**Overview:**

Highcarder is a casual/semi-competitive card game where the main premise is to survive as many “antes” as possible while gathering the highest score possible by playing poker hands from an ever-dwindling deck of cards. The aim of this web application is to develop a simple, yet addicting gameplay loop that would keep the player engaged and aim to reach for the highest possible score and level. The game is inspired by other card games and deck builder roguelikes like Texas Hold ‘em Poker, *Uno (YEAR), Persona 5 Royal’s (20XX)* Tycoon Minigame, Slay the Spire (YEAR), and mainly the recent *Balatro (2024).*

**Usability and User Experience.**

My aim with Highcarder was to create a game that is simple to pick up yet offers strategic depth. I took inspirations from focused gameplay looks, settling on a deck-builder card game with poker-like elements after being influenced by Balatro, which was coded entirely in Lua using the Love2D framework, and without a game engine. To create a similar experience, I used JavaScript and the p5.js library, which provided similar flexibility but with the added benefit of object-oriented features such as classes.

Gameplay is built around the following loop:

* Start with a 52-card deck and 7 cards in hand.
* Play a poker hand to progress each round; 5 rounds form an ante.
* Score requirements increase each ante and must be met to progress.
* After each ante, choose upgrades: Booster Packs (add cards), Refinements (edit cards), Perks (meta/resource upgrades), or Abilities (passive effects).
* Additional mechanics include burning, freezing, or skipping upgrades, storing upgrade points, and difficulty increases via added debuffs every 5 antes.
* The game ends when the player runs out of cards; there is no win condition, encouraging high-score chasing.

The upgrade system, combined with the familiarity of poker, creates a satisfying balance of freedom and strategy. A variety of upgrades supports different playstyles, encouraging experimentation. The absence of a win condition drives replayability, while the leaderboard—stored via localStorage—gives players a sense of progress and self-improvement.

A screenshot of a game

AI-generated content may be incorrect.

A screenshot of a game

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Gameplay of Highcarder + Leaderboard.

Highcarder uses a single input method—mouse or touch—making it fully playable on desktops and partially functional on mobile devices. The game is optimised for desktop devices due to a minimum canvas size of 1100x600 pixels. While the canvas stretches across the webpage and automatically resizes to fit the browser window, reducing the size below the minimum causes important visual elements to overflow beyond the visible area.

The game also displays active abilities on the left side of the screen and active debuffs on the right, using HTML elements positioned outside/above (via z-index) the p5.js canvas so they remain fixed to the viewport and always visible. Both panels also feature accordion drop-down functionality, with either the arrow icon or title clickable to expand to collapse them. The titles will always be visible, along with a count of how many abilities or debuffs they have. This allows players to hide the panels if they obscure gameplay elements, particularly on smaller screens while they keep key status information accessible. This approach improves usability by ensuring important modifiers are always visible yet easily adjustable to suit player preferences.

To resolve this, the HTML enforces a minimum window size of 1100x600 pixels. This ensures the game is always displayed within its intended layout, keeping all interactive elements visible and preventing cramped or partially hidden UI elements. This will, however, force the user to scroll horizontally or vertically to view the full game, which can disrupt the intended seamless experience. Though unfavourable, this method does ensure functionality on smaller screens and on mobile devices.

Because the game is built entirely within a p5.js canvas, it does not support full keyboard navigation (e.g. arrow keys or Enter) for gameplay. The keyboard is only currently used for entering a player when submitting score to the leaderboard. Adding full keyboard controls was considered, however this would have expanded the scope of an already ambitious project and required more development time than was available.

Players can quickly learn the game through the “HOW TO PLAY” tab, which is always accessible at the top of the page. This section clearly explains every major feature, including how the upgrade system works, the full list of playable poker hands (excluding secret hands), how scoring is calculated, and the rarity rates for upgrades. By providing this information in one easily accessible location, players can quickly grasp the rules and mechanics, then experiment with different strategies, explore upgrade synergies, and even test the limits of how far they can “break” the game.

A screenshot of a computer

AI-generated content may be incorrect.

“How to Play” tab.

**Technical Implementation**

The code of Highcarder is designed to work as a connected system rather than isolated modules. Most functions interact through the global variables initialized at the start of the game (defined in globals.js), which hold core game data such as the deck, hand score, upgrades, and game state. These variables act as the main engine towards the game, allowing different parts of the code to access and update the same information.

Due to the size of the application’s codebase (over 5,000 lines), this section will focus on the more relevant, significant, and technically complex parts of the implementation. Not every function or file will be discussed in detail.

A screenshot of a computer program

AI-generated content may be incorrect.  
A portion of Highcarder’s global variables.

**Cards:**

Cards in Highcarder are represented by a dedicated Card class, which stores its “rank”, “suit”, selection state, position, and dimensions. The class uses the “suitMap” and “rankMap” objects (defined in card.js) to convert each rank and suit into numerical indices, allowing the cards to be drawn from a sprite sheet efficiently within the “draw(x, y)”. This method is what mainly renders the card, highlighting it gold if selected. “contains(mx, my)” checks if the mouse is over the card for player interaction.

A screen shot of a computer program

AI-generated content may be incorrect.

