Casters & Coders

DESIGN DOCUMENT

Team Number: sddec23-13
Client & Advisor: Mat Wymore
Team Members/Roles:

- Brennan Seymour
- Branden Butler
- Edward Dao
- Wenqin Wu
- Theng Wei Lwe
- Max Bromet

Team Email: sddec23-13@iastate.edu
Team Website: https://sddec23-13.sd.ece.iastate.edu/#

Revised:8/12/23 Version 2

Executive Summary

Development Standards & Practices Used

- The IGDA Crediting Standards Guide. This standard will help us to officially credit all contributors effectively.
- The IGDA Crunch, Unsustainable Work, and Management Abuse Standard. This standard lays out how to avoid unsustainable or excessive working practices, to better respect group members.
- Much of our project will use C#, so we will commit to Microsoft's .NET C# Coding Conventions. This will help us maintain code-style consistency while we collaborate.

Summary of Requirements

- Functional:
    - The Player will write scripts and save them.
    - The Player will browse their saved scripts and choose to edit them.
    - Scripts will interact with puzzles through a predetermined api.
    - The Player will advance through the game by solving puzzles.
    - Some of the puzzles will be optional. These will typically be more difficult than normal puzzles.
    - The game will be playable on OSX, Windows, and Linux-based operating systems.
- Qualitative
    - The game should be fun to play.
- Resource:
    - The game should be pretty lightweight to run.
    - It should achieve 60 frames per second when running on the minimum specs.
    - Min Specs:
        - AMD A10 5800k
        - 4G RAM
        - Radeon HD 7660D iGPU
        - 10 Gigabytes storage
- Economic:
    - Casters & Coders will always be free and open source.
    - We will use free and self-made assets only.
- UI:
    - The game should be highly accessible.

    - We will choose colors which are friendly to color-blindness.
    - It will be possible to play the game with only a keyboard, although mouse controls will also be available.

Applicable Courses from Iowa State University Curriculum

- SE 309: Software Development Practices
- SE 319: Construction of User Interfaces
- SE 329: Software Project Management
- SE 339: Software Architecture and Design

New Skills/Knowledge acquired that was not taught in courses

List all new skills/knowledge that your team acquired which was not part of your Iowa State curriculum in order to complete this project.

- Knowledge of the Godot game engine
- Combining multiple programming languages in one monolithic executable
- Video game design

Table of Contents

Development Standards & Practices Used 2
Summary of Requirements 2
Applicable Courses from Iowa State University Curriculum 3
New Skills/Knowledge acquired that was not taught in courses 3
Table of Contents 4
1 Team 6
1.1 Team Members 6
1.2 Required Skill Sets for The Project 6
1.3 Skill Sets covered by the Team 6
1.4 Project Management Style Adopted by the team 7
1.5 Initial Project Management Roles 7
2 Introduction 7
2.1 Problem Statement 7
2.2 Requirements & Constraints 7
2.3 Engineering Standards 8
2.4 Intended Users and Uses 9
3 Project Plan 9
3.1 Project Management/Tracking Procedures 9
3.2 Task Decomposition 10
1. Engine and Scripting System 10
2. Environment Design and Asset Creation 10
3. Puzzle design 10
4. Mechanics of the Game and Level Implementation 10
5. User Interfaces 11
4 Design Context 11
4.1 Prior Work/Solutions 11
BitBurner <https://danielyxie.github.io/bitburner/> 11
CodeCombat <https://codecombat.com/play/> 11
4.2 Design Visual and Description 12
4.3 Functionality 13
4.4 Technology Considerations 14
4.4.1 Game Engine 14
4.4.2 Scripting language 14
4.5 Design Plan 15
5 Testing 15
6 Implementation Details 16
6.1 Game Logic 16
6.2 Scripting System 16
6.3 User Interface 17

7 Appendix **17**
7.1 operation manual 17

1 Team

1.1 Team Members

- Branden Butler
- Brennan Seymour
- Edward Dao
- Wenqin Wu
- Theng Wei Lwe
- Max Bromet

1.2 Required Skill Sets for The Project

- Software development/ object-oriented programming knowledge
- Experience in multi-language programming
- Experience in or ability to learn video game design
- Experience in UI design

- Puzzle-creation skills
- Artistry skills

1.3 Skill Sets covered by the Team

- Software development/ object-oriented programming knowledge
    - Branden Butler
    - Brennan Seymour
    - Max Bromet
    - Wenqin Wu
    - Theng Wei Lwe
    - Edward Dao
- Experience in multi-language programming
    - Branden Butler
    - Theng Wei Lwe
- Experience in video game design
    - Branden Butler
    - Brennan Seymour
    - Max Bromet
    - Wenqin Wu
- Experience in UI design
    - Brennan Seymour
    - Theng Wei Lwe
- Puzzle-creation skills
    - Max Bromet
    - Edward Dao
    - Brennan Seymour
- Artistry skills
    - Edward Dao
    - Brennan Seymour

1.4 Project Management Style Adopted by the team

We will be aligning with the Agile project management style.

