2.2 Methods

Object Events

Introducing methods created by the user and triggered during particular events was a difficult task throughout the design process, but we also wanted the methods to be easily understood by the user. That is why we conveniently named the methods as events so that the user understands that events are scripts that will get executed at appropriate times. In addition, we decided that the name of each method should be descriptive and specify when the script will get executed.

The system will prompt the user to enter the correct name for the event that will be executed at a particular time. If a user wants to run a script when the game starts, for example, the user needs to build a script called "onGameStart." Using this naming pattern will help the user understand how to split code in a more straightforward manner.

Object Collisions

Figure 11: Events of an object and scripting behavior when triggered.

Figure 12: Collisions of an object and scripting behavior when collide with given object.
The object also contains collisions; to register a new collision, the user must establish a new "Collision" on the object and specify the second object of the collision by providing the name of the other object, since we follow a name convention. When a collision happens, the script that the user provided will get executed, as depicted in Figure 12.

### 3.2.3 Built-in Objects

There are various specific objects that the user has no control over when it comes to their creation or removal, as briefly stated in Chapter 3.2. These are essential objects for both the user and the system. Therefore, we make a concerted effort to keep them as simple as possible and to avoid introducing new ones as much as possible.

Each of these objects has a defined functionality, and in many cases, this object is the only means for the user to complete a certain task. At this moment, the user has access to two system defined objects them: the stage and the keyboard.

The stage is an object that includes information on the stage that the user is presently working on, as its name implies. This information will define the current stage's width and height, as well as the position of the window that the user can see on the screen currently. It can, of course, be used to store game-related information as well.

The user's only option for adding scripts that receive input from the keyboard with the supplied events is to use the object keyboard. The object is pre-filled with "pressedKeyA" and "unpressedKeyA" events to make it easy for the user to grasp how to utilize and split the code appropriately.
3.3 Game Logic Scripts

![Figure 13: Scripting area with visual programming on the left and JavaScript on the right.](image)

The scripting mechanism is undeniably one of the most critical aspects of our system. It is critical, especially for a programming teaching system, to provide the user with an entertaining and straightforward approach to develop a script. With this in mind, we developed a multi-language scripting system that can handle both visual and text-based programming languages.

Our system allows for the simultaneous execution of scripts in various languages in the same game instance. That way, if our user decides to switch languages in the future, they will not have to rewrite all of their scripts.

When the goal is to establish an ever-growing mindset, providing different ways to obtain the same outcome is critical, and it also helps the user comprehend that there are more ways to produce the same result. Figure 13 shows that both scripts, one built with Blockly and the other with vanilla JavaScript, provide the same result. Both techniques utilize the system's supplied API, as seen in the figure, Blockly with the pre-defined blocks on the left and pure API calls through JavaScript code on the right.

Last but not least, the user's scripting mechanism is always capable of providing a "preview" of the actual JavaScript code that will be executed. Since all languages must be converted to JavaScript before they can be used on our system, we know that such representation must always exist.
Architecture of the Game Logic System

The scriptManager must be able to switch between languages while the system is still running in order to deliver the capabilities we described. As a result, our scriptManager uses a proxy object named "currentLanguage" to access the languages API. The user can switch between languages by changing the “currentLanguage” to a language that meets the API’s requirements.

To introduce a new language, it must satisfy five different functionalities: 
- `currentScriptAsText`, which returns the text representation of the current script,
- `currentScriptAsCode`, which returns the current script as JavaScript code,
- `executeCode`, which executes the code,
- `clearAndLoadFromText`, which clears the scripting area and loads the given text for the editing tool,
- `injectInDiv`, which orders the current script.

To provide many languages at the same time, the user can “force” the proxy to point to a given language just for a single script, which means that multiple objects can be scripted in different languages without the need for a manual proxy change.

### 3.3.1 Hierarchical Organization

Every experienced programmer understands the significance of separating source code into separate files. This reduces repeated code, reduces possibilities for error, and, most significantly, helps the programmer’s understanding of the source code. As a result, we chose to provide the user with a great number of tabs where code can be written, and as we described in paragraph
3.2, a new scripting tab would appear for the user to utilize each time a new property or method is linked to the object.

We added this dividing method to help the user split the source code without forcing them to worry about the code structure since we believed it is too early for an inexperienced user to add this degree of abstraction into their project.

### 3.3.2 Visual and Textual Scripts

![Visual blocks sample to interact with our system’s API.](image)

In order to provide a visual programming experience in our system, we had to include Blockly[21], a client-side JavaScript library. Blockly, created by Google developers, provides a toolbox that includes a basic set of blocks such as loops, math, and logical operations. Additionally, Blockly’s creators provide an API that allows users to extend the language by defining and generating new blocks corresponding to certain code samples. We created 50 new blocks using this API, ranging from modifying an attribute value to object duplication.

Even though visual programming languages are a great place to start, we believe that learning a textual scripting language is a requirement for anybody interested in programming. To simplify the transition to real-world programming, we introduced support for JavaScript, which is a high-level textual language.
3.4 Game State System

Game state is vital in our system for a variety of reasons. It may be used to save time, provide helpful information, take snapshots, and travel back in time. That is why it was one of the first features to be added to assist us in the system's development.

As a game state, we define each object with its properties, methods, and scripts together with animators (whenever required) and any information that the particular managers want to preserve. We will use various information to achieve our goal depending on how the game state is used, as indicated in Table 1.

<table>
<thead>
<tr>
<th>Objects</th>
<th>Animations</th>
<th>Managers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Save game</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Helpful tooltips</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Object snapshot</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Scene snapshot</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Time travel (State Snapshot)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Time travel (State Deltas)</td>
<td>Differences</td>
<td>✓</td>
</tr>
</tbody>
</table>

*Table 1: Important information to get from game state according to the usage.*

3.4.1 State Capture

We saw the three types of data we need to be able to capture in the previous paragraph. To capture all three groups, we must first locate the bare minimum of information required to return to that state. We do not save data that can be deduced from the rest of the data we have. This means that we must obtain a specific subset of data for each information group.

For objects, we record their current properties, methods, and scripts, as well as their name, id, and category. This information is necessary to ensure that an exact clone of the object can be created without losing any information. Someone can use the objectManager API to retrieve this information efficiently.
We use the same method for animations. Therefore, we must capture the method that will be executed on each iteration, the method that will be executed at the end of the animation, the last time the animation was iterated, the current repetition of the animation, and, of course, the name and animation that is currently running. Note that the method that would be executed on start is not saved because the animation has already begun, and we will not need it when restoring the animation.

The last group is the managers, which are an essential element of the current state of the game system because they may be used for both editing and gameplay. The difficulty with saving state information for managers is to allow the person who created the manager to choose which information to save. This was accomplished by providing a method named onSave from the super-class "Manager" that the system would invoke for each installed manager in order to acquire the relevant information.

