The Icarus Protocol

Prototype Project Report

Lincoln Middle School

Collin Nelson;

Anna Ueti;

[11.18.22]

Table of Contents

Introduction

I. Project Introduction

In 2021, the games industry reported estimated revenues of about 198.40 billion dollars, solidly positioning themselves as one of the world's most immensely popular and valuable forms of entertainment. This growth is only projected to continue rising as well, with market research firms anticipating revenues as high as 339.95 billion by 2027 (*Gaming market size, share: 2022 - 27: Industry growth* 2021). The power of games to capture the minds and attentions of players across the world cannot be discounted, and given this power, there is also incredible potential for games to capture the attention of players and direct it towards learning and personal advancement as well as towards entertainment.

There is a powerful overlap between games and education, an overlap that innovators have only recently begun to truly explore. The feedback loops and reward models designed and honed to keep people engaged in modes of entertainment can be useful for intriguing students in class work with a reputation for being uninteresting. Our team aims to explore the potential of games as a tool for education by producing a fully featured game designed to teach basic programming skills to students at a middle school or early high school level with no prior programming experience. While games of a similar nature exist, they often fall into one of two traps which our team sees as pitfalls. Some, while employing the surface level appearance of a game, fail to truly embody the game design principles that make games powerful for learning; Others use a proprietary scripting language that has lessened impact in teaching actionable programming skills. In the construction of this project (working title: "Icarus Protocol") we aim to solve this problem by producing a game that is a genuinely fun and interesting experience, while also serving as an effective tool for teaching real Python programming.

II. Background and Related Work

In order to evaluate current leaders in the same domain as our project, we must first define that domain. For the purposes of this document, we have narrowed down this definition to "*interactive gamified services designed to teach programming.*" Given this definition, our research highlights three representative and distinct examples of highly successful existing products in the same domain.

The first notable leader in the domain, and the closest analogue to our work, is the service CodeCombat (CodeCombat, 2013). They create a number of educational tools, but the original product "CodeCombat Classroom" is of the most relevance to us. Their game, originally released in 2013, teaches programming to a similar age group to our target demographic using a series of fantasy-themed levels covering programming concepts in Python, C++, and Javascript. They have also implemented several potentially useful features, such as integration with Google classroom for teachers. Due to the larger scale of their company, they have also been able to fund studies on the effectiveness of this learning model in the classroom. The most recent of these, released in 2019, shows that 90% of teachers agreed that this learning model kept students engaged, and 97% agreed that the gain to students was worth their time. (Danks et al., 2019) It also contains several other useful insights about factors such as ideal session time which will be useful in constructing our project. While a survey of CodeCombat's curriculum shows some overlap with the intentions of *Icarus Protocol*, the structure of their lessons lends itself to AI programming, which while generally good for teaching basic concepts, makes it

difficult to teach concepts such as data conversion or file operations. Our hope is that *Icarus Protocol* can innovate in this field by using a narrative context that lends itself to a broader array of curriculum.

The second state-of-the-art contemporary of note is the online coding school *Codecademy (2011)*, which teaches many different professional programming languages. While it doesn't specifically target middle schoolers, it also teaches lessons assuming no prior experience with the relevant coding languages. They are set apart by the variety of lessons they offer, and the breadth of curriculum available on their site. While they opt to not integrate full gamification into their service, they do employ progress bars and completion badges to strengthen game-like feedback loops.

The third related work that I would like to highlight in our research is the simple online game creation platform *Scratch (2007)*. This MIT project allows people to create games with a simple all-inclusive editor. It is not in and of itself a game, but it teaches programming concepts through the game creation process. It utilizes a proprietary building-blocks programming language with which the students can script the behavior of sprites created in the program. This tool is particularly important to our project as it is one of the tools currently being used by Lincoln Middle School during their units on programming. It has the advantage of being very powerful, and broad in the concepts it can teach, but cannot function autonomously since it doesn't come with a structured curriculum. It also teaches fundamental programming concepts, but opts not to use a real programming language to reduce complexity. With *Icarus Protocol*, we aim to provide an equally useful tool to accompany or serve as an alternative to Scratch in the classroom at Lincoln MIddle School, but one which is capable of autonomous function and which teaches true actionable programming skills.

In summary, while there are several notable existing services that fall within the provided definition of the project domain, very few accomplish the true intersection of game, educational tool, and real programming simulation that we aim to create in *Icarus Protocol*. Our project is distinguished from services like CodeCombat, which do also accomplish these things, by the curriculum that it is enabled to tackle, and the platforms it is designed to operate on. It is also distinguished by its distinct theming and narrative elements, which are unique from any prior service or tool uncovered in my research.

In order to accomplish the technical aspect of the project there are a few tools and frameworks we will have to master or improve our skills in. The most important is Unity. As the engine for the game, experience with its setup, debugging, and optimization tools will be required to make the game function, and function efficiently enough to be smoothly playable on the often outdated computers that are available to students. In addition, in order to integrate real Python programming into the game we intend to use the IronPython framework. Our team does not currently have any experience among us with this framework, so skills will need to be learned as the project progresses. The final notable technology with which our team is not already familiar is the Chrome OS and the process for building executables for it. While we will certainly be able to build for Windows, our desire to make the project as available to students as possible will require us to look into the requirements for Chrome applications, and potentially the process for submitting work to the Chrome Store.

III. Project Overview

There is incredible potential in games and game design principles to forward the field of effective education. Since the early 2000s, researchers have been talking about how

educational games should be “like the school corridor, where kids experiment, interact, create, and share” (Squire & Jenkins, 2003), and even earlier science fiction authors like Orson Scott Card envisioned a world in *Ender’s Game* where students learned complex battlefield strategy through interactive simulations. (Card, 1985). *Icarus Protocol* aims to explore the use of games to engage kids in educational settings by placing them into the role of a starship AI on a quest to repair its malfunctioning vessel and discover the source of the damage.

In doing this there are several key overarching objectives that define the nature of the project. The first of these is that the game must be *fun*. Many other similar games fall into the pitfall of being coding simulations first, and games second. They ignore many of the feedback systems and incentive structures that make games effective at prompting player learning, and this causes them to lose some of the impact they could otherwise have. *Icarus Protocol* needs to employ these systems in full, existing as equal parts educational tool and game, and using systems like level scoring, completion percentages, secrets, and challenge modes to make the experience exciting and engaging to play.

Additionally, *Icarus Protocol* would be failing in its purpose if it isn’t an effective tool for teaching the intended programming skills. While being fun to play, the game must also be demonstrably effective at allowing students to gain transferable and extendable coding skills without needing to be accompanied by traditional lectures or homework assignments. It should be able to operate independently (i.e. without assistance from a teacher) and help students to solve their problems and correct their mistakes.

The project will be designed and built from the ground up in the Unity Engine, which will serve to accelerate the process of game development, improve the general quality of the final project, and allow for simultaneous builds to multiple key platforms. Design and development will take place on Windows, and the game will be built for Windows PCs, optimized to be smoothly playable on laptops available to students. In addition, if features are ultimately compatible, a secondary build will be constructed and deployed for Chrome OS, intended to be played on the student-issued chromebooks used by Lincoln Middle School.

The game itself will consist of a sequence of levels, each themed after a different malfunctioning system of the starship. Each level represents a single coherent concept that the game intends to teach to the students. As a baseline for simple programming skills, the game will have levels on ideas such as arithmetic statements, function calls, if statements, loops, and lists. Each level will be constructed of a series of “phases”, of increasing complexity, beginning with an introduction to the concept and culminating to a programming task that requires the synthesis of the new concepts with concepts introduced in previous levels.

The game should use reward systems to incentivize the player to achieve full level completion, and should also use systems which reward novel, intelligent, or especially efficient solutions to the more complex programming tasks. In addition, the game should be flexible in accommodating optionally difficult challenges for students who feel they have a firm grasp of the material and wish to stretch their abilities.

All of the above objectives should be shaped and guided, though not inherently constrained by the Washington State 6-8th grade learning standards for computer science, particularly educational standards 2-AP-10 through 2-AP-19, which describe the learning goals and standards in computer science classes for students between the 6th and 8th grade in Washington State.

IV. Client and Stakeholder Identification and Preferences

Our primary client is Lincoln Middle School, where Mr. Davis, the school's primary technology teacher, will be our contact liaison. The final project will be predominantly used by LMS as an introductory tool for students to gain experience with the Python programming language. There are multiple stakeholders within Pullman School Districts, which include but are not limited to Lincoln Middle School.

In order to enhance core programming fundamentals retention in middle school students, we're working with Lincoln Middle School to create an educational video game that uses positive feedback loops and a reward model framework. This project will require a released build of the game, and the deployment of the game to Windows PC for the Lincoln Middle School computers.

Potential clients would include other K-12 educational institutions. To appeal to these institutions, it would be important for the final build of our software project to be easy to download and install. Additionally, it would be greatly beneficial if the curriculum adheres to the CSTA's guidelines for 6-8 grade education. (Ospi, 2018)

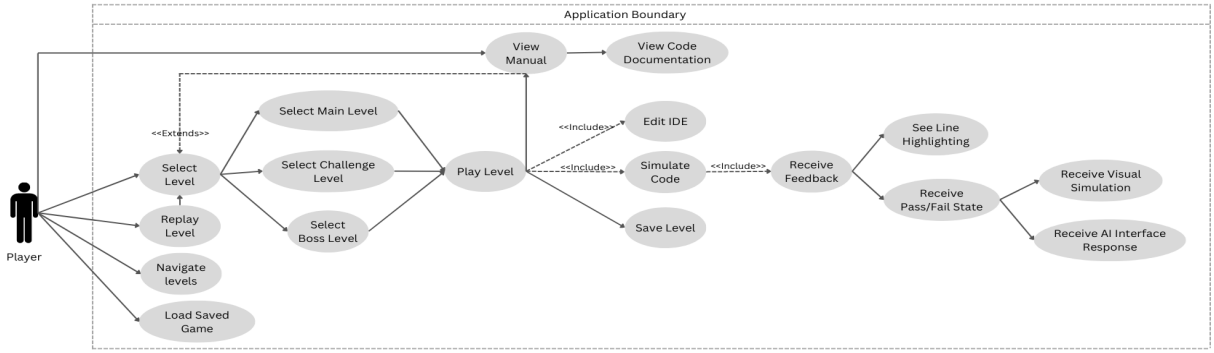
Finally, all stakeholders of the project would benefit from clean and well-documented code that is easy to deploy and to extend. The *Icarus Protocol* team will endeavor to treat these needs as prerequisites as we fulfill the preferences identified above. The needs of our primary client (LMS) will be prioritized first, but the needs of other institutions will be considered throughout the design and development process.