1.5 Initial Project Management Roles

Our group is split into several teams, listed below with their team leaders and members. There is some overlap of members between teams as the workload is not evenly split among the four teams.

- Language Embedding
    - Team Lead: Branden Butler
    - Other Members: Brennan Seymour
- Puzzle Design and Concept Art

    - Team Lead: Max Bromet
    - Other Members: Edward Dao
- Storyline
    - Team Lead: Wenqin Wu
    - Other Members: Theng Wei Lwe
- Game Environment and Logic
    - Team Lead: Theng Wei Lwe
    - Other Members: Edward Dao
- User Interfaces
    - Team Lead: Brennan Seymour
    - Other Members: Theng Wei Lwe, Max Bromet

2 Introduction

2.1 Problem Statement

Our project aims to solve the problem of making programming more accessible and engaging for students. Traditional methods of teaching programming can be dry and difficult to understand, which can discourage students from pursuing this valuable skill. By gamifying the process and creating a fun and interactive way to learn programming, our project can help bridge this gap and make programming more approachable and enjoyable for students. This can help increase interest in STEM fields and prepare students for future careers in technology.

2.2 Requirements & Constraints

The requirements of our projects are listed below as follows:

- Functional:
    - The Player will write scripts and save them.
    - The Player will browse their saved scripts and edit them.
    - Scripts will interact with puzzles through a predetermined API.
    - The Player will advance through the game by solving puzzles.
    - Some of the puzzles will be optional. These will typically be more difficult than normal puzzles.
    - The Player's progress will be saved automatically between play sessions.
    - The game will be playable on OSX, Windows, and Linux-based operating systems.
- Qualitative
    - The game should be fun to play.
- Resource:
    - The game should be pretty lightweight to run.

    - It should achieve 60 frames per second when running on the minimum specs.
    - Min Specs:
        - AMD A10 5800k
        - 4G RAM
        - Radeon HD 7660D iGPU
        - 10 Gigabytes storage
- Economic:
    - Casters & Coders will always be free and open source.
    - We will use free and self-made assets only.
- UI:
    - The game should be highly accessible.
    - We will choose colors that are friendly to color blindness.
    - It will be possible to play the game with only a keyboard, although mouse controls will also be available.
- Stretch Goals (If possible within our time frame):
    - The Player will be able to import previously written scripts and call functions from them.
    - There will be multiple save files.
    - The game will be playable in a web browser.
    - The game will support multiple scripting languages.

2.3 Engineering Standards

The Engineering standards that are like to apply to our project are as follows:

- The IGDA Crediting Standards Guide. This standard will help us to officially credit all contributors effectively.
- The IGDA Crunch, Unsustainable Work, and Management Abuse Standard. This standard lays out how to avoid unsustainable or excessive working practices, to better respect group members.
- Much of our project will use C#, so we will commit to Microsoft's .NET C# Coding Conventions. This will help us maintain code-style consistency while we collaborate.

2.4 Intended Users and Uses

Students: Students who are interested in learning programming can benefit from our project as it offers a fun and engaging way to learn the concepts. By making programming accessible and enjoyable, more students may be encouraged to pursue careers in technology and related fields.

Teachers: Teachers can use our project as a tool to supplement their traditional teaching methods. By incorporating a game-based approach to learning programming, teachers can help their students better understand the concepts and engage them in the learning process.

Educational institutions: Educational institutions such as schools and universities can benefit from our project as it offers a new and innovative way to teach programming. By providing access to our game, educational institutions can enhance their curriculum and better prepare their students for future careers in technology.

Tech Industry: The tech industry can benefit from your project by having a more diverse pool of talent with programming skills. By making programming more approachable and enjoyable, your project can help increase interest in the field and ultimately contribute to the growth and innovation of the industry.

Non-Technical Office Workers: Someone who works in an office may be able to use scripting languages to automate tasks or leverage existing tools to assist them in the workplace. If they're not already familiar with scripting, our game may provide a more approachable way to learn it.

3 Project Plan

3.1 Project Management/Tracking Procedures

The has adhered to the agile project management style. Since the project is a video game, we will be undergoing a lot of testing and debugging. There can also be changes to the requirements such as adding a new feature late into development. With a team size of 6, this has helped us to engage in highly parallel development, since teammates can rapidly respond to changes made by another.

The team has resorted to using GitHub and Discord to help keep track of progress done throughout each member. Discord is the main tool for communication whereas GitHub is primarily used for version control. We also use markdown files in Github for the bulk of our documentation.

GitHub has been an important part of the agile methodology as well. The team has been adopting 2-week sprints throughout the semester. GitHub was the main tool used to create the user stories and the story board. Members were assigned story cards with priority levels, and code branches were created for each user story card. At the end of the bi-weekly sprint, the team undergoes a sprint retrospective to go over the latest sprint and talk about what went well and what could be improved. This process was repeated until the final presentation deadline.

