Casters & Coders

DESIGN DOCUMENT

Team Number: sddec23-13

Client

Advisers: 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: 4/23/23/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 Player's progress will be saved 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 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.
- 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.

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

1 Team 7

1.1 Team Members 7

1.2 Required Skill Sets for Your Project 7

1.3 Skill Sets covered by the Team 7

1.4 Project Management Style Adopted by the team 8

1.5 Initial Project Management Roles 8

2 Introduction 8

2.1 Problem Statement 8

2.2 Requirements & Constraints 9

2.3 Engineering Standards 9

2.4 Intended Users and Uses 10

3 Project Plan 10

3.1 Project Management/Tracking Procedures 10

3.2 Task Decomposition 11

3.3 Project Proposed Milestones, Metrics, and Evaluation Criteria 12

3.4 Project Timeline/Schedule 13

3.5 Risks And Risk Management/Mitigation 14

3.6 Personnel Effort Requirements 15

3.7 Other Resource Requirements 18

4 Design 19

4.1 Design Context 19

4.1.1 Broader Context 19

4.1.2 User Needs 20

4.1.3 Prior Work/Solutions 20

4.1.4 Technical Complexity 21

4.2 Design Exploration 21

4.2.1 Design Decisions 21

4.2.2 Ideation 23

4.2.3 Decision-Making and Trade-Off 24

4.3 Proposed Design 25
4.3.1 Design Visual and Description 25
4.3.2 Functionality 26
4.3.3 Areas of Concern and Development 27
4.4 Technology Considerations 29
4.5 Design Analysis 29
4.6 Design Plan 29
5 Testing 30
5.1 Unit Testing 30
5.2 Interface Testing 30
5.3 Integration Testing 31
5.4 System Testing 31
5.5 Regression Testing 31
5.6 Acceptance Testing 31
5.7 Security Testing 32
5.8 Results 32
6 Implementation 32
7 Professionalism 32
7.1 Areas of Responsibility 32
7.2 Project Specific Professional Responsibility Areas 35
7.3 Most Applicable Professional Responsibility Area 36
8 Closing Material 37
8.1 Discussion 37
8.2 Conclusion 37
8.3 References 37
8.4 Appendices 38
8.5 Team Contract 38

List of figures

1 Gantt Chart 14
2 Work Estimate Breakdown 18
3 Design in a Broader Context 19
4 User Needs 20
5 2D Dungeon Environment 25
6 Script Editor Mockup 26
7 Basic Gameplay Loop 27
8 Proposed Godot Node Graph 30
9 Areas of Responsibility 34

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 Your 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
- Story writing experience
- 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

- Story writing experience
    - Theng Wei Lwe
    - Wenqin Wu
- Artistry skills
    - Edward Dao

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 team is expected to adhere 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 will help us to engage in highly parallel development, since teammates can rapidly respond to changes made by another.

We plan on using GitHub and Discord to help keep track of progress done throughout each member. Discord will be the main tool for communication whereas GitHub will be primarily used for version control. We will also use markdown files in Github for the bulk of our documentation.

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. *Story, Environment Design, and Asset Creation*
    2.1. Create a storyline to go with the game
    2.2. Create rooms/map for the game
    2.3. Write dialogue for character interactions
    2.4. 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. Design "Challenge" puzzles, which will be optional in-game, and lower-priority for development.
    3.6. 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.

3.3 Project Proposed Milestones, Metrics, and Evaluation Criteria

Milestones:

1. *Engine and Scripting System*
    1.1. Engine choice locked in
    1.2. Scripting language locked in with a hello world example
    1.3. Functional script saving and loading
    1.4. Functional in-game script editing via basic text editor
    1.5. Script API is formally defined in documentation
    1.6. Script API is implemented and integrated with the host engine
2. *Story, Environment Design, and Asset Creation*
    2.1. Complete storyline
    2.2. Completed map design consisting of multiple rooms adjacent to each other
    2.3. Designs of NPCs, player characters, and rooms fully implemented
3. *Puzzle design*
    3.1. Learning pathway completed (order of programming concepts to be taught)
    3.2. Introductory, Exploratory, and Challenging puzzles designed for each concept.
4. *Mechanics of the Game and Level Implementation*
    4.1. Fully functional player character
    4.2. Interactable objects/characters implemented
    4.3. Puzzles implemented to be solved using chosen scripting language
5. *User Interfaces*
    5.1. Fully functional user interface including menus, chat bubbles, and player information
    5.2. In-game IDE for user input with basic highlighting and code completion.

3.4 Project Timeline/Schedule

This chart assumes a sixteen-week timeline and 1-week sprints. Subtasks which have strict dependencies on other subtasks indicate these with a task number in parentheses. Some tasks are expected to be finished before others, in which case team members will be reassigned to help with other tasks in progress.

Note that this chart is not a steadfast plan but a starting point to work off of.

Task/Subtask	Sprint															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Engine and Scripting System																
1.1 Prototype scripting systems	X	X														
1.2 Choose engine, tool, and language			X													
1.3 Design scripting API			X													
1.4 Scripting MVP				X	X											
1.5 Improvements						X	X	X	X							
Story, Environment Design, and Asset Creation																
2.1 Storyline	X	X	X													
2.2 Game Map (3.2)				X	X	X										
2.3 Dialogue							X	X	X	X						
2.4 Source assets											X	X	X			
Puzzle Design																
3.1 Puzzle framework	X	X														
3.2 Curriculum	X	X														
3.3 Introductory puzzles		X	X	X												
3.4 Exploratory puzzles					X	X	X									
3.5 Challenge puzzles								X	X	X						

3.6 User feedback and polish								X	X	X	X	X	X			
Mechanics																
4.1 Player Controller	X	X														
4.2 Tileset environment	X	X														
4.3 Interactable objects			X	X	X											
4.4 Single puzzle & systems (1.4)(5.2)						X	X	X								
4.5 Additional puzzles									X	X	X	X	X	X	X	X
User Interfaces																
5.1 Style Guide	X															
5.2 IDE MVP		X	X	X												
5.3 Dialogue system					X	X	X	X								
5.4 Game menus									X	X	X	X				
5.5 IDE Polish													X	X	X	X

Figure 1: Gantt Chart

3.5 Risks And Risk Management/Mitigation

We have identified potential risks for each task along with potential mitigation strategies, listed below:

1. *Engine and Scripting System*
    a. (1.2) Select an engine, tool, and prototype to flesh out.
        i. (risk: 0.1) Our decision here could prove to be unworkable further down the line, requiring us to switch systems and possibly engine. Our mitigation strategy for this is two-pronged. First, we intend to design an API for interacting with our scripting system which is generic enough to fit *any* scripting system with little effort (see 1.3). Second, we have begun preliminary investigations into this topic early, evaluating a variety of tools as a part of research. We believe that this head-start will help us to more surely make the right decision here. Because of these strategies, the risk is reduced.
2. *Story, Environment Design, and Asset Creation*
    a. (2.1) Create a storyline to go with the game

i. (risk: 0.5) Once gameplay and puzzles are designed, the story may not fit well to the game developed. However, we have lots of extra free time scheduled for this set of tasks, so additional work can be put into reworking the story. This should be dealt with *before* significant work is done for task 2.4, so that limited refactoring is necessary.

b. (2.4) Create or source assets for the game

   i. (risk: 0.2) We may not be able to find suitable assets. This is a fairly acceptable situation however. Our requirements do not include artistic cohesion so we can reasonably repurpose ill-fitting assets.

3. *Puzzle design*

   a. We do not anticipate any significant risks associated with puzzle design.

4. *Mechanics of the Game and Level Implementation*

   a. (4.4) Implement a designed puzzle, creating reusable systems and integrating the scripting system.

      i. (risk 0.6) We may discover an issue in existing game systems, the scripting system, or the in-game IDE which needs to be addressed before this task can be completed. This would create a significant delay. Mitigating this risk, we believe that our agile framework will help us to deal with such unexpected delays. Additionally, the developer assigned to this task can spend their downtime assisting with these unexpected squashing issues.

   b. (4.5) Continue implementing puzzles.

      i. (risk 0.7) We may not have time to implement all the puzzles which we design. In fact, this is fairly likely. To mitigate this, we will focus on developing Introductory puzzles first, and move forwards to Exploratory and Challenge puzzles afterwards to flesh out the game. This way, the later content is less fundamental to our game and can be cut with less consequences.

5. *User Interfaces*

   a. We do not anticipate any significant risks associated with user interfaces.

3.6 Personnel Effort Requirements

Task/Subtask	Hours	Context
Engine and Scripting System		
1.1 Prototype scripting systems	18	Preliminary work on these is already done, deflating time requirements.
1.2 Choose engine, tool, and	2	After thorough prototyping this should be a

language		thoughtful, but relatively easy decision.
1.3 Design scripting API	4	We will need to design the scripting API carefully, as a lot of the project depends upon it.
1.4 Scripting MVP	20	This is an important blocking-task, so it should be prioritized.
1.5 Improvements	60	"Improvements" is sort of nebulous by design. It includes bugfixes as well. This may well take more or less than 60 hours.
Story, Environment Design, and Asset Creation		
2.1 Storyline	8	The storyline should start broad and become more specific as gameplay elements materialize. This task will be somewhat spread out, and may take more than 8 hours because of this.
2.2 Game Map (3.2)	16	The game map is a real logistical challenge, especially as room layouts may change across puzzle design and implementation.
2.3 Dialogue	24	Dialogue will be very important to our game - it is the main source of guidance for the player. As such, there should be a lot of it, and it should be good.
2.4 Source assets	8	We will need a lot of assets to fit the scope of our game. Finding fitting ones for free could take some involved searching.
Puzzle Design		
3.1 Puzzle framework	8	The puzzle framework is important preliminary work. It will help better define bounds in which all puzzles will work. This design document should be thorough, though it may be modified later.
3.2 Curriculum	16	The curriculum is the most essential heart of our game, and while it is fairly straightforward, it will involve a lot of detail.

3.3 Introductory puzzles	48	Introductory puzzles will take the longest as we will have to come up with novel, effective puzzles to teach every concept in the curriculum.
3.4 Exploratory puzzles	30	These will reuse gameplay elements from the introductory puzzles, adding some complexity that tests the player's understanding. Because the gameplay elements will already exist, extending them will be more straightforward.
3.5 Challenge puzzles	24	Challenge puzzles are an extension of the exploratory puzzles. They will require less work for two reasons. First, they will be optional, so if one of these is too hard, that's acceptable. Additionally, these will be the last to be implemented, so time constraints will likely mean that we won't be able to implement very many.
3.6 User feedback and polish	40	This is an open ended task, but is pretty involved. It involves playtesting levels and redesigning puzzles based on player feedback.
Mechanics		
4.1 Player Controller	4	This is a straightforward task with many relevant guides and resources.
4.2 Tileset environment	6	This is a little more involved than the previous, but is also well documented.
4.3 Interactable objects	8	This system will be somewhat simple, allowing the player to interact with objects in the environment in a generic way. However, there is little built-in engine support for this feature, so it will have to be designed and crafted by hand.
4.4 Single puzzle & systems (1.4)(5.2)	12	A single puzzle is not large, and if our supporting systems work well then this should go smoothly. However, we are likely to uncover bugs in those systems in this task. Additionally, this will involve creating resources that can be reused for all subsequent puzzles.
4.5 Additional puzzles	100	This task is open ended. Depending on the

		scope and scale of our puzzle designs, it could go on for much longer. Several developers will be able to work in parallel on this task, which should make it achievable.
User Interfaces		
5.1 Style Guide	6	The style guide should be created thoroughly, and will accelerate all future UI work by providing a helpful design framework.
5.2 IDE MVP	8	The IDE MVP will take significant work, but being a minimum viable product, it can be extremely simple at this stage.
5.3 Dialogue system	24	The dialogue system should be very robust to accommodate the amount and quality of dialogue. It will likely take involved work.
5.4 Game menus	8	The game menus will be simple. They present a decent amount of work with low complexity. The most difficult part of this task will be full keyboard navigation.
5.5 IDE Polish	32	The IDE can be polished to a very high degree - look at modern code editors. However we only really need syntax highlighting and an "available APIs" bank, as well as a rudimentary filesystem for saving/loading scripts. Code completions would be nice but are optional.

