Delve

Final Report- CS 467 Online Capstone

Chara Group:

Jason Anderson	Sean Cortes	Joshua Nutt
anderja6@oregonstate.edu	*cortess@oregonstate.edu*	*nuttj@oregonstate.edu*

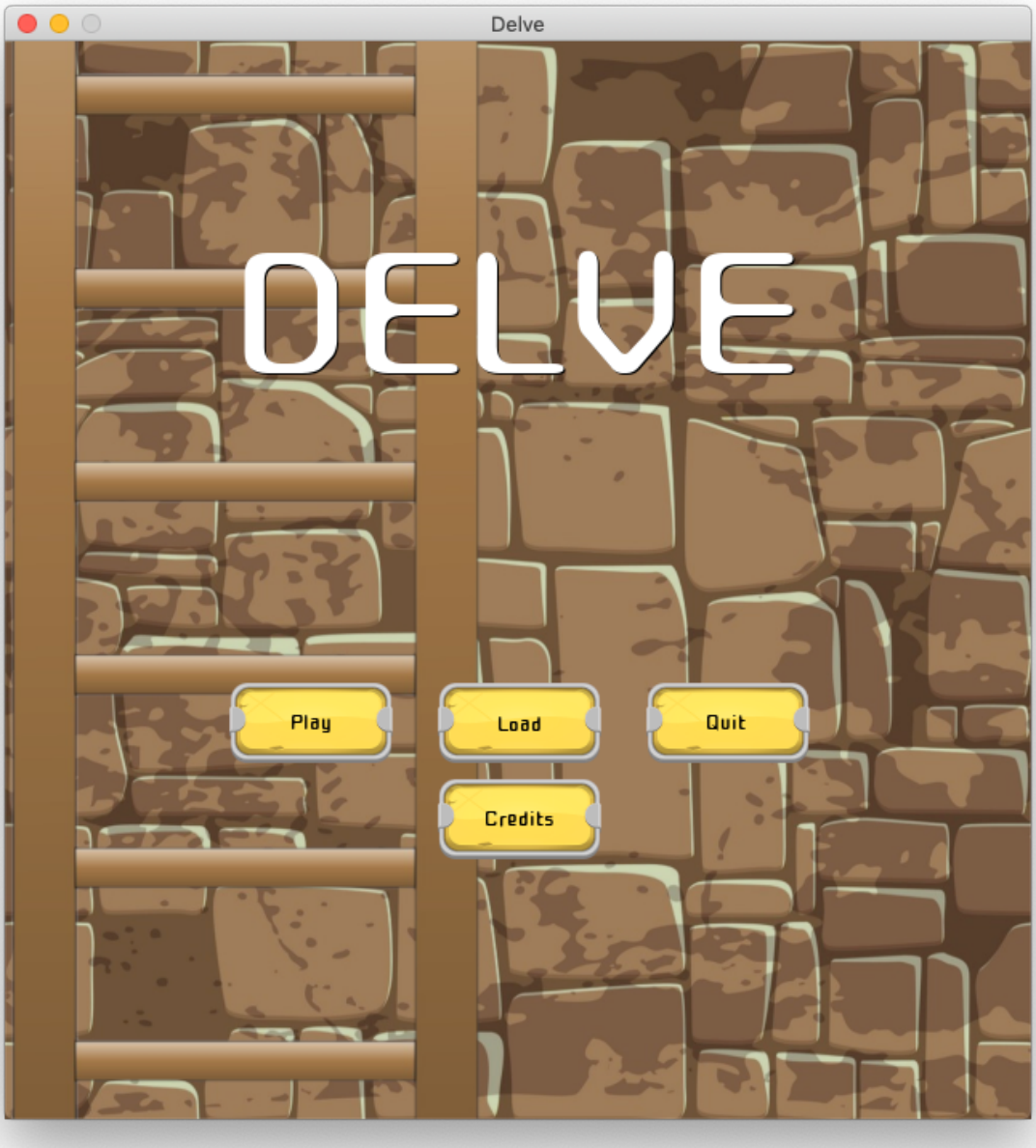

Figure 1: Main Menu Scene

I. Introduction

Our program is a 2D game called Delve. The user controls a player who must go through 10 levels of a dungeon to retrieve a chest of gold at the end. Levels are made up of a 20 by 20 grid of tiles. The objective for each level is to unlock an exit so the player may reach a ladder to the next level. An exit will be unlocked one of two ways depending on

the level: either by obtaining a key and using it to unlock the exit, or activating a switch to unlock the exit.