Cards are then stored and managed through three main arrays:

* “deck” – contains all cards currently available to draw. It is filled with a standard 52-card set (via the “generateDeck()” function) and is reduced as cards are drawn into the hand.
* “hand” – holds the cards available for the player to play in a round. Cards are drawn into the hand via the “drawHand()” function. Cards are shown and rendered visually via the “drawHandUI()” function.
* “selected” – tracks which cards in the hand have been selected by the player for play.

Various functions interact with these arrays to retrieve, update, and store game data:

* The “drawHandUI” function handles rendering the cards for visual display, calculates the horizontal spacing to centre cards and updates the position based on game state.
* Card interaction is handled through via p5.js mouse functions with added “drag-and-drop” functionality to allow players to order cards: “mousePressed()” sets the held card, “mouseDragged()” enables the drag-and-drop reordering and handles its functionality, “mouseReleased()” updates both the “hand” and “selected” arrays accordingly to reflect the reordering of cards.
* The “playHand()” function handles how cards and played hands are scored, interacting with “evaluateHand()” to check which hand the player has played before returning the appropriate information (hand type, name, and score).

**Hand Evaluation**

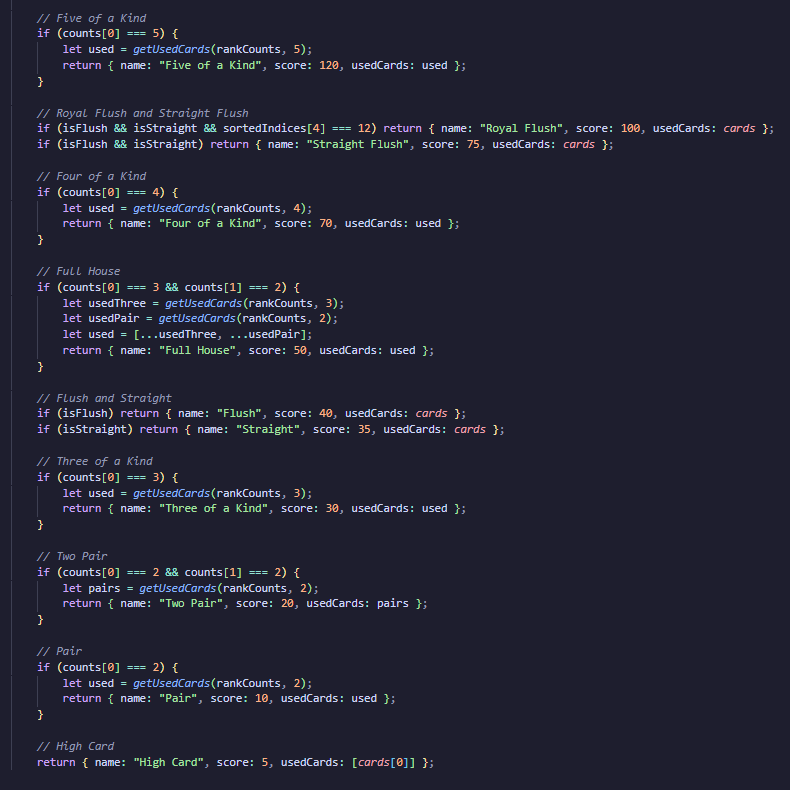
Card and Hand type evaluation is done within the evaluateHand() function. The function determines the type of poker hand formed by the player’s current selection and then returns the hand type. It works by going through a series of checks:

* Counting duplicates (counts).
* Checking if all suits match (isFlush).
* And checking if ranks are in order (sortedIndicies & isStraight).



evaluateHand(cards) function, part 1.

A switch statement based on the number of selected cards (1-5) determines which poker hands to check for, allowing for partial hand plays (4 cards or less). To check for poker hands, it is based on a set of if statements that check the relevant hand conditions (counts, isFlush, isStraight) in descending order of strength. For example, in the five-card case it checks for Flush Five, then Flush House, then Five of a kind, etc. This ensures the highest-ranking valid hand is always selected.



This is separated from playHand() which handles scoring and game progression to keep the evaluation logic self-contained and easy to maintain. It also allows the implementation of new hand types.

**Leaderboard**

Though the intention was for other player’s scores to be viewable amongst others, for assessment purposes, scores are stored via localStorage. Scores are stored as a JSON array containing the player’s names, scores, and number of antes played through and survived. Scores are saved via “saveScore()”, which is triggered by clicking the relevant game button visible on the game over screen. When a score is saved, the list is updated, sorted in descending order by score, and written by to localStorage. On the Leaderboard page, scores are then retrieved, parsed, and dynamically displayed in a table for the player to review.

**Critique**

Sound  
More focus on HTML and CSS – achievements page maybe/unlockable  
More features that had to be cut because going over scope/too ambitious.  
Maybe animations  
inclusion of more upgrades -> legendary and cursed upgrades (not included because ran out of ideas/scope got too ambitious which lead to not enough time.  
upgrades aren’t as well balanced as I’d like it to be (more focused on high ranking cards rather than other strategies) – this is a game design flaw rather than a user experience flaw, and a result of poor planning (throwing random ideas at the wall/taking too many ideas from already existing games, most notably balatro)