3.4.2 State Restore

It is critical to restore the game state that was captured, especially on a live system. For us, restoring a game state can take many forms: it can be done on a single object, the entire game, or even a completely new game. At first glance, restoring a game state appears to be much easier than capturing, but it has its own set of challenges to solve on a live system.

Starting with object restoration, the technique varies depending on whether the object is still present or destroyed; if the object has been destroyed, an exact duplicate must be constructed from scratch with the same name and id, as well as properties, methods, and scripts. On the other hand, if the object is currently alive, we must be careful not to trigger events hooked on "onChange" (as we detailed in paragraph 3.2) since this could result in some undesirable consequences.

Changing and manipulating the current animations may cause side effects when it comes to the animation restore. For example, consider the following scenario: we had an animation running on an object with an onFinish function to remove the item, and we wanted to restore the animation; if we did not remove the animation without triggering the onFinish callback, the
object would have been deleted, resulting in an error. By avoiding this issue, we may safely replace the currently running animation with a new one that we just constructed using the data we collected and then utilize animation timeShift to get the timing back to where it was.

For the managers, our state restore system follows the same procedures as it did for state capture, invoking the super-class method "onRetrieve" to get the data that the same manager had saved.
3.5 User Interface

We believe user interfaces are a very significant and vital aspect of the system, both as an educational tool and a development tool. Therefore, we created user interfaces that a person with prior knowledge of relevant products can immediately understand the purpose of as well as making them as non-experienced friendly as possible.

In the following paragraphs, we will emphasize the most significant UIs that we provide to users and the suitable use-cases and components that make up the UIs.

3.5.1 Live Property Sheets

Figure 16: Live property sheet user interface.
The most prevalent and expected user interface in related technologies is the property sheet. It is typically used to add additional properties, scripts, and methods to objects similar to ours with the twist that changes made to our property sheet will affect the game immediately as it runs.

The system's live property sheet is detrimental to the development process because it gives important information as well as the ability to add new or modify existing properties and methods of a single object.

The main layout and structure of the property sheet can be seen in Figure 16. The region denoted by the number 1 contains information about the object which is currently focused. We also offer a quick action toolkit based on the currently focused object, which is the region denoted by the number 2.

Each of the areas highlighted 3, 4, 5 is utilized to enhance the system's liveness. Area 3 allows the user to modify the object’s current state, whereas area 4 allows the user to add a new property or method to the object, and area 5 allows the user to edit or remove an existing property.

### 3.5.2 Visual Scripts Workspace

![Figure 17: Visual scripts workspace.](image)
We tried to keep the scripting workspace as clean as possible so that the user could understand it with ease. Furthermore, we attempted to keep it as close as possible to the standard tree form that users may encounter later in an IDE.

When disassembling Figure 17, we can see the area on the left side where all of the possible scripts for the current object are displayed, and the user can click on them to begin scripting. We can also see the name of the currently selected script, which is denoted by the number 2. Next, there are two function buttons on the right. The one with the number 3 is the save current script button, while the one with the number 4 allows the user to switch languages and preview the current JavaScript code. Finally, as shown in the region indicated with the number 5, there is a coding area where the user can insert blocks in order to code.

### 3.5.3 Time Travel Controller

![Figure 18: Time-Travel Controller User Interface.](image)

The User Interface dedicated to time travel is separated into smaller discrete regions that each have a specialized purpose, as seen in Figure 18. A sliding bar can be found in the area marked with the number 1 and can be used to select a specific frame of the recording. The buttons in the area labeled with the number 2 are being used as inputs for various functions such as rewind,
pause, continue from here, playback. Number 3 is supplementary frame and time information for further assistance when the timing is critical during playback.

In addition to the replay capability with code and data inspection, we took it a step further by allowing users to record many sessions in a row that can be switched back and forth. The User Interface identified with the number 4 in Figure 18 can make use of this.

Finally, we created a more advanced tool to provide the user with additional meta-information about what happened on the current object throughout the recording, as well as tracking changes in code and data and highlighting changes that occurred on the frame the user is currently viewing.

### 3.5.4 Live Game Assets

![Figure 19: Live game asset user interface.](image)

Numerous asset categories may be required when designing a game. Our system features a user interface called "inventory" that keeps all of the live game assets that are available to the user in order to convey all of this information in a logical manner. This is especially beneficial for inexperienced users since it will be easier to recognize when they should look anything up on this UI. Since the concept of collections and grouping has been ingrained in our minds as humans since childhood.

There are seven separate groups viewable through this UI, as shown in Figure 19, and the user can switch between them by selecting the group from the left-hand panel. We will not go into detail about each tab in this paragraph since each one will be thoroughly assessed for its functionality in other chapters.
3.5.5 Live Options Menu

The only non-essential UI we will include in this paragraph is the live options menu because we believe customization is critical for a system like ours. Allowing users to customize the system to their specific needs will make them feel more linked and welcomed. Options are also necessary to alleviate some of the annoyances that users may experience. For example, if a user feel the need to press a key bound that we provided, they can simply alter it to something else.

As seen in the top left corner of Figure 20, our system currently offers four different types of customizations through options. The first is called "Colors", and it allows the user to select a color for the overall theme of the system. The second is called "AddOns", and it allows the user to turn UIs on and off as needed. The third feature is "Keybinds", which allows the user to assign any key or key combination to an action. The fourth category is named "Options", which provides the user with several options and allows them to choose whether or not to enable or disable those options.

Finally, we would like to point out that it is called live options menu since each modification made will have an immediate influence on the entire system, regardless of the change.

Figure 20: Live options menu user interface.

---

2 Its existence is not required for the usage of the system.
Chapter 4

Live Development

Since the conceptualization of the system, we had the idea of someone being able to pause and being able to script and edit an object that is currently on the game and then the user would be able to continue and repeat that process until all the parts of the game are complete. In order to provide the liveness status that we wanted, we had to provide several ways for the user to be able to interact and change things as he or she pleases. At the same time, liveness helps with the testing since the user can test different situations live without recompiling, rebuilding and rerunning the whole game.

Providing a live system to a novice user is a challenge by itself since, novice users tend to make small and tedious mistakes that may lead to fatal errors for the system and the game itself. In order to prevent something like that from happening we are providing several backup mechanisms (Chapter 5, Paragraph 4.3, Paragraph 4.4).

In particular, we will demonstrate what tools and features we provide to the user to give the ability to change the game live. Scripting the object behavior on certain occasions, being able to change object properties and change the position of the object, utilizing the clipboard both as editing feature and gameplay mechanic at the same time and of course change the animations that the user has registered. Also, we will also explain how we managed to incorporate the live changes to have the appropriate impact on the game.

4.1 Live Programming

One of the most important parts of our live system is providing the ability for the user to script the game behavior using a VPL or a textual programming language and then the actual script to be a part of the game.
To accomplish this, we must save two primary parts on each object, for each script, in order to let us retrieve and execute the specific code script. The first is the representation required by the individual coding tool to render that specific piece of code that the user saved and provide the ability to examine and update the code again. The second part is the translation of the given code in JavaScript (as provided by the coding tool), which is important to reduce the overall cost of the translation because a specific script may be executed many times, and translating it over and over will result in a lot of unnecessary translations.