Figure 2: Work-Estimate Breakdown

3.7 Other Resource Requirements

1. Free-to-use external sprites, tile maps, art to complement the game design and theme
2. Tools for version control
    a. GitHub
    b. We'll mirror to the class GitLab for evaluation
3. Tools for communication
    a. Discord
    b. GitHub
4. ISU students to fill out surveys and test early versions of the product
5. Old computer to be used for testing the game on low end hardware
6. Computers to develop on

7. Game engine for development, and libraries for language embedding.

4 Design

4.1 Design Context

4.1.1 Broader Context

Area	Description	Examples
Public health, safety, and welfare	By providing an educational resource, we are directly enriching public welfare. We can take our players' welfare into account by doing things such as giving warnings if there are going to be flashing lights.	Players who engage with the game will learn valuable concepts which can be applied in personal and professional settings.
Global, cultural, and social	Our project is not directed towards any specific cultural group. However, due to time and resource limitations the game will be presented entirely in English, which might exclude non-English speaking people.	Our project will not be culturally charged or controversy-provoking in any way.
Environmental	What environmental impact might your project have? This can include indirect effects, such as deforestation or unsustainable practices related to materials manufacture or procurement. Beyond consuming electricity, this project should not have a significant impact on the environment. We can work to make our game lightweight and optimized to reduce power consumption	Increasing/decreasing energy usage from nonrenewable sources, increasing/decreasing usage/production of non-recyclable materials We will work to make Casters and Coders as efficient as possible to cut down on electricity use.
Economic	What economic impact might your project have? This can include the financial viability of your product within your team or company, cost to consumers, or broader economic effects on communities, markets, nations, and other groups.	Product needs to remain affordable for target users, product creates or diminishes opportunities for economic advancement, high development cost creates risk for organization Players who play this game will learn key programming concepts,

	Programming is a skillset that is in very high demand in our modern world. By teaching people how to code with our game, we are helping our players broaden their skill sets.	which will make it easier to go learn more advanced concepts in the future

Figure 3: Design in a Broader Context

4.1.2 User Needs

Group	Needs
Prospective coders with no programming experience	Gradual introduction to basic programming concepts because they have no knowledge of any of the core concepts needed. They will also need clear "best practices" to avoid common pitfalls.
Curious beginner coders	Introduction to somewhat more complex concepts because they might already know the basics and might be curious to learn more
Workers looking to augment their workflows	The game to use a real language so they can easily transition to real-world gains quickly

Figure 4: User Needs

4.1.3 Prior Work/Solutions

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.

4.1.4 Technical Complexity

1. The project's design contains multiple components that must be divided among several team members.
    a. Scripting Language Embedding - Technology to embed an existing language into the game to parse code written by the users. Requires extensive knowledge of game engines, programming languages, and software engineering.
    b. Godot/Unity Scripts - Component of game that is required to parse game logic and user input in real-time. Defines behavior of components in-game and behind the scenes. Requires extensive knowledge of game engine, C programming, and game physics/math.
    c. Puzzle and Storyline - The main content and gameplay that users will be interacting with. Contains mind-stimulating information and mechanics to engage users. Requires extensive skill in high-level order game design, technical problem-solving skills, and conceptual understanding of various genres and styles.
2. The problem will require the team to work together using a version control tool such as GitLab. This matches the current industry standard of utilizing collaboration tools for software development such as GitHub to maintain an up-to-date environment for software development.
3. The problem scope requires the team to design a completely original and new product from scratch. This is common in the software industry as companies are always looking to create new and innovative products for their consumers. The team is challenged in the same way to analyze the problems, brainstorm solutions/ideas, and execute the desired steps.
4. The team will be adhering to an agile development methodology which is commonly practiced in multiple software companies. Each team member will be given individual tasks that have to be completed within a specific time limit to reach a milestone. Development of the project will cycle often around debugging and testing to ensure compliance to the requirements and desired output.

4.2 Design Exploration

4.2.1 Design Decisions

Game Style

We looked at a variety of styles for our game to follow. Here are the most notable categorical decisions.

Realtime or Time-Controlled

In a real-time system, the scripts being written must be polled or registered to callbacks; this adds a layer of complexity to scripting but also grounds it more in the game world. It lets players tangibly interact with their scripts to a greater degree.

In a time-controlled system, more typical programming can be done. Here, a player-written script would be called once and operate until completion, then terminate. This allows a more controlled environment with more typical programming conventions but reduces a player's ability to interact with their script *after* writing it.

We decided to go with a realtime system, but design some puzzles in a time-controlled way by letting them register a callback which is run once by an in-world button, or other similar activation method.

Action or Puzzle

In an Action based game, players' scripts would control some gameplay element which is used dynamically and combatively. The most straightforward application of this is that of a typical RPG, but those games rely on number-creep for progression, which programs can achieve arbitrarily. Alternatively, we could gate progression by requiring more and more complex script behaviors, but at that point you're writing a puzzle.

In a Puzzle based game, players' scripts control gameplay elements in the world, and scripts must be able to perform certain tasks to overcome contrived puzzle scenarios. This style of game gives us far more control over how players engage with programming concepts, so that we can gate their progress with finer-grained programming concepts.

We decided to go mainly with a Puzzle based style of gameplay. We think that it best suits our needs as an educational game. However, we are still interested in having highly physical puzzles which require some degree of player interaction, or possibly skill.

Puzzle Formats

We will need a fairly standard format for our puzzles to streamline development and to keep a consistent player experience. We have decided on two styles of puzzle to include.

First, we will have test-based puzzles. These are more conventional and constrained, running a battery of tests against the player's script and judging success based on their outcomes.

The second format will be based in world traversal. In this style of puzzle, the player's script will control a number of game elements through an exposed API, and their success will be measured by whether they can get to the room's exit. Obstacles in these puzzles will be situations like a bottomless pit to cross, a trap to outmaneuver, or a monster to defeat.