Team Member Introduction

Anna Ueti is a Senior at Washington State University and will graduate with a degree in Computer Science in the Spring 2023. After graduation, she intends to pursue a full-time junior developer position at Schweitzer Engineering Laboratories. She has interests in web and front-end development, and has experience in these two areas from her three year internship as a Software Engineer Intern at Schweitzer Engineering Laboratories. Anna's technical skills include proficiency in scripting languages such as css and html, and the object-oriented language C#. Anna's responsibilities include team coordination, and UI design and implementation.

Collin Nelson is a Junior at Washington State University pursuing a degree in Computer Science, and plans on graduating in the Sprint of 2024. Collin had interests with game design and plans to join a major game studio after graduation. Collin has previous experience with game design within his personal projects. Additionally, he has program design experience and technical skills in C# from his internship at Schweitzer Engineering Laboratories, where he has been a Software Engineering Intern for the past 4 years. Collin's responsibilities include team management and communication, as well as architecting the game's back-end code.

V. Use Cases



*See the glossary for more in depth descriptions of diagram stakeholder

Select Level

Pre-condition	- On level select screen
Post-condition	- Level information is displayed - Play button is available
Basic Path	1. Locate level icon in level list 2. The user selects a main mission by clicking 3. The level name, and progress is displayed
Alternative Path	- In the 2nd step, the user selects a challenge level. Instead of level progress by phase the complete and incomplete objectives are displayed.
Related Requirements	- Level-Select - Main Levels - Challenge Levels

Replay Level

Pre-condition	- On level select screen
Post-condition	- Level information is displayed - Play button is available
Basic Path	1. Locate a completed level icon in level list 2. Click on level to show data about level

Alternative Path	
Related Requirements	- Level-Select

Navigate Levels

Pre-condition	- On level select screen - Level information for a level is being displayed
Post-condition	- New level information is displayed - Play button is available
Basic Path	1. Locate a different level icon in level list 2. Click on level to show data about level
Alternative Path	1. The user interacts with hotkeys (such as arrow keys) or with menu buttons to navigate to the next level or previous level.
Related Requirements	- Level-Select

Load Saved Game

Pre-condition	- On main menu screen
Post-condition	- Previously completed levels are reflected in level list
Basic Path	1. Locate the Continue button 2. Click on the Continue button
Alternative Path	
Related Requirements	- Saving and Loading - Cloud Save and Load

View Manual

Pre-condition	- On level select screen
Post-condition	- Manual of code examples is displayed
Basic Path	1. Locate manual icon 2. Click manual icon
Alternative	1. At the precondition step, Player is on the Level screen

Path	
Related Requirements	- Accessible In-Game Python Documentation

Play Level

Pre-condition	- Selected level and main level type
Post-condition	- Receive level feedback
Basic Path	1. The player enters the level, the screen displays a section for the python IDE, the ship simulation, and the instruction/tip panel. 2. The player click on the panel for the python IDE, typing to modify the python code 3. The player clicks on Simulate button 4. The Python background tests check and determine whether the player's code is successful or not. 5. The code is successful, the ship simulation plays an animation for the task being successful 6. The level data is saved
Alternative Path	- At the 5th step, the code is not successful, the ship simulation plays an animation for the task failing, and the player returns to the 2nd step.
Related Requirements	- Functional Python IDE - Operational IronPython Integration - IDE Syntax Highlighting

VI. Functional Requirements

1. User Interface and Python IDE

Functional Python IDE:

Description	The application needs to contain a functional Python IDE for writing and simulating code in-game. This IDE needs to allow the user to write code either freely, or by filling in blanks in a fixed code-block. The resulting code then must be packaged as a python string to be simulated by the engine.
Source	Internal requirements elicitation among members of the team.
Priority	Priority Level 0: Essential and required functionality.

IDE Syntax Highlighting:

Description	The Python IDE should implement some level of basic syntax highlighting. This highlighting should be able to recognize and distinguish between strings, integers, and applicable keywords, highlighting them with different text colors to make programming easier. In addition, background colors or lines in the IDE should be used to indicate the tab index of a block of code, helping to distinguish what is contained within certain code blocks.
Source	Internal requirements elicitation among members of the team.
Priority	Priority Level 1: Desirable Functionality

Level-Select:

Description	The application needs to feature a UI that allows the players to select a level, view level details, and choose to play the level if they wish. This UI needs to accommodate the potential addition of levels outside of a linear mission order, should such levels be added to the game. This UI must also limit players to selecting levels they have unlocked by completing certain prerequisite conditions.
Source	Internal requirements elicitation among members of the team.
Priority	Priority Level 0: Essential and required functionality.

Accessible In-Game Python Documentation:

Description	The game must include a documentation window in its UI, accessible from the menu screens or from within missions. This documentation must include explanations of the concepts introduced in the games, including example code and detailed instructions and tips. These entries should be itemized by concept, and should unlock as the concept is introduced in the game, hiding information that is not yet relevant to the user.
Source	Internal requirements elicitation among members of the team.
Priority	Priority Level 0: Essential and required functionality

2. IronPython Integration

Operational IronPython Integration:

Description	The game must contain a successful integration of IronPython, a popular Python to .NET integration, with our C# code. This integration must allow us to run user-generated code, which must be sandboxed to prevent

	unintended access to the game code or the operating system. Errors must be handled gracefully, with no interruption to the game itself.
Source	Internal requirements elicitation among members of the team.
Priority	Priority Level 0: Essential and required functionality.

3. Core Game Systems

Main Levels:

Description	The game must feature a collection of at minimum 5 main “levels”. Each level will consist of multiple phases of escalating complexity, introducing, developing, and synthesizing a distinct concept. Specifically the game must at a minimum introduce the concepts of statements, conditionals, loops, and lists.
Source	Internal requirements elicitation among members of the team.
Priority	Priority Level 0: Essential and required functionality.

Boss Levels:

Description	The game will end with at least one “boss level”. The goal of this level is to synthesize a variety of already-introduced concepts into a single mission. This is distinguished from main missions by the fact that it includes no new concepts, but instead uses multiple phases of increasing difficulty to challenge existing knowledge.
Source	Internal requirements elicitation among members of the team.
Priority	Priority Level 1: Desirable functionality.

Challenge Levels:

Description	The game will include a variety of “challenge missions”. These missions are short, but difficult missions consisting of a single phase. This phase contains a unique challenge based on knowledge acquired in previous missions, but doesn’t introduce any new concepts. These challenges can be used to develop skills and provide additional completion objectives for motivated users.
Source	Internal requirements elicitation among members of the team.
Priority	Priority Level 2: Extra features or stretch goals.

Saving and Loading:

Description	As the player completes objectives, the game should automatically store their accomplishments, including facets of level completion, level unlocks, and the last simulated code for each level. This saved data should be automatically loaded if the player selects to “continue” from the main menu. This data will be stored locally in an accessible location of the user’s computer.
Source	Internal requirements elicitation among members of the team.
Priority	Priority Level 0: Essential and required functionality.

Cloud Save and Load:

Description	The game will store the saved data somewhere attached to a cloud storage service (i.e. Google Cloud tied to the user’s Google account) along with local save. If the player is utilizing this cloud storage then the game will sync the local save data with the cloud save data upon startup.
Source	Internal requirements elicitation among members of the team.
Priority	Priority Level 2: Extra features or stretch goals.

VII. Non-Functional Requirements

Self Containment:

The system shall be fully self contained. In this context what that means is that completing the game and learning the material shouldn’t require any outside Googling, or instructor help. All the resources and help should be provided in-game.

Fun:

The game shall be fun to play. It should be an experience that students enjoy going through, that teaches while engaging the player in a satisfying and enjoyable way. This can be measured through playtest surveys and player metrics.

Effective Teaching:

The system shall be an effective teaching tool. An average student playing through all of the main missions and engaging with an adequate amount of side-content should exit the experience having learned actionable skills in alignment with grade-appropriate standards. (*Ospi,* 2018)

Actionable Skills:

The system shall teach skills which extend outside of the realm of the game. An average student who completes the game and demonstrates understanding of in-game concepts, should also feel more confident in their understanding of Python, and their ability to use real programming principles.

Efficient Performance:

The system shall perform efficiently enough to be playable on machines available to students at LMS. “Playable” means that the game should generally maintain a framerate of >30 fps without clear stuttering or slowdowns during normal gameplay.

Intuitive Use:

The game should be intuitive to use for most students, and should not require the student to read detailed installation instructions, or operating procedure in order to understand how to play the game and interact with its systems.

Visual Interest:

The system shall be visually interesting and engaging to look at. This means that most actions should result in some kind of visual and/or auditory feedback to the user, and that success and failure should be marked with distinct visual effects.

VIII. System Evolution

One of the major factors of consideration within this design is Software Evolution. Part of the nature of this project is user play testing with the students within Lincoln Middle School (LMS). Once the alpha build of the game is available, our team will have a small group of testers at LMS play it and give feedback regarding their understanding and any problems that arise. Additionally, our team will debrief with our contact liaison, Mr. Davis, who is the student’s technology teacher, about the teaching environment while the students were testing and any other observations and feedback he has. Our team will do this testing process several times. With each iteration, we will refine, refactor, or modify the game so that the experience is improved. With this process in mind, we will create code that is modular and well documented so that it is easy to modify and extend, as needed. Furthermore, our team is aware that this is a preliminary testing design and is subject to change in the future.

