Master's Project Report

Hunter McGarity

Prepared For:

Dr. Mustakimur Khandaker

CSCI 7200 – Master's Project

University of Georgia

Athens, GA

August 2, 2022

Project Summary

As my Master's Project for CSCI 7200, I chose to build a two-dimensional video game. The game itself is written in the Rust programming language and falls within the category of "dungeon crawler" games, wherein the player must navigate a series of rooms and corridors in search of items and/or an exit. This genre of video game is not novel, but has been around for decades, with perhaps the most well-known title being *Rogue*, hence the other designation for the genre: "Roguelike". The project adapts to modern game design methodologies, including Entity Component System design and data-driven design. The main idea for the game, as well as some of its core mechanics, comes from the book *Hands-on Rust*, by Herbert Wolverson. This book walks the reader through not only the basics of the Rust programming language, but also the nature of ECS design and game deployment via web assembly. In preparation for the project, I worked through the book and built the game described therein according to the author's instructions. For the Master's Project, I decided to take this game I had built using Wolverson's book as a springboard of sorts and focus on heavily expanding on the original work to truly make it my own. Such expansion is even suggested by Wolverson in the book's final pages. As such, while my work relies in part on Wolverson's as foundation, I feel that my changes and additions expand significantly on the game enough to make it another entity entirely. I chose this project path not only out of immense interest in Rust game design, but also so that I could learn and apply such concepts within the brief summer semester.

Project Motivation

The motivation for this project was not sudden. Ever since my teenage years, I have had an interest in game design. This interest was not in game design as a career, but rather as a potential hobby and an outlet to express my creativity. Having played many games in my

younger years, I felt that when I began studying computer science and programming it would be the perfect opportunity to blend both and create a “passion project” of sorts. However, I never found the appropriate time or motivation to undertake such a project during my undergraduate career and so the Master’s Project course finally provided me the perfect opportunity for this pursuit.

The decision to write the game in Rust, however, was rather sudden. In Spring 2022, I took CSCI 8245 – Secure Programming, where the focus was on the Rust language and how it can be used to eliminate software vulnerabilities that have plagued C/C++ applications for decades. Since I have an interest in Cybersecurity as well as Software Development as potential careers, I naturally wanted to learn as much as I could about the Rust language. This, coupled with increasing adoption/recognition of Rust by major companies in the field, was all the motivation I needed to write the project in Rust and hopefully further my knowledge of the language.

A further insight is that the game’s narrative elements, presented in the intro/outro screens, is inspired by a short story that I composed for a class my freshman year at the University of Georgia. The story itself was well-received by my professor and nearly published in a campus magazine, and so I wanted to expand on the lore of that fictional universe but was not sure how for a long time. Again, this project presented a good opportunity for a long-time goal.

Project Goal

The overall goal of the project was to demonstrate my knowledge of the Rust language, while also gaining knowledge of modern game design methodologies and pursuing a long-time

passion project. To the first end, I became more familiar with various aspects of the Rust language that truly set it apart from non-secure programming languages. These include borrowing and ownership, mutability, macros, multi-threaded environments, and others. All of these aspects of the language are used to varying extents in my final project. Having finished the project, I now feel much more confident in my abilities to design and implement software in the Rust language and to understand secure programming concepts.

In terms of game design, the game relies on the widely popular model of Entity Component System design. I had heard this term before in passing mention, but only started delving into the specifics of it in preparation for this project. Through exposure and adaptation to this design philosophy, I learned what it takes to create a video game in the modern game industry, as well as the importance of several design principles like obfuscation, code re-use, and code-readability. I also learned that the ECS model, once fully implemented, allows for quick addition/modification of existing game features, rather than hours of careful work to implement something new. Again, I do not plan on game design as a career for myself, but I do feel more confident in my game design abilities thanks in large part to learning ECS through this project.

The final goal was to simply enjoy the project development cycle by making a long-time passion project. Years ago, I had created video game experiences for distribution online through games that were packaged with proprietary content editors. This made creation of user-generated content easy, and it also sparked my interest in game design. In my time as an undergraduate computer science student, I had the opportunity to produce interactive software and the occasional basic game but nothing that quite satisfied my drive for more. Through this project, I was finally able to create a complex but entertaining game that I consider a good demonstration

of my knowledge of not only programming but also game design. As such, the project stimulated my knowledge as well as my creativity.