The player will encounter puzzles in the dungeon in the forms of blocks, switches, and doors. Blocks can serve as a puzzle, with the player having to push the blocks to navigate through them to reach a different part of the room. Switches are used to open doors and exits, but are only activated while the player or a block is on top of the switch. In addition, there are two types of enemies the player may encounter in the dungeon: bats and ghosts. The player can attack and kill these enemies, and these enemies can attack the player, with each attack draining one of the player's three hearts. Once the player is out of hearts, the player must restart the level from the beginning. Hearts always replenish when a player reaches a new level.

We began the project without any experience using pygame or developing games at all. We began by planning out the general style of our game, and developing a list of objects that we would need to build our game using an object-oriented paradigm. We then created a plan to develop the objects in phases, and assigned the objects for each player to create.

Creating a game requires a lot of assets besides the code files. To develop the game, we had to acquire and develop game art and sounds. Some of the game assets were taken from various repositories around the web, but most of our game tiles were created by one of our team members from scratch, which required learning to create pixel art. Other images were edited by the team to change color, or to create backgrounds.

Creating each level required creating the tile map that would serve as the environment for the stage, and then populating that map with the game objects the player can interact with. Originally, we drew the map and placed the objects by using a text files, with each tile represented by a character on a 20 by 20 grid. As the number of tiles and objects grew, we needed a more efficient way to draw the maps and place objects, so we decided to replace the text files with .tmx files generated from the Tile program. This made it easier to populate stages, as we could draw the maps and determine the placement of objects using the graphical editor, and the created some code loops to read the .tmx files and generate the game levels.

The initial game design had the first few stages being tutorial stages for the introduce the main objects and interactions of the game. To help the player, we created an object to print instructions to the screen to guide the player in the basic interactions of the game.

We developed a menu interface to allow the player to access different functions of the game. The players starts at the main menu screen, and can use the buttons on that screen to start or load again. The character can also pause the game during gameplay to bring up a pause menu which allows the player to restart the level, save their current

progress, or exit to the main menu. Since the game has multiple stages, we wanted the player to be able to save their current game so that they could return to it later. This required creating a method for handling files, and an method to encode the files so that they could not be easily edited by the player.

We also created a credits screen to show the develops and to attribute the assets we used that required accreditation, as well as a victory scene to display to the player after they have beaten the game.

II. Setup/Installation

There are two methods which you may use to run delve:

A. Using installer: **(PC ONLY)**

Pre-requisites: None

1. Navigate to 'dist' folder: `/dist`
2. Run 32-bit installer: `delve-1.0-win32.msi`
3. Navigate to installed path
4. Run main.exe

B. Running python script: **(PC or MAC):**

Pre-requisites:

Python 3.6.1+

Pygame

```
python -m pip install -U pygame --user
```

1. Run main.py:

```
python main.py
```

III. Game Instructions

In-Game Controls:

W A S D	Moves the player on the screen
Spacebar	attack and enemy or interact with an object
'+' or '-'	Increase or decrease the volume
P	Pause the game to bring up the Pause menu

The game begins from the Main Menu. From there the player can click one of four buttons:

- Play - Starts the game from the first level
- Load - Allows the player to load a previously saved game
- Quit - Exits the program
- Credits - Displays the credits

Pressing the P key on the keyboard brings the Pause menu. From here, the player can click one of four buttons:

- Back to Game - Unpauses the game and returns the player to the current level
- Save Game - takes the player to the Save Game screen to allow them to save the game. The program can save up to three games
- Restart Level - Allows the player to restart the level
- Main Menu - Quits the current game and returns the player to the main menu

IV. Assets

Assets used include the object and characters sprites, map tiles, menu buttons and backgrounds, and sounds. The majority of the tiles and sprites were created by the team.

Figure 2: Player Spritesheet

Figure 3: Bat (Enemy) Spritesheet

Figure 4: Ghost (Enemy) Spritesheet

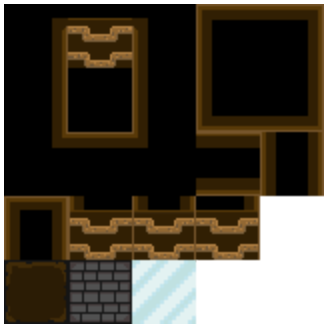

Figure 5: Cave Tileset Spritesheet

The ice tiles, treasure chest, menu backgrounds, and buttons were built from online assets. The sounds were acquired online and were edited for volume and length, and some files were converted to a Pygame friendly format.

Besides art and sounds, other files used are the .tmx files generated by the Tiled editor, and the .sav files which are encrypted save game files created by the program to store the player's current progress.

Licensing: The assets listed below required attribution according to their licenses and are included in the Credits scene of the game:

- Game over screen background vector created by kjpargeterat www.freepik.com
- Treasure chest sprite by goo30 at www.opengameart.org licensed under CC0 1.0
- Free fantasy GUI by pzUH at www.opengameart.org licensed under CC0 1.0

For the audio implementation, the following sounds were used:

- Click2 sound created by Sebastian at www.soundbible.com licensed under CC BY 3.0

These sound volume levels were altered and any extra silence at the beginning and end of the files were trimmed.

- Rock Scrape 2.wav by Benboncan | License: Attribution | https://freesound.org/people/Benboncan/sounds/74441/

These sounds were used under creative commons 0, and have had their levels changed and silence trimmed:

- Game sound by chris_schum | License: Creative Commons 0 | https://freesound.org/people/chris_schum/sounds/418149/
- Retro Game sfx_jump bump.wav by mikala_oidua | License: Creative Commons 0 | https://freesound.org/people/mikala_oidua/sounds/365672/
- metal sound, fighting game by evilus | License: Creative Commons 0 | https://freesound.org/people/evilus/sounds/203454/

This sound has also had some parts removed that were clipping and popping in order to make for a smoother listening experience.

- Game Over Sound by TheZero | License: Creative Commons 0 | https://freesound.org/people/TheZero/sounds/368367/

V. Programming Languages, Structure, Libraries and Tools

Language and Code Structure

Python/Pygame: The program was primarily developed in Python using Pygame. The main structure of the game uses a handler to call scenes, which can be either game levels or menus. This is handled by the Game object in the main.py file, which calls the MainMenuScene upon launch. From there, the player can start the game from the first level or load a previously saved game. The Game object also includes the run method, which serves as the main game loop. This game loop calls a series of functions on the currently loaded scene:

1. **clock.tick()**: This function handles how fast the game updates. It can be controlled by passing a frames-per-second argument to it, which will only update the game a number of time each second equal to the frames-per second
2. **handle_events**: this function gets the input from the mouse and keyboard. It exits the game if the player clicks the X at the top right of the screen. It also detects which key the player presses, and calls the appropriate functions that are mapped to each key. For the menu scene, it also handles interactions between the buttons and the mouse.
3. **update()**: This function updates the game state. This is where the enemies on the screen move, animations are updated, and collisions are detected
4. **render()**: This is the function that draws the map and game objects to the screen

Other python files in the program include:
Scenes.py: This files includes the classes for each level of the game, and contains the handle_events, update, and render functions describe above. Each scene is its own subclass that inherits from the Scene class. The constructor for the class loads the data from the map files to generate the tile map, and places the player and other game objects on the map.

Map.py: This files contains the Tilemap class, which loads the data from the .tmx files created in Tiles to generate the game map.
Menu.py: This file contains the menu classes, as well as helper classes like the Background and Buttons class. The MainMenuScene serves as the base class, and the other menus include the Pause, Game Over, Save Game, and Load Game menus. It also contains the Credit and Victory scenes, which are not actual menus, but share enough attributes with the menu class that they were created as subclasses of the

MainMenuScene. This file also contains methods for encrypting and decrypting the save game files.

Settings.py: This file contains global variables that are used throughout our program. This includes tuples for basic colors and compass orientations, the size of the screen, frames per second, and the file and folder locations for game assets.

Sprites.py: This file includes the classes for the sprite objects, which are the main objects in the game, including the player, block, switches and doors. It includes the necessary update render functions for these objects, which are used to update animations, test collisions, and draw the objects to the screen

Enemy.py: This contains the classes for the enemy sprites. It includes methods to move the enemy along a predetermined pattern that is set in the Scene subclasses, as well as methods to handle attacks between the player sprite and enemies.

Helper.py: This contains methods to draw text to the screen, which is used by the menu class and by the tutorial levels when instructions are drawn to the screen. It also includes methods for updating the sprite animations, which are used by other classes in the sprites,py and enemy.py files.

Item.py: This file includes the inventory class, which is used to store the keys the player can acquire and use in the game.

Setup.py. This file is used the the cx_Freeze program to create the game distributable.

File Structure

The following subfolders of the program hold the game assets:

Image: Holds the art assets that are used for the sprites, tiles, menu backgrounds, and buttons

Maps: Holds the .tmx and .tsx files that are generated by the Tiled editor and integrated into the game using the pytmx plug-in.

Music: Holds the audio assets used by the game

Save: This contains the .sav files, which are generated by the program to store the player's game progress

Libraries/Tools

Tiled: Tiled is free third-party software that allows for easy creation of tiled maps. The original design of the game used a series of text files to map tiles to specific characters. As more tiles and layers to the map were added, this became unwieldy, so the Scene objects were changed to work with the .tmx files generated by Tiled using the pytmx library. Tiled allows multiple layers to be added to a map, as well as show where objects will be placed so that they can be generated by the game. Certain stages, such as the stages that contain ice, required the placement of stones that served as rocks that the player could collide against to stop them from sliding on the ice. This required drawing the ice on one level, drawing the stone on another level, placing the invisible ice the player slides across on a third level, and then placing the walls and game objects on a fourth level. This required four text files in the previous method, but Tiled allowed all four layers to be created in a single file using a graphical interface.

CxFreeze: Set of scripts/modules for freezing Python scripts into executables or creating MSI installer packages. Allows for a user to run the game without needing to have to have Python or any additional libraries to be installed onto their system.

Git/GitHub: Utilized for version control. Each week we created a new branch while merging the previous week's branch.

Piskel: Piskel is a free sprite editor. All sprites and their sheets which were not attributed to external resources were created in-house using this editor. This program supports a simple layer, import and export system.

Gimp: Gimp is an open-source image manipulation program that was used to edit some of the assets for the game. The graphical user interface used on the menus were created from assets found online. The buttons were edited in Gimp to change the color, and the background images on the menu were created by layering separate images on top of each other. It was also used to edit some of the sprites in the game by posting multiple animations on a single file.

Audacity: Audacity is a free, easy-to-use, multi-track audio editor and recorder. This was used to edit all of the audio files used in Delve.

VI. Team Member Contributions

Jason Anderson:
Jason was responsible for developing the player controls in the handle events section of the main game loop. He implemented the block object and wrote the code for the block's interaction with the player, switches, and doors. He wrote some unit tests for the block object. He designed implemented and debugged a tutorial level for boxes. The game has several early levels to help the player understand how to interact with different concepts in the game. Jason then created level 9 for the game which combines multiple concepts to create a challenging level requiring multiple objectives to complete. For example, the player has to navigate through enemies and ice to find the correct switch to open the door to access a key to complete the level. He also implemented sounds in the game, which required sourcing the audio files, converting them to pygame friendly formats, and editing them for volume, length and clarity. He also implemented the volume controls to allow the player to adjust the volume of sounds and music in the game. Jason was also responsible for the poster and part of the final write up.

Sean Cortes:
Sean was responsible for creating the overall boilerplate, implementing various objects (player, enemy, item, inventory), creating pixel art, and implementing object animations. The creation of the game's boilerplate started through heavy inspiration from pygame tutorials found online. The boilerplate quickly shifted to a more customized codebase as we began adding more features to our game. He implemented the player and enemy objects and all necessary parameters and class functions such as an attack/interaction button for the player, movement logic, and a naive movement algorithm for enemies. Interactions and animations between the two objects were also created such as the player and enemies attacking each other. Sean also implemented a generalized methodology for reading in a spritesheet and animating each 'living' object to have an idle walking, attack, and damage animations. In order to meet the deliverable requirement, Sean researched cx_Freeze and integrated its functionality into the program. Finally, Sean helped with general debugging tasks and helped distributing tasks during our weekly meetings.

Joshua Nutt:
Joshua was responsible for developing the various menus in the game. This required acquiring art for the GUI's and creating the button class to call the methods assigned to the button after it was clicked by the user. He always set-up the file handling and encryption for the game save files so the player could save their progress. He researched methods for fading text and background onto the screen for the credits and victory screens. Joshua was also responsible for creating the tile objects and worked on

integrating the .tmx files into the program. He also developed the ice tile objects, which required altering the player sprite and box movements so that they would slide across the ice until colliding with an object. He also debugged and provided extensive user testing for the ice tiles. He also created four of the levels and sourced and edited the sprite art for the ice levels.

VII. Conclusion

Our project did not have any major deviations from the original project plan. Tasks were re-assigned to group members based on schedule, and we used more tools than were originally planned, but we feel we did a good job of mapping out the necessary structure of the game and the required in-game objects during the planning phases of the project. Our approach to the code structure was object-oriented in nature, even though Python does not directly support OOP. We planned out base objects and tried to maximize our utilization of inheritance. This is most apparent by our use of GameObject and having all major objects inheriting from that class (e.g. Player, Enemy, Block).

We did make some changes to the program as we went along. Our original methodology for generating the map was by transcribing a textual representation of our map. We collectively deemed this methodology inefficient, specifically since it limited our scalability for integrating new obstacles while keeping object interactions consistent. Quick research pointed towards Tiled as the solution to our problem. The Tiled editor saved time and made it easier to generate the levels and puzzles.

As development progressed, we also decided to implement greater complexity to the interactions between the objects in our game. Originally, switches only worked to open the exits, but were modified to work with other doors in the game to add an additional layer of interactivity.

We discovered that developing a game requires extensive work outside of developing the gameply code. We had to spend time sourcing and editing assets, and creating our own, which required finding and learning new software to develop and manipulate those assets. We also spent time creating the menus and file handling to give structure to the program, which took additional hours on top of developing the main gameplay

Overall, we enjoyed making this game and working together to distribute the workload. While it was a time consuming project building the game and working on the assets, we enjoyed being able to implement a program in a language we had not implemented large projects with.