3.2 Task Decomposition

1. *Engine and Scripting System*
    1.1. Prototype language embedding systems using a range of tools.
    1.2. Select an engine, tool, and prototype to flesh out.
    1.3. Design the API for scripting interaction.
    1.4. Implement the scripting system to minimum viable product, enough for puzzle development to begin with a stable API.
    1.5. Flesh out the scripting system further, improving stability and adding features.
2. *Environment Design and Asset Creation*
    2.1. Create rooms/map for the game
    2.2. Write dialogue for character interactions
    2.3. Create or source assets for the game
3. *Puzzle design*
    3.1. Design a framework for describing puzzles and fitting them into a logical order of increasing complexity and introducing concepts.
    3.2. Decide on a "curriculum" for the order of concepts to be taught.
    3.3. Design "Introductory" puzzles for each concept, teaching and introducing.
    3.4. Design "Exploratory" puzzles for each concept, expanding and fleshing out.
    3.5. Refine designs and gather feedback.
4. *Mechanics of the Game and Level Implementation*
    4.1. Create a player controller with basic movement.
    4.2. Design a tileset-based environment which can be repurposed for any puzzle.
    4.3. Implement interaction mechanics. (interacting with game objects)
    4.4. Implement a designed puzzle, creating reusable systems and integrating the scripting system. This is dependent on task 1.5.
    4.5. Continue implementing puzzles.
    4.6. Apply assets, replacing "programmer art"
5. *User Interfaces*
    5.1. Develop a style guide to help reinforce consistent UI and to accommodate keyboard-based navigation.
    5.2. Create the in-game IDE as a minimum viable product, so that it can be used for puzzle development as early as possible.
    5.3. Develop a basic system for talking to NPCs and inspecting objects in the world. (ideally driven by text files)
    5.4. Develop a main menu, settings menu, and pause menu.
    5.5. Flesh out the in-game IDE, including some form of highlighting & completion.
    5.6. Build an in-game text log

4 Design Context

4.1 Prior Work/Solutions

The following prior work were used as light referencing for the general idea of the game we have implemented.

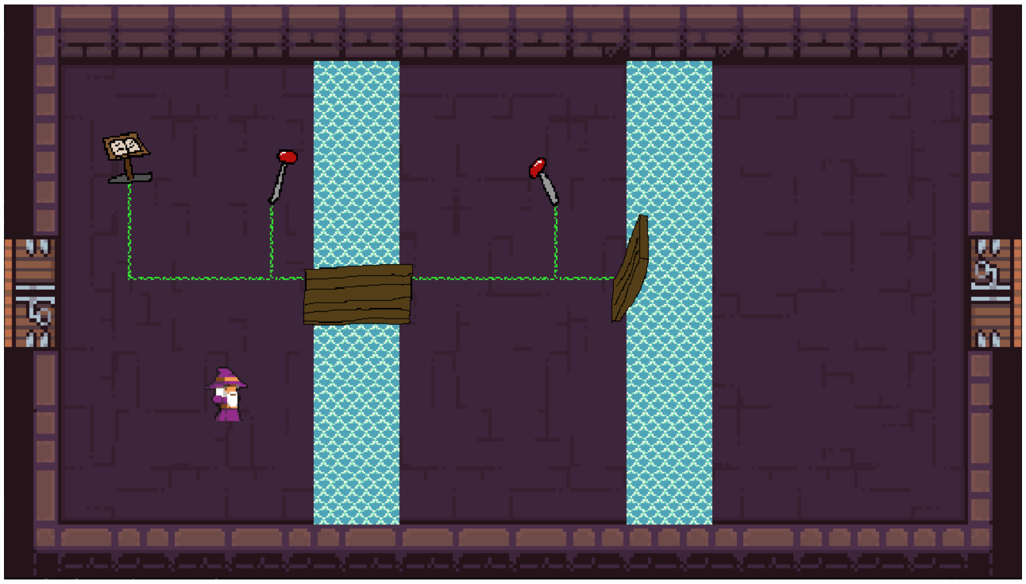
BitBurner <https://danielyxie.github.io/bitburner/>

A cyber hacking-themed video game primarily based around a text interface and 90's-era "hacker" aesthetics. The game involves the player using Javascript to "hack" in-game businesses and accrue large sums of money. The primary shortcoming of this game is that it does a poor job of teaching programming to those unfamiliar with it - it has two versions of its scripting API with few explanations of the differences between them, and each requires a significant amount of "magic" commands and boilerplate.. The text-based method of interacting with most of the game can also be off-putting for most gamers, who are generally used to point-and-click style GUIs. Our solution plans to slowly introduce every core concept of programming without any boilerplate or "magic." It will also include a more traditional player character that moves around the screen, with a separate text entry panel for the scripting component.