4.2.2 Ideation

One design decision that we spent a lot of time considering options for was the choice of user-facing scripting language. We identified multiple options based on the feasibility of integration with the engine and ease of use for the player. The options we considered and researched are listed below:

- Python: This language provided some of the easiest binding capabilities to our engine choices and is also well-known among the programmer community for being easy to learn. We considered its odd scoping syntax, whitespace-based, against our personal preference for braces-based and determined the trade-off was insignificant for its ease of use. We also discussed potential pitfalls around its type system, which is duck-typed and dynamic-typed, and determined that while a statically typed language may remove some error cases, it also removes some of the ease of use for brand-new users. Being interpreted means we also did not need to worry about embedding a compiler toolchain.
- Typescript: We found that TypeScript was very easy to use as well, and its static type system was easy enough to understand for most new users. We also theorized that static typing would allow our embedded editor to provide hints and autocomplete for the user to assist them. The compiler was able to be embedded with ease as well. We were, however, wary about using this language because the compiler adds additional complexity. On Unity, we used the Clearscript package, which we found introduced a significant penalty for crossing the boundary between the engine and embedded scripting.
- Lua: This was the language we found most easy to embed in the engines available to us. We did find that its syntax was a little off-putting, 1-indexing could lead to problems when transitioning to other languages, and its reliance on metaprogramming to emulate other language features could lead to extreme confusion for beginning programmers.
- Kotlin: Kotlin was the hardest language to embed in our engines. In Unity, we attempted to use the IKVM project to run JVM code inside Unity's .NET environment. This allowed us to run a simple Kotlin hello-world program, but attempting to use the experimental scripting library caused the entire engine to crash. Godot was a bit nicer but still had problems, there's a community-driven Kotlin language binding, but it's in Alpha state. We were able to get a hello-world program working with minimal effort, but once again, the scripting library caused issues. We have been in contact with the developers of this language binding and believe we have a fix for this issue, so research on Kotlin is still ongoing.
- Webassembly: While not a language in itself, Webassembly opens the doors to a much larger pool of languages to integrate and would provide us a single interface to easily swap between languages. We were able to integrate Wasmtime into Unity but found that the native library nature of Wasmtime could pose problems for a

cross-platform game. We also decided that embedding a full compiler toolchain that compiles to Webassembly could be potentially problematic.

4.2.3 Decision-Making and Trade-Off

We evaluated all of our potential language options with respect to multiple criteria. The most important criterion was that the language could be embedded in our chosen engine, as looking at languages that weren't feasible in the first place would just be a waste of time. The second most important criterion was the ease of learning from the perspective of a new programmer. We evaluated this criterion against several subcriteria: lack of boilerplate or "magic" ceremony functions or constructs, readability of code, conciseness of code, and overall syntax "feel." We also evaluated each language with respect to its feature set, specifically its inclusion of object-oriented and functional paradigms as well as more advanced features like comprehensions or operator overloading.

Our current choice is Python. It satisfies most of our criteria for the chosen language. It is easily embeddable via IronPython, it does not require any special ceremony for simple scripts, and most beginning programmers find it easy to read. Python also includes support for both object-oriented and functional programming paradigms. The drawbacks of our other options are listed above.

4.3 Proposed Design

4.3.1 Design Visual and Description

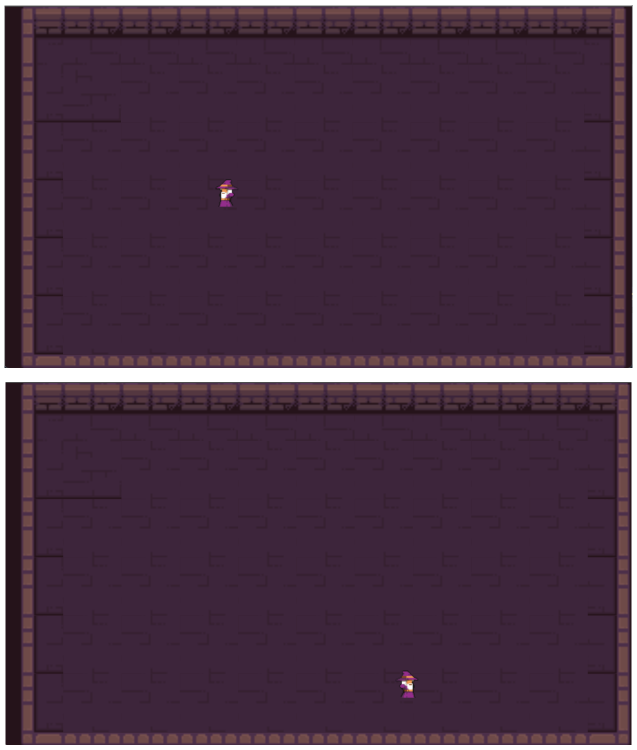
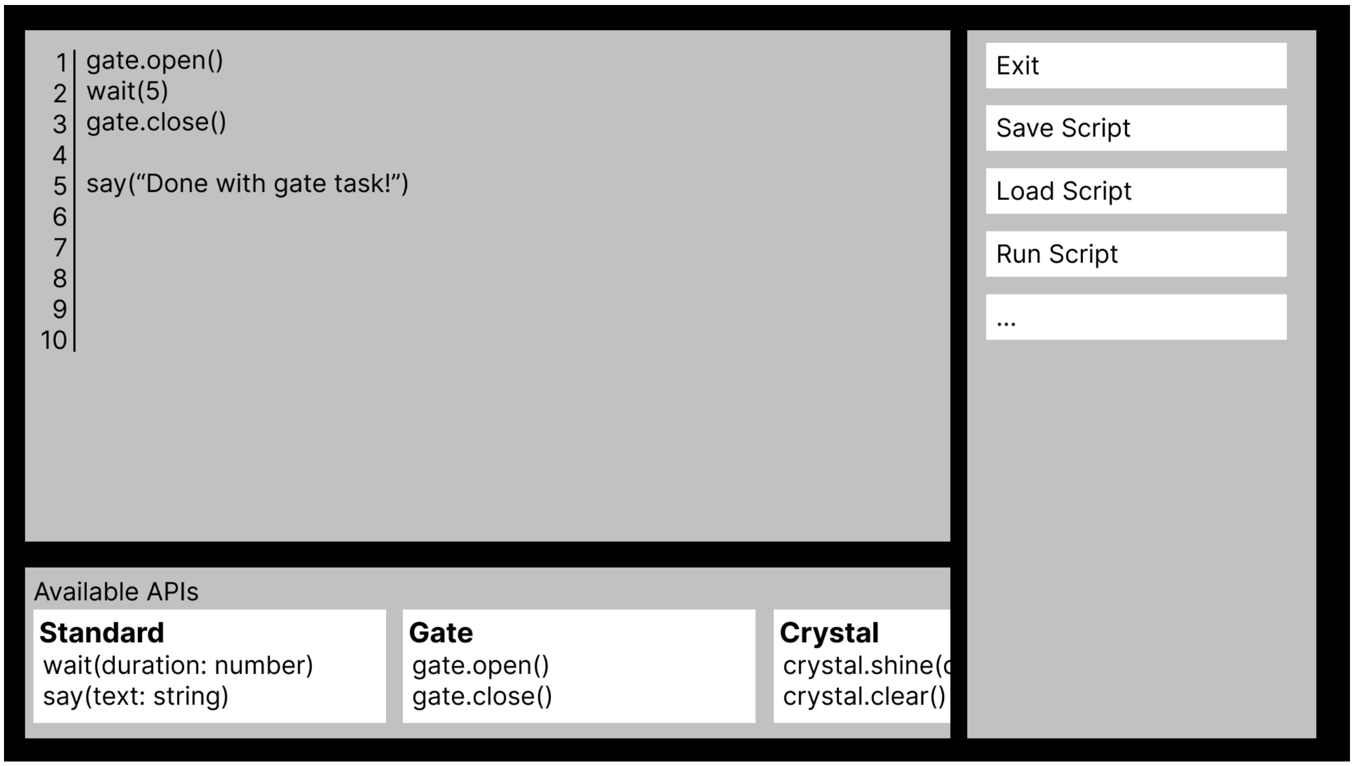