This technique also has a very interesting byproduct in that it allows anyone to add any editor they want as long as there is a provided "translation" to JavaScript from the editor. The only restriction is that this editor must allow the user to describe our API in the language (as we did by providing new blocks for Blockly integration). Because multiple languages can be used within the same game instance, this is particularly useful for teaching new languages (as shown in paragraph 3.3).

Our Case Study (Chapter 6) of Super Mario is a great example of how powerful a feature like that can be. Because we did the development of this game, we had the opportunity to test the live programming and it was very interesting that the user could be progressing in the game while scripting the game logic. Also, as experienced programmers, we chose to write the majority of the game in a textual language such as JavaScript, but we were also able to code some scripts in VPL to ensure that everything works exactly as intended.

4.2 Live Editing

Live editing allows users to interact with objects in order to change the value of a property in a unique way. A property’s value can be changed in a variety of ways, ranging from using a UI to custom actions such as dragging the object with the mouse (changing the x and y attribute of the object).

The challenge in enabling live editing comes from the fact that by altering a value, certain things may need to happen. Even something as easy as moving an object, which may appear to be trivial, takes meticulous planning because it may result in several events (such as grid recalculation).
We need to store various pieces of information in order to implement live editing in every object and property. First, we must save the current value of the property as well as the scripts that go with it. Second, in order for everything to work properly after a modification, we must adhere to a precise pipeline: To begin, we apply the change to the property, which triggers the user scripts (and occasionally system code) that react to that specific change, and then we send an event through blackboard to notify everyone who would be interested in that change.

4.3 Live Collections

We provide live collections because a live system is responsible to provide a mechanism to monitor what is currently happening within the system. Live collections offer a view of the system as a whole. It is quite difficult, especially for inexperienced users, to remember what is now "active" within the system.

The collections always stay updated with the system's status, whether currently showing the objects, the snapshots or even the animations that exist within the system. Our case study showed that the collection is a vital part of the liveness of the system since it does not only provide a way for the user to have a more abstract and quantized view of the system but also provide certain actions to utilize (like removing an object, clearing the clipboard, etc.).

4.4 Live Animation Creation and Editing

Figure 21: Preview of currently registered (created) animations.
Animations play a significant role in the user's immersion in any game. That being said, we wanted to give the user the ability to create new animations within the system utilizing the imported films.

The user can select a specific film, which is made up of numerous bitmaps, and apply the required values for the delay, dx, dy, and id while previewing the animation, as shown in Figure 21. After the animation is completed, it is immediately ready for use via code and in the previously stated live collections.

Our system allows the user to alter and update previously created animations (Figure 22) by entering new values for delay, dx, and dy. From then on, every instance of that animation that is currently running in the game (as long as the game is still running) will receive the changes on the next iteration of the film and will continue with the updated version.
Chapter 5

Time Travel

Figure 23: Time travel example with 487 frames and currently previewing frame 361 extra information.

One of the most challenging parts to use, especially if you are not an experienced programmer, is the debugging techniques. Furthermore, determining what went wrong and what needs to be changed is a difficult task, made even more difficult by the fact that our system operates in a single mode during both play and editing, necessitating the development of a solution to assist our users in "debugging" and having a better overall developing experience.

The solution we devised is a backup model that allows the user to begin a recording session and then proceed to perform any action that could cause a problem. After stopping the recording, the user will be able to study every frame (from code to data) and will be able to either continue with a new recording or continue playing the game from this point forward.

Figure 23 shows a single frame from the recording, which has 487 total frames saved and took 8.1 seconds to complete, and the user is currently monitoring the extra information that was provided for each change that occurred on object "Mario." In paragraph 3.5.3, the user interface for time travel was clearly detailed.
5.1 State Deltas

The disparities between two consecutive states are referred to as state deltas. We consider as difference and take into account any alteration that has an immediate influence on the game. Given that the game engine operates at 60 frames per second, simply recording the differences that occurred in each frame reduces a significant amount of time and memory required.

In order to construct time travel, we interpret state deltas by observing every action that occurs in the game and keeping a record of every change. At the end of each frame, we save the changes that occurred during that frame into a data structure so that we can retrieve that information later. We will have the first state as is and all the state deltas in between at the end of our recording session. We are able to reproduce every potential frame of the game as it was initially, given the data that we saved.

As previously stated, this method is both fast and memory friendly, but it is also very slow (in comparison to other alternative methods) when attempting to reproduce a certain time-frame because it must internally play all the differences in order to return to the appropriate state.

The nature of state deltas provides us with yet another brilliant notion for how to assist the user by presenting additional information. We could communicate that information with the user and explain what caused that action, since we are mandated to save all delta states. This can be used to help the user understand why a certain action occurred and what can be done to change it as we can briefly see in Figure 23.

5.2 State Snapshots

In computer science, a snapshot is the state of a system at a specific point in time. They are commonly utilized as database backup solutions because they can provide useful information and serve as a great reference point if something goes wrong.

We demonstrate how to use state snapshots to create a time travel feature by recording the state on each frame and then effortlessly replaying each frame. Of course, putting so much emphasis on each frame will eventually degrade the game's overall performance while recording, as well as result in large memory footprints due to the large amount of data that must be kept,
since the amount of data that needs to be kept grows in proportion to the number of items a user has in the game.

Snapshots have the advantage of storing all of the necessary information to restore the game to an exact frame, which is critical when the user wants the preview mode to be faster and contain a lot of extra information.

As stated in paragraph 3.4, we do not need to preserve all of the information that is there in order to save the game state; instead, we can save only the data that is essential and data that can be easily extrapolated can be skipped. Because there are many items that don't carry a lot of information, we can drastically reduce the cost of the snapshot by doing so. Depending on the size of the game and the number of objects in it, this can save up to 70% of the memory required for each snapshot.

### 5.3 Snapshots vs Deltas

Despite the fact that both snapshots and deltas can achieve the same result, they work in quite distinct ways and have very different effects on the game during recording and replay. There are benefits and drawbacks to utilizing either option, therefore the decision to choose one over the other should be entirely up to the user and how they want to use the time travel feature.

We have already mentioned that state deltas are faster to capture and have a smaller memory footprint, but snapshots, on the other hand, allow for much faster replay inspection and store a lot more information for the game itself, as seen in Table 2.

<table>
<thead>
<tr>
<th></th>
<th>Capture speed</th>
<th>Memory footprint</th>
<th>Show a specific frame speed</th>
<th>Information on specific frame</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>State Deltas</strong></td>
<td>Very high</td>
<td>Very low</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td><strong>State Snapshots</strong></td>
<td>Low</td>
<td>Very high</td>
<td>Very high</td>
<td>Very high</td>
</tr>
</tbody>
</table>

*Table 2: Deltas vs Snapshots on important metrics.*
We provide both, because each situation requires different approach. Our proposition is that someone should use state deltas for gameplay feature or if the user wants to try something and wants to fall back to safety and use state snapshots when thorough inspection is needed on every aspect of the game.