Additionally, our team understands that the lesson requirements should be modular, as the passage of time may affect the expected standards from our client or the CSTA 6-8th grade learning standards for computer science (*Ospi,* 2018). As such, our team will design the game with modularity in mind, making it easy to update and improve curriculum as needed.

Solution Approach

IX. Project Design Introduction

This section of the document serves as a comprehensive overview of the inter-relational design of the gamified project being developed in collaboration with Lincoln Middle School. It will focus on three main design elements, Architecture, Data, and User Interface. For Architecture, each subsystem within the overarching design of the project will be thoroughly explored and explained, with a focus on the subsystem's concepts, algorithms, and interface properties. For Data, the primary data types and database files will be diagrammed and evaluated. And for User Interface, each page will have a mocked version and a detailed description of components and functions.

The purpose of this section is to create a guideline for the developers to follow throughout the development phase of this project. The developers will be able to perform verification tests during development based on the detailed description of

i. Subsystems descriptions

ii. Subsystem relationships

iii. Data type objects

iv. Data type and Database relationships

v. Database schema

vi. User Interface Mocked pages

Additionally, another purpose for this design document is for stakeholders. With these designs created and documented within this solution, stakeholders will be able to cross reference the prototype with the intended goals and design outlined here.

Our team aims to explore the potential of games as a tool for education by producing a fully featured game designed to teach basic programming skills to students at a middle school or early high school level with no prior programming experience. While games of a similar nature exist, they often fall into one of two traps which our team sees as pitfalls. Some, while employing the surface level appearance of a game, fail to truly embody the game design principles that make games powerful for learning; Others use a proprietary scripting language that has lessened impact in teaching actionable programming skills. In the construction of this project (working title: "Icarus Protocol") we aim to solve this problem by producing a game that is a genuinely fun and interesting experience, while also serving as an effective tool for teaching real Python programming.

X. System Overview

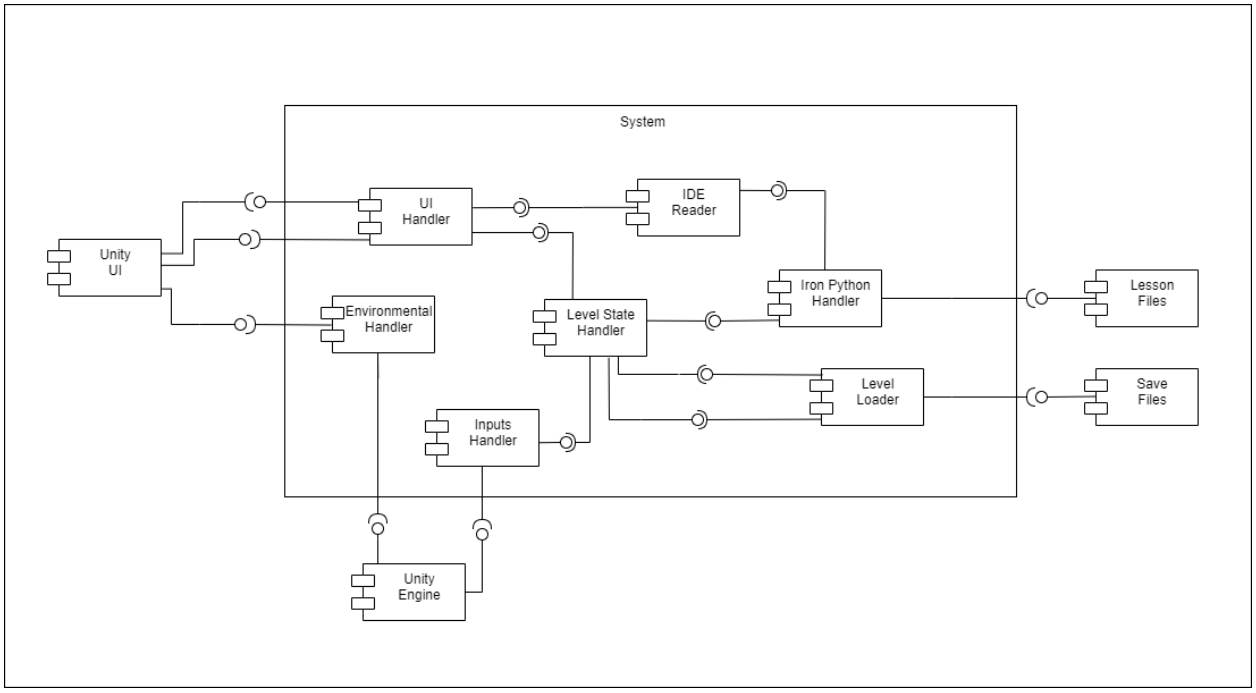
This project's main functionality surrounds the idea of teaching python to students with a novel and engaging experience, focusing on skill and knowledge retention through the avenue of gaming. The functionalities of this project can be loosely divided into three major categories;

User Interface and Python IDE, IronPython Integration, and Core Game Systems. Our design methodology is heavily influenced by the engine specific requirements caused by using the Unity Engine to drive development. The effect of this influence on our system architecture and component design patterns will be more thoroughly discussed in the Architectural design overview section. In regards to the design of our data structures, there are two major considerations that will be discussed. The first is the method of storing user progress in reliable and extensible save files. The second major consideration is the structure of our level data, which will have to be designed to be both highly flexible and easily modified without recompilation of the primary executable, and the core consideration when designing our UI will be the strengths and limitations of the Unity UI system. While this system is flexible enough to be useful for our purposes, its limitations will have to be taken into account when creating UI layouts. The specifics of our component interfaces, data structures, and UI layouts will be discussed in the following sections.

XI. Architecture Design

1. Overview

The Pluribus Doctrina Team has chosen to use an Object Scripting Model (Esposito, 2018) for this project. Since the application is developed in the Unity Engine, the architecture model must conform to the engine specifications. Given these requirements, the team has decided on an engine specific model that will utilize the strengths of the Unity Engine. Within the context of solely working on the game in the Unity engine, the team couldn't see another pattern that would take advantage of the Unity system as well. Below is a component diagram that illustrates the overarching architecture of our game. This diagram's purpose is twofold, one, it serves as a guide for the developers to reference and base development around, and two, it provides a visual representation of the component architecture that is decomposed in the following section. Here is an overview of the components, as seen in the diagram below. The Unity UI will communicate with select system handlers. Such systems would be the UI Handler, which will facilitate any UI element such as page layouts and icons, and the Environmental Handler, which modifies environmental properties such as camera positioning. From the UI Handler, there is a connection to the Level State Handler, which is the main subsystem that handles the levels which the player interacts with. The Level State Handler connects to the IronPython Handler, which facilitates the interpretation and simulation of the user's python code and loads lesson files, the Level Loader, which saves and loads levels as the player progresses, and the Inputs Handler, an abstraction for user input from unity engine. Additional in-depth descriptions of these subsystems will be elaborated on in the following section.



2. Subsystem Decomposition

2.1. [UI Handler]

2.1.1. *Description*

The UI Handler subsystem manages any user interaction with the UI elements as well as UI details such as the displayed layout and icons. It handles inputs from the Unity UI external component such as actionable functionality that is tied to UI elements or player code entered into a UI element.

2.1.2. *Concepts and Algorithms Generated*

Similar to many systems designed in Unity, the UI Handler is dispersed across many sub-classes. The most prominent of these sub-classes would be the UI-Layout class that dictates to the Unity UI which layouts should be displayed. The UI Handler also consists of the scripts tied to each UI elements' input events. In alignment with the Unity Engine's design philosophy, each button or UI element may have one or more atomic scripts which handle event actions. For the purposes of simplicity these are all considered to be part of the UI Handler subsystem. These classes will handle the interaction between the Unity UI elements and other subsystems.

2.1.3. *Interface Description*

Services Provided:

Service Name	Service Provided To	Description
UpdateLayout	Unity UI, Level State Handler	The UpdateLayout service will allow the Unity UI or the Level State Handler to call for an

		update to the page layout, this will occur for the transitions between the home page, the level select page, the manual page, and the level page, with any variations due to level phase.
BundlePlayerCode	IDE Reader	The BundlePlayerCode service will package the code written in a UI element into a compiled data object that is passed to the IDE reader, this will happen once the player has selected to simulate.

Services Required:

Service Name	Service Provided From
ModifyUI to UI Handler	Unity UI

2.2. [Environmental Handler]

2.2.1. *Description*

The Environmental Handler system manages all the interaction between the Unity UI and the application system. It also handles the interactions between the internal environmental objects and the Unity Engine. It will handle services pertaining to both the Unity UI and Engine systems, such as camera direction and scope.

2.2.2. *Concepts and Algorithms Generated*

Following a similar structure to the UI Handler, the Environmental Handler is decomposed into many sub-classes. However the one largest class is the Camera controller class. Given the strengths of working in a Game Engine, the game objects, such as layouts and UI elements, will always be on screen and the camera will display or obscure these objects as needed. Other scripts that are considered to be part of this subsystem could include motion scripts on environment objects or scripts controlling lighting and particle effects.

2.2.3. *Interface Description*

Services Provided:

Services Required:

Service Name	Service Provided From
ModifyCamera to Environmental Handler	Unity UI
ModifyEvents to Environmental Handler	Unity Engine

2.3. [Input Handler]

2.3.1. *Description*

The Inputs Handler's responsibility is to abstract the inputs from the Unity Engine for the internal application system environment. It will handle inputs from the Unity Engine and provide a sanitized version to the level State Handler.

2.3.2. *Concepts and Algorithms Generated*

Following the design philosophy of the Unity Engine, the Inputs Handler will also consist of several smaller classes. These classes will be tied to the script that is paired with game objects. The responsibilities of the Inputs Handler, while on any game object, is to act as a sanitizer for the raw inputs from the Unity Engine. It will then pass the inputs to the Level State Handler.