Figure 5: 2D Dungeon Environment

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 will be able to interact with the environment to complete puzzles and earn progress throughout the game.



```
gate.open()
wait(5)
gate.close()

say("Done with gate task!")
```

Figure 6: Script Editor Mockup

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.

4.3.2 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.

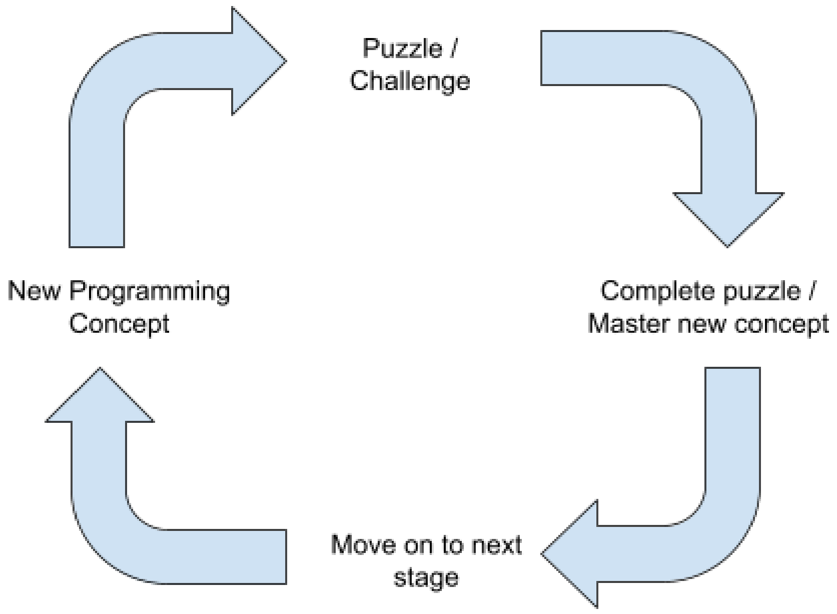


Figure 7: Basic Gameplay Loop

The current design 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.

The design also allows the game to continue being fun and stimulating, as desired in the non-functional requirements. By introducing new concepts and harder stages, users will be inclined to put in more effort, making it more rewarding and fun.

4.3.3 Areas of Concern and Development

Our primary concern is ensuring the scripting system works and teaches the users how to program in a progressive and slow manner. The scripting is core to the game's purpose, and if it isn't functional or is too difficult for a beginner to use, our game has failed at that purpose.

This concern breaks down into two issues: the technical implementation of the scripting engine must be performant, robust, and easy to develop with, and the puzzles themselves must be designed carefully to be intuitive and teach concepts well.

To address the technical side, we will spend considerable resources on the scripting system and UI. We plan to test as much of the system as possible and provide UI mockups to external parties for feedback.

As far as design, all we can do is come up with a lot of ideas and get plenty of player feedback during the process to figure out what works and what doesn't as early as possible.

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. Our options, research notes, and conclusions for each are listed below.

4.4.1 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. Here are our findings for a variety of languages:

- Python
    - Very simple, easy-to-understand syntax for basic operations.
    - Good support for object-oriented concepts and patterns.
    - Good support for functional paradigms: lambda functions, higher-order functions, currying.
    - Interpreted language, so we don't need to worry about bundling a compiler.
    - Dynamic typing makes types often nebulous and introduces many potential pitfalls for players.
    - IronPython interpreter allows embedding in .NET environments
    - Godot has a Python binding as well
- Kotlin
    - Nice succinct syntax, certainly less boilerplate-driven than Java.
    - Has great support for all popular programming paradigms.
    - Compiled language, which runs in the JVM. So we would have to bundle a compiler and a JVM implementation.
    - Good Godot support, so if we chose Godot, there exists tooling to use Kotlin.
    - We would need to leverage the scripting libraries to execute embedded code, which are highly experimental and may not work as expected.
- Javascript
    - Great functional programming support.
    - Interpreted language, like Python.
    - Dynamic typing creates many pitfalls, especially with Javascript's judicious type coercion.
    - The prototype-based object system provides a poor introduction to object-oriented programming.
    - Would require embedding a Javascript engine, like V8. Most engines we've found are not reimplementations like IronPython is for Python but rather wrap V8, and thus can introduce additional compilation complexity.
- Typescript
    - Fantastic, robust type system, fixing our main gripe with Javascript.

    - Requires a transpilation step, but transpilers exist in Javascript, so if we get Javascript running, then Typescript support is nearly free.
- Lua
    - Approachable, lightweight syntax.
    - Highly unusual syntax, which would not translate well to other programming paradigms.
    - Not widely applicable in real-world programming.
    - Poor support for object-oriented paradigms, emulated through Tables which are a Lua-specific feature.

We've narrowed down our choices into two viable options: TypeScript and Python. Because of TypeScript's complex build tooling, we've decided to attempt to use Python first.

4.4.2 Game Engine

- Unity
    - Most well-known engine, with lots of community support.
    - Plenty of tools for high-quality visual effects.
    - Uses an unusual skew of C#, which makes importing libraries painful.
- Godot
    - 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.
- Impact
    - Runs natively in-browser, sidestepping platform support.
    - Built on Javascript, which makes embedding Javascript and Typescript fairly painless.
    - Niche, with limited tools and limited community support.
    - Less features than the other engines.

Currently, we are most likely to use Godot. We like its robust design, and having many language bindings lets us use a wider variety of tools.