Commercially, a hybrid approach is being used [47] to capture and move back in time in games like Braid [46], which is excellent because it is both fast on capture and solves the greatest drawback of state deltas (as shown in Table 2): that going to a specific frame is slow. In the case of a 10-minute recording, this means that in order to display a single frame, you must apply all of the changes that occurred in the meantime internally. However, saving a snapshot at predetermined intervals, requires that the system only repeats a subset of the deltas internally and utilize the snapshot as a "save point" on the recordings.

We would be happy to give such a solution, however as previously stated, the requirements are determined by the game and how you wish to approach the problem. Furthermore, because our time travel is primarily intended to be used as a debugging tool for the user, supplying additional information is extremely helpful.

5.4 Dealing with Commands

Working backwards in time has its own set of challenges that must be carefully managed because some commands are not simply reversible, necessitating the usage of more precise design patterns to handle the problem. Handling events like object deletion or creation, to be more specific, is tough since you must "undo" the action.

We use the command design pattern [54] to handle commands that are not native language operators but instead imply an action on an object, and we can undo the command by using the unique ids we assigned to the object.
5.5 Chained Recordings

Another innovation that we brought was the chained recordings which gives the ability to the user to record a session, then make any edit required and continue with a new recording. Afterwards the user is able to go back to any recording and "redo" the history from that recording.

The idea is easy to comprehend by the user since it is a common usage even for everyday users since the undo/redo on normal editors as well as the go back to a specific page on a browser works exactly the same way. Chained recordings also provide a more advanced feature for users that really want to dive into the time travel mechanics and want to have a more hardcore development experience.

5.6 Time Travel in Testing

In this paragraph, we will look at how time travel can be used as a testing technique, how it is designed to be used, and how a non-experienced user can utilize it with ease and without having a cumbersome experience.
Figure 25 depicts the intended use of time travel and how it should be used by a user. It was difficult to come up with a way to manage time travel in such a way that the user could readily get the grasp of it. We were looking for ways where the user might have already experienced time travel, and the video player was by far the greatest fit. That is why we give a replay and play ahead button for previewing the frames in the same way that the user would handle a video, and also having the benefit of being able to halt at any point (as someone is able to do on a normal video player). From that point, there are other variations, the most notable of which is that the user can interact with the objects, study any aspect of the game, and obtain additional information on why a particular action occurred (Figure 23). Of course, the user can alter everything in the game, and then choose whether to continue playing or start a fresh recording with the new changes applied.

The value of the time travel feature stems from the fact that attempting to test a piece of code might have consequences in the overall game. With time travel someone can not only go back and try again safely but also be able to preview what happened and what went wrong which is both assisting in the solution and giving the user valuable lessons on the impact that a certain code might have.
Chapter 6

Case Study

To put our system to the test, we intended to explore a variety of ways, ranging from the most well-known platform game (Super Mario) to a simpler game. In this chapter, we will look at the several levels of difficulty that each game may require, as well as how experience can influence the outcome and how easy it is to create a game using the system.

The manner an educational system is used, will vary depending on the user's experience and familiarity with the system. As a result, in this chapter, we will go over the challenges a user might have while developing the game, as well as how the system is attempting to prevent and assist the user in overcoming these challenges and eventually producing what the user had in mind.

A common observation made by many people was how different the system looks and feels while making different games. That, in our opinion, is due to the fact that each game has its own set of needs, and users tend to employ different features based on the type of game they want to create and the tools they have at their disposal.

The case study we will present was created by us, with all of the features at our disposal and a thorough understanding of the system. As a result, in each paragraph we will outline how a user without that expertise or knowledge of the system would approach the same problem in order to arrive at the same result.

We chose Super Mario as a case study since it is one of the most influential games and has been the face of platform games for many years. Super Mario incorporates the majority of platform game principles, which is one of the reasons it was the best selection for our case study.

It was critical to evaluate some of our system's most crucial features for our case study. The main focus was on features like live editing, time travel, clipboard, object snapshots, and scene snapshots, as these are not often used and tested in similar systems, and it was crucial to see how much they influenced the development and the usability of the system.
6.1 Incremental Development Process

The first stage was to set up the films and bounding boxes, which is a crucial step if you want to make a game with animations. To do so, we will need to import a JSON file containing the information shown in Figure 26, which includes the film's x, y, width, and height in the sprite sheet. Because the aforementioned setup is difficult for someone without prior knowledge, we plan to provide a bundle that the user may be able to import online with all of the necessary information already inside.

Continuing with the setup, we constructed objects that we will later use as templates to speed up development. This is a more challenging step because users with no experience would begin developing at that point.

The next step for us is to consider the game's layout and how many scenes are needed, as well as how we will connect all of these worlds. We also considered how the camera would move to convey the appropriate information.

Before we begin the development process, we needed to build and test several animations that we knew we would utilize, especially with regards to our protagonist and opponents.
The development process requires some setup and thinking before starting to create; we provide several tools to help with that goal, and especially plan ahead for a better experience later on. Of course, new users are not expected to plan ahead of time, which may make the procedure more difficult for them as time goes on.

### 6.2 Basic Terrain and Game UI

The basic terrain and a minimal game UI were the first things we produced. We used several template-objects and made numerous copies of them to fill our area the way we wanted it, by copying and pasting and then dragging the items to the appropriate spots within the game area. The basic terrain in our case study looks like Figure 27 with all the objects in place.

We must decide what properties shall a block have after establishing the game area with the required objects, since the game is live even if non-experienced users cannot readily understand it at this point.

It takes a lot of time and effort to think about the game, how everything will be related, and, of course, how the game will play when creating the terrain and game UI. As a result, we put a lot of emphasis on object creation and manipulation by including snapshots, a clipboard, and a live property sheet. Even after the processes of player construction or game logic have begun, the user will have these tools to assist on further edit of the terrain.

Figure 27: Game area after the terrain and a simple game UI was introduced.
6.3 Player and Characters

Following the creation of the terrain, the player and non-player characters (NPCs) were added to the game. Characters are more sophisticated than plain terrain blocks because they usually have a lot of animations, an AI, and a specific role in the game. We considered every potential condition a character may be in, as well as any possible property we might need later, when creating a character. We can save a lot of time and thought later on when the most challenging stage of the game development begins if done this way.

In this case study, we created Mario, along with all of his necessary states (idle, moveRight, moveLeft, jumping, dead), as well as all of his enemies, which included goombas, which must be killed by jumping on their heads, piranhas which are plants that emerge from pipes, and turtles which required two hits to kill.

We thought that providing pre-defined objects for a specific game within the bundle of the animation we specified in the previous paragraph would be quite beneficial while we were working on this phase of the project. This will considerably assist a new user because it will avoid the onerous process of anticipating the game's requirements, allowing the user to concentrate on learning how to code and create interactions between game objects.