2.3.3. *Interface Description*

Services Provided:

Service Name	Service Provided To	Description
GetPlayerInputs	Level State Handler	The GetPlayerInputs service will abstract the interaction process of the player's input and the internal application's properties. This process will take the input from the Unity Engine as a result of player actions and will facilitate and sanitize what is provided to the Level State Handler.

Services Required:

Service Name	Service Provided From
GetPlayerInput to Inputs Handler	Unity Engine

2.4. [IDE Reader]

2.4.1. *Description*

The IDE Reader, similar to the Inputs Handler, manages packaging the user's input, specifically the input in the Python text editor UI element. The UI Handler will provide the IDE Reader the raw text from the player, which the IDE Reader will package into a compiled object in the internal application system. Afterwards, the IDE Reader will provide this object to the IronPython Handler. This adds a layer of abstraction which allows for sanitization of the user's text so as to

avoid unintentional interactions between the code written by the player and the code written by the developers.

2.4.2. *Concepts and Algorithms Generated*

The IDE Reader is responsible for the sanitizing and packaging of the player code, written in the UI element for the Python IDE, into a compiled object. Additionally the IDE Reader is responsible for providing this object to the Iron Python Handler for compilation. The IDE Reader will consist of one main class that is tied to the script of the UI element for the Python IDE. While most of this project's architecture compliments that of the Unity Engine, the IDE Reader is only used in an instance of the Python IDE UI element within a level.

2.4.3. *Interface Description*

Services Provided:

Service Name	Service Provided To	Description
CompilePlayerCode	Iron Python Handler	The CompilePlayerCode service will take in the raw text input of the player, and "compile" it into a data object, which is then provided to the IronPython Handler. This will occur once the player has selected to simulate.

Services Required:

Service Name	Service Provided From
BundlePlayerCode to IDE Reader	UI Handler

2.5. [IronPython Handler]

2.5.1. *Description*

The IronPython Handler is responsible for the interaction between the python code that the player has written and simulation of that code. This component will take input from any instantiation of the IDE Reader, as well as any given lesson file, which will consist of an external python file for the specific lesson. It will then provide the Level State Handler the results of the Python simulation of the player's code.

2.5.2. *Concepts and Algorithms Generated*

Unlike many other components in our system, the IronPython handler consists of only one class. This one class will take inputs from two places, the IDE Reader class which will provide a compiled data object to run the Iron Python simulation on, and a lesson file, which will consist of python environmental objects and methods specific for the lesson. The Iron Python Handler will also provide to the Level State Handler the results of the Iron Python Simulation.

2.5.3. *Interface Description*

Services Provided:

Service Name	Service Provided To	Description
SimulatePlayerCode	Level State Handler	The SimulatePlayerCode service will simulate the compiled data object from the IDE Reader, and provide the results to the Level State Handler. This occurs after the player selects to simulate.

Services Required:

Service Name	Service Provided From
LoadLesson to IronPython Handler	Lesson Files
CompilePlayerCode to IronPython Handler	IDE Reader

2.6. [Level Loader]

2.6.1. *Description*

The Level Loader subsystem is responsible for the loading and creation of save files. An important aspect of this project given the nature of video games is saving and loading. Players should be able to save their progress within the phases of a level and load their previously completed data for levels. The Level Loader is responsible for transposing the internal level data into a JSON file that can be accessed and read at a later time. It is also responsible for the reverse action, whereby given a save file, the Level Loader will read and interpret the file and update the internal level data to reflect the progress in-game.

2.6.2. *Concepts and Algorithms Generated*

Similar to the Iron Python Handler, the Level Loader subsystem will consist of a single class. This class will take an input of the internal class data from any instance of the Level State Handler and will transform the given data to a JSON file format as a Save file. Additionally, the Level Loader will take a JSON file input and translate it into data objects that update the internal level representation with previously saved states.

2.6.3. *Interface Description*

Provide a description of the subsystem interface. Explain the provided services in detail and give the names of the required services.

Services Provided:

Service Name	Service Provided To	Description
SaveFile	Level State Handler	The SaveFile service will take the input of the Level Loader's data objects progression then it will output the data transformed into a JSON file format and save it to a local storage location.
LoadFile	Level State Handler	The LoadFile service will take the input of the JSON file from the Save Files subsystem, then it will transform the JSON data into a compiled data object and provide it to the Level State Handler. Afterwards, the Level State Handler will update its internal data objects based on the compiled JSON object.

Services Required:

Service Name	Service Provided From
GetLevelData to Level Loader	Level State Handler

2.7. [Level State Handler]

2.7.1. *Description*

The Level State Handler is the main gameplay subsystem that is responsible for managing the services from the UI Handler, Iron Python Handler, Level Loader, and Inputs Handler to create a cohesive and fun level. It will manage the other components for displaying the level layout, running the player python code, the simulated results, and saving level progress at key milestones.

2.7.2. *Concepts and Algorithms Generated*

The Level State Handler subsystem will consist of a single class attached to the Level UI Component. Similar to the IDE Reader, the Level State Handler will have multiple instances for each Level that the player can play, such as main, challenge, and boss levels. Additionally, it will provide Level data objects to the Level Loader in order to create or update a player's save file, which can be loaded in later.

2.7.3. *Interface Description*

Services Provided:

Service Name	Service Provided To	Description

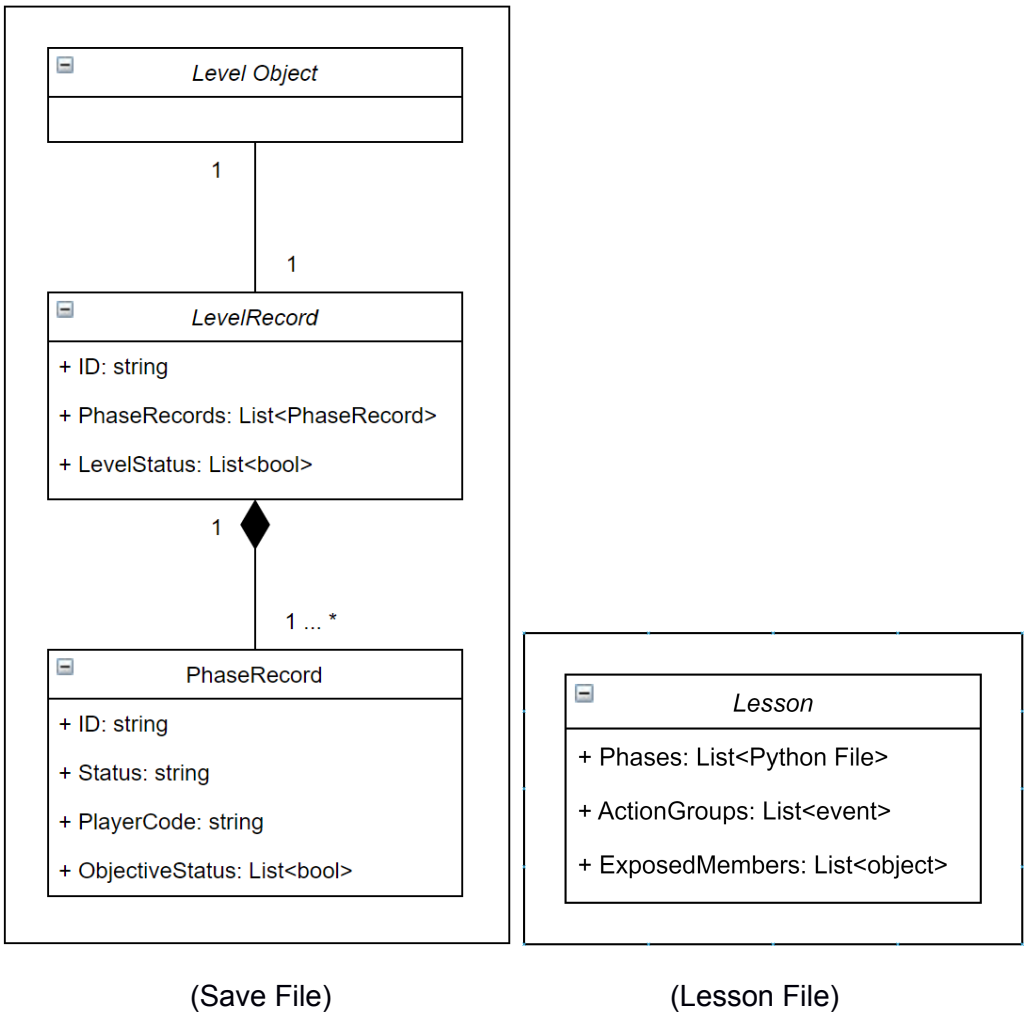
GetLevelData	Level Loader	The GetLevelData service will provide the data objects of the level's progress to the Level Loader. This Level data will then be converted into a JSON file and saved to a local location.

Services Required:

Service Name	**Service Provided From**
UpdateLayout to Level State Handler	UI Handler
GetPlayerInput to Level Loader	Inputs Handler
LoadFile to Level Loader	Level Loader
SimulatePlayerCode to Level Loader	Iron Python Handler

XII. Data Design

For this application, there are two main data structures of concern, both which interact with external subsystems. These two structures pertain to the interaction with Lesson files and Save files. For more information about the internal subsystem interactions see the System Architecture diagram and description above.



(Save File) (Lesson File)

For the Save File data structure, there will be a Level Object that functions as an abstraction for the LevelRecord object. This LevelRecord will consist of three main properties: ID, PhaseRecords and LevelStatus. The ID consists of a static identifier that associates this saved record with a particular level in-game. The LevelStatus object will contain a list of boolean values that correspond with the completion of each phase, this will be used to establish how much progress has been completed and which phases have been or need to be completed. The PhaseRecord object will contain four properties: ID, Status, PlayerCode, ObjectiveStatus. The ID property is very similar to the ID property of the LevelRecord, and will serve to uniquely identify the phase. The Status will be a string or enum that indicates the completion of the phase, this could be incomplete, partially complete, or complete. The PlayerCode will be a string representation of the player's code that was last compiled, this will allow players to continue from the last line of code they wrote at a later time. And lastly, the ObjectiveStatus object will be a list of boolean values, each that correspond to the completion status of the objectives for this phase. Given this structure, the LevelRecord is a composition of the PhaseRecords, where every LevelRecord contains at least one PhaseRecord.