4.5 Design Analysis

We have not yet implemented our design.

4.6 Design Plan

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

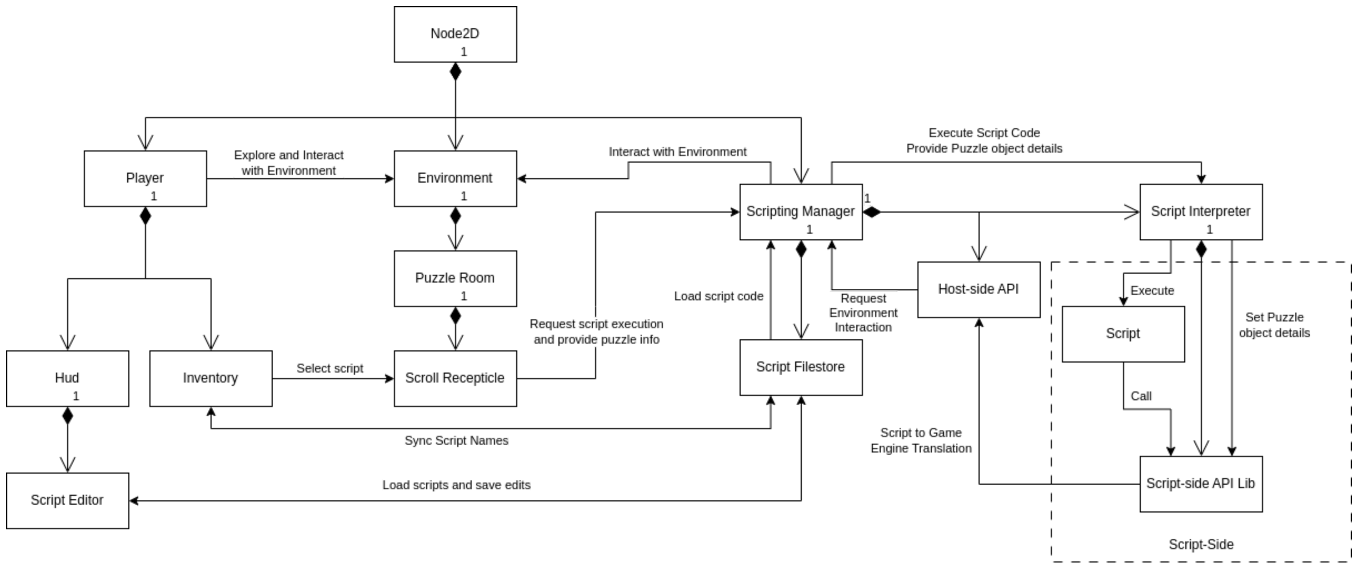


Figure 8: Proposed Godot node graph

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

5.1 Unit Testing

We will be unit testing the user-facing scripting component. Thanks to using a real language with an embedded engine design, we can split everything written in that language into a separate sub-project and unit test it separately. This includes the API between the scripting language and the game engine; the engine side will be emulated using mocks.

We will also be unit-testing the host side of the interface API and the script manager singleton via the Godot Unit Test (GUT) tool. Dispatch of commands to game objects will be emulated during tests.

5.2 Interface Testing

- We will write interface tests to insure that the different game systems are interacting correctly

    - This includes testing to insure things such as the spells the player codes trigger the correct animations
    - This includes testing to insure things such as the inputs the player makes give the correct outputs in the game
- We will test with accessibility filters to make sure that the game is accessible to people with sensory impairments
    - We won't have puzzles that are reliant on audio cues to solve so that people with hearing impairments are able to play.
    - We will allow the player to add filters to the game to help with certain kinds of color blindness

5.3 Integration Testing

The most critical integration path is the path from a user script to a game object. A user script command passes from the script engine to our script-side interface API, then to the host-side API, to a script manager object, and finally dispatched to the affected game object. This path will be tested thoroughly via the Godot Unit Testing (GUT) tool, and if this path is faulty, the entire center point of the game becomes faulty.

5.4 System Testing

Since our project is a game with more subjective requirements than other projects might have, our system testing will not be automated. Instead, we plan to send out development builds to interested parties for game testing and feedback. Such a strategy mirrors what "triple-A" game studios do with their own game testers. An advantage of having live testers is that we can get subjective feedback on problems we never considered. Automated testing has the fault that only potential problems thought of by the designers are tested, and so they tend to ignore certain failure classes.

5.5 Regression Testing

The current strategy is to ensure that the current functionality works perfectly before adding new features, which means we will always have a working version. The CI/CD system connected to our repo will also perform a battery of tests on the embedded scripting system for every pull request. We will only merge the pull request if all tests succeed, protecting against regressions in our most critical system.

5.6 Acceptance Testing

For functional requirements, we will make sure that our game meets the criteria that we set out to meet. We will also write tests to ensure that the game's systems interact correctly.

For nonfunctional requirements we will be testing based on the criteria we wrote out at the beginning. More specifically, we will profile the game on a lower end device having exactly our stated minimum specs. We will require that the frame rate exceeds a lower bound of 60 frames per second for 99% of the time when running on this machine.

To test functionality, we will use visual testing because all the functions are straightforward and can be determined if it's working or not just by running the game.

Additionally, we intend to playtest the game. In the survey we sent out at the beginning of the semester, we asked recipients if they would like to test an alpha build of the game. Those that agreed will be sent copies of the game alongside a survey to rate their satisfaction.

5.7 Security Testing

We are planning to make the game open-source and free to play, so security is not applicable in this project. Additionally, this is a singleplayer game, so if players wish to cheat, we hope only that they enjoy themselves while doing so.

5.8 Results

We have not yet conducted testing.

6 Implementation

We have begun implementing the user-facing scripting system using the Python Godot bindings through PluginScript. We are also simultaneously designing puzzles, writing the overall storyline, and getting a shared project up and running. Soon we will be working on prototypes for the environment and character.

7 Professionalism

This discussion is with respect to the paper titled "Contextualizing Professionalism in Capstone Projects Using the IDEALS Professional Responsibility Assessment", International Journal of Engineering Education Vol. 28, No. 2, pp. 416–424, 2012

7.1 Areas of Responsibility

We selected the SE code of ethics.

SE:

1. Public. Software engineers shall act consistently with the public interest.

2. Client and employer. Software engineers shall act in a manner that is in the best interests of their client and employer, consistent with the public interest.