6.4 Game Logic

We intended to recreate the Super Mario world as closely as possible to the original, which resulted in some really fascinating outcomes. We were able to recreate the game entirely with the blocks provided by utilizing the available features as well as the live development.

Our game logic development began with simple methods like "addOneOnScore" on the object "Score", which simply raises the score and plays the iconic Super Mario coin sound. After that, we provided a script for "onScoreChange" that updated the UI text with the modified value for the user to see. Then we added the game logic that when a coin collides with a player, the coin's pick coin animation is played, and the addOneOnScore method is triggered at the end.

We introduced a relationship between the object and a higher level of abstraction, the Score, at this stage, with very little and easy-to-understand code. We then proceeded to develop
enemies that kill or shrink Mario using the same principles. As a last stage, we added the flag motion, which marked the end of the Scene and allowed us to go on to the next.

Game logic is by far the most difficult aspect of the development process, and it is also the phase in which we have included the most functionality to assist the user in overcoming any issues that may arise during the process. Even as experienced developers, we had to make extensive use of the features in order to achieve the results we desired. Our first thought, especially whenever we added a new script, was to start a recording and make sure everything was operating properly. Snapshots were on the same level; we used them a lot when we wanted to see if we had the desired result. For instance, if we wanted to test the pace of the ending animation, we could simply use the snapshot, update the object properties and scripts in real time, and try again with alternative values until we were satisfied.
Chapter 7

Conclusion and Future Work

Conclusion

In this thesis we present a game development environment that focuses on introducing users with little to none prior knowledge, into programming and coding concepts through game development.

Our system incorporates all the necessary tools to create a simple game with animations, grid manipulation, sound management and much more. It aims to provide an enjoyable experience for the user by adapting to their skills and tries to introduce new concepts using familiar ones.

By integrating live development into the game environment, we are offering a different way for the user to develop games compared to the most popular Visual Programming Tools such as Scratch and Tynker.

Interacting with objects allows the user to add a new script or change the properties of objects while the game is still running. It also provides the user with one-of-a-kind feedback that will assist them in dealing with any challenges that may emerge during the development process.

Additionally, we introduced time travel as a new method for debugging a live system with the aim to help the user to thoroughly analyze what went wrong without requiring prior experience with debugging or debugging facilities.

Finally, we used our newly introduced features to create different mini games and more notably a remake of the Super Mario video game, demonstrating the development process.
Future Work

Given the wide usage of tablets and smartphones by the younger generation, we could use that as an opportunity to introduce them into programming. As future work, we intend on porting our current system into an application, with changes to the user interface to accommodate the smaller screen size, that will allow the users to achieve the same outcome on their smartphones.

Furthermore, we believe it is critical to present learners with semi-built games as well as to provide them with the opportunity to be creative with premade games. Especially for beginners, it is beneficial to provide such games and let the user experiment with simple modifications such as manipulating properties and making small changes to scripts.

Finally, we would like to carry out an experiment within a real-life class, including students of different backgrounds, in order to better test and evaluate our system’s features and tune our it accordingly.
Bibliography


Appendix

Extra Quality of Life Provided Features

A system's target audience is determined by the features it offers. That is why we spend so much effort attempting to find a balance in order to assist as many people as possible from diverse cultural origins and upbringings. To do this, we attempted to make clear links to the actual world to act as a bridge to effectively take them to the end goal, which is to understand programming and programming ideas. At the same time, we want our system to evolve alongside the user. That is, either the features will get “smarter” or new features will be added to offer an additional layer of difficulty.

Animation Creation

Figure 28: Preview of all the loaded films from the configuration file.
Computer scientists imagined the screen as a canvas that could display images with tiny changes in between, tricking the mind into thinking the image was moving and changing over time, providing a very fascinating illusion [22].

That is the main reason, allowing the user to input their own sprite sheet and then be able to generate animation within our system and then apply it directly within the game was a top priority for us.

To produce the films correctly, our system requires a configuration file that provides the films' basic bounding boxes, film groups, and the spire sheet from which to extrapolate the bitmap. From there, the user will be able to use these groups within the system to construct animations by selecting the delay, dx, dy, and, of course, the number of repeats before the animation finishes.

After constructing the animation, the user can use the provided API to bind and play an animation to a specific object, either in a block form (if coding in VPL) or in a text form. It is worth noting at this point that we only allow one animation to be active on an object in order to keep things simple for the user.
Object Snapshots

Figure 30: On the left side we can see the User Interface used for preview and editing purposes. On the right side we can see two ways to take an object snapshot.

The concept of snapshotting is widely used throughout the area of computer science [17] [18], from concurrency control in certain circumstances to even as a fallback point in case of a failure in others. We chose the latter as a snapshotting approach and use it as a fallback point to which someone may always return safely.

The main reason we sought a snapshotting mechanism in our system is because as the mobile era progressed, more and more individuals were exposed to smart phones. People are increasingly enthusiastic about the idea of photographing something they wish to remember forever. As a result, we believe that this concept is simple to grasp from the user's standpoint, and that it will assist them in overcoming any obstacles that may arise.

Our snapshotting system allows a user to take a snapshot of an object that contains all of the data, codes, and methods that the object contained at the time of the snapshot. After that, the user can return the object in that state (even if the object is removed).

The user can take a snapshot in two methods, as shown in Figure 30. The first is a button on the object information menu (shown as number 1), and the second is a button on the context menu (shown as number 2), which is accessed by right-clicking on an object. We also provide a User Interface that allows you to preview and edit the snapshots you've taken.
Scene Snapshots

![Scene snapshot interface](image)

*Figure 31: Scene snapshot User interface for scene swap.*

Building the scene is one of the most important aspects of game creation. In order to do so, a new technique for switching between scenes had to be created, especially in a live context. We determined that the ideal option was to allow the user to construct scenarios that could be snapshotted and used to transition between them. The key reason we chose this method was that it allowed the user to build on the information they had gained from the object snapshotting we described earlier in order to make the concept easier to learn and comprehend.

Copy & Paste with Clipboard

![Copy & Paste interface](image)

*Figure 32: Two ways to copy an object.*

Users increasingly expect quality-of-life features from system developers, and without a doubt one of the most useful quality-of-life features is the copy and paste mechanism, which allows the user to copy an object and its properties and produce a new exact copy of that same item.

For us, preserving all of the data associated with an object is adequate for copying it, as it contains all of the scripts, methods, and data that will be passed to the new object. Of course,
the new item will differ from the prior object in two ways. First, the name will be different since we have a unique name policy, and we have embraced the practice of appending a unique id to new objects' names as a rule. Second, when the new object is created, it will be assigned a unique ID.

We strongly advise users to utilize the API connected to "this" when referring to the object itself in order for copy and paste to work without serious flaws. This manner, through object transfer, the same scripts can be kept without the need to change the code.

**Object Renaming**

![Image of renaming process](image)

Figure 33: Renaming process using provided UI.