Regarding the Lesson data structure, there are three main elements that it consists of. Firstly, Phases - a List of Python files -, ActionGroups - a List of Events -, and ExposedMembers - a List

of Objects. The Phases property contains python files that dictate what functions and variables the player has access to in the UI Python IDE element and can contain additional python files depending on the amount of phases in a lesson. The ActionGroups holds events that trigger modifications in the Unity UI subsystem. And the ExposedMembers list allows control over class members that are exposed to the user to interact with.

XIII. User Interface Design

The Pluribus Doctrina Team has created a preliminary UI mockup for the game, with a focus on the main layouts for each page that the player will interact with. For the User Interface Design, the team gave heavy consideration for the nature of the application, a video game, and the audience, middle school students. As development continues, the team has decided to follow a minimalistic format with bold colors to draw attention to important items. From the images in the Appendix, the player is first greeted with the start screen (Image 1). This page displays four items, one of which is the title of the game and the other three will be inter-actable buttons. These buttons consist of a start button, which will move the player to the Level Select page with no previous progress, and a continue button, which will prompt the player to choose a save file to load. After choosing one, the player will be moved to the Level Select page with the previously saved progress adde. Finally it will include a quit button, which will terminate the executable, quitting the game. The second page in the player game loop is the Level Select page (Image 2), this page has two main elements: level selection, and level description. The level selection, positioned on the left, will indicate which level is selected and the level description, positioned on the right, will give a brief description of the level, display level progress, provide a begin or continue button depending on the state of progress made in that level, as well as a restart button, that will reset any progress made. Once a level is selected, the player will then be moved to a Level page (Image 3). This layout will have three main elements: Python IDE, Simulation, AI response. The Python IDE will let the player write code within it and will contain a button to initiate code simulation, the Simulation will provide visual feedback to the player regarding the outcome of the python code they wrote, i.e. if it worked or if it did not, and the AI response will provide instruction, tips, and feedback to the player. The final page is the Manual page, which is accessible in either the Level Select page or the Level page. This page contains two main elements, similar to the Level Select page. While the left positioned element is the same as the Level Select page, the right positioned one, will instead contain information about the lesson selected. This would include a written explanation of the python documentation and examples of what was taught in that lesson. The final image in the Appendix, Image 5, is the overlay screen for the game, the two elements that it contains are a menu button that will allow the player to choose to quit the game or to save their progress, and a manual button that will open the manual layout.

Some additional aspects to note, this is a partial UI mockup there are still parts of the UI that are undetermined by both the team and the client, one such example is the background imagery. In the partial UI, there is a background of stars, however, this is temporary pending confirmation of the team and client. Additionally, this partial UI mockup doesn’t denote any animations for layout change, actionable events, or simulations. However, these aspects will both be handled in the team's video demonstration farther in development.

Testing and Acceptance Plans

XIV. Testing and Acceptance Plans Introduction

1. Section Overview

Icarus Protocol is a game being made in collaboration with Lincoln Middle School, designed to teach basic Python programming skills to students with little to no prior programming experience. The game is designed to be fun and engaging, focusing on the key design objective of “putting fun first” to create an experience that increases knowledge retention and student enjoyment by allowing them to develop their skills through an activity that feels more like a game than like a series of lessons. In the game students will play the role of a ship AI, trying to repair the broken software systems of the ship by correcting the ship’s incorrect code. Doing this introduces and teaches a variety of programming concepts and challenges the students to extend their knowledge through optional challenges and boss levels. The students will repair the code by writing real snippets of Python code, which will be executed dynamically at runtime and evaluated in real time to determine if the player’s code successfully solves the problem at hand.

The game is being constructed in the Unity Engine, which imposes some particular challenges for testing and validating the game scripts. We consider automated tests crucial for the core IronPython integration that handles the ability of the game to run user code dynamically at runtime, as well as the save/load code that manages player save data. Most other scripts would be good to test, but mostly stand to interact with UI elements or manipulate objects within the Unity game engine, making them both less necessary, and more difficult, to test.

2. Test Objectives and Schedule

The objective of tests in this kind of project is generally to gain confidence that the code we’ve written will successfully and consistently execute the desired behavior and fulfill all requirements at runtime. Automated tests especially serve the additional purpose of serving to safeguard the project against being broken by future updates. Our automated and manual tests will accomplish both of these purposes, helping to ensure that all core features are operating correctly each time that we deploy a new build of the game. We feel this is especially important to have established as we move into the second half of our development process, which will involve directly interacting with users in a series of beta-tests to receive feedback and rapidly iterate on our build. High quality tests now ensure that our iteration later can be more agile and more reliable.

The project being constructed in Unity leads to us needing to adopt some very specific testing technologies. For unit testing we will have to use the Unity Test Framework (Unity Technologies), a special C# unit testing framework designed to work with and within the Unity Engine. We can also use this framework to develop some isolated level of integration testing, although full integration testing will be utterly impossible because most of our code interacts with the external Unity Engine systems that we have no reason to do integration tests on. Where applicable, some integration tests will be used to ensure that closely related components created entirely by us are in fact integrating properly. We will also need to create and document extensive manual testing procedures. Most of our system testing will have to be done manually, due to the lack of automated tools for testing compiled Unity executables. This will have to

include manual tests run by ourselves and during the acceptance testing process to ensure that the product performs all expected functionalities as intended when interacted with by a user, as well as procedure for performing formalized playtests with middle schoolers, gauging their engagement, the ease of use, and receiving feedback from them on their experience.

Our testing process will deliver 3 major deliverables. The first is a suite of unit tests covering all non-trivial non-UI scripts produced when creating the functionality of the application. The second is a document containing the testing procedure including operational instructions and expected results for all manual functional tests and playtests, and the final deliverable is a github CI pipeline created using the GameCI framework for running our pull-requests through a tested CI pipeline (GameCI).

It should be noted that the GameCI framework is a third party tool not associated with the Unity Engine. We also have yet to confirm the viability of this as an approach. Unity's built-in CI system is unfortunately a paid feature that we aren't prepared to pay monthly for. GameCI seems promising and professionally developed, but we may encounter problems that make the use of this system untenable. If these types of issues (package conflicts, licensing issues, etc.) occur we may have to opt to not use CI in our project as a result. This would obviously not be ideal, and it isn't our intention to go this route, but Unity-enabled CI is still a problem that is actively being solved, and all existing solutions have their own flaws and concerns.

3. Scope

This section discusses our plans for how we intend to test and create test material for Icarus Protocol. It will cover our general testing and deployment process, as well as our testing plans for various forms of essential product testing. While this document does not exhaustively cover all of our test frames and test inputs, examples of some of our tests can be found in the appendices of this document.

XV. Testing Strategy

Project testing will be designed to create full automated tests of all core functionality of the game, running these tests through a CI pipeline for continuous integration. The major components of the core functionality are the IronPython integration, and the Save/Load system. Smaller nonessential or non-core functionality may be tested, but due to time constraints may be allowed to run without automated tests. Because Unity automatically handles many errors during runtime without interrupting the flow of the game, errors in these components will not compromise the overall operation of the finished system. Our specific testing procedure is broken down below into 2 distinct processes. The first process describes how we develop, run and push code tested through automated unit/integration testing, as well as manual FT and acceptance testing. The second process is a preliminary outline of our process for performing unit tests, to be finalized later with the help of Lincoln Middle School as we move into the alpha and beta testing phases of the project.

Developer Testing Process

1. **Developer Writes Code:** We do not intend to use TDD for this project. As such, we begin by creating and attaching the feature implementation to the Unity Project.

Implementation is considered complete once the feature appears to successfully accomplish all required functionality.
2. **Determine Test Cases:** For each core functionality, we will create at least 2 test cases. One tests the ordinary expected operation of the unit, and the other tests exceptional or invalid behavior. Nonessential or small functionality, if tested, is acceptable if it only includes a test case for its ordinary behavior.
3. **Run Tests:** The developer should run the tests on the code, running all tests currently in the project, including the ones that they have recently added.
4. **Fix Issues:** If any of the tests fail in **Step 3**, the developer should identify the source of the failure and repair the bugs, rerunning tests as needed until all tests pass.
5. **Developer Pushes Code To Remote:** The developer will push their code to the remote. Assuming that we are successfully able to establish a Unity-compatible CI pipeline, the push will be run through this pipeline and results will be shown. A branch is only eligible for merge if its most recent commits pass all tests run in the CI.
6. **Developer Makes a Pull Request Against Main:** The developer makes a PR against the main branch, and adds at least one other developer as a review. The branch should not be eligible for merge unless and until the reviewer(s) perform a code review and provide approval.
7. **Merge Branch:** The branch will be merged into main assuming it passes all previous steps, and the CD pipeline will deploy a new release of the game to github, if applicable.

Playtesting Process

1. **Create Playtest Build:** A developer will create a build of the game set up for the playtest. This could be a fresh build of the game, or a modified build designed to omit or skip certain content to allow testers to skip relevant playtest material.
2. **Deliver Build & Allow Time For Testing:** With the build delivered, a time period potentially of several weeks should be allowed for students to test the material.
3. **Deploy Feedback Forms:** As testers complete the content under test, they should be prompted to fill out a form. This form should ask questions designed to directly and indirectly gauge their interest in the game, the amount of fun they had playing it, the amount they felt that they learned, and whether they are retaining knowledge from the game.
4. **Collate Feedback Data:** Feedback should be collated into a feedback record document, and results should be analyzed for useful insights into what can be changed or improved for the next iterations.
5. **Make Any Necessary Changes:** Based on the analyzed feedback, make changes to the content, UI, or player experience to improve the experience for the next iteration of playtesting.