3. Product. Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.

4. Judgment. Software engineers shall maintain integrity and independence in their professional judgment.

5. Management. Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.

6. Profession. Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.

7. Colleagues. Software engineers shall be fair to and supportive of their colleagues.

8. Self. Software engineers shall participate in lifelong learning regarding October 1999 85 Software Engineering Code of Ethics and Professional Practice the practice of their profession and shall promote an ethical approach to the practice of the profession.

Area of Responsibility	**Definition**	**SE Code of Ethics**	**Difference from NSPE Code**
Work Competence	Perform work of high quality, integrity, timeliness, and professional competence.	Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.	NSPE focuses more on a sense of honor and loyalty to one's employer, whereas the SE code focuses more on professionalism.
Financial Responsibility	Deliver products and services of realizable value and at reasonable costs	The 5th principle, management, states that software engineers shall make accurate assessments on cost on the project they plan on working on. They shall also keep in mind their	NSPE doesn't directly address cost management, instead simply asserting that an engineer should be loyal and trustworthy. I think that Financial Responsibility is a

		scheduling and skills needed for the work to reasonably achieve to finish.	direct implication of this.
Communication Honesty	Report work truthfully, without deception, and understandable to stakeholders.	The 4th principle states that engineers must not deceive others and must not commit acts of bribery or fraud. It also states engineers should only endorse documents whose subject matter they are intimately familiar with	NSPE suggests in many sections that an engineer must not be deceptive or alter or skew facts. The SE code is somewhat less thorough in this regard, but addresses it directly.
Health, Safety, Well-Being	Minimize risks to safety, health, and well-being of stakeholders.	The SE code of ethics holds the health, safety, and welfare of the public as its primary purpose, and the 8 principles are in service of this purpose.	The first NSPE canon is to hold paramount the well being of the public. The fourth is to act as a trustee to client and employer. The SE code is similar, but puts more emphasis on the public good than that of the employer than NSPE does.
Property Ownership	Respect property, ideas, and information of clients and others.	Software Engineers shall uphold the best interests of their client and employer, while not engaging in deceptive activities.	The fourth NSPE canon is to act as a faithful trustee to both employer and client. The fifth is to avoid deceptive acts. In this way the codes treat these manners virtually identically.

Sustainability	Protect environment and natural resources locally and globally.	Software engineers shall first and foremost act in the best interests of the public. This includes an obligation not to harm the environment.	In the NSPE code, engineers are encouraged to act in the interest of sustainability, but this tenant is more prevalent in the SE code overall.
Social Responsibility	Produce products and services that benefit society and communities.	Avoid falsifying details of product, documents, and methodologies by being truthful,ethical, and taking responsibilities of software development concerning public relations.	The NSPE code goes into far more detail about exactly how an engineer can communicate with the public, what matters they can communicate over, what disclaimers to add, and more. The SE code makes simpler, broader statements in this regard.

Figure 9: Areas of Responsibility

7.2 Project Specific Professional Responsibility Areas

- **Work Competence**: This principle does apply to our project because we are designing a video game that follows the goals that we have set up and what our clients want from this video game, while fulfilling the objectives that are set from this course. We expect this project to be picked up by a later team and thus our code must be well-designed and easy to understand. There are also a lot of interfaces between modules, especially in the area of the user-facing scripting, and thus keeping the quality of work high will lessen any integration issues between modules.
- **Financial Responsibility**: This principle does not apply to our project in terms of financially. We are doing the entire project free of cost. Underneath SE Code of Ethics 5th principle, management, does apply to our project. We will need to be able to manage our time and access our skills to see if we can manage to complete the project. For this semester of the project we will be spending time on researching and learning about our project and what needs to be done to complete it for the next semester.

- **Communication Honesty**: This principle does apply to our project as we must not deceive our own team members or our adviser/client about our capabilities or understanding. Our project group is doing well in this regard, we each have a list of related experience, and when we notice a deficiency our members gladly take up learning opportunities. We communicate with each other and our client regularly and describe any difficulties we've encountered.
- **Health, Safety, Well-being**: This principle somewhat applies to our project, our game does not have inherent safety risks but we must ensure we don't stray into narrative storylines that can affect mental wellbeing. As of right now our project is doing well in this area, we are moving towards a mentally-stimulating puzzle solver game.
- **Property Ownership**: We will not process any users personal information, so there won't even be an opportunity to be untrustworthy with clients' information. The game is also a low-risk application, so even the most catastrophic failures on our part would be incredibly unlikely to cause any real harm. Performance: High.
- **Sustainability**: Since our product is software-only and small-scale, there is little risk of environmental damage. However, we are still doing what we can to protect the environment. All the design documents and diagrams are designed and stored in cloud drive so there is no paper getting used and wasted. Additionally, we will strive to make the game performant, so that it does not create excessive power use. The performance of this section: High.
- **Social Responsibility**: This professional responsibility applies to our project because as developers of the video game, we are responsible for the information that is being displayed to the users or the public. Our team is performing highly in this area of responsibility as we ensure that the information contained in the video game is valid, correct, and accurate. This allows the users to have faith in our product and the team behind it.

7.3 Most Applicable Professional Responsibility Area

One responsibility that is very important to our project is Social Responsibility. To our project specifically, we have a responsibility to teach the public proper programming techniques and paradigms in a real language. Teaching incorrect theory, ineffective approaches to coding, or a language no one will use are things we consider directly damaging.

Programmers who learn incorrect or ineffectual techniques will carry incorrect information forward into their daily lives. We have demonstrated social responsibility by taking time to consider all of our language options, weighing them against the usefulness in daily life, and devising puzzles that teach the correct way to use the language as well as the correct way to use programming in general.

8 Closing Material

8.1 Discussion

Our project is not yet complete but we have made excellent progress. We've planned a solid design that fulfills all of our functional requirements. We believe we have a solid plan for making our game fun and engaging, and we have spent a good amount of time considering accessibility options. We are on track for fulfilling all of our requirements. We believe that if this game is successful, it will provide a novel and entertaining option for people across the world to learn basic programming concepts, something that is becoming more and more essential in the modern workplace as technology proliferates all we do.

8.2 Conclusion

We have come up with unified decisions regarding the game engine (Godot) and programming language (Python) of choice. We came up with potential storylines and puzzles to complement the video game. Our goals are to create a flexible language embedding interface with an implementation for Python, a number of interactive puzzles teaching basic programming concepts, and a wide variety of UI elements to facilitate this.