Renaming is an expected feature of such a tool, but it has its restrictions because we always want to maintain our unique naming convention. A major issue with renaming is that many functionalities are accessible using name indexing in order to make things easier for the user (for example collision registration). In this sense, the user should be aware that using the renaming feature may have unforeseen consequences.

As shown in Figure 33, the renaming process is as simple as one might expect, beginning with a simple edit button next to the object's current name, followed by the user being prompted to type the new name into the text field, and finally, if every convention is being followed (unique naming), the new name is applied and all changes are made in the system internally.
Populate and Manipulating Object Data and Methods

![Before and After](image)

*Figure 34: Before a new flag data is added there is no “on extra change” after it is added a new script showed up.*

Because the object is an instance of its category template, the data and methods of objects in the same category will differ dramatically. Because each object has its own data and methods, this is the case. Making it simple to produce and change data and procedures is critical in this regard.

As mentioned in paragraph 3.2, when a new method or data is added, a new script is created automatically, as seen in Figure 34.

**Audio Management**

The auditory system is particularly effective in creating immersion [16] in games, according to research. Someone can also claim that sound and music are critical in creating a game's environment. That is why we created a simple sound system that allows users to add specific sounds to the system and then activate them using code on specific occurrences.
Grid Terrain

![Image of Grid Terrain](Figure 35: Live grid preview (red overlay over the solid objects) top without active preview bottom with.)

Most game developers are aware of the importance and necessity of the grid, particularly in platform games. Most of the time, a grid is employed to create the illusion of a game area and the location where the game should be played.

In our case, the grid is made up of objects with the “isSolid” flag set to true. These things are identical to all other ordinary objects, with the exception that they cannot be passed through by other objects. The grid is updated each time an object with that flag activated is moved to ensure that it is constantly consistent.

As always, we provide a mechanism to go around the grid by applying the “moveThroughGrid” flag to an object, which allows it to move freely regardless of whether or not there is a grid.

Moving Platforms

When it comes to 2D games, moving platformers have always been a fan favorite, and we wanted to give a novice user the possibility to add them into the game. Moving platforms, on the other hand, are incredibly difficult to code for a novice user.

To address this issue, we offer a user-friendly movable platform system. Simply, by setting the flag "isPlatform" to true, the object will serve as a moving platform, and any objects attached to it will move the same amount as the platform when it moves.
Object Search

Figure 36: Search result that shows objects and objects that contain attribute.

Figure 37: Search within a popup for faster indexing.

We needed to find a means to enable quick access to objects that weren't immediately accessible with the click of the mouse as part of our goal to provide a positive user experience and maintain our design strategy to never let the user be more than 3 clicks away from the intended result. That is why, we incorporated the old fashioned but always reliable search that (almost) everyone is familiar with.

However, we wanted to offer more than simply a search for the objects that are currently in the game. We provided the user the option of searching for objects depending on their properties. The rationale behind this feature is that if there are a lot of objects, the user may remember a method name or a data name rather than the object name after a while. As demonstrated in Figure 36, it also allows for a rapid glimpse to see which objects hold that certain
property. Of course, a search tool is useful in many situations, which is why a search bar is frequently used within list windows, as illustrated in Figure 37.

**Mouse Interplay**

Because the mouse is such an important element of the design and creation process, we try to use it as much as possible, from moving objects around to using it as a gameplay component. Because it is the simplest way to concentrate an object in order to modify a value or manipulate a script, the mouse is usually employed throughout the development phase of the process. It can also be used to open the context menu, which allows you to perform quick actions on that location or object.

**Restrict Axis Movement**

We integrated the feature to prevent needless movement of objects, which was inspired by editing tools like Figma [23], Photoshop [24], and even game engines like Unity [13]. In fact, if the user just wants to move an object along the Y axis, pressing the "Control" button will limit the object's movement along the X axis. And with "Shift," the object will only move on the X axis while restricting movement on the Y axis.

**Movable User Interfaces**

![Image of Movable User Interfaces]

*Figure 38: Demonstration of why moving User Interfaces is important.*
Following the evaluation, many users stated that the User Interface can get in the way when they are attempting to develop something in the game, and that it can be tedious at times. As a result, we made every UI element in the game area moveable.

The user is now able by pressing “Control” and drag the User Interface element to move it to a new place so access to objects won’t be restricted as shown in Figure 38. On the top part the objects are not accessible if another object is focused but if we move the coding area then the objects are easily accessible again.

**User Interfaces Toggling**

![Figure 39: Enabled and disabled User Interfaces that the user is able to toggle.](image)

As with most large systems, there are numerous UIs involved, each serving a specific purpose, making it difficult for a user, particularly one with no prior familiarity with such systems, to navigate through it. As a result, we offer a User Interface Toggling System.

Based on the user's expertise and level of confidence, the user should be able to hide or change which UIs are visible. Of course, the user may always opt in for a UI that is disabled by default and out for one that is enabled.

**Record and Replay Keystrokes**

The ability to record and replay each keystroke at precise intervals is a clever idea inspired by Tool Assisted Speedrun tools (aka TAS) [26]. At first look, it may not seem vital to provide such a
capability, but it might be very useful to be able to duplicate exactly what happened in order to reproduce a bug when certain actions and game manipulation are required.

Recording and replaying can be used for a variety of purposes, including providing a tutorial or even as a gameplay mechanic. Because it uses an API, the user can also plan ahead for certain actions to take place at a specified time.

**System Guidance**

![Create object (1/4) guidance prompt](image)

*Figure 40: Guidance prompt to help the user achieve a certain goal.*

It is critical to provide a rapid and easy-to-understand walkthrough of the system, especially for systems created for those with no prior familiarity with similar systems. That is why, as shown in Figure 40, we designed a system guidance that will assist the user through the system while delivering meaningful instructions on how to accomplish certain goals.

**System Translation (Multi-Language)**

![System translation examples](image)

*Figure 41: Examples of editor translation.*
We wanted everyone to be able to use the system, which meant that non-native English speakers had to be included. That is why, in order to feel more secure and confident, we implemented a translation system that will translate all of the text that the user may view on the screen into their native language. Only two languages have been fully integrated thus far (English and Greek).

Editor Scaling Based on Screen Size

![Figure 42: Scaling example with a reduced screen width.](image)

Most inexperienced programmers do not consider the potential of someone playing their game on a different screen size while building it. As a result, we decided to provide a scaling method for the editor based on the user's screen size, while keeping the ratio and view window the same by adding extra greyed out space where required as seen on Figure 42.
HTML Injection as Game Objects

When designing a game, we frequently want to integrate something from the internet, which is why we allow users to inject any HTML element they want into the game, which are then handled as game objects. To improve the experience even more, this can be used to integrate iframes, basic div components, or even more advanced HTML.

When injecting an HTML element and moving on to saving the game, the game object system is responsible for saving the element as is so that it may be recreated later using the same information.