CodeCombat <https://codecombat.com/play/>

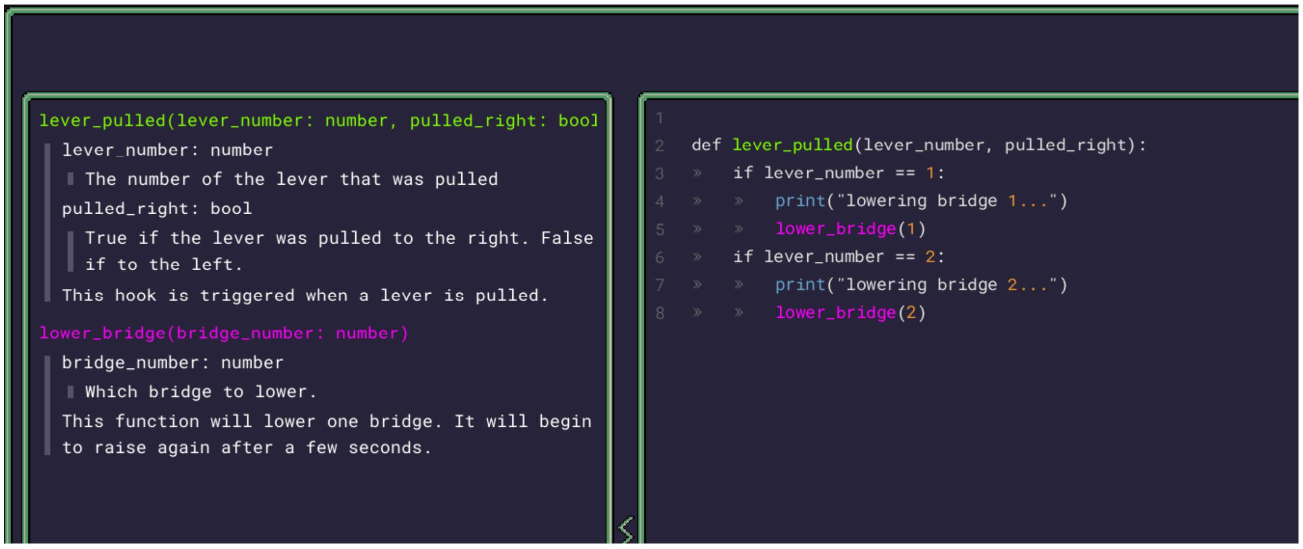
A fantasy puzzle game with an early 2000's "flash game" aesthetic. Players write scripts to control a character who must walk around a dungeon, avoid obstacles, and defeat enemies. This game achieves all of our goals at a surface level, by integrating gameplay puzzles with coding concepts, but we feel that it fails to be engaging on any deeper level. The game's levels are highly separated, giving the player little sense of investment, there is little to no plot, and the game is very resource-intensive. Additionally, the game costs money to play, making it less accessible to players.

4.2 Design Visual and Description



2D Dungeon Environment

The current design approach will be a 2D, top-down, sprite video game that users can control using their keyboards. The approach of the game design is having a sprite character that the users will move around a map full of rooms that are adjacent to one another. Users are able to interact with the environment to complete puzzles and earn progress throughout the game.



```
lever_pulled(lever_number: number, pulled_right: bool
    lever_number: number
        The number of the lever that was pulled
    pulled_right: bool
        True if the lever was pulled to the right. False
        if to the left.
    This hook is triggered when a lever is pulled.
lower_bridge(bridge_number: number)
    bridge_number: number
        Which bridge to lower.
    This function will lower one bridge. It will begin
    to raise again after a few seconds.
```

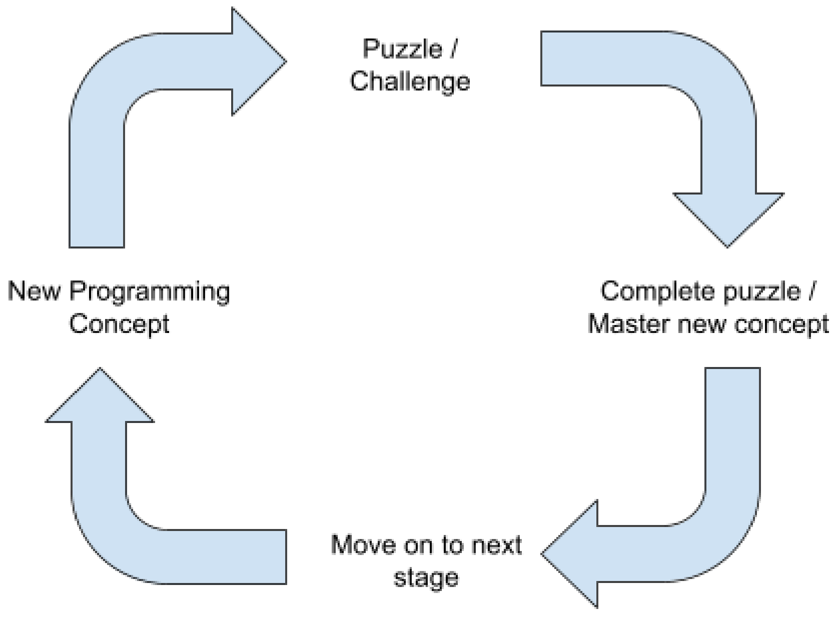
```

def lever_pulled(lever_number, pulled_right):
    if lever_number == 1:
        print("lowering bridge 1...")
        lower_bridge(1)
    if lever_number == 2:
        print("lowering bridge 2...")
        lower_bridge(2)
```