Process & Technical Details

As previously mentioned, the project's development began with working through the *Hands-on Rust* book by Herbert Wolverson. The process of the book, and of my own learning through the summer, began with explaining the importance of a design document and separating development goals from stretch goals when starting development. I developed my own design document for the project, and frequently referenced it throughout the project's development. As for the technical details, the book began with an introduction to two-dimensional graphics rendering in Rust through the author's own **bracket-lib** Rust crate. The bracket-lib crate has found popular use among Rust game developers primarily for dungeon crawler games such as this project. Despite the other two-dimensional rendering crates available, like **macroquad** or **amethyst**, I chose to adhere to the bracket-lib crate for ease of transition from the author's work into my own work. The crate contains a number of useful modules and structs, such as the Point struct which is used in position-related operations in the project, as well as the RandomNumberGenerator, which is instrumental in the procedural-generation of the project. The book also gives an introduction and overview of the Entity Component System model, and the project itself relies on the **Legion** Rust crate for this model. Other ECS crates have recently surpassed Legion in terms of developer support, attention, and features, but I again chose to adhere to Legion for the ease of transitioning from the author's work into my own.

As outlined by Wolverson, the ECS architecture is an increasingly popular approach to handling the sheer volume of game data, as it is finding more and more use in popular game engines such as Unity. An *entity* in the ECS architecture can represent anything from a player to

an enemy or even an item. However, entities lack associated logic, instead being essentially an identification number like an integer. A *component* represents a property that an entity can possess. Entities can have a varying number of components associated with them, and each of these components if given functionality via *systems*, which query the entities and components. Finally, a *resource* is shared data that is made available to multiple systems (Wolverson, pp. 103-104).

From this point, each chapter in the book walks the reader through a new aspect of game design, all compounding to create a basic dungeon crawler game. Through this process, I learned about implementing procedural map generation, creating distinct dungeon enemies, implementing turn-based combat, and giving the player an inventory of items, all of which relied heavily on the introduction or modification of entities and components.

Once I had finished working through the book, I began to make my modifications for the project. One of the first features I wanted to pursue was giving the player attack armor that could absorb damage from enemies until it was broken, thus safeguarding the player's health. For this feature, I needed to introduce a new component struct into the project that would contain the player's current armor level as well as their maximum armor level. I then had to inject this new "player resource" into the system responsible for spawning the player in each level. This is an overview of the process for implementing new components, of which there are quite a few in the project. Once armor was implemented and visible to the player's heads up display, I had to make significant modifications to the game's combat system to ensure that armor was decreased when the player received damage, and that the player's health was protected by the armor. To finish out the attack armor feature, I introduced new items for the player to find that, when used, would replenish a portion of the player's armor. These items required another new component struct to

provide armor to the player. Finally, I had to tweak the spawning schema of healing items and armor items to emphasize replenishing armor over replenishing health, which I felt gave the game a more tactical approach to combat and item scavenging.

For other mechanics, such as poison-resistance and increasing the player's field of view, I followed a similar process to that of the attack armor. That is, I introduced new components and items, while making modifications to appropriate systems to ensure proper functionality. A great desire of mine was to alter the game visually by introducing new themes or tilesets. To this end, I introduced two new map themes for the more difficult levels of the game, each with its own sprites for walls, floors, and the environment. I also wanted to introduce new enemies and more complex enemies, which meant finding new sprites for them firstly. I decided to use the **GIMP** bitmap editor to introduce new sprites into the game. Specifically, I would locate a new sprite I wanted to use on an attributed spritesheet, then use GIMP to crop out the sprite and add it into the project's spritesheet, which in turn was read by the bracket-lib crate to render the sprite in the game world. This did present one notable issue: certain color patterns, such as bright green and brown, appear to render incorrectly through bracket-lib. The issue appears to be unaddressed by the crate's author, and it is unlikely that it will be fixed in the near future. This caused me to enter a process where I would add new sprites only to find out that they did not render correctly in the game. Thus, the process of introducing new artwork into the game was long and arduous, but still rewarding. Through this process, I introduced new sprites for new enemy types, for floor and wall textures in the new themes, as well as for decorative tiles in all themes.

To address enemy complexity, the player is encouraged to use their mouse to hover over enemies and items to see a tooltip, which will reveal useful information on enemies such as enemy name, enemy health, and damage the enemy is capable of dealing to the player. However,

this was not enough to satisfy the goal of enemy complexity. To this end, I decided to implement special enemy types, each of which would have a special characteristic the player would need to consider before engaging them. Examples of these special characteristics are stealing score from the player, knowing the player's location from anywhere on the map and pursuing accordingly, ignoring player armor when attacking, and decreasing player field of view when attacking. Each of these required new components and injection into the spawning system, as well as special instructions for combat. Some of these special enemy types proved too difficult to deal with when allowed to spawn on every level, so I also decided to relegate them to certain levels and themes. It was at this point I decided to limit the game to four levels, one for each theme, with each level having a special item or enemy type only found therein. This required modifying the game's theme selection code to be sequential rather than random. This also presented another development challenge: I wanted to introduce a new component to the player that would keep track of the player's level in the game to spawn the next theme accordingly, but the spawning system failed to satisfy a very confusing trait bound in the Legion crate, so I could not add any more components to the player than were already present. As a work-around, I added a new data member to an existing Map object resource to track the current theme. This approach worked flawlessly for the intended purpose.

There are a number of other complex features that were added to the game, such as locked door tiles that the player can only traverse when they are carrying a key item, as well as poison floor tiles that damage the player unless they have used an anti-poison item, or even an item that when used extends the player's field of view, allowing them to see more of the environment from a distance. Each of these features, and others like them, needed their own set of new components and modifications to existing systems or even new systems entirely. For each

of these, my general procedure was to identify which existing systems or components could be modified to create the new functionality, and then to make said modifications while also making additions to supplement them.

The project relies heavily on a concept known as **data-driven design**, which loads as much data as possible from external files and uses said data to populate the game's entities. An ECS architecture is data-oriented since one of its main focuses is on storing data in memory and then providing efficient access to it (Wolverson, p. 269). For this project, all information about spawned entities, including all enemies and items, is stored in a .ron file, which itself is deserialized with the **serde** crate in Rust. For every new spawned entity that was added to the game, I had to find a way to structure that entity's data in the .ron file and incorporate that data into the game's systems for spawning into the game world. The biggest challenge here was from the special enemy types, since they required dedicated components to be attached to them when spawning so that they could be tagged as capable of their special functions in combat.

Challenges & Limitations

The most difficult feature to implement was game audio. For this I used the **rodio** crate, as it is reputable for basic game audio implementation. To be specific, I wanted to implement ambient music for each of themes to further differentiate them from one another while also introducing sound effects for various actions, such as items being picked up or enemies being slain. Although the rodio crate documentation states that each sound is played in a newly spawned thread, I found that the sounds would not play unless their parent thread was paused to wait for the sound to finish, which in a video game is unacceptable as it would mean pausing the entire game just for a sound to play.

I tried a number of approaches for the game music, but none seemed to work. I simply could not figure out a way to play the music during a level, then stop the music track and start another when the player progressed to another level. Because of this, and the fast-approaching end of the semester, I decided to cut the game music feature entirely and focus solely on sound effects. I implemented a number of effects for everything from picking up a sword to drinking a potion, including unique death sounds for each type of enemy. The way I implemented this was having a dedicated function to spawn a thread, which would in turn play the sound. Because the rodio crate spawns a new thread anyway while playing a sound, this means the first spawned thread is a "buffer" thread whose only purpose is to spawn another. This was necessary because I could put this intermediate thread to sleep for the sound to play without pausing the main thread where the gameplay is taking place. I acknowledge the inefficiency of this approach since it essentially wastes system resources by creating a thread whose only purpose is to create another, but in the context of sound effects, each of which is three seconds or less, I believe the effect is negligible because all sound threads and intermediate threads terminate quickly before there is a chance for multiple ones to build up or cause much performance overhead. Perhaps in future work on this project I can find a more efficient way to do this.

Another challenge I encountered was related to the spawning of the player as well as items within prefab structures. Prefab structures are hard-coded level geometry that can appear in certain areas of the game world. I expanded on this system as well by adding more varied prefab structures and enabling the spawning of different prefabs across the levels, rather than a single prefab being spawned on every level. Nevertheless, I found that sometimes multiple items would spawn atop one another within a prefab but be rendered as a single item. This proved a problem since I limited the player's inventory to four items, because the player could pick up one of these

stacked items and accidentally receive up to three or four, potentially breaking the inventory system. I managed to solve this issue by logging all prefab item spawn coordinates in a vector, then checking that vector's contents against all new items being spawned at level generation. If the item wanted to spawn at a location where one had already been spawned, permission was denied.

I also encountered the major issue of the player occasionally spawning inside a prefab structure. This was a problem because all prefabs require a key item to enter or exit. If the player spawns inside a prefab and has no key item, they are trapped, and the game is broken. I managed to address this by altering the prefab spawning mechanism to ensure that the player's chosen spawn location is not within the proposed coordinates of the prefab, and if it is then a new set of coordinates are chosen to try and spawn the prefab. This raised the secondary issue of a player being locked into a prefab. The game allows a player to discard items from their inventory, which by nature means that a player can discard a key while inside a prefab. This would normally result in the player being locked inside a prefab, but I found that by adding a Boolean value to the Map system resource, I could keep track of whether a player was carrying a key or not. Using this information, I could log all coordinate points that lie within a prefab structure and then allow the key carry Boolean to be set to false if a player discards the key while outside of a prefab. Otherwise, the player discards a key inside a prefab but the key carry bool, which when set to true allows the player to pass through a door, remains set to true until the player exits the prefab, at which point it is set to false. This was possible through an entirely new system file created to handle the key-related actions.

One final significant challenge was in implementing a feature that was not given in my design document nor in the project's README.md file. I had earlier in the semester

implemented a scoring mechanism, which was absent from the foundation code, where the player could accumulate score by picking up certain items and slaying enemies. However, this score did not carry any meaning since it was not recorded anywhere and was lost on victory or game over. I decided to change this by logging the player's scores in an external .txt file. When the player is victorious or defeated, their final score is displayed on the outro screen along with whether or not this is a new high score, checked against the previous scores logged in the file. Since this score-checking and notification only takes place at the end of the game, I decided to implement this feature as an expansion to the system that already handled the ending of the game, because implementing it in another system could lead to the process happening on every game tick/execution cycle, which would be woefully inefficient since the process involves file i/o.

The system activates on victory or defeat and works by reading in every line of the .txt file and filtering each line's characters to only include numeric characters. Then, each of these filtered strings is parsed for a numeric value and placed into a vector. I find the maximum element in the vector and then compare it to the player's score from this play of the game. If the player's score is greater than the vector's max element, then the player is notified of a new high score, otherwise they are not. Regardless, the player's new score is logged into the file so long as it is not a duplicate of a pre-existing score. The main challenge here was file manipulation and the problem of dealing with non-numeric characters in the file, which is partially addressed through the character filtering, but ultimately this relies on the assumption that the player will not manually open the .txt file and input scores they did not actually achieve.

Conclusion

This project was a monumental exercise in the learning and application of modern game design principles and, as an effect of using the Rust programming language, of secure programming paradigms. Through the project's development I gained and applied knowledge of Entity Component System design as well as data-oriented design, with an emphasis on code readability and code reuse. I also grew more familiar with the Rust programming language and the unique features that make it a secure programming language: features such as the ownership model, mutability, and multi-threading mechanisms. Thus, I feel that the project gave me critical insight into game development but also furthered my knowledge of secure software design, itself a highly valuable skill in the industry. Beyond the technical learning experience of the project, this was also a great opportunity to stimulate and express my creativity through the art of game design. While not the most challenging aspect of the project, one of the most enjoyable was the process of designing the new environments, mechanics, and enemy types, particularly the special enemy types which carry special attributes and require planning when the player encounters one. This was an outlet for creativity unlike most of my other projects developed in my time at the University of Georgia.

Through the design, implementation, and troubleshooting of new features and mechanics in the dungeon crawler game, I feel that my knowledge of key computer science concepts has been strengthened. The project was not without its challenges, but it is the overcoming of these challenges that was the most rewarding development. In its current state, the project does have certain limitations, but perhaps future work can adequately address these shortcomings, while also exploring the transition from turn-based gameplay to real-time as well as porting the game to a more popular Rust game engine.

References

Hands-on Rust. Wolverson, Herbert. 2021. The Pragmetic Programmers, LLC. Print.