We can distinguish an iframe inserted in the game as a game object in Figure 43, with all of the typical game object functionality (properties, events, scripts). It features an internet-retrieved page that can be changed while it is in focus.
System’s API

As mentioned throughout the thesis, our system provides an API for a range of uses. There are various locations where someone can contribute code to increase the system's capabilities and usages. The API we provide is served by a single object that can be imported and used in conjunction with the blackboard to allow for complete system interaction.

Accessing API

To use the API, you must first import the Engine.js file, which is located in the system's src folder. Then, by just gaining access to the manager, anyone can utilize the offered API, which we shall list below.

The following is an example of how to use the API after importing Engine.js as AK:

```javascript
AK.ObjectManager.getObjectByName("player");
```

For the API will use the following as types:

- GObject = Game Object.
- Callable = Function or String (that can be evaluated).
- TimeCall = An id that will help the engine identify the callback.

Core Managers API

Animation Manager

- getAnimationCategory(catID: String): Returns the constructor of a given category for animations.
- getAnimatorCategory(catID: String): Returns the constructor of a given category for animators.
- progress(gametime: Number): Progress all running animators.
- timeShift(dt: Number): Timeshift all running animators.
- getAllFilms(): Returns all the films that have been loaded.
- getFilm(filmID): Returns the film with the provided ID.
- getAnimation(animID: String): Returns the animation with the provided ID.
- removeAnimation(animID: String): Removes the animation with the provided ID.
- getAllAnimations(): Returns an array with all the animations.
- getAllAnimators(): Returns an array of the running animators.
- restoreAnimators(anRep: Array<Animators>): Destroys all the currently running animators and apply the new ones.
- playAnimation({object: GObject, anim: Animation,.onStart: Callable, onFinish: Callable, animator: Animator}): Play animation on a specific object.
Collision Manager
- **installCollision(obj1ID: String, obj2ID: String, code: Callable):** Adding a new Collision to the manager between two objects.
- **removeCollision(obj1ID: String, obj2ID: String):** Removing the collision.
- **getAllCollisions():** Returns an object with all the collisions.
- **getCollisions(objID: String):** Returns an object with all collision involving the object with the given id.
- **getCollision(obj1ID: String, obj2ID: String):** Returns the collision (if it exists) of two objects.
- **setCollision(obj1ID: String, obj2ID: String, code: Callable):** Update the code to be executed on collision of two objects.
- **updateObjectName(objName: String, newName: String):** Update the name of an object within the collisionholder.

Input Manager
- **keyPressed(key: String, keepFiring: Boolean):** Mark the key as pressed and choose if you want to keep adding it on pressed on each getPressedKeys() call.
- **keyReleased(key: String):** Mark the key as released and it will return it as released key on the next getReleasedKeys().
- **getReleasedKeys():** Returns an array of all the keys released since the function was last called.
- **getPressedKeys():** Returns an array of all the pressed keys since last checked (also includes the keys that are marked as keep keepFiring).
- **setCombo(id: String, keys: Array<String>):** If these keys are being pressed at the same time this combo will be returned as pressed.
- **getCombo(id: String):** Returns the keys for that specific combo.
- **removeCombo(id: String):** Removes the combination with the given id.
- **addOnKeyPress(cb: Callable):** Adds a function that will be called when a key is pressed and will pass the key as the first argument. Returns a unique ID for the cb.
- **removeOnKeyPress(cbID: String):** Remove onKeyPress cb with the provided ID.
- **addOnKeyRelease(cb: Callable):** Adds a function that will be called when a key is released and will pass the key as the first argument. Returns a unique ID for the cb.
- **removeOnKeyRelease(cbID: String):** Remove onKeyRelease cb with the provided ID.
- **isCurrentlyPressed(key: String):** Returns true if the key is currently pressed.

Object Manager
- **objects:** Return an object that contains all the alive objects currently on the game.
- **getObject(objId: String):** Return the object corresponding to this id if it's alive.
- **getObjectByName(objName: String):** Return the object corresponding to this name if it's alive.
- **rename(obj: GObject, newName: String):** Rename an object if the new name doesn’t exist.
• `isSystemObject(objId: String)`: Return true/false if the object is listed as `systemObject` (like scene, collision, keyboard).
• `addSystemObject(objId: String)`: From now on `isSystemObject(objId)` will return true for this id.
• `renderAll()`: Will render all the objects to the screen. (it's called by Engine each loop)
• `addToWorld(newObject: GObject)`: Add `newObject` to the world. (Usually called within the initialisation of the object itself)
• `removeFromWorld(aliveObject: GObject)`: Removes `aliveObject` from the world. (Usually called within the deconstruction of the object itself)
• `addConstructor(name: String, constr: Callable)`: Installs a new constructor for objects. (Installed by ObjectManager itself)
• `removeConstructor(name: String)`: Removes an installed constructor from the manager
• `getConstructor(name: String)`: Returns the constructor with that name.

**Save Manager**
• `setEngine()`: This function is crucial in order to load the game because it loads the information needed. Must be called before Engine.start().
• `saveObjects()`: Will save the objects to Database if you are database is available or download a JSON file.
• `saveGame()`: Saves the current game in the database or downloads a new JSON file with the game data.
• `getObjects()`: Will retrieve the objects either from Database or from the JSON file.

**Scripting Manager**
• `getCurrentEditorID()`: Returns the id of the current editor.
• `setNewEditor(editorID: String)`: Sets the editor with the provided editorID as the new default editor.
• `currentScriptAsText()`: Return the current script on editor as Text (whatever representation the current editor wants as string).
• `currentScriptAsCode()`: Return the current script on editor as Code (Javascript code as string).
• `executeCode(codes: Object{String, String}, currentObjectID: String)`: Executes the code saved from `currentScript`.
• `clearAndLoadFromText(codes: Object{String, String})`: Clear the editor and load the new text on editor.
• `injectInDiv(div: DOMElem):` Called from UI in order to notify that the scripting UI shall be added there.

**Sound Manager**
• `addSound(tag: String, path: String)`: Install a new sound to the soundManager
• `removeSound(tag: String)`: Remove a sound from the manager
• `getSounds()`: Return all the installed sounds
• `playSoundOnRepeat(tag: String)`: Plays a sound on repeat until it is stopped by either `stopSoundOnRepeat` or `stopAllSounds`
• `stopSoundOnRepeat(tag: String)`: Stop a sound that is playing on repeat
• `playSound(tag: String)`: Play an installed sound. (If this sound is already playing it will restart)
• `stopSound(tag: String)`: Stop a sound that is playing.
• stopAllSounds(): Stops all the sounds currently playing.

Time Manager
• update(): This function is updating the game time and keeps track of FPS internally. Must be called each frame once.
• callIn(callback: Callable, args: Array[any], delay: Number): Similar to setTimeout(cb,delay) but according to game timer and it triggers the callback at the end of each update if the new timer is after the delay.
• cancelCallBack(returnedObjectOfCallIn: TimeCall): Similar to clearTimeout.