We will face a variety of challenges. We will need to implement our language system in a robust way, we'll need to design puzzles which are engaging and use it well, and we'll need to create a UI that is intuitive, stylish, and helpful to the player along every step of the way. These are not simple systems, and their implementation will take a lot of careful development, with regular review and revision.

8.3 References

[1]

Godot Engine, "Tutorials and resources," Godot Engine documentation, 15-Apr-2022. [Online]. Available: https://docs.godotengine.org/en/3.5/community/tutorials.html. [Accessed: 23-Apr-2023].

[2]

S. Mukherjee, "Godot Languages Support," GitHub, Mar. 28, 2023. [Online]. Available: https://github.com/Vivraan/godot-lang-support. [Accessed: Apr. 23, 2023]

[3]

Unity Technologies, "Unity - Manual: Unity User Manual (2019.2)," Unity3d.com, 2019. [Online]. Available: https://docs.unity3d.com/Manual/index.html. [Accessed: Apr. 23, 2023]

8.4 Appendices

Market survey conducted by us: Casters & Coders Survey (Responses)

Language research notes: Language Research Notes

8.5 Team Contract

8.5.1 Team Members:

1) Theng Wei Lwe

2) Wenqin Wu

3) Brennan Seymour

4) Branden Butler

5) Max Bromet

6) Edward Dao

8.5.2 Team Procedures

1. Day, time, and location for regular team meetings:

Every Thursday, 2:30p.m. - 3:30p.m. (virtual & face-to-face) Durham 353. Advisor included in meeting every other week (bi-weekly)

2. Preferred method of communication updates, reminders, issues, and scheduling:

Preferred method will be Discord for simple communication; Face-to-Face for more involved discussion.

3. Decision-making policy:

Majority vote

4. Procedures for record keeping:

Team members rotate every week to be in charge of meeting minutes. Minutes are shared/archived through google docs.

8.5.3 Participation Expectations

1. Expected individual attendance, punctuality, and participation at all team meetings: Each individual is expected to attend weekly meetings. Individuals are allowed up to 10 minutes being late. All members are expected to participate and voice opinions during meetings.

2. Expected level of responsibility for fulfilling team assignments, timelines, and deadlines:

All team members are expected to fulfill tasks that are assigned, meeting deadlines and keeping up to date with the timelines.

3. Expected level of communication with other team members:

Each team member is expected to respond to messages in Discord within a day.

4. Expected level of commitment to team decisions and tasks:

Each team member is expected to give 100% commitment towards team decisions and tasks assigned to them.

8.5.4 Leadership

1. Leadership roles for each team member (e.g., team organization, client interaction,

individual component design, testing, etc.):

- Branden Butler: Script embedding and overall system architecture
- Theng Wei Lwe: Ensure team's keeping up with deadlines
- Edward Dao: Supporting team members working on the game designing and Unity aspect of the project
- Wenqin Wu: Making sure team members research out the right resources when faced with an issue.
- Brennan Seymour: Game systems architecture and implementation.

2. Strategies for supporting and guiding the work of all team members:

- Those with more experience should help teach those with less
- Regular check-ins to ensure all team members are on the same page and aren't getting stuck.
- Code reviews and at least two approvals for all PRs.

3. Strategies for recognizing the contributions of all team members:

- Keeping track of pushes on Git
- The weekly reports done

8.5.5 Collaboration and Inclusion

1. Describe the skills, expertise, and unique perspectives each team member brings to the team.

- Branden Butler: Very deep knowledge of programming languages, significant experience in designing event-driven or message-passing /

Actor-based systems. Significant experience with distributed development and leading small teams

- Edward Dao: Experience in Advance and basic knowledge of java, c, and c++, experience with leading a small team for web development, experience in front end and back end development, and little exposure to game design (not in unity).
- Theng Wei Lwe: Experience with web development frameworks such as Angular and React. Strong knowledge and experience in Java, JavaScript and PHP. Full-stack software development work experience.
- Wenqin Wu: Experienced with the game engine we are using, Unity, had medium level of game development and C#. Medium level of 3D game development and low level of 2D game development. BrainStorm with game design. C,C++,Java,database and backend development.
- Brennan Seymour: Thorough knowledge of the Unity game engine, two games finished and published on Itch.io, many more started and left unfinished. 4 years of professional web development experience. Even tempered, ie. chillin'. Some experience with shaders and OpenGL/WebGL programming.

2. Strategies for encouraging and support contributions and ideas from all team members:

- Mainly communicating through Discord to ask for assistance and supporting other members.
- Sharing ideas on Discord and the weekly meetings

3. Procedures for identifying and resolving collaboration or inclusion issues

- Through the use of Discord, we have a team server to mainly communicate with each other. For more private communication we can use direct messaging.

8.5.6 Goal-Setting, Planning, and Execution

1. Team goals for this semester:

- Successfully set up the development environment.
- Come up with a complete storyboard.
- Create lots of architectural documents & diagrams.
- Design one good puzzle for each of the following programming concepts: if/else, looping, callbacks.
- Get a working prototype of the scripting system running with the selected language.

2. Strategies for planning and assigning individual and team work:

- Split tasks up to several categories and assign individuals to each of them

- Since we have six members, we'll assign three goals per week to three dynamically-allocated pairs. (we'll write a custom allocator of course)

3. Strategies for keeping on task:

- Weekly meetings to update each member's task status
- Push changes often and perform code reviews as soon as possible
- Help each other out if stuck.

8.5.7 Consequences for Not Adhering to Team Contract

1. How will you handle infractions of any of the obligations of this team contract?

Members will be held accountable and shall buy food/snacks for the rest of the group, to be consumed at the next group meeting. ($20)

2. What will your team do if the infractions continue?

Report the offender to the instructor/TA. Frown at them.

8.5.8 Signatures

a) I participated in formulating the standards, roles, and procedures as stated in this contract.

b) I understand that I am obligated to abide by these terms and conditions.

c) I understand that if I do not abide by these terms and conditions, I will suffer the

consequences as stated in this contract.

1)	Theng Wei Lwe	DATE	2/19/2023
2)	Wenqin Wu	DATE	2/19/2023
3)	Brennan Seymour	DATE	2/19/2023
4)	Branden Butler	DATE	2/19/2023
5)	Max Bromet	DATE	2/19/2023
6)	Edward Dao	DATE	2/19/2023