XVI. Test Plans

1. Unit Testing

The team will generally follow traditional unit testing procedures, however where the team's unit testing methodology will diverge in areas that are Unity Engine specific. The team will be using the Unity Test Framework, formerly known as the Unity Test Runner, as the avenue of testing for this project (Unity Technologies). In order to consider the code sufficiently tested, the team will be required to test all core game functionalities. A core game functionality is defined as a game object functionality that would render the game unplayable if non operational, it is non-trivial and a non-UI script. Additionally, the team will evaluate the relevance and impact of all non-essential

units. If the developer feels it necessary, more unit tests may be developed. These units under review would be scripts pertaining to game objects that are highly frequented or provide essential user functionality, but are not considered core game features. For this section, we will defer to individual developer discretion when considering the extent of additional unit tests.

2. Integration Testing

Given the nature of working within the Unity Test Framework, our integration testing will mirror the Unit testing section however with more limitations. Due to requirements of the Unity Engine, our team must diverge from standard integration testing processes, testing isolated code clusters without the ability to fully integrate the application. Additionally, the unique structure of the project may lead to developer discretion in whether or not to attempt to integrate less amenable script structures.

3. System Testing

3.1. Functional Testing:

The team's functional testing plan is primarily reliant on manual testing implemented and tested by the developers. For this section, the team will predominantly focus on the previously outlined Requirements and Specifications document, which contains developer and stakeholder expectations for project functionality. Each functional requirement, outlined in the document mentioned above, will be associated with one functional test. This section of the document is subject to revision if there is an update to or restructuring of the project's functional requirements. Given the nature of working with the Unity Engine, these tests will be manually validated by the developers. In the case that a functional test should fail, the developer who is testing that component will provide a description of the test conditions and request the original developer to reevaluate and solve the error.

3.2. Performance Testing:

Leaning into the strengths and unique qualities of the Unity Engine, our team is opting to use the Unity Profiler tool (Unity Technologies). The tool measures and provides performance information about the application in areas such as the CPU and memory. This is especially important for the Icarus Protocol game given poor performance doesn't reflect well on the gameplay functionality of the application. Additionally, this tool will provide the resources for the developers to monitor the performance of the application under variable loads. In regards to the non-functional requirements specified in the Requirements and Specifications document, our team will manually test the performance of these aspects. Given the non-functional tests rely on qualitative metrics rather than quantitative, developer discretion will be given.

3.3. User Acceptance Testing:

In collaboration with Lincoln Middle School, the team will employ play testing for several iterations before the final release of the game is completed. Our testing strategy for user acceptance testing, will be outlined below with sections for provided resources, instructions for testing groups, and plans for feedback and revision. This outline mirrors the Playtesting Process detailed above, however this description focuses more on the developer view.

A. Resources
    a. A build of the application (for each testing student)
    b. A google form for feedback (for each testing student)
    c. A google form for teacher feedback (for testing facilitator)
B. Instructions
    a. A small select group of students will be chosen for this testing iteration.
        i. Note: a student will only be able to participate in a testing group once.
    b. Each student in the testing group will be provided a working build of the Icarus Protocol game.
    c. A to be determined timeframe will be provided for testing.
    d. After the student finished the game or by the end of the testing time frame, the student will be provided an exit google form for feedback.
    e. After all students have completed the game or the duration of the testing time frame has expired, the facilitator will be provided an exit google form for feedback.
C. Revision
    a. The team will receive and review all feedback forms
    b. The team will then convene and deliberate on modifications to pursue based on the testing group feedback.
        i. If a new feature is pursued or a bug is found, the team will provide documentation for the addition or modification and will continue with updating the project accordingly.

*This outline is intended to be reiterated for testing purposes.

The final iteration of the user acceptance testing will involve positive responses from the students and facilitator, as well as positive skill growth from the beginning of the game to the end and demonstration that all required functionality is functional in the final build.

XVII. Environment Requirements

Our testing environment is predominantly self-contained with Unity Engine proprietary tools. As mentioned above, both unit and integration testing will use the Unity Test Framework and their testing strategies will generally mirror each other.

However, the team will be using an external component, GameCI if possible, in conjunction with the Github CI pipeline. This framework will provide a testing pipeline that will allow developers to guarantee that tests are run and passed before committing and merging work.

There are no specific hardware requirements.

XVIII. Alpha Prototype Description

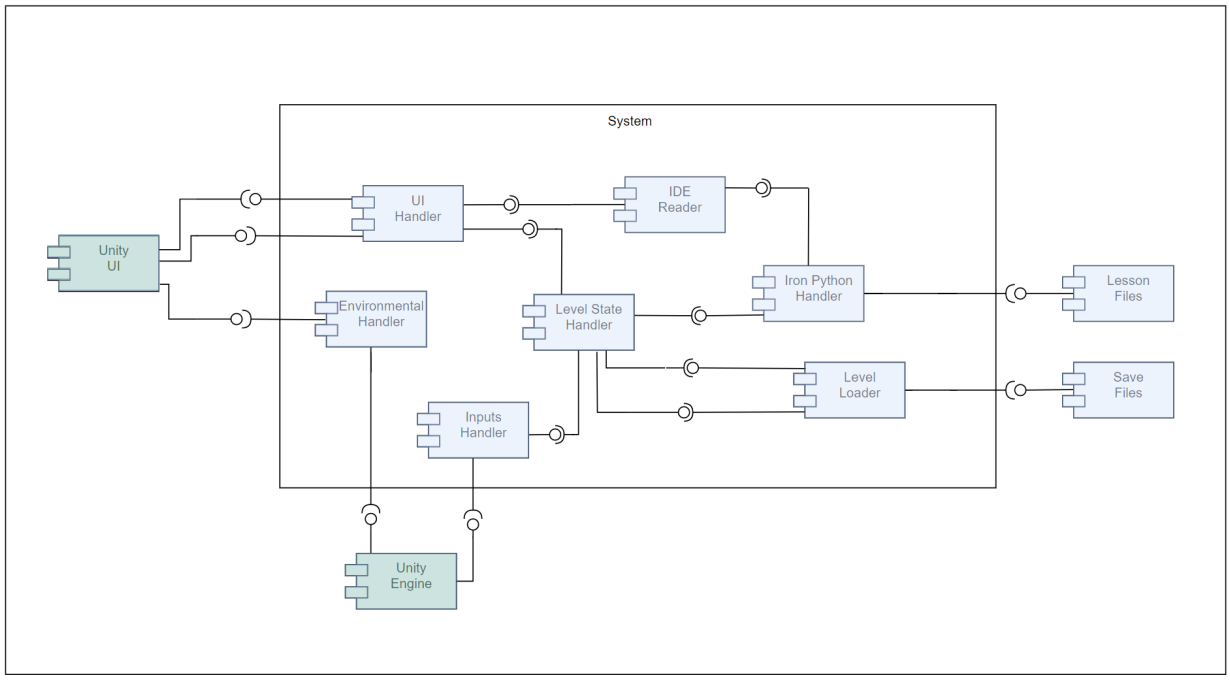
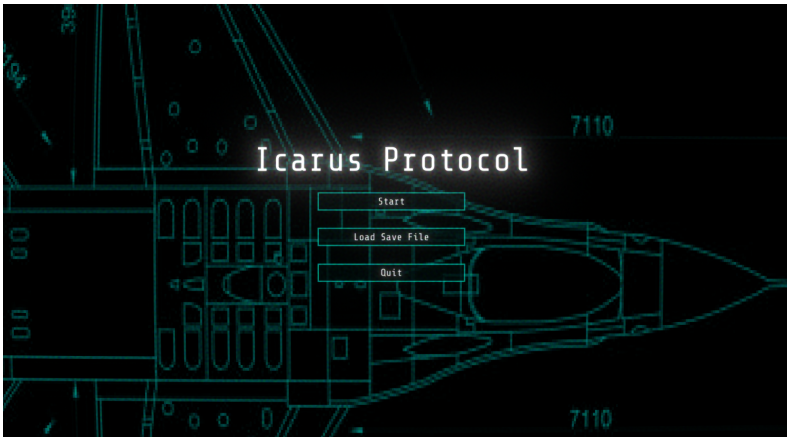


Figure XVIII.I

The above diagram, Figure XVIII.I, illustrates the subsystem decompensation identified earlier in this document. For this updated version, all subsystems identified in blue shading have been implemented by the Pluribus Doctrina team, and any subsystems in green have already been created and are being used by the team. Each subsystem implemented by the team will be addressed to a greater length and detail later in this document. The team and client have agreed to this alpha prototype, where the code foundations are implemented but not graphical and gameplay polished details, such as sound design and animations. All core game functionalities have been implemented as well as external files that correspond to save files and lesson files, additionally preliminary UI layouts and UI component functionalities have been created and tested. These subsystems are subject to modifications and updates as the team refines and polishes the Icarus Protocol game in the future.

1. UI Handler

1.1. Functions and Interfaces Implemented



The UI Handler is one of three subsystems that functions on an abstract level, the others being the Environmental Handler and the Input Handler. These three systems are the interactions between the external Unity systems such as Unity UI and the Unity Engine that allows the Unity platform to interface with the code the team writes. The UI Handler is decentralized on each UI component that dictates the ability for that component to have an attached code file that describes the behavior it should execute.

1.2. Preliminary Tests

Given the decentralized nature of this subsystem, manual testing was performed on each component prior to integration of that component, in order to confirm that the expected behavior was consistent with the actual behavior. As specified in the testing and acceptance plans section of this report, there will be automated tests for this subsystem in the future.

2. Environmental Handler

2.1. Functions and Interfaces Implemented

Similar to the UI Handler the Environmental Handler dictates the interaction between the Unity Engine and the system that the team creates. It predominantly manages minor file updates for the Unity Engine's internal files as it deems there is an environmental change due to new code that has been integrated.