Extension Managers API

Clipboard Manager
• push(item: GObject): Push information to clipboard and also on collection.
• top(): Returns the clipboard item.
• getCollection(): Returns all the items on the collection.
• clearClipboard(): Clears all the clipboard collection.
• removeItem(index: Number): Removes the item in the position “index”.
• copy(obj: GObject): Copy Object to clipboard.
• paste(): Create a new Object based on the copied object on clipboard.

Calculation Manager
• distanceTwoPoints(p1: Point, p2: Point): Get the distance between two points.
• distanceObject(obj1: GObject, obj2: GObject): Get the distance between two objects (center).
• getObjectCenter(obj: GObject): Returns the {x, y} of the center of the provided object.
• distanceObjectWithMouse(obj: GObject): Get the distance between an object (center) and the mouse.
• getMouseX(): Get the mouse X value.
• getMouseY(): Get the mouse Y value.

Grid Manager
• calculateGrid(): Recalculates the grid.
• getGrid(): Returns the grid.
• objectsOnPlatform(platformObj: GObject): Returns an array of attached objects.
• isPointInGrid(x: Number, y:Number) calculates if a point overlaps with the grid.

Input Automation Manager
• startRecording(): Starts a recording session.
• stopRecording(): Stops and returns the recorded session.
• startInputAutomation(recordedSession: RecSession): Replays the actions of the recorded session.
• stopInputAutomation() Stops the input automation that is currently running.
Pause Manager

- `pause()`: Pause most of the actions happening except input and rendering.
- `resume()`: Resume the game and resume animations.
- `addOnPause(cb: Callable)`: Adds a callback to be triggered on pause.
- `addOnResume(cb: Callable)`: Adds a callback to be triggered on resume.
- `isPaused()`: Returns the state of the game if it is currently paused.

Quantizer Manager

- `move(obj: GObject, x: Number, y: Number, ms: Number)`: Will start an animation to move an object smoothly from the current position by x,y in ms milliseconds (0 for instant teleportation).
- `moveTo(obj: GObject, x: Number, y: Number, ms: Number)`: Will start an animation to move an object smoothly from the current position to x,y in ms milliseconds (0 for instant teleportation).

Timewarp Manager

- `setTimewarpMechanism(twID: String)`: Switch between the timewarp mechanisms.
- `saveTimeFrame()`: Saves the current timeframe.
- `startRecording(interval)`: Start recording and save frame each interval (0 for each frame).
- `stopRecording()`: Will pause the game and will give the ability to rewind/replay or even step to a point in the recordings.
- `isRecording()`: Returns a bool if the timewarp is currently recording.
- `playForward(startingTimestamp: Number, speedfactor = 1)`: Play each frame from the frame on the given timestamp until the end with the original delay that the game had (except if you change speedfactor).
- `playBackward(startingTimestamp: Number, speedfactor = 1)`: Play each frame from the frame on the given timestamp until the first with the original delay that the game had (except if you change speedfactor).
- `stopPlayback()`: if there was a playback currently playing will stop showing the rest of the frames.
- `showSnapshot(timestamp: Number, frame: Number)`: Display the frame on the given time Stamp on the screen. if frame is provided it will show the end user the frame that is currently showing.
- `getRecordedTimestamps()`: returns an array with all the timestamps that the recording managed to collect.
- `resumeFromRecording(timestamp: Number)`: Display the frame on the given timeStamp and then resume the game and the animators so the user is able to continue the game from this point.
- `isReoccuring()`: Returns true if the current timewarp mechanism is reoccurring.
- `saveTimeline()`: If the current timewarp mechanism is reoccurring then it saves the current recording.
- `getTimelines()`: If the current timewarp mechanism is reoccurring it returns all the saved timelines.
- `clearTimelines(index: Number)`: If the current timewarp mechanism is reoccurring clear the timelines after the index.
- `setTimeline(index: Number)`: If the current timewarp mechanism is reoccurring then set the current timeline to be index.
• getCurrentTimeline(): Returns the index of the current timeline.
• getDiffs(): Will return the Delta States that happened on this frame.

Snapshot Manager
• snapshotObject(obj: GObject): Snapshots given object.
• getSnapshots(obj: GObject): Get all snapshot from that object.
• getAllSnapshots(): Return the whole collection of snapshots.
• removeAllObjectSnapshots(): Clears the collection of snapshots.
• removeObjectSnapshots(objID: String, snapID: String): Removes a specific snapshot from the collection.
• resetObjectToSnapshot(objID: String, snapID: String): Reset an object to a specific snapshot.
• snapshotScene(optionalName: String): Takes a scene snapshot.
• resetSceneToSnapshot(sID: String): Reset scene to specific snapshot.
• getAllSceneSnapshots(): Returns the collection of the snapshotted scenes.

Tutorial Manager
• setTutorial(steps: Array<Steps>): set the current tutorial to the given steps.
• showStep(index: Number): Show a specific step from the tutorial.
• finishTutorial(): Finishes the tutorial.

System’s Blackboard

Our system includes a blackboard that may be used to send out events to possible listeners. It is mostly used for reactive programming in response to certain activities or to save extra information.

Someone must include the blackboard from utils / blackboard.js in order to monitor a variable. If we import the blackboard as bb, an example of its use would be:

`bb.installWatch("events", "nameOfEvent", callback(val));`

If the variable "events", "nameOfEvent" is set after that, the callback will be executed and the value set as the first argument will be sent. A user can also initiate an event by altering the value of a variable. For the same instance:

`bb.fastSet("events", "nameOfEvent", "this is a test"); // Trigger watches on "events", "nameOfEvent" and also set the variable name as "this is a test".
bb.fastGet("events", "nameOfEvent"); // Shall return “this is a test”`

Important Blackboard Variables

**events**

• last: Provides registry for every event on the game.
• contextMenu: Context menu was accessed by the user.
• gridUpdated: Update that the grid changed.
• setOption_{optionName}: Update if an option with the name “optionName” was changed.
• setValue_{optionName}: Update if an option with the name “optionName” was changed.
- addObject: When a new object is added.
- removeObject: When an object is being removed.
- renameObject: When an object is being renamed.

**Engine**

- self: Return the engine itself.
- {ManagerName}: Returns the manager.

**actions**

- {ActionName}: Returns the Action.

**UI**

- getLoadedUIs: will return a function that upon call will return the currently loaded UIs.
- getUIs: will return a function that upon call will return all the UIs.
- hideUI: will return a function that upon call hides a specific UI.
- installUI: will return a function that upon call installs a new UI.
- loadUI: will return a function that upon call will load a specific UI.
- removeUI: will return a function that upon call will remove a specific UI.

**state**

- FPS: Returns the FPS of the system.
- focusedObject: Returns the currently focused object.
- gameTime: Returns the gameTime of the current frame.
- mode: Returns the current mode of the system.

**settings**

- {settingName}: Will return the state of that state.

**urlParams**

- {paramName}: Will return the state of that url param.

**assets**

- {assetName}: Will return item of that asset.