Script Editor

The player will enter a script-editing menu to write a script for any given task, which will display all the available API elements which the script can interact with. When the script is run, this window will close so that the player can watch its effects play out in the game world.

4.3 Functionality

We intend to have the player load up the game on their computer, and then enter into the gameplay loop. The gameplay loop will consist of the player exploring, finding puzzles, solving said puzzles, and then continuing on with their exploration. The puzzles will be coding puzzles, and the player will have to learn new programming concepts to solve said puzzles. By exploring, the player can find information that teaches them about key programming concepts.



Basic Gameplay Loop (old design)

Last semester, our initial design plan requires users to completely master a concept before moving on to more complicated concepts. These allow the users to build upon the strong basic fundamentals and have more challenging/advanced puzzles as they progress. This also prevents users from getting stuck on a particular phase with a lack of basic understanding. This allows the project to meet its functional requirements.

However, we have managed to implement a totally different playstyle. We decided to leave basic boilerplate coding introduction to popular online resources such as W3Schools. The game itself will contain basic method introductions in the puzzles as well as very descriptive comments in the code to help guide players through the stage. The current design also allows for a very robust and huge potential in further development of the game in the future.

Players are allowed to input code that can interact unlimitedly with the system and game itself. This opens doors to opportunities such as a coding playground and sandboxing. For example, a player could even program their very own snake game in the game itself by utilizing pixel generation, simple game logic, and a timer library imported in Python.

The design also allows the game to continue being fun and stimulating, as desired in the non-functional requirements. Instead for completely new beginners, players with a few weeks of learning in introductory courses can be introduced to this game to help them play around more with interactive codes that are visually appealing and stimulating, as opposed to the traditional text-based style programs.

4.4 Technology Considerations

The primary technologies that we needed to consider were our choice of user-facing scripting language and our choice of game engine.

4.4.1 Game Engine

Our team has utilized the **Godot** game engine. The following explains several benefits and reasoning behind this choice of game engine.

- Still well known, also has a lot of community support.
- Less support for high-budget effects.
- Better support for 2D games.
- Has a variety of language bindings, dramatically expanding our possibilities for libraries.
- The Node-based component system is more flexible than Unity's GameObject-based component system.

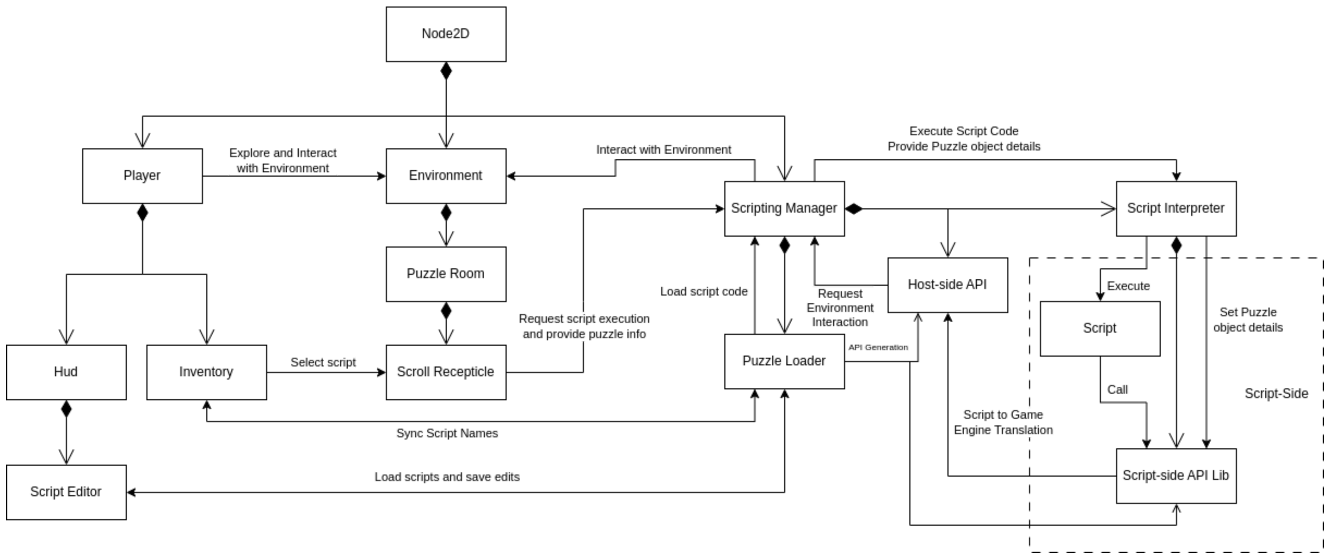
4.4.2 Scripting language

We need to select an appropriate scripting language to teach. We've considered a number of possibilities depending on the approachability of each language and what tools are available for embedding. Ultimately, we have decided to go with the Python scripting language because it:

- Is very simple, easy-to-understand syntax for basic operations.
- Has good support for object-oriented concepts and patterns.
- Has good support for functional paradigms: lambda functions, higher-order functions, currying.
- Has interpreted language, so we don't need to worry about bundling a compiler.
- Has dynamic typing makes types often nebulous and introduces many potential pitfalls for players.
- Godot has a Python binding as well

4.5 Design Plan

Godot represents games as a collection of nodes organized in a graph. As part of our design we created a node graph that fulfills our requirements.



Godot node graph

The node graph can be broken down into three primary sections: the player, the environment, and the script execution system, flowing left to right in the above graph. Each node communicates with others through signals, represented as simple arrows, allowing a loose coupling between nodes. Each node may have children, represented in this graph as composition arrows.

5 Testing

We have initially proposed to utilize the built-in Godot Unit Testing (GUT) library and framework that comes with the game engine. However, due to the time constraints and the size of the end game result, we decided to test solely based on running the game. Collision testing were all performed during the creation of assets, tilemaps, and objects. These collisions are known to work as testing can be done instantaneously when launching the room/scene.

For the Python Script API manager, we performed testing through “Hello World” implementation by feeding the manager some predefined code such as print(“Hello World”), and logged the output to the console. By having the manager run these code and some predefined python files in the local directory, we successfully test that the manager is running and working as expected.

Lastly, for the user interface code implementation, we only need to test that the user-written scripts are updating the actual file location in the local directory where the scripts are being read from. When the user opens the file in the game, they edit the code and exit. We can check that the local JSON file is updated and that the user can re-open the code in the game to see the saved changes.

6 Implementation Details

We have seperated the work into 3 different sub-branches and each of the branch did a very good job towards the final delivery.

6.1 Game Logic

The game development project has made significant strides, laying a strong foundation for an immersive gaming experience. The core elements, including the meticulously designed tilemap, characters, and objects, form the backbone of the game's visual and interactive world. Crucially, the implementation of character movements, dungeon scenes, room traversal logic, and dialog boxes adds depth and engagement to the gameplay. The meticulous attention to detail in setting up object interactions, collision masks, and layers ensures a seamless and realistic gaming environment. Moreover, establishing a framework for integration with the UI and Python Scripting System teams paves the way for efficient collaboration and future enhancements, promising a well-rounded and dynamic gaming experience.

6.2 Scripting System

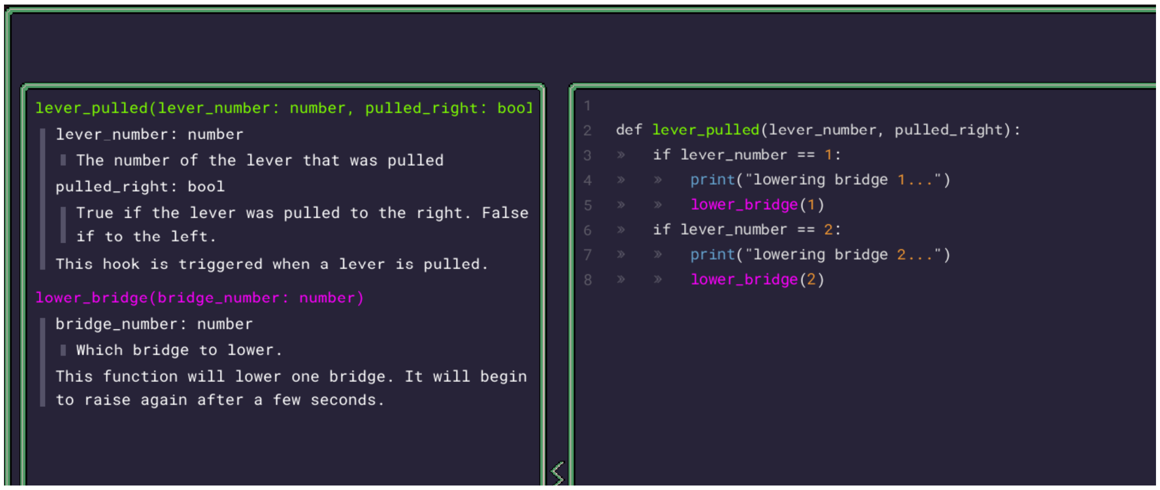
The scripting system has been fully implemented. We used the Godot Python plugin to enable us to write node scripts in Python. Thus, interfacing between the user scripts and the puzzle environment was incredibly simple. We split the system into three primary components: the puzzle loader, the python manager, and the user script. The puzzle loader is written in GDScript, and handles loading the puzzle definition JSON files from the filesystem. It also handles saving and loading the user’s scripts from the filesystem.

The python manager handles a couple of core functionalities: it generates the context for the user scripts, including the functions and expected user callbacks; it handles signal management; and it manages the map of running puzzles. The context is generated when loading the puzzle, it takes the loaded definition string and parses it into a dictionary. The parsed dictionary contains the names of input and output functions and variables, which the manager uses to dynamically generate a context dictionary. This dictionary is then used as the globals and locals for the user script.

The user script is executed inside an exec() call, with try/except wrappers around both the bytecode compilation and the actual execution to prevent user error from crashing the game. The only objects accessible to the script are those within the generated context, so users don’t accidentally interface with non-exposed game logic.

6.3 User Interface

The user interface team has managed to develop a fairly simple code editor interface for players to utilize. Recalling the image provided earlier in the report,



```
lever_pulled(lever_number: number, pulled_right: bool
  lever_number: number
    The number of the lever that was pulled
  pulled_right: bool
    True if the lever was pulled to the right. False
    if to the left.
  This hook is triggered when a lever is pulled.
lower_bridge(bridge_number: number)
  bridge_number: number
    Which bridge to lower.
  This function will lower one bridge. It will begin
  to raise again after a few seconds.
```

```python

def lever_pulled(lever_number, pulled_right):
    if lever_number == 1:
        print("lowering bridge 1...")
        lower_bridge(1)
    if lever_number == 2:
        print("lowering bridge 2...")
        lower_bridge(2)
```

Players are able to write code with basic syntax highlighting and indentation that is provided by the game engine, Godot itself. The game engine proves really convenient as tools are provided for responsive UI layouts, rich text entry, and rule-based dynamic text formatting and highlighting. The Godot text editor was actually built using the same tools exposed to the engine.

The window on the left contains read-only documentation that players can utilize to understand more about the stage itself. This is populated dynamically based on JSON "puzzle definitions" which give unique information about each puzzle. The window on the right contains an editable text field where players can write Python code, which will interact with the game system when closing the code editor window. The updates all happen instantaneously and provide a seamless experience for the players.

7 Appendix

7.1 OPERATION MANUAL

The game is fairly easy to set up and run. First, it requires that the user has Godot installed on the computer. This project runs on the Long-Term Support (LTS) version which is 3.5.3.

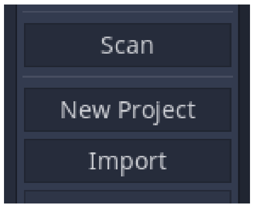
The Godot installation link: https://godotengine.org/download/3.x/windows/

The link provided defaults the installation on a Windows OS device. However, scrolling down, you can find options for installation on Linux, MacOS, and even Android.

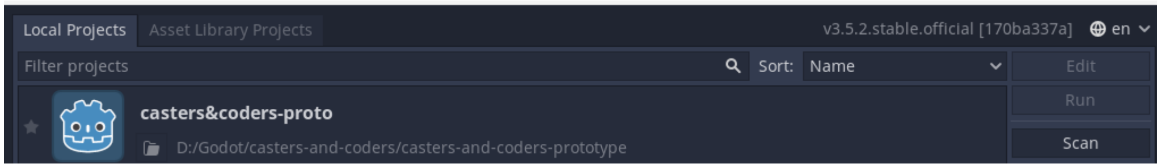
Next, you can acquire the Godot project simply by visiting the official GitHub repository of Casters and Coders.

- The GitHub repository link: https://github.com/sddec-13/casters-and-coders.

Simply fork the repository and clone it on your local machine to have access to the code. Once cloned, you can go ahead and start up the Godot game engine.
On the right-side menu, click on Import and Browse to the folder where you have cloned the GitHub repository. Inside the *casters-and-coders* directory, navigate to the specified file to import the godot project *casters-and-coders/casters-and-coders-prototype/project.godot*



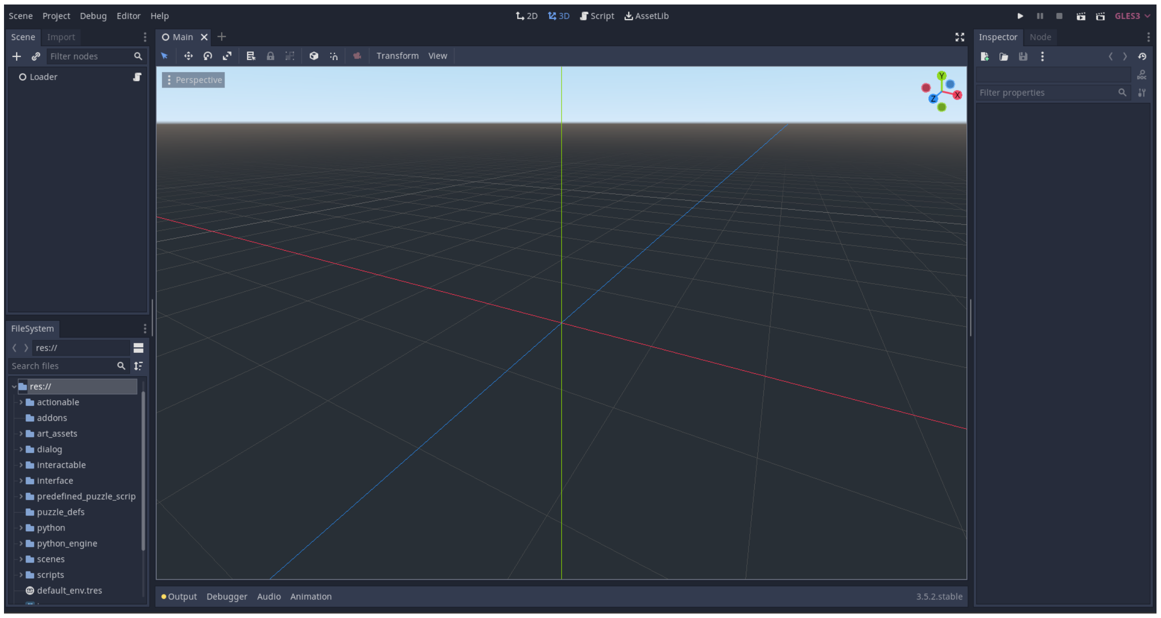
Once imported, you should be able to view the project on your menu as such



casters&coders-proto
D:/Godot/casters-and-coders/casters-and-coders-prototype

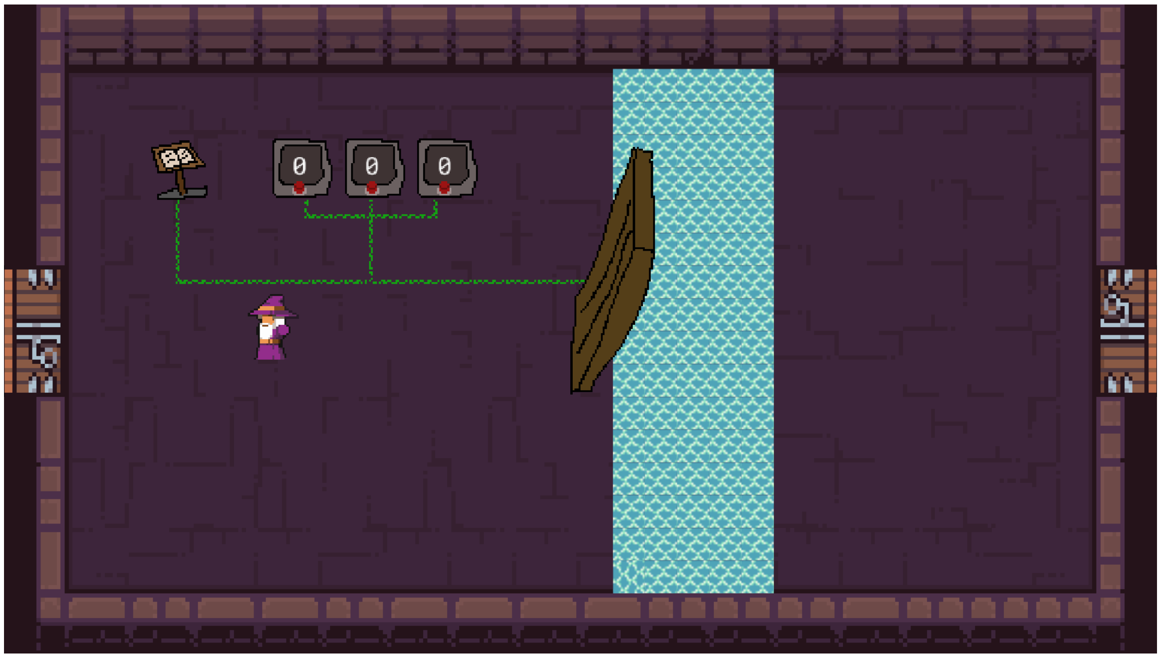
Double click on the project to open it up in the editor. If you wish to run the game on its own, you can also click on it once, and click on *Run*

Inside the editor, you will see something like this



On the bottom-left corner, you can see the project's directory tree where all the file, code, and assets are located. To run the project, simply click on the *Play* icon on the top right corner of the editor. Similarly, you can also press F5 to launch the game.

Once you run the project, you start at the first puzzle room of the game. It should look like this:



Editing the project itself requires some basic knowledge of Godot and its scripting language GDScript. There are a wide variety of resources available on the Internet, including Godot's official website.