2.2. Preliminary Tests

For the Alpha prototype of the Icarus Protocol game, manual tests were performed in order to confirm that environmental changes made in developer code are reflected as a change in the Unity Engine. Similar to the UI Handler, in the future, there will be automated tests for the Environmental Handler.

3. Input Handler

3.1. Functions and Interfaces Implemented

The Input Handler is the third of the more abstracted subsystems that focus on direct interaction with the Unity Engine or Unity UI systems. The Input Handler manages the interaction between the user, the Unity system, and the developer's code, this could be selecting a UI component or entering text or clicking a specific keyboard button such as 'esc'.

3.2. Preliminary Tests

As outlined in the testing and acceptance section of this report, the team took the Input Handler under manual testing before integration into other sections of the developer's codebase. This subsystem will also have automated tests in the future.

4. IDE Reader

4.1. Functions and Interfaces Implemented

The IDE Reader was implemented in the IDE controller script, which handles two main functionalities: code input and code output. The input and output configurations have variants for fill-in-the-blank style questions. For code inputs, the IDE Reader will be provided starting code in the form of a text file for a level phase, it will then reformat the text into starting code that the player can interact with. In the case that there are fill-in-the-blank characters within the starting code the IDE Reader will format it accordingly. For the outputs, the IDE Reader - on call from the Simulate button - will take the code (starting and player's) and bundle it into a package readable by the IronPython Handler.

4.2. Preliminary Tests

As specified within the testing and acceptance criteria section of this document, we implemented numerous manual tests to confirm that the IDE reader has the intended functionality. Tests similar to the ones illustrated in the Appendix were performed to confirm that code from the phase files was displayed properly (both the normal and the fill-in-the-blank variant), and that once the simulate button was pressed the code in the IDE box was packaged and provided to the Iron Python Handler.

5. IronPython Handler

5.1. Functions and Interfaces Implemented

The IronPython Handler manages the python code simulation and interaction with exposed C# members. It interfaces with the IDE Reader, which provides the code to simulate, and the Level State Handler, which the IronPython Handler provides the result of the code simulation to. The IronPython Handler manages the bi-directional relationship between the python code and the C# code. In order to sandbox the user and still allow them to trigger effects in the game, the team elected to use the IronPython Handler to manage the interactions between the python simulated code and action groups it can trigger. This allows the team to expose specific action groups to the Handler without exposing them to the user.

5.2. Preliminary Tests

Manual tests were performed on the IronPython Handler in order to confirm that the IronPython Handler could interact with specified exposed C# members and action groups, as well as determine the resulting state (success or failure) from the bundled player's code.

6. Level Loader

6.1. Functions and Interfaces Implemented

The Level Loader does two main functions - saving user progress and loading previous user progress- through the SaveAndLoad class. Saving can be accessed in two ways, autosaves are incorporated into the game for when any phase has been completed and the player can call the Level Loader for saving by selecting Save Game in the Pause Menu. Loading the game can only be accessed by selecting the Load Saved Game in the Starting Menu, where the Level Loader will read from the autosaved file that is located on the local machine.

6.2. Preliminary Tests

The level loader was confirmed to perform the correction functionality through a series of manual tests done by the developing team. It was confirmed that a player's progress was autosaved, and that they could manually save a version of their progress. Additionally, it was confirmed that upon loading a previous game all saved data was correctly loaded and displayed in the game.

7. Level State Handler

7.1. Functions and Interfaces Implemented

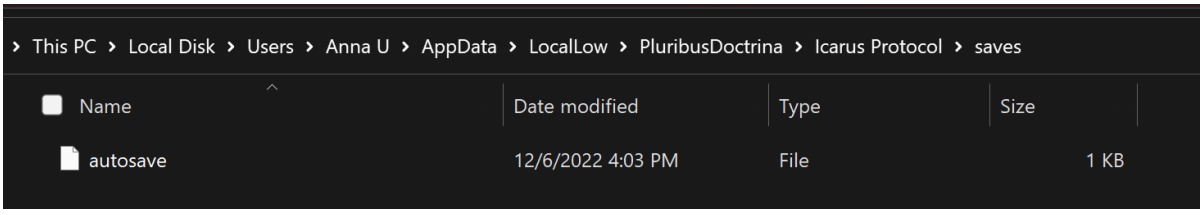
The level state handler is implemented in the Level Player Controller, for which it manages the user facing response and developer facing backend of the results from the IronPython Controller (success and failure). It also handles the transition to the next phase, and marks the previous phase complete (with a new autosave) upon success from the IronPython Handler simulation. As well as in the case that the player completed the last phase in a level it will return the player to the level select page and mark the level as completed.

7.2. Preliminary Tests

Like many of the other preliminary tests for this project, the development team decided that for the first semester there would only be manual tests while in the future, automated tests would be implemented. The Level State Handler was confirmed to function correctly through these manual tests. The team tested that this subsystem could handle either final state (success or failure) and the corresponding transition to another phase or the level select page upon success.

8. Save Files

8.1. Functions and Interfaces Implemented



For the project's purposes, a new format of save file needed to be created to store the player's progress. The team created a unique save file format that the SaveAndLoad class manages for the bi-direction relationship between the save file and the game. For saving, the team locates a local folder on the machine it is on to save to, and it uses JSON.NET to convert the current level

progress into a json object that is stored. For loading, the team uses the same package JSON.NET to reconvert the object and update the game to reflect the loaded game.

8.2. Preliminary Tests

Save Files were manually confirmed to interact with the LevelLoader subsystem.

9. Lesson Files

9.1. Functions and Interfaces Implemented

```
lifeSupportActive = False

def simulate():
	global lifeSupportActive

	try:
		parent.execute_user_code()
	except Exception:
		pass

	if lifeSupportActive is True:
		parent.end_simulation(0)
	else:
		parent.end_simulation(1)

def simulate_tick():
	return 0
```

Similar to the save files, lesson files would need to be made for the purpose of this application. There are three files that relate to a phase and many phases make up a lesson. One of these is the text files that by the team's discretion have certain conventions that indicate in-game formatting, such as the newline character and the '~' character. Another is the starting python code that is provided to the user, a note about this component there is some overlap with controlled exposed C# variables and functions that need to be manually integrated. The final is a unit test file that when given the user's code it runs code and determines if a success or failure state is reached.

9.2. Preliminary Tests

Save Files were manually confirmed to interact with the IronPythonHandler subsystem.

XIX. Alpha Prototype Demonstration

1. Summary of Demonstration

The demonstration includes a walkthrough of all level phases: main, challenge, boss. Additionally, it shows the functionality of all menu options: starting and pause - this includes saving, quitting, loading, and returning to either the starting page or the level select page. Another part of the demonstration will show the manual and all manual pages. The team will also demonstrate how to load a previously saved game.

2. Comments or Suggestions

3. Questions and Responses

XX. Future Work

The future work can be distinguished as two main categories: graphical updates and sound design.

Graphical Updates:

- Code Animation
    - Add animation during the time when the player's code is being run that indicates if they are succeeding or failing
- Button Animation
    - Add transitional animations for the buttons
- Syntax Highlighting
    - Add colors to the syntax of the player's code (which includes both starting code and player code) that correspond to specific keywords like True, False, or known variables

Sounds Design:

- Background music
    - This music will change as need for the themes of the game
        - Lo-Fi background music
        - Sinister background music
        - Epic background music
- Code sounds
    - Adds sounds for the animations of when the players run their code
    - Adds sounds for success and failure states
    - Adds sounds to buttons that can be clicked

XXI. Glossary

Alpha-test: The preliminary testing stage that validates that the Icarus Protocol performs as expected.

Beta-test: The final testing stages that ensure the Icarus Protocol is qualified for final release.

Building-blocks Programming Language: A method of scripting or programming behavior that involves connecting predefined behavior blocks. While it may utilize coding logic, it requires little to no handwritten code on the part of the user.

CI/CD: Continuous Integration/ Continuous Deployment. This provides the team a method to ensure tests are passing as software is updated by developers.

Feedback Loops: A term in game design which refers to using the player's success or failure to adapt and adjust the output of other systems. This could mean adjusting up the difficulty of challenges if the player appears to be succeeding too easily, or providing hints only when the player shows signs of struggling.

Functional requirements: The requirements for the Icarus Protocol that can be tied to a software functionality implemented by the team.
Gamified: Incorporating certain aspects of game design, such as lives, points, levels, or skill trees in an attempt to make ordinarily non-entertainment activities more engaging or rewarding to perform.

HUD: Heads Up Display. A user interface element that displays information visually to a player.

IDE: Integrated Development Environment: A software or part of a software which can be used to develop other applications. In our case the IDE is the component of our game that allows you to develop functional Python code.

Level: A self-contained stage of the game, consisting of one or more phases, and teaching a complete and distinct concept. Analogous to a lesson for the purposes of education.

Non-functional requirements: The requirements for the Icarus Protocol that are general, often qualitative, and not related to a specific functionality.

Non-trivial: This refers to methods that have core game functionality that the game requires for completion.

Object Scripting Model: A niche software architecture pattern defined by the attachment of atomic scripts to objects primarily managed by another system (i.e. the Unity Engine).

Player: Player is a term referring to our primary application user.

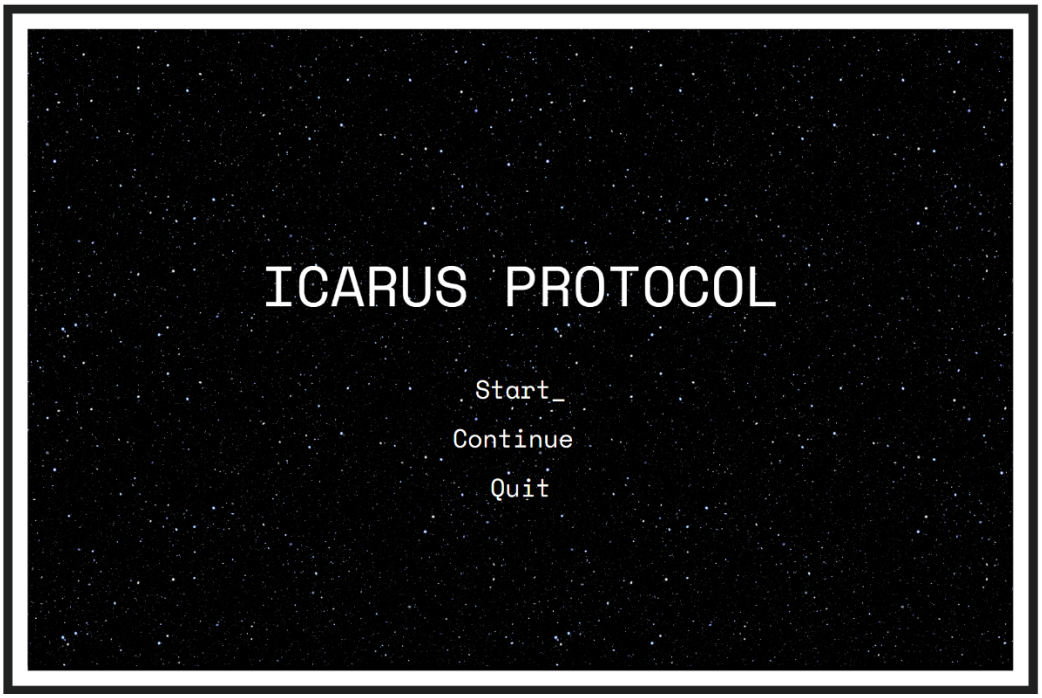
Reward Models: A game's reward model is the structure and array of visual, auditory, and mechanical rewards that are given to the player to incentivize success, as well as the specific situations in which rewards are given to incentivize the desired behavior.

Scripting Language: A programming language, usually an interpreted language, which can be integrated in with a compiled executable to allow for the adjustment or extension of existing features or content without recompilation of the primary executable or exposure of its source code.

TDD: Test Driven Development. A type of software development characterized by writing tests first.

UI: User Interface. The space where the user and the game system interact and communicate.

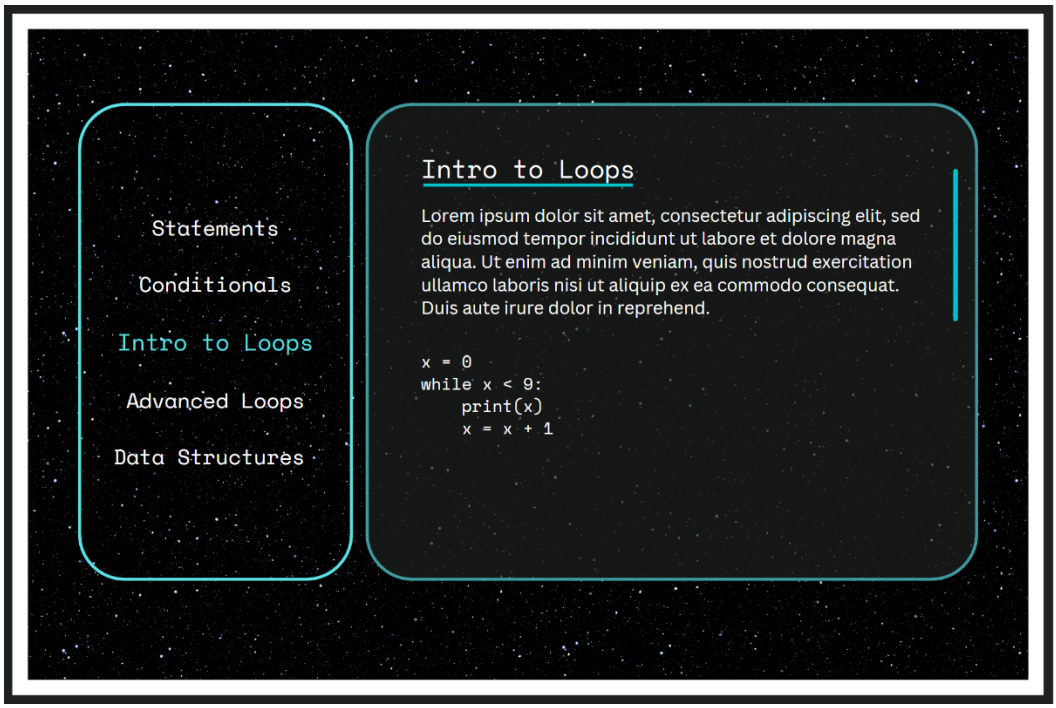
XXII. Appendix



(Image 1)

(Image 2)

(Image 3)



(Image 4)

(Image 5)

Test Cases:

Main Levels

Description:

This is a series of manual test cases for selecting a main level.

Assumptions:

- The game has been started with a new game (no previous save file data)

Test ID	Expected Inputs	Expected Outputs	Actual Outputs	Pass / Fail
01	Click on Life Support	The Life Support description box opens to the right with no phases highlighted as completed.	The Life Support description box opens to the right with no phases highlighted as completed.	P
02	Click on Navigation	The Navigation description box opens to the right with no phases highlighted as completed.	The Navigation description box opens to the right with no phases highlighted as completed.	P
03	Click on Reactor Core	The Reactor Core description box opens to the right with no phases	The Reactor Core description box opens to the right with no phases	P

		highlighted as completed.	highlighted as completed.	
04	Click on Artillery	The Artillery description box opens to the right with no phases highlighted as completed.	The Artillery description box opens to the right with no phases highlighted as completed.	P
05	Click on Shield Generator	The Shield Generator description box opens to the right with no phases highlighted as completed.	The Shield Generator description box opens to the right with no phases highlighted as completed.	P
06	Click on the description of the level	Nothing happens.	Nothing happens.	P
07	Click on background image of the ship	Nothing happens.	Nothing happens.	P
08	Click on the level name in the level select list that was previously clicked	Nothing happens.	Nothing happens.	P
09	Press the ‘s’ keyboard button	The level button that is down the list from the previously clicked level button is now highlighted.	The level button that is down the list from the previously clicked level button is now highlighted.	P
10	Press the ‘w’ keyboard button	The level button that is up the list from the previously clicked level button is now highlighted.	The level button that is up the list from the previously clicked level button is now highlighted.	P
11	Click the Start button in the level description	The corresponding level’s first phase UI layout is transitioned to.	The corresponding level’s first phase UI layout is transitioned to.	P

Manual Page

Description:

This is a series of manual test cases for selecting a manual entry.

Assumptions:

- The game has been started with a new game (no previous save file data)

Test ID	Expected Inputs	Expected Outputs	Actual Outputs	Pass / Fail
01	Click on manual	Manual layout page is	Manual layout page is	P

	overlay button	transitioned to, with nine unique manual entries.	transitioned to, with nine unique manual entries.	
	Click on a manual entry	The corresponding manual entry is displayed with the entry name and description on the right side of the screen.	The corresponding manual entry is displayed with the entry name and description on the right side of the screen.	P
	Click on manual overlay button	Previous layout page (level or level select) is transitioned to	Previous layout page (level or level select) is transitioned to	P
Assumption: Test Level is now complete				
	Click on manual overlay button	Manual layout page is transitioned to, with ten unique manual entries.	Manual layout page is transitioned to, with ten unique manual entries.	P

XXIII. References

"Coding games to learn python and JavaScript," *CodeCombat*, 2022. [Online]. Available: https://codecombat.com/home. [Accessed: 20-Sep-2022].

"Codecademy Homepage," *Codecademy*, 2011. [Online]. Available: https://www.codecademy.com/. [Accessed: 20-Sep-2022].

Computer Science Teachers Association (CSTA), "Computer Science 6-8 Standards," *Ospi*, 2018. [Online]. Available: https://www.k12.wa.us/student-success/resources-subject-area/computer-science/computer-science-k-12-learning-standards. [Accessed: 20-Sep-2022].

D. Esposito, "An introduction to scripting technologies and Object Models," *An Introduction to Scripting Technologies and Object Models*, 20-Sep-2018. [Online]. Available: https://www.itprotoday.com/devops-and-software-development/introduction-scripting-technologies-and-object-models. [Accessed: 05-Oct-2022].

"Gaming market size, share: 2022 - 27: Industry growth," *Gaming Market Size, Share | 2022 - 27 | Industry Growth*, 2021. [Online]. Available: https://www.mordorintelligence.com/industry-reports/global-gaming-market. [Accessed: 20-Sep-2022].

"IronPython," *IronPython.net /*. [Online]. Available: https://ironpython.net/. [Accessed: 27-Sep-2022].

K. Squire and H. Jenkins, "Harnessing the power of games in education," *InSight*, vol. 3, 2003.

n/a, “GameCI,” *GameCI · The fastest and easiest way to automatically test and build your game projects*. [Online]. Available: https://game.ci/. [Accessed: 28-Oct-2022].

O. S. Card, *Ender's Game*. New York: Tor, 2021.

R. Torres Bonet, “A Better Architecture for Unity projects,” *A better architecture for Unity projects*, 03-Jul-2018. [Online]. Available: https://www.gamedeveloper.com/disciplines/a-better-architecture-for-unity-projects. [Accessed: 05-Oct-2022].

“Scratch Homepage” *Scratch*, 2007. [Online]. Available: https://scratch.mit.edu/. [Accessed: 20-Sep-2022].

S. Danks, B. Fraumeni, and M. Smrekar, “CodeCombat Implementation Study,” *McREL International*, Feb. 2019.

U. Technologies, “Profiler Overview,” *Profiler overview*. [Online]. Available: https://docs.unity3d.com/Manual/Profiler.html. [Accessed: 28-Oct-2022].

U. Technologies, “Unit testing,” *Unity*. [Online]. Available: https://docs.unity3d.com/Manual/testing-editortestsrunner.html. [Accessed: 28-Oct-2022].