Design Review 1

Hexagons

Detailed Design

The hexagons serve two functions: they take user input through the button matrix, and they display the tile's resource status through the LED and 7-segment display. There are two applications of the hexagons: "field tiles" that represent resources on the island, and "harbor tiles" that represent trading ports.

There are nineteen "field tiles" that will display a hexadecimal number from 2-C (to represent the typical numerals 2-12 in a single digit) and the resource that they produce, plus an additional LED to represent the presence of the robber.

There are also nine "harbor tiles": one for each resource type to offer a 2:1 exchange of a specific resource, and four "generalist" harbors that enable a 3:1 exchange of any resource. These are located "outside" the island and may be fit as additional hexagons to the outside of the board (much like earlier editions of the game). The harbors may optionally be distributed randomly, but are typically distributed between nine distinct locations, preventing the need to create a harbor tile for every coastal location. Since the primary information on a harbor tile is the resource type (or "generalist", which may reuse the "robber" or "desert" status) and a number for the exchange rate, the hardware for the "field tiles" may be reused for dynamic harbors.

The devices and connections involved in the hexagons are outlined in detail in Figure 1, the current working schematic.

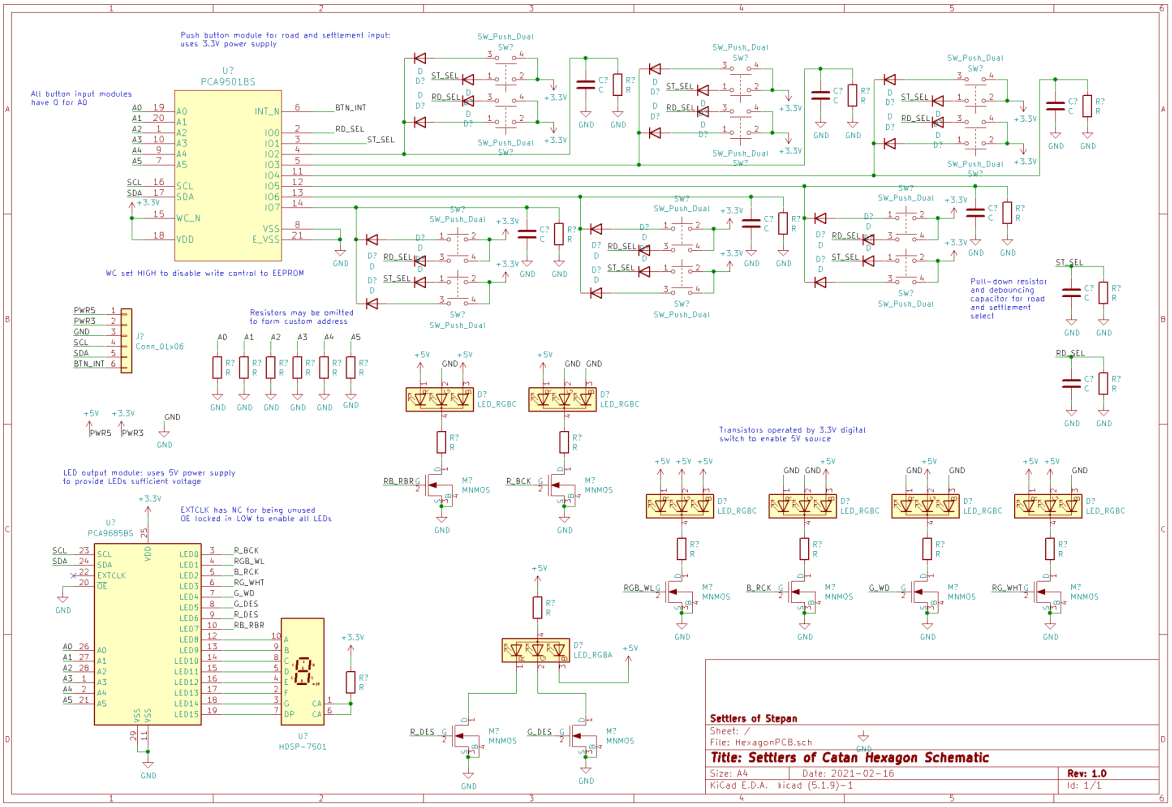


Figure 1: A schematic depicting the hardware and pin assignments of the hexagons.

Going through the schematic, the components of the subsystem operate as follows:

- The button input matrix is processed by the PCA9501BS and communicated through the I2C bus to the Raspberry Pi. The diodes by each of the respective buttons create a set of hardware "OR" gates, so six of the IO pins have a "pair" of buttons (one settlement and one road) that trigger it, and then the remaining two pins are used with "OR" gates to distinguish whether the corresponding settlement or road is being pressed. Pull-down resistors and decoupling capacitors are utilized to stabilize the button inputs.
- The external connector provides both a 3.3V and 5V power supply, which are needed for various components, and also has pins for ground, serial data, serial clock, and the PCA9501BS interrupt pin. These connections provide power and communication capabilities to the hexagon.
- The set of resistors for the addressing pins are used to give each hexagon a unique address. The two IC's share the addressing connections because they each have a built-in bit corresponding to being an input or output, which can distinguish between the two devices. Pull-down resistors may be added to create zeros in the address, and the internal pull-up in each device will create ones.
- The LED modules each correspond to a different resource and will be color coded (users can also better distinguish between them based on non-electric symbols placed on the game board). The module includes an RGB common cathode LED with the appropriate

pins connected to either power or ground to create a primary or secondary color based on which portions of the LED are used. A 5V power supply is used to allow the green and blue colors to be displayed, and a resistor is added in series to protect the LEDs from excessive current. NMOS transistors are controlled at their gates by the PCA9685BS to control which LEDs are lit by a 3.3V PWM signal.

- The orange LED for the desert is different; rather than a common cathode, it uses a common anode. This allows the red and green pins to be controlled independently and create orange, which is neither a primary nor a secondary color. This would not work with a common cathode because the NMOS transistors need to have their sources connected to ground to enable proper control from the PCA9685BS.
- The PCA9685BS reads I2C communications and uses them to determine which LEDs it should enable and what to display on the 7-segment display.
- The 7-segment display has a common anode and uses inverse logic on the PCA9685BS to determine what to display.

All of these individual modules work together to determine button inputs from the users when appropriate and to display the status of the tile (resource, presence of the robber, and number on which a resource is produced).

Major Hexagon Components

The two major hexagon components are the ICs: the PCA9501BS and PCA9685BS. These facilitate the inputs and outputs of the hexagon. All other components are auxiliary and support these two components.

PCA9501BS

The PCA9501BS input expander takes the inputs from the buttons and uses I2C to communicate the input to the Raspberry Pi. When a change is detected, it sets the interrupt hardware flag, which would trigger the Raspberry Pi to check the input matrix on the PCA9501 and interpret it to determine which button was pressed.

The pinout/major connections are described in the table below

Pin	**Connection**
VDD	Connected to the 3.3V power plane to provide a VDD that meets the PCA9501 requirements
VSS & E_VSS	Grounded to the system's common ground
A0-A5	The hexagon's personalized resistor array to synchronize the PCA9501 address with that of the hexagon. The fixed bit of 0 distinguishes the PCA9501 from its corresponding PCA9685.
SCL & SDA	The external I2C bus to enable I2C communication

WC_N	Hardwired to 3.3V to disable writing to EEPROM, as it is not required.
INT_N	Direct connection to an IO pin on the Raspberry Pi to indicate that this specific device is receiving a signal via an interrupt
IO0-IO7	The IO pins are connected to the outputs of the hardwired OR gates to receive data from the input matrix created by the array of buttons, as outlined below.

The "input matrix" created by the array of buttons can be summarized in the following table:

Road Selected	01, 100000	01, 010000	01, 001000	01, 000100	01, 000010	01, 000001
Settlement	10, 100000	10, 010000	10, 001000	10, 000100	10, 000010	10, 000001

The current requirements are calculated in the table below. "Across System" describes the entire system of hexagons, whereas "Per Device" describes the contribution of an individual hexagon. The I2C and button currents are not multiplied by the total hexagons (nineteen tiles plus nine harbors) based on the assumption that only one hexagon will communicate at once and only one button will typically operate at a time. This results in an aggregate current of 7.68 mA from the PCA9501BS across the entire game board.

Feature	**Current**
Standby Current (Across System)	1.68 mA
Standby Current (per Device)	60 uA
Read Current	1 mA
I2C Output	3 mA
Input Current from Buttons	400 uA
Interrupt Current	1.6 mA
System Total (Button Matrices)	**7.68 mA**

PCA9685BS

The PCA9685BS serves a twofold function as an output expander based on I2C input from the Raspberry Pi: it interprets the serial input to send PWM signals that enable the status LEDs for

the hexagon's resource, and it uses inverse logic to set the 7-segment display for the tile's production number.

The pinout/major connections are on the table below:

Pin	Connection
VDD	Connected to 3.3V to comply with specifications. Although the PCA9685 is capable of using a VDD of 5V, it must use VDD of 3.3V in order to understand the I2C communications.
SCL & SDA	Connected to the I2C bus for serial communications.
A0-A5	The hexagon's personalized resistor array to synchronize the PCA9685 address with that of the hexagon. The fixed bit of 0 distinguishes the PCA9685 from its corresponding PCA9501.
LED0-LED7	Connected to the NMOS transistor gates that control each of the seven resource indicators. Using these gates allows the 3.3V signal to control an LED that is powered by a 5V supply.
LED8-LED15	These pins use inverse logic to control the different segments and decimal point of the 7-segment display.

Before current can be calculated, the number of active and inactive segments must be calculated. Because each game uses the same set of numbers for all of the tiles, the total number of active segments may be calculated to give a more accurate estimate of current draw than assuming all segments will be active.

Numeral	Segments Used
One 0 or X (Desert)	6
One 2 (Resource)	5
Two 3's (Resource)	10
Two 4's	8
Two 5's	10
Two 6.'s	14
Two 8's	14
Two 9.'s	14
Two A's	12

Two b.'s	12
One C	4
Five 2's (Harbor)	25
Four 3's (Harbor)	20
Total Segments Used	**154**
Total Segments on Board	224
Total Segments Unused	**70**

The total current usage of the PCA9685 across the entire system is calculated on the table below.

"Per Device" indicates how much current is drawn from an individual hexagon, and "Per Segment" is used to indicate the current drawn for a 7-segment display segment. The "Per Device" values are multiplied by the 28 total hexagons to find the system total, and the "Per Segment" values are multiplied by the total number of segments in the system for the total.

The I2C feature and robber LED are not multiplied by the hexagons because there is only one robber on the entire board and only one I2C communication at a given time.

All of this results in a total current of 5.54 amps across the system for all PCA9685.

Feature	**Current**
Operating Mode Supply Current (total)	168 mA
Operating Mode Supply Current (per Device)	6 mA
I2C Current	28 mA
LED Output Current (total)	700 mA
LED Output Current (per Device)	25 mA
LED Output Current (Robber)	25 mA
LED Off-State Output Current (total)	1.4 mA
LED Off-State Output Current (per Device)	50 uA
7-Segment Supply Current (total)	4.62 Amps
7-Segment Supply Current (per Segment)	30 mA

7-Segment Off-State Current (total)	700 uA
7-Segment Off-State Current (per Segment)	10 uA
System Total (Hexagon Display)	**5.54 Amps**

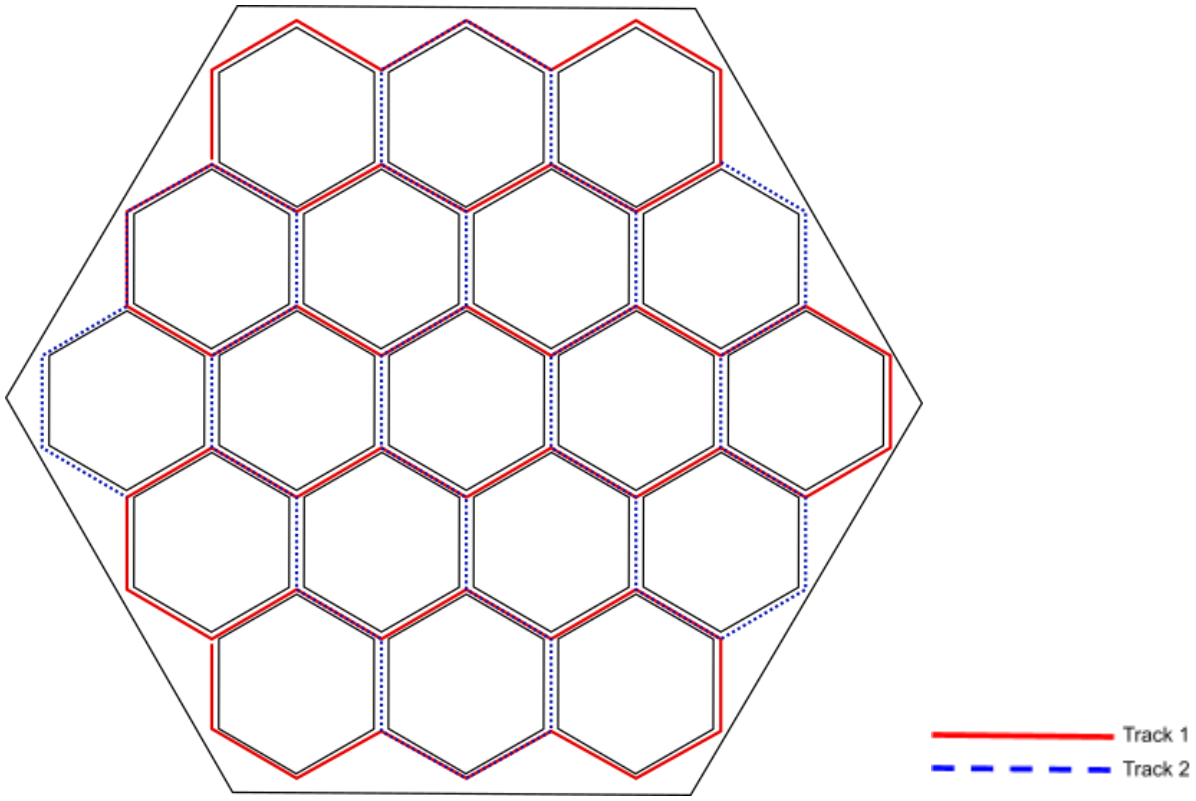
Problems to Solve

There are a few problems/issues that still need to be resolved, but should be solvable during the prototyping phase:

- Values for passive components (resistors and capacitors) have not yet been determined. More detailed assessment of the system, standard values, and prototyping should reveal appropriate values for these components to fill in.
- There is a minor inconsistency between the common anode desert LED and the common cathode LEDs on the hexagons. This is purely aesthetic and should not pose any issues, but the orientation of the common cathode LEDs may be adjusted if bulk ordering or the use case motivates an adjustment to common anode LEDs.
- Although the current draw can be supported by an outlet, the PCA9685 draws a significant amount of current. Most of this current is used on the 7-segment display, so during prototyping experiments with pull-up resistor values may be conducted to see if the display can operate with a lower current, which would significantly impact the current draw and enable additional customization by allowing more segments to be powered simultaneously. These experiments would be viable because they would operate on a smaller scale (single-segment), so the total power required for the experiment would be much lower than if conducted with the entire system at once.

LED Strip

Layout



The board will use four one-meter strips of LEDs connected to the Raspberry Pi in parallel. The path they will follow along the individual hexagons is shown above with each “track” being one path that doesn’t overlap. On the sides of the hexagons where the tracks overlap one of the strips will be cut and extended with wires so as to not overlap LEDs. There will be seven LEDs per hexagon side. In order to curve the strips around the corners the strip will be cut and then connected with wires.

Coding

Programming for the LEDs uses a Raspberry Pi Python library, which was designed for controlling the WS2812 LED Strips that we are using. Using Serial Mode in PWM, you can control the LEDs out of the WS2812 in an array. For our Pi, the GPIOs, which will work for our PWM communication are the 12, 18, 40, or the 52 pin.

In order to program the LEDs in a customizable way, you have to, first, create a specific structure, which is defined by the Pi’s library, for the LED Strip’s LEDs. Then, you can initialize the structure as our LED array, and it’s just a matter of calling color modifying functions, identifying specific LEDs to light up given certain logic, and getting the timings of the colorings correct.

Problems

We encountered 1 problem while working with the LED Strip so far. The initial information, which we had found online showed that the LED Strips required an incredible amount of current. After we did our own testing on the LED Strip running all lights at white color, we found that, on an appropriate brightness level, only about 1A of current is drawn per strip, so a normal power supply should suffice.

Game Logic

Game State

Settlers of Catan is a state based game which means that you can know everything you need to know about a game just by looking at the current state, you don't need a history of past events. Also, because we want to save the game in the middle and easily transfer pieces of the state to the web site, we want the state to be in a simple and serializable form. Thus, we'll use python lists and dictionaries to save the game state so it can be easily navigated and saved to a json file. For the rest of this section, when I show the state I will show it in json format.

The overall structure will be a dictionary of important lists and other global state which will look like this:

```
{
  hexes: [...],
  roads: [...],
  settlements: [...],
  developments: [...],
  harbors: [...],
  players: [...],
  robber: ###,
  longest_road: ###,
  largest_army: ###,
  turn: ###
}
```

Hexes

The `hexes` list holds dictionaries that contain the information that gets displayed on the Hex tiles (minus the robber which is stored in the top level). The dictionary's position in the list will indicate what position it is on the board as all the positions will have an id number that corresponds to a position in this list. A hex dictionary will contain information about the hex's adjacent roads and settlements, its tile type, roll number, and i2c address. A hex dictionary will look like this:

```
{
  roads: [##, ##, ##, ##, ##, ##],
  settlements: [##, ##, ##, ##, ##, ##],
  tile_type: “type”:
  roll_number: ##,
  address: 0x##
}
```

Roads

The roads list holds dictionaries that contain information about the roads in the game. The dictionary’s position in the list will indicate what position it is on the board as all the positions will have an id number that corresponds to a position in this list. A road dictionary will contain information about the settlements the road connects, adjacent hexes, and owner of the road. A road dictionary will look like this:

```
{
  settlements: [##, ##],
  hexes: [##, ##],
  owner: ##
}
```

Settlements

The settlements list holds dictionaries that contain information about the settlements in the game. The dictionary’s position in the list will indicate what position it is on the board as all the positions will have an id number that corresponds to a position in this list. A settlement dictionary will contain information about adjacent roads, adjacent hexes, type (none, settlement, city), and owner. A settlement dictionary will look like this:

```
{
  roads: [##, ##, ##],
  hexes: [##, ##, ##],
  type: “type”,
  owner: ##
}
```

Common to Hexes, Roads, and Settlements

When specifying the adjacent objects in a list, as seen in the previous 3 dictionary examples with the roads, settlements, and hexes keys, the numbers in the list correspond to the objects ids. The order of the ids will indicate the location of the adjacent objects relative to the dictionary that holds the list. To get the order for each of these lists you start at 12’oclok and go clockwise, adding the ids of adjacent objects as you pass them. If there is an empty position where there

usually would be a position (like a missing hex tile because there is a cost), you record the id as -1.

Developments

This is a randomly ordered list of strings that represents the “deck” of development cards. The string will indicate the function of the card in the game.

Harbors

The harbors list holds dictionaries that contain information about the harbors in the game. The dictionary’s position in the list will indicate what position it is on the board as all the positions will have an id number that corresponds to a position in this list. A harbor dictionary will contain information about adjacent settlements, trade ratio, and resource type. A harbor dictionary will look like this:

```
{
  settlements: [##, ##],
  ratio: ##,
  resource: “type”
}
```

Players

The players list holds dictionaries that contain information about the players. The order in the list will indicate the turn order of the game. A player dictionary will contain information about owned roads, settlements, resources, development cards, color, name, ip address, and total victory points. A player dictionary will look like this:

```
{
  roads: [...],
  settlements: [...],
  development: [...],
  ip_address: “127.0.0.1”,
  name: “name”,
  color: [##, ##, ##],
  victory_points: ##
}
```

Other

The robber key indicates the id of the hex that the robber occupies. The longest_road indicates the id of the player with the longest road. The largest_army indicates the id of the player with the largest army. Turn indicates the id of the player whose turn it is.

Hex API

The hex API will contain useful abstractions to help display information on the game board.

IC Abstractions

I've written classes to abstract the behavior of the ICs to make them easier to work with in python. This will allow us to easily manipulate everything on the hex PCBs.

Functions

What will follow is a listing of functions that will be used to interact with the board.

`def set_state(hexes, robber):`
This function will take the list of hex dictionaries and the position of the robber and set everything on all the hexes so that it matches with what is passed in.

`def set_roll_number(hex_id, roll_number):`
This function takes a hex_id to indicate the appropriate hex and sets its 7-segment display to match the roll_number.

`def set_tile_type(hex_id, type):`
This function takes a hex_id to indicate the appropriate hex and sets its LEDs to indicate the type passed to it.

`def move_robber(from, to):`
Turns the robber indicator off on the hex indicated by from and turns it on on the hex indicated by to.

```
def get_hex(player, callback):
def get_road(player, callback):
def get_settlement(player, callback):
```
These buttons will tell the Pi to get ready to receive a selection of the corresponding type. When the selection is made, the callback function passed will get called with the player and id as the arguments.

Web API

This is an outline of the functions that the server will provide for the website. I will be using python functions syntax to make this outline but they won't look exactly like this in the code because of how the server works. All of these will return a status code (200-ok, 40x-error) and the content that is passed to the website. If you want to see how all these functions will be used and kinda fit together, see the next section on Web GUI.

`def get_notifications(player):`
This function will get called repeatedly by the website so it can get popup notifications from the game.

`def add_player(name, color):`
This function will add a new player to the game.

```
def get_player(id):
def get_player_by_name(name):
def get_player_by_ip(ip):
def get_player_by_color(color):
```
These functions will search the player list and return the player dictionary to the frontend of the appropriate player.

```
def get_players():
```
Returns the list of player dictionaries.

```
def shuffle_board():
def shuffle_cards():
```
These will add some randomization to the game.

```
def request_trade(from, to, transaction):
def trade_decision(accept, transaction):
```
These will deal with the trading between players and with the bank and harbors. Transaction will be a dictionary with information about the trade like an id and the proposed resources.

```
def activate_robber(robber, hex, victim):
def robber_discard(player, resources):
```
These functions will handle the robber logic.

```
def resource_roll():
```
This function does the virtual dice roll and notifies the players with their resources. It also indicates to the game logic to move to the next player's turn.

```
def build_road(player):
def build_settlement(player):
```
These functions will handle the logic for building roads and settlements on the map.

```
def buy_dev_card(player):
def use_dev_card(player, card):
```
These functions will handle the logic for buying and using development cards in the game.

```
def get_everything():
```
Passes back the entire game state.

Web-based GUI

Introduction

It should be noted that the interface examples shown below are mainly for aesthetical and UI flow purposes--some changes will be made between now and the final product, which cannot be shown yet because they depend on other subsystems which have not been completed. These include the game API, game board, and Wi-Fi LAN. So, the screenshots seen here are simulating what would be visible to the user in the final product. Additionally, the aesthetics of the design will undoubtedly be modified from their current state.

Conceptual Design and Interfacing with Other Subsystems

The web-based graphical user interface that will supplement the physical game board is comprised of three pages: the main (“index”) page, the game page, and the debug page.

Upon connecting to the game, a player will be presented with the index page:

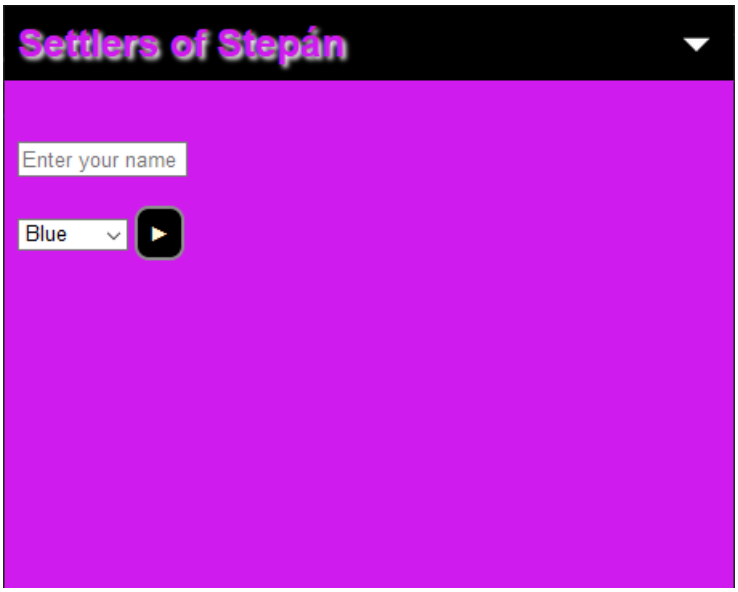


Figure 2. The index page; where player info is entered

Here, the player first enters their player name and chooses an available color from the four that the game offers. The game API will alter the list of which colors are still available to choose from. Upon clicking the arrow, the web GUI will link this game player to the device being used--this will be explained in the next section. Next, the player views a list of players who have added themselves to the game.

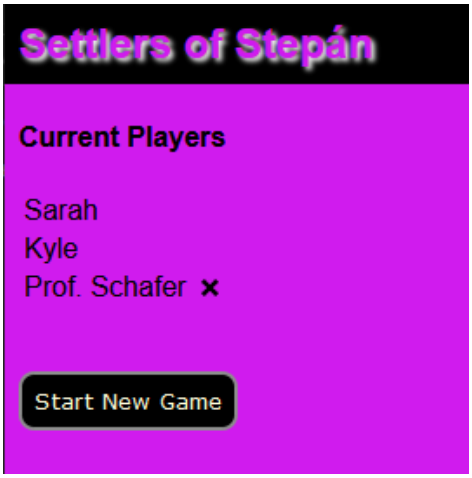


Figure 3. The index page; list of players before starting the game

There is an option to delete oneself from the game (in case one no longer wants to play, or has misspelled their name), which takes you back to the previous page. Only the player themselves can remove themself from the player list. Any player can click the “Start New Game” button. There is a confirmation dialog,

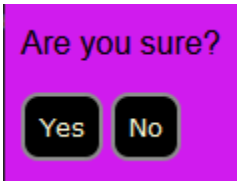


Figure 4. Game start confirmation dialog

and choosing “Yes” will begin the game (using the APIs) and redirect the web browser to the game page. Clicking “No” will return the player to the player list.

After the process of determining player positions and which player begins, which happens on the game board, the game page will be presented. For the purposes of this document, we will assume the role of a player who is not going first.

Figure 5. The game page; when it is not your turn

Displayed is a set of information that one needs to know throughout the game, which supplements the physical game board. We can also see whose turn it is, indicated in red. All of this information is pushed to the web page via the game API as a JSON object. Before moving on through the game, we should look at some things that can occur on the GUI when it is not your turn, or at any point.

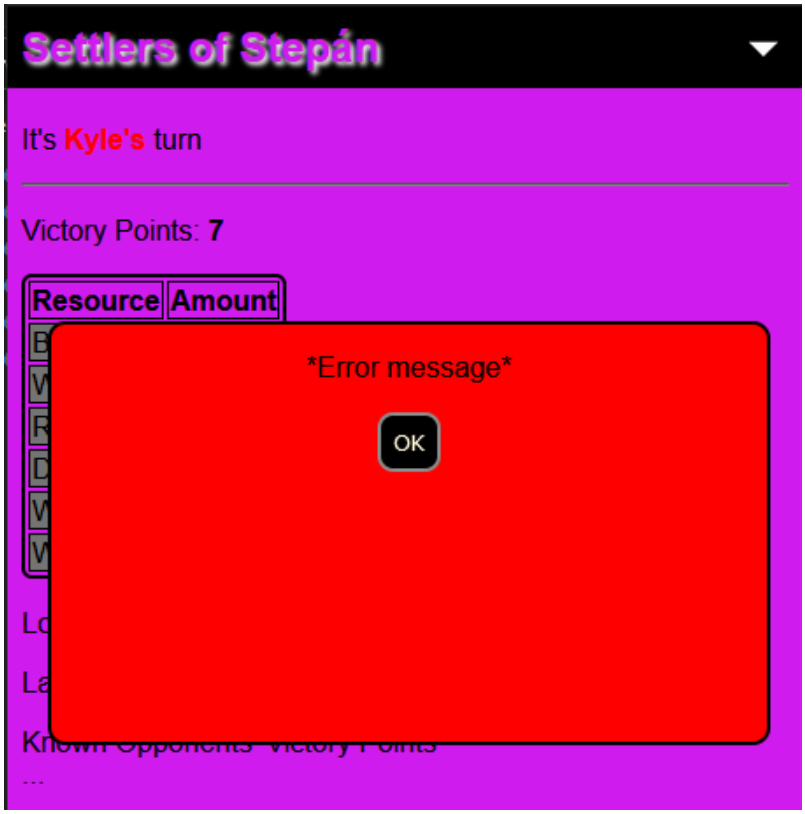


Figure 6. The error pop-up on the game page

First, an error message will be displayed if an illegal action is attempted, or if something goes wrong with the game itself. The message will be provided by the API.

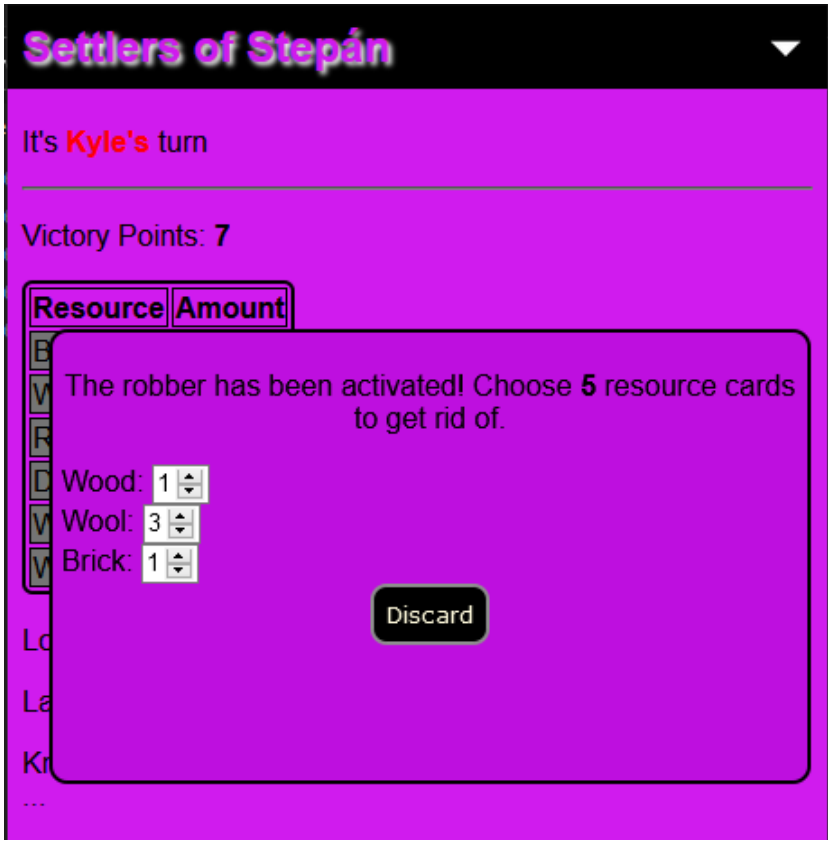


Figure 7. The robber-card-discarding pop-up on the game page

Another thing that can occur while it is not your turn is that the robber is activated, in which case every player gets rid of a certain amount of their resource cards. This dialog displays information about how many cards you must discard, and allows you to select however many of each type you would like to discard, based on what you have. Again, all of this information is passed back and forth by the game API.

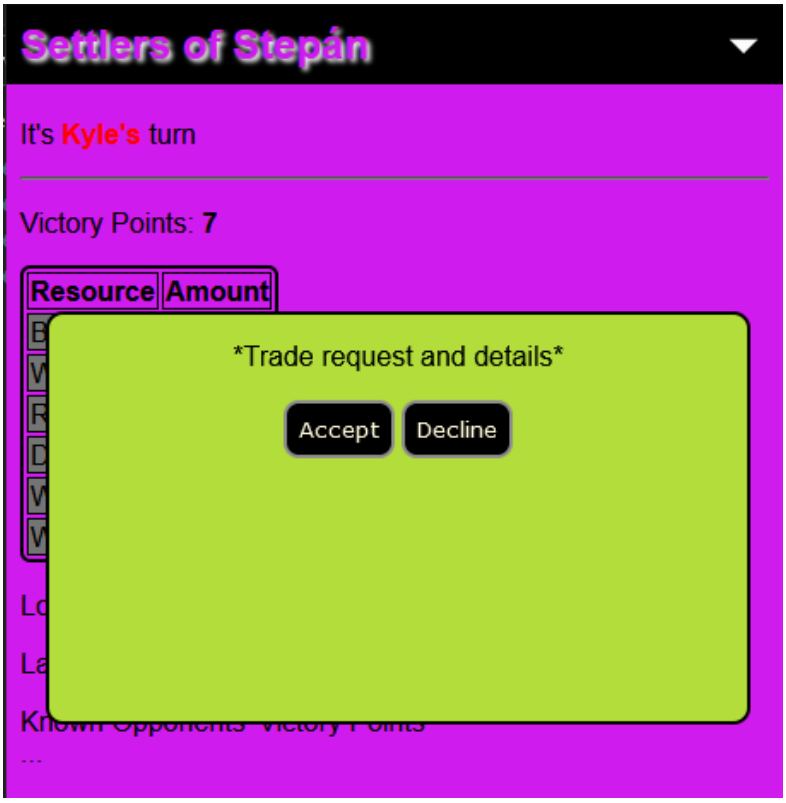


Figure 8. The trade request pop-up on the game page

Finally, someone can request to trade resources with you (a “domestic” trade). This is something you would negotiate by discussion, but then implement in the game using the GUI. The player whose turn it is will select you as their trading partner, after which the game API will cause this pop-up on your device where you can accept or decline the terms of the trade.

Now, we can move on to the various parts of the GUI concerned with a player’s turn, which is broken up into three phases. The first phase is the “resource roll” phase, which is the first thing a player will see when it becomes their turn:

Figure 9. The game page at the beginning of the Roll phase

The main action you must perform here is rolling the dice (virtually). However, you can also view or use development cards at any point in your turn, so this option is displayed as well. Upon “rolling the dice”, the result is displayed.

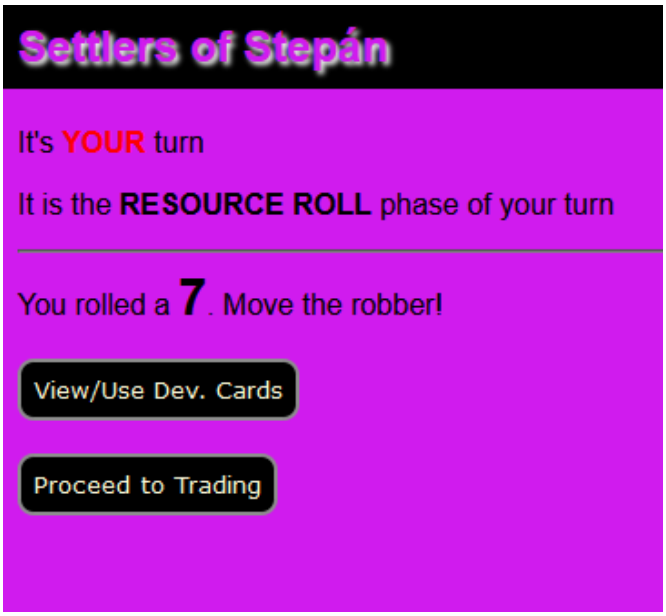


Figure 10. The game page after the dice are “rolled”

In this example, a 7 is rolled, which activates the robber and requires some additional steps from all of the players. Part of this was described earlier, but there are some actions required of the player who rolled the 7 as well. The first of these is on the game board, which involves selecting which hex to move the robber onto. After this, the player steals 1 resource card at random from a player who has a settlement or city adjacent to this hex. If there is only one player who does, the requisite steal will occur automatically. However, a pop-up will be presented if the player has to choose between multiple other players to steal from:

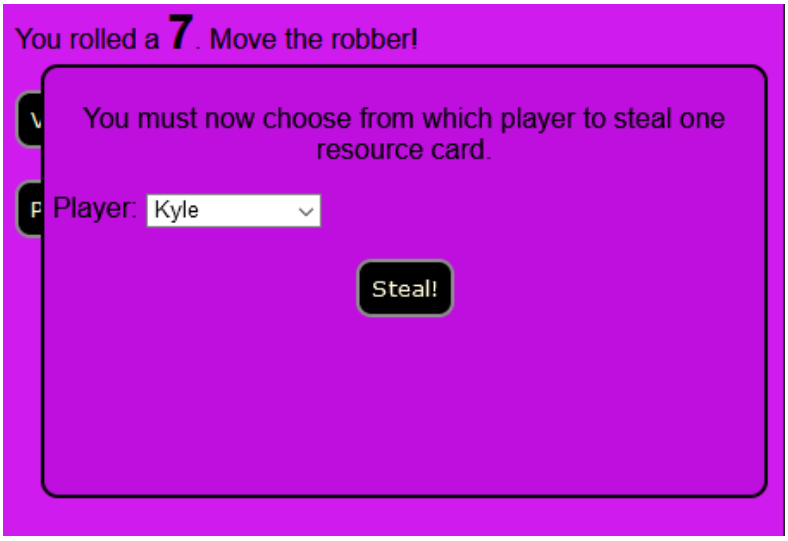


Figure 11. The game page if the player needs to choose who to steal from

After the player is done with this phase of their turn, they click the “Proceed to Trading” button as seen in Figure 10, which is the second phase:

Figure 12. The game page (partial) at the beginning of the Trade phase

This phase of the turn mainly involves trades with other players or with harbors (“maritime” trades). As well as being able to view all of the static info needed to make gameplay decisions, and view or use development cards, one can propose a trade with another player (“Propose Trade”) or initiate a maritime trade (“Maritime Trade”).

Figure 13. Domestic trade pop-up

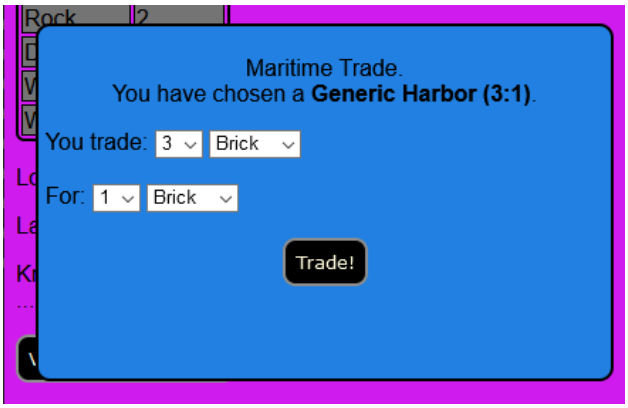


Figure 14. Maritime trade pop-up

As with all of the other user interactions, the variables related to making a trade are communicated from the game API to supply the dialog box with the correct information, and then the user’s choices are sent back to it to make the changes in the game itself. One note is that the maritime trade first involves the selection of a location on the game board, which then determines what information is filled in on the pop-up.

At this point, we can take a look at what the development card interface looks like:

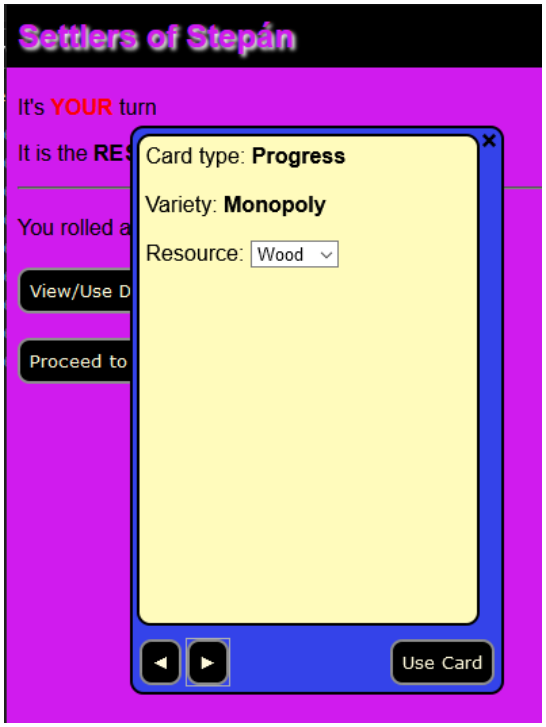


Figure 15. The game page with the development card pop-up

The user can cycle through all owned development cards (sent to the GUI as objects by the game API), and view any relevant information about type and other variables. The user can also set variables if needed before using a certain type of card. Note that there will be a numbering system with which to differentiate a user's cards, which is not visible in this prototype, but this cannot be implemented until the game software is constructed.

Upon completion of trading activities, the player clicks the "Proceed to Building" button seen in Figure 12.

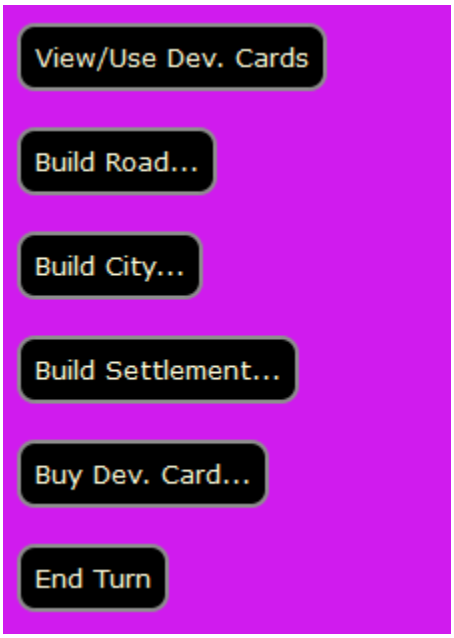


Figure 16. The game page (partial) at the beginning of the Build phase

In addition to the game information that was displayed in previous phases, there are buttons associated with the four actions unique to this phase of play: building a road, city, or settlement, or buying a development card. The first three do not require any interaction with the web GUI, but with the physical product itself. After buying a development card, though, a message will be displayed conveying the type of card that was purchased:

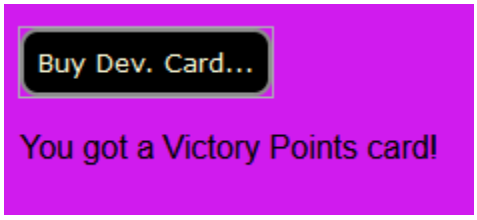


Figure 17. The message appearing after buying a development card

This way, it will be easier to identify the new card when a player views their development card collection. After the conclusion of any building or buying activities, the player ends their turn with the "End Turn" button seen in Figure 16. As stated before, all of these interactions involved the game API, which will be elaborated upon in the next section.

Aside from the gameplay, there is one more aspect of the web GUI: the debugging section. This is accessible from the drop-down navbar visible on all pages:

Figure 18. The navbar after being expanded

Upon clicking the link, the user is taken to the debug page.

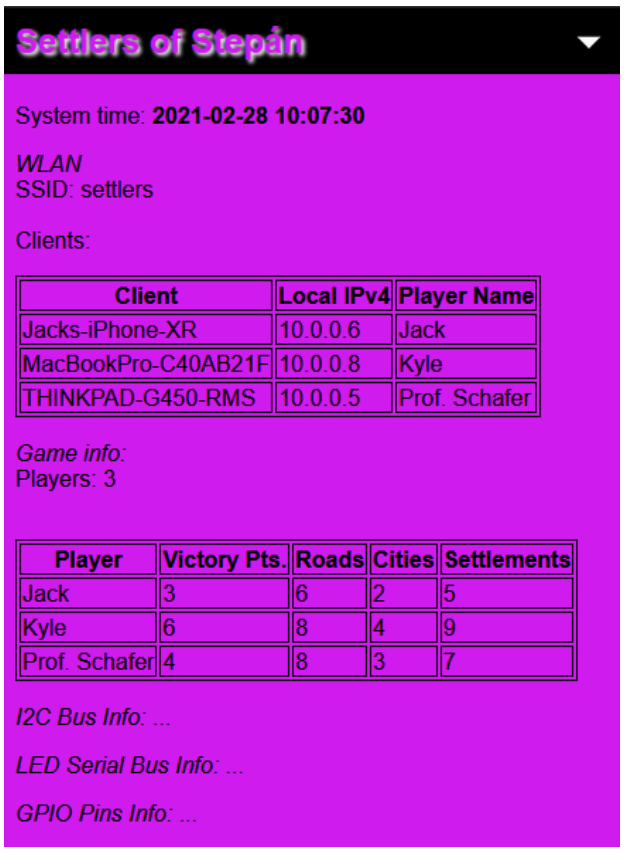


Figure 19. The debug page (partial)

Here, users can view information about the Raspberry Pi Zero W which is running the game, as well as variables concerning the gameplay itself. This includes information about the WLAN, I^2C bus, LED serial control, GPIO pins, power, and more. Some of this information cannot be demonstrated in this prototype since we do not know what information we will be able to display. In addition, a user will be able to perform various functions associated with the game,

board, and the Pi computer more generally, in case any troubleshooting, repair, or in-game diagnosis needs to be done. These functions have not yet been determined.

Languages

The languages used in this subsystem include:

- HTML
    - Structure of the GUI and its pages
- CSS
    - Formatting
- JavaScript
    - Pushing and pulling information to/from the game API
    - Dynamic behavior of the GUI elements as the user interacts with them
- jQuery
    - Dynamic behavior of the GUI elements as the user interacts with them

Player Association

One key functionality that has not been described yet is the method by which players' information will be kept secret like in the original game, and how the web GUI will identify each player. This will be accomplished by one of two methods, yet to be determined:

- Giving the player's devices a cookie and treating the game as a web "session"
- Tracking the IP address of each player's device

Either way, these simple methods will allow a 1:1 association between a player's device (i.e. their web session) and the corresponding player in the game data. This way, the players will be accessing the same web pages simultaneously, but will be served information unique to their player in the game. Correspondingly, any actions the player performs on their device will be identifiable by the game API as coming from that "player" in the game. Essentially, the UI's web pages will be templates, and will be filled in with the correct information based on which player's device is accessing it.

This device-software linkage, and the dynamic, per-device population of the web pages may require the use of another language, such as PHP. This is yet to be determined.

APIs

The game will be written in Python, and will be running on the Pi. As mentioned previously, there will be an API written in JavaScript for the GUI to interact with the game information itself. This will be comprised of functions for requesting certain types of information, and passing it on to the GUI for display, and for entering information and actions. These

functions will be very simple, and their being written in JavaScript will make it very easy to integrate directly into the existing web interface, especially given that the current buttons and pages system is implemented in jQuery, which is a JavaScript library.

WLAN

This minor subsystem falls under the major Web GUI subsystem, and will be touched upon just briefly.

The Wi-Fi chip on the Raspberry Pi Zero W can be easily operated as an access point, rather than a simple client. There are many Linux packages for this purpose, *RaspAP* being a popular one. Regardless of whichever one we choose, the implementation will be quite simple. However, this remains to be done, since the Pi's Wi-Fi capabilities need to be kept in a client mode so that group members can access it remotely to work on the project. We will switch it over to being a Wi-Fi access point relatively late in the design process.

Then, all that remains to be done is to redirect any device that connects to it to the home/"index" page of the GUI before a game is started, and to collect each client's IP address if this is needed for player-device association.

We foresee no problems implementing this minor subsystem, and have already begun researching the ample examples of its use, as this is a very popular thing to do with embedded computers such as a Raspberry Pi.