Title: Card Game Project Writeup

Name: Gabriel Donnan

Number

Contents

[Analysis 3](#_Toc25653441)

[Identification 3](#_Toc25653442)

[Computational Approach 3](#_Toc25653443)

[Research 4](#_Toc25653444)

[Stakeholders 7](#_Toc25653445)

[Interviews 7](#_Toc25653446)

[Hardware Spec 8](#_Toc25653447)

[Software Spec 9](#_Toc25653448)

[Requirements 9](#_Toc25653449)

[Limitations 10](#_Toc25653450)

[Design 11](#_Toc25653451)

[Design of required features. 11](#_Toc25653452)

[1)Card Buying System 11](#_Toc25653453)

[2) Having different types of cards with different effects 14](#_Toc25653454)

[3) Cards which trigger effects at different intervals 16](#_Toc25653455)

[4) Placing limitations on the size of the players’ boards, hands, mana and currency 25](#_Toc25653456)

[5) Tutorial Section 32](#_Toc25653457)

[6) Ability to end the game when an end condition is reached 34](#_Toc25653458)

[7) Ability to combine cards to make them more powerful 37](#_Toc25653459)

[8) Implementation of local multiplayer against another human player 40](#_Toc25653460)

[Implementation 41](#_Toc25653461)

[Game Loop 41](#_Toc25653462)

[First Iteration 41](#_Toc25653463)

[Player Class 46](#_Toc25653464)

[Card class 50](#_Toc25653465)

[First iteration 50](#_Toc25653466)

[Second Iteration 51](#_Toc25653467)

[Testing 53](#_Toc25653468)

[Evaluation 54](#_Toc25653469)

# Analysis

## Identification

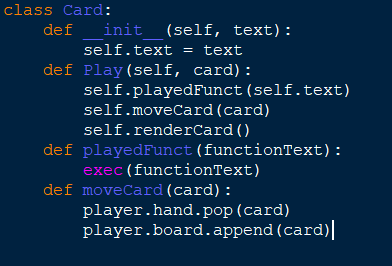
Currently I believe the market for digital card games is relatively bare, with only two main games dominating the market. I believe this leaves room for the creation of a new digital card game, using the simple combat mechanics employed by many card games in the past but using the digital medium to allow for more complicated effects and mechanics that are not amenable to the physical medium, specifically pseudo-random effects such as random generation. As such I believe that creating a game to fill this role would be beneficial to those who may believe the current game market to be stale as it could revitalize an interest in games of this type.

## Computational Approach

The problem of making this is, I believe, best solved by computational methods. This is for a variety of reasons, firstly the game is easily broken down into consecutive logical operations, which is exactly how computers operate, following instructions linearly. For example when a card is played this can easily be broken down into some function calls that can be made, firstly any effect the card might have on play must be enacted so a function called playedFunct() can be called to do this, then the card must be removed from the list of cards in your hand then added to the list of cards on the board, this can be done via a simple function call or even a single line statement which can pop and append the card (though one-liners like this can be disadvantageous when debugging as they can often seem more logically complex than they actually are). And finally the card has to be rendered on the board so a function renderCard(card,position) can be called, allowing the card to become visible on the board. Every facet of the gameplay can be broken down logically as such, as can the menus outside of the game.

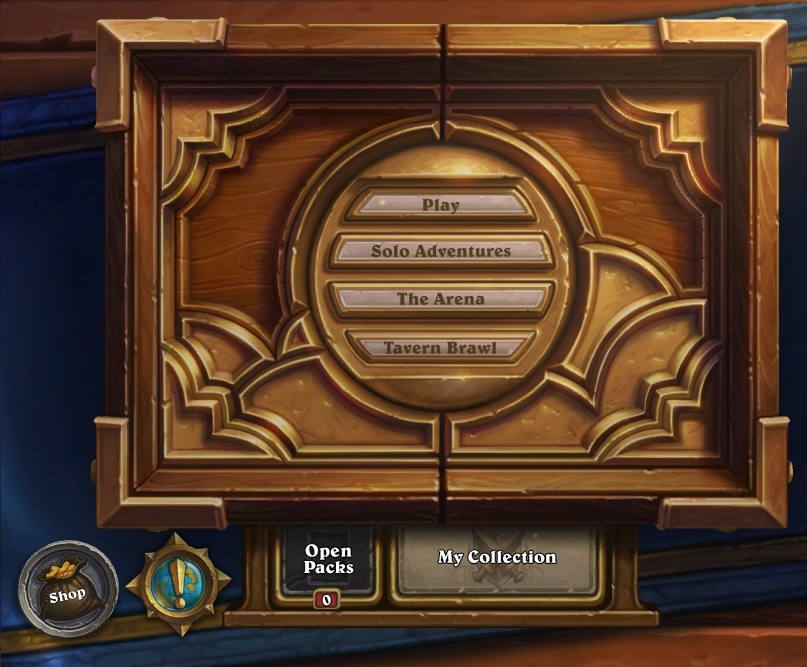
The reinforcement learning based AI which will be playing the game is not only best suited to computers as it can interact linearly with the game, it is impossible to make without them as you would need some other device capable of intelligent decision-making to be playing the game, which is limited to only humans and computers. Using a human here would defeat the purpose of having an AI opponent as the game is meant to be playable by yourself. The AI’s training makes use of iterative self-improvement (iteration being one of the most key computing concepts), performing mathematical and logical operations to achieve this at a speed not physically achievable by non-computational methods. The AI operates on a basic level by taking data from playing the game and optimizing itself so that it can create an output which will be the most likely to secure it the ability to win the game, this satisfies the computational approach as it is simply enacting a mathematical function on the input data to create an output

The game will be made in an object-oriented manner as I find this to be the best method to create the game whilst maintaining an intelligible, logical, code flow. Each card for each player can be instantiated as an object of the class “Card” which allows the cards to have their own unique properties (such as a name, cosmetic text, health, cost, attack, etc.) and were I not using an object oriented structure like this it would be much more difficult to logically allow the game to function. This process can be shown to be effective when using the example of playing a card again as using objects the cards themselves can have a .play() function, which is important as if it was necessary to use a global function the player number would need to be specified as each player has cards and the syntax within this function would be much more complicated, requiring many IF statements to determine which player is playing and what card they’re playing. Once the card played is determined when using objects any effects the cards may have on other cards can be easily enacted as they can directly adjust the properties of the other cards, but when not using an object oriented approach there are no individual card properties to edit, and as such everything would have to be kept in lists and adjusted this way, resulting in much more unreadable code, greater difficulty in adding more cards and inefficient code.

 *Below is a mock-up of the Card class structure needed to play a card:*

## Research

When looking into creating a game such as this I must first look at other similar games to see what inspiration I can derive from them but also more importantly what I can improve on comparatively to them. As part of my research I played and looked at the features of the two most popular online card games (Hearthstone and Magic: The Gathering, Arena) and the recently released DOTA Underlords, which is not a card game but uses an interesting unit buying system which I feel can help solve the problem of card drawing randomness which often negated some amount of the skill involved in gameplay for the first two games I looked into.

 Firstly I looked at Hearthstone, which is a game which is designed to look as smooth and user-friendly as possible, meaning it is an automatic attractor for those newer to the card game community or those who appreciate the polish put in to it.

*The Hearthstone main menu screen*

Hearthstone’s simple user interface is extremely appealing and as such I have decided to use a more simplified visual main menu structure rather than using text or any more complicated layouts which I feel needlessly confuse the user. The game has a pack opening system and an overarching card “collection” which indicates the cards which you do and do not own, and therefore the cards which you can and can not put into decks. I feel this is unfair to the end user and instead have opted to allow each user to have access to all the cards in the game.

The game’s concept is relatively simple, you draw 3 or 4 cards at the start of the game (dependant on whether you go first or second), you start at 1 mana crystal, gaining one per turn until you have 10. Mana crystals can be spent by playing cards and will refresh at the start of each player’s turn. There are two main card types, minions and spells. Minions are creatures which are summoned on to the board which can fight for you, either attacking your opponent or their minions. Spells are instant effects which do not necessarily summon creatures but can affect them or either player’s character. When the first player reaches 0 health the game ends.

I have decided to simplify cards into one type instead of using minions and spells. Cards will always summon a creature and can have effects similar to those of hearthstone’s spell cards attached to them, thus meaning the game is more easily understood. I have also opted to use a similar combat style, however unlike in hearthstone, you must always attack the creatures on the board before attacking the opposing player, allowing the game to last for longer and for creature on creature combat to be a larger facet of the game.



*A standard board state in Magic: The Gathering, Arena*

The second game I researched was Magic: The Gathering, Arena. Magic: The Gathering is one of the longest standing card games ever made and its advent into the market of digital card games has given me an opportunity to see what works and what doesn’t comparatively to Hearthstone.

The game functions slightly differently to hearthstone in that instead of mana crystals you gain as the turns advance you have the ability to play “land” cards which allow you to play other cards by spending them each turn. It is also different in the fact that your opponent gets to dictate the path on which your attacking creatures attack, meaning the game is also reliant on bluffs and not knowing your opponent’s decision-making.

I believe that whilst this does promote the skillful aspect of the game that my stakeholders want, bluffing and strategies such as this are almost impossible against an AI opponent as it does not make decisions in the exact same way humans do and as such I am allowing the player full control of their creatures’ attacks. In Magic: The Gathering damage inflicted to creatures is not permanent due to the game originating in physical form so it was nearly impossible to keep track of damage inflicted to so many things. Since my game will be digital the damage inflicted to creatures will last between rounds as computers can keep track of this very easily.

 One feature of Magic: The Gathering that I wish to implement if I have time is the ability to be able to stack like units on top of one another to remove clutter on the board and allow the game to look more visually appealing, which I believe is a key factor in player accruement.

*A standard board state in DOTA: Underlords, with purchased units displayed at the bottom*

The final game at which I looked for my research was DOTA: Underlords, a recently released game based on the popular mod DOTA: Autochess. This was the only game I researched that was not a card game, but rather just a strategy game in a more broad sense to see if I could incorporate any features currently unused by digital card games but applicable to strategy games in general to allow mine to stand out.

One particular feature that stood out to me while researching this game is the unit purchasing system in which each round you are offered a selection of units which you can buy with gold you collect over the course of the game, when applied to a card game instead of drawing a card each turn I feel that this would be extremely beneficial to the skill requirement of the game and thus the long term enjoyment levels that can be achieved. Adding this feature also removes the need for deckbuilding, which becomes simply a game of statistics instead of an interesting facet of the game with which to play due to the easily analysable gameplay of card games.

## Stakeholders

When looking into stakeholders for my game a number of my peers expressed interest in such a game given certain features they wanted were implemented. One particular interested party is a friend of mine named Vishaan who has a background in games that involve longer term strategy so as such a card game was a natural attractor for him. The game perfectly suits his desire for strategizing as there are nigh on infinite possible iterations of the game and its state so a good understanding of strategic thinking is rewarded.

He also enjoys games with a high degree of customization as evidenced by his investment in RPGs with character customization in the past, so the deck customization that is possible appeals to him as well. He hopes that in the way the game is coded it will be easy to add new cards to allow the game to feel fresher as new cards can rejuvenate a game to ensure it’s long term enjoyability. He also prefers games in which the opponents feel more real and as such I have decided to use reinforcement learning AI as opposed to the standard model of using strings of conditional statements to make the decisions the opponent makes less rigid.

Another potential interested party is my younger brother Micah as he has enjoyed digital card games in the past but feels like the current available market for them is too bare for his liking, as such he wants a game with different gameplay to the other two major online card games of Hearthstone and Magic: The Gathering, Arena. In order to achieve high enough difference from these games he feels that the game I am making should have mechanics with more choice to replace the classic method of drawing cards implemented by other games which can lead to a high amount of luck being involved in gameplay as opposed to pure skill so as such I have decided to instead opt for a turn based card buying system instead which, whilst it still has some variance, rewards skill over luck a greater proportion of the time. He has also requested that the game have a screen that will scale to any screen size automatically so it does not take any additional effort to use it on different devices and as such the game will be able to be enjoyed by a wider range of device owners who can now use any device without the game looking any different between them.

The third stakeholder I have identified is another friend of mine who very much enjoys gaming but is not currently very involved in card games or strategy games in general and as a result of this lack of current investment into games of this type he has said he would enjoy it if the game were to have an explanation of the rules included within it to make his learning process easier. In order to fulfill this request I have decided that I will add a basics tutorial section to the game accessible via the main game menu so that the game can be picked up by new players with greater ease.

### Interviews

I conducted short interviews with my stakeholders in order to identify what exactly they wanted to see from the finished product of the game:

My first interviewee was Vishaan, below is a record of the interview that was conducted:

Me: What, for you, makes a game enjoyable?

Vishaan: Well, in my opinion as long as the game is of a type that I like it just has to not be completely reliant on luck but have enough variation so it doesn’t feel stale

Me: How do you think one could tackle the problem of luck versus variance in a card game?

Vishaan: Im not 100% sure, but if I had to guess I would say I would want not to have to build a deck of cards so the games I play don’t always feel the same for each deck

Me: Alright, and what do you feel, if anything, that current card games are doing wrong that I could improve on?

Vishaan: Yeah there is one thing that annoys me about card games right now, most games do this actually and its really offputting. Its that their AIs for enemies don’t feel like they’re actually a challenge and I think if they seemed more human then I could look at them as more of a valid opponent

## Hardware Spec

The game will not be extremely hardware intensive as all the AI’s training will be done on my computer beforehand so the intense training process is not necessary to carry out on the end user’s machine. If someone were to attempt re-training it or training an entirely new neural network to use instead of mine (this can be done relatively easily and would normally be done to adjust the difficulty level of the AI opponent as less training would mean more difficulty and vice versa up to a point) I would recommend a system using:

* Dedicated GPU with no less than 4GB of VRAM
  + GB+ of VRAM would be recommended but can be very expensive
  + A more powerful GPU is more impactful on training performance than a more powerful CPU
* CPU with at least 4 cores with a clock speed of 3GHz
  + Improving the CPU will increase the performance of training with more cores being more impactful than higher clock speed but an upgrade of CPU will almost always be worse than an upgrade of GPU

These hardware specs are important in the event of a retraining in order to attempt to train it in a practical amount of time (i.e. less than a week). The only real requirement when you are not retraining is that the hardware is able to support all the necessary software at a basic level for the computer to run it which, according to Intel (<https://software.intel.com/en-us/distribution-for-python/system-requirements>), Is at least one gigabyte of secondary storage (SSD or HDD are both acceptable as the read/write speed will not affect the performance very much) for the base python and an additional 2 to 3 gigabytes for the rest of the libraries that need to be imported such as pytorch.

The system should also have:

* A processor at least as powerful as a single core 1.6GHz processor
  + This is normally the power level of the lowest price point ARM or Intel Atom processors
* Standard peripheral devices required for computers to function normally
  + At least one monitor on which to see the gameplay
  + A keyboard (even 80% size keyboards still work)
  + A mouse

## Software Spec

There are relatively few pieces of required software for the game:

* A python version of 3.0 or more recent
  1. Syntax changes heavily in some common statements like print or input from 2.7 to 3.0 and as such the code would not be able to run on any older versions of python
* The most recent versions of the python libraries I use
  1. Pygame
  2. Pytorch
  3. Random
  4. Math
* An operating system which supports all the libraries I have listed (i.e. the standard operating systems of Windows, most Linux distributions or Mac OS)

## Requirements

The game, upon completion, should have:

* Necessary Features for the game to function:
  1. Card buying system
  2. At least 10 different card types
  3. Cards which trigger effects on:
     + Death
     + Play
     + Attack
     + End of turn
  4. Ability to play cards from your hand onto the board
  5. A limit on the space on your board and in your hand
  6. Tutorial section
  7. Ability to attack using your played cards to damage the opposing player
  8. AI implementation of the second player
  9. Ability to win or lose the game given certain criteria have been fulfilled
  10. Ability to combine cards to make them more powerful
  11. Local multiplayer versus another human player
* Important features the game should have but are not necessary for it to function:
  1. A menu with which to navigate, at the very least containing the ability to:
     + Play a game
     + Adjust your settings
     + View the tutorial
  2. Ability to adjust the position of cards on your board
  3. Tooltip boxes which appear when items are hovered over
  4. Ability to combine multiple of the same card into a single more powerful version of themselves
  5. Ability to pause the game
  6. Menu of all the cards and their combinations
     + Given enough time this could also be an interactable visual menu, though this is more of a stretch feature
* Features that would be nice to have but less likely to be finished:
  1. The ability to retrain the AI from within the game menu
  2. Different game modes to the standard mode of play
  3. In-game custom card creation feature
  4. Ability to save the game state and return to it later
  5. Animations for actions such as playing cards
  6. Passive effects for played cards
  7. Stacking alike units

## Limitations

Whilst the game will be feature-rich and strategically in-depth as I can make it, there will of course be limitations to the things that I will have been able to complete or add to the game within the time frame I have to make it.

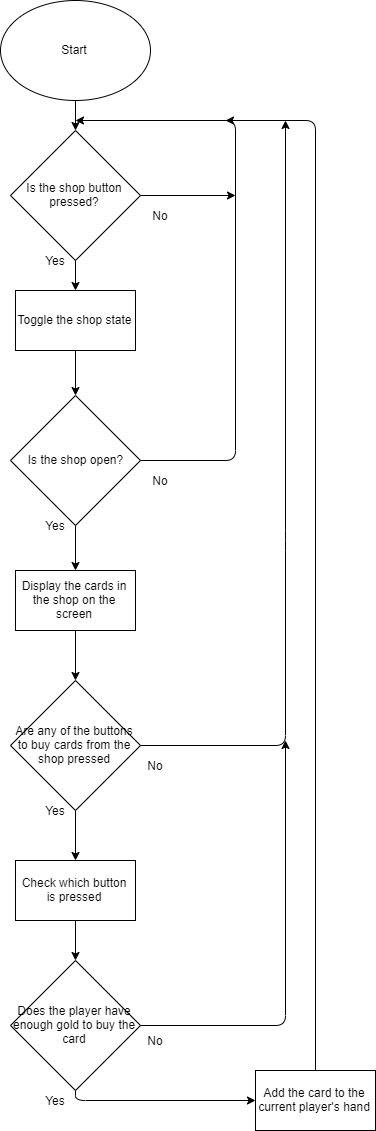
Such limitations include:

* No sound
* No custom animations for different effects
* No online or LAN multiplayer
  1. I might possibly implement local multiplayer, however online or LAN multiplayer is much too complicated to do within such a time frame
* Only English language support

# Design

## Design of required features.

### Card Buying System

This is a system to allow the current player to buy cards from the shop via clicking buttons associated with them in order to be played later.

If mouse.collide(button) and mouse.clicked() then

Shop.open = True // toggles Boolean value

If Shop.open == True then

Cards.display() //Displays cards in the shop

Cards.displayButtons() // Displays the buttons associated with the cards in the shop

For button in Cards.buttons:

If button.get\_pressed() == True and PlayerCurrency > button.card.currency then

Player.buyCard(button.card) // Buys the card which the button is linked to

PlayerCurrency -= button.card.cost // Deducts the cost of the card from the players gold count

Else

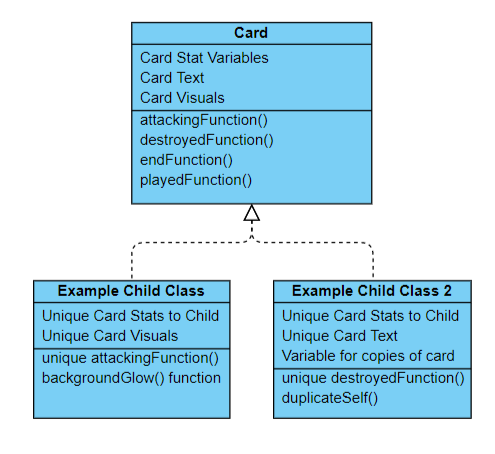
DisplayError(“You don’t have enough currency to buy that”)

##### Test Plan

|  |  |  |  |
| --- | --- | --- | --- |
| Number | Item | Purpose | Expected Result |
| 1 | Clicking shop button | Check that button functions as required to toggle the shop | The shop goes from not being displayed to being displayed when the button is pressed |
| 2 | Clicking on the card itself when shop is open | Check that clicking on the card in the shop does not buy it, play it if an instance is in your hand or any other such possible bugs | Clicking on the card itself should not do anything, as the buttons associated with the cards are what are used for buying them |
| 3 | Clicking on the buy buttons associated with the cards when you have enough currency to buy them | Check that when you have enough currency to buy the card it can be added to your hand and removed from the shop | The card is removed from the shop and added to your hand, deducting the amount of gold it costs from your balance |
| 4 | Clicking on the buy buttons associated with the cards when you do not have enough currency to buy them | Check that when you do not have enough currency to buy the card it does not get purchased from the shop and an error message is displayed | The card remains in the shop and a copy of it is not added to your hand, an error message is displayed telling you that you do not have enough gold to purchase that card right now |
| 5 | Clicking in the locations in which the buy buttons would be located when the shop is not being displayed | Check that the buttons cannot be interacted with if the shop is invisible | Nothing will happen, the cards in the shop at that location will not be bought |

### 2) Having different types of cards with different effects

This is managed using a class as a blueprint for cards, allowing them to have effects which trigger at different times, variables common to each card but also functions exclusive to each.

Class Card:

Initialise(health, attack, cost, text, picture):

Card.health = health // Declares class variables

Card.attack = attack

Card.cost = cost

Card.text = text

Card.picture = load(picture, size) // Loads the picture as an image

attackingFunction()

Null // The function does nothing and the child classes will modify these via polymorphism to make them do things specific to their needs

destroyedFunction()

Null

endFunction()

Null

playedFunction()

Null

Class ExampleChild(Card):

Initialise(health, attack, cost, text, picture):

Card.initialise(health,attack,cost,text,picture) // Initialises child class using parent class

ExampleChild.IsDuplicated = False // Adds an additional variable which does not exist in the parent class

DuplicateSelf()

ExampleChild.isDuplicated = True

ExmpleChild(x,y,z,a,b,c) // Creates a duplicated instance of the same class

endFunction()

ExampleChild.health += 1 // Changes the inherited endFunction to do something different from the default of doing nothing

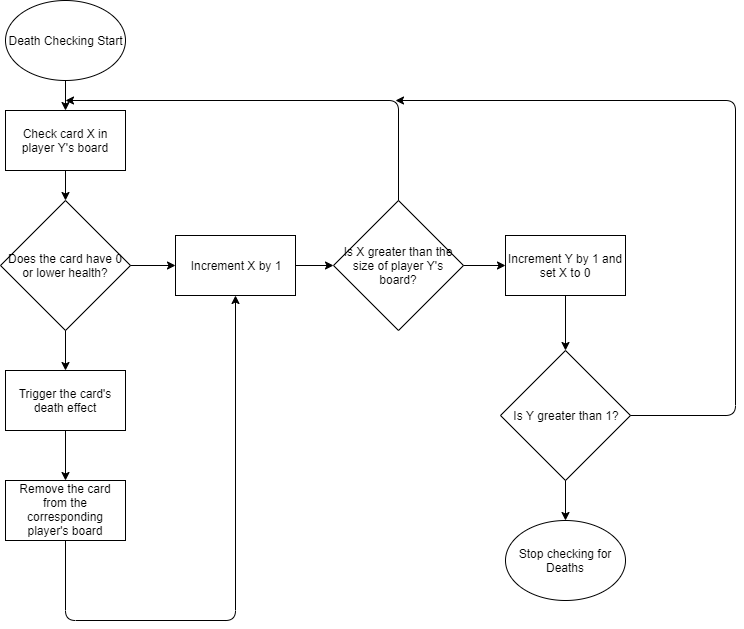
##### Test Plan

|  |  |  |  |
| --- | --- | --- | --- |
| Number | Item | Purpose | Expected Result |
| 1 | Create an object from the parent card class | See if objects can be created and used as cards even when they are not declared as the unique card child classes | A template card is created that can be used as a card functionally but it will not be able to have any differing methods from normal |
| 2 | Create an object from a child card class with every function changed and call each | Make sure that changing each function via polymorphism for a child class works as intended | A card is created with changed played, destroyed, attacking and end functions, when each one is called the intended effect is enacted in the correct manner |
| 3 | Create an object from a child card class with every class variable changed | Make sure that changing each class variable works as intended for the card | A card is created with the default functions but different variables (i.e. different picture, different health values, etc.) |
| 4 | Create an object from a child card class with additional functions such as duplicating themselves | Making sure that adding additional functionality outside of the basic four functions that cards come with works as intended | A card is created with default stats and the default functions other than the newly added function. This function, when called, will perform its task as intended and no error will occur |

### 3) Cards which trigger effects at different intervals

#### 1) On Death

This simultaneously checks whether destroyed cards need to be removed from the board and, if so, triggers their effect by calling a class function that all cards have but is changed via polymorphism to the specific card.



For card X in playerBoard[Y]:

If X.health < 1: // Checks if the card is dead

X.onDeath() // Triggers the card’s death effect

playerBoard[Y].remove(X)

if index(X) > length(playerBoard[Y]) // Checks if it is checking outside the range of the current player’s board

Y++

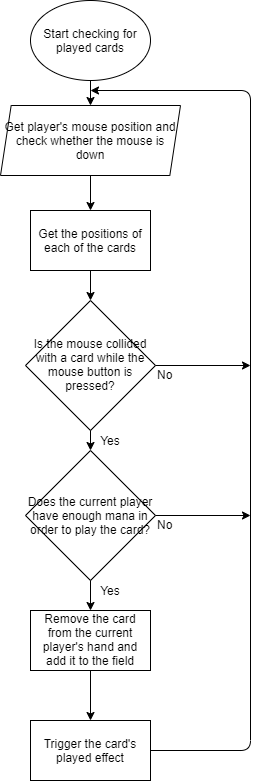
X = 0

##### Test Plan

|  |  |  |  |
| --- | --- | --- | --- |
| Number | Item | Purpose | Expected Result |
| 1 | Checking an empty board | To check if the program tries to remove cards despite lack of any or if there’s an index range error where the max index is 0 | No error will occur, i.e. no cards will erroneously try to be removed and the indices will not exceed their limits |
| 2 | Checking a full board of living cards | To test the edge case where both players’ boards are full, i.e. that no index range error occurs and no cards are removed | No cards are removed as they all are alive (i.e. have health values greater than 0)  No index out of range error occurs. |
| 3 | Checking a full board of dead cards | To test whether the program can effectively remove multiple cards at a time from multiple boards | All cards are removed from both players’ boards |
| 4 | Checking a board with some living and some dead cards | To test whether the program can differentiate between living and dead cards interchangeably | The dead cards are removed from the correct players’ boards and the living cards are ignored |
| 5 | Checking a board with some living and some dead cards of the same name | To test whether the program can differentiate between living and dead instances of the same card and does not remove the living ones erroneously | The living instances of the cards remain untouched and the dead ones are removed, the death effects are triggered only for the cards that died |
| 6 | Checking a board where one player has a full board of cards and the other has an empty board | To check whether the maximum indices of the board length can change in between players | No index out of range error occurs, dead cards are removed and living ones are ignored |
| 7 | Checking a board with intermediate numbers of cards on both players side with some living and some dead | To make sure the program functions outside of purely edge cases | All dead cards are removed and all living cards are ignored |

#### 2)On Play

This checks whether the current player is able to play their card and, if so, triggers the On Play effect unique to that card



For card in playerHand[currentPlayer]:

Position = card.getPosition() // Gets the position of each card

If mouse.touching(Position) and playerMoney > card.cost and mouse.isDown() == True: // Checks if cursor is touching the card and whether the mouse is down. Then subsequently whether the player can afford the card

