SRE Final Project - Balatro

Luke Severs, Jacob Sharp, Camp Steiner

Goals

Our overall intention for this project was to create a mod for the hit roguelike deck-builder ‘Balatro’. This was then broken down into the following goals:

- Reverse Engineer Balatro
- Understand how Jokers, Vouchers, Hand types, and Challenges in Balatro are represented and create our own
- Understand how Balatro calculates the score of a hand and make a calculator program for both your current hand and the optimal score
- Have fun :)

Report

The reverse engineering portion portion of the project was both easier and more difficult than we planned. Balatro is written using the LÖVE2D game engine, which is a beginner-friendly Lua-based framework. As a result, the distributed Balatro.exe is simply a wrapper around the Lua runtime and the LÖVE framework. When reverse engineering the executable, we found that it did little other than setup the Lua runtime then load and execute the Lua source files, which were thankfully extremely easy to locate.

We discussed and attempted several methods of writing our mod. The most interesting and perhaps “proper” method would have been to intercept the Lua DLL calls to force the Lua runtime to recognize our own Lua files, but this proved not feasible due to lack of knowledge about DLL injection/rerouting and lack of time to learn it. Instead, we opted for a slightly unusual method of mod injection that abuses the way LÖVE packages Lua files. The executable is technically a self-extracting archive that bundles the source code along with the runtime, so we were able to temporarily extract, edit, and repack that archive of source code in order to make our own modifications to the game.

Using this injection method, we wrote a series of patches and new Lua scripts that added a new joker, challenge, hand, and voucher to the game. These additions act in a similar way to the previous cards such as adding mult or changing hand size.

For the cards, the jokers, vouchers, planets, and more types of “blueprints” are stored in game.lua as a permanent representation of each card’s attributes. Card.lua pulls that information when creating card objects, and assigns a type based on what was called for creation. When created, Card.lua assigns the information and calls to the localization packs for text to be displayed in game. Card information can be accessed from a “base”

in the Card class that contains info like “value” (a string representation of rank) and “suit” (a string representation of suit) as well as other info like id (which is numbers 2-14 for 2-Ace respectively). Modifying this base allowed us to set all of the cards in hand to a King of Spades as demonstration of how easily manipulated cards are.

The scoring in-game is handled by a function called evaluate_play() in state_events.lua, but this function also performs many UI changes and affects things like joker level and dollar state. Instead of being able to hook this directly, we had to instead copy the function and remove any code that wasn’t purely for the purposes of getting the resulting score of the hand. We also were able to remove a lot of redundant code in the process since it wasn’t especially well written to begin with. What we ended up with still calls other functions like get_poker_hand_info and evaluate_joker to calculate the score, but stringing those functions together had to be done manually.

To calculate the optimal score, we simply loop through all possible hands with a series of nested for loops and calculate the score for each one. The highest score will be deemed the optimal hand, and if there is a tie it will prioritize the hand with the fewest cards since any extra cards weren’t contributing to the score and don’t need to be played.

Retrospective

The card.lua and game.lua interactions sound simple but they took some time to be uncovered. The developer didn’t create the most unified way of sharing data across classes. Some data or logic was somewhat hidden away, such as hand determination logic in the misc_functions.lua file. The developer did not use many central handlers, and as such some parts of the game remain harder to mod than others. I looked at making a boss blind active across all blinds in a challenge, but due to the setup of the program, there was not an easy way to instantiate it.

This game does not follow a software architecture pattern that allows for easy refactoring. Additional cards and hands of a similar type can easily be added, but changing any larger game functionality requires a major overhaul of how the code is structured. With more time, we would have liked to explore other ways to inject our modded code as well as other card interactions, such as blinds or adding new joker types. There was also not a super clean way to hook the scoring function since it directly modifies the game state, but the way the score function was copied and modified probably isn’t ideal since it could become inaccurate with updates. However, we were happy with what we were able to accomplish given the limitations of the existing code, and being able to inject added content and an optimal score counter met all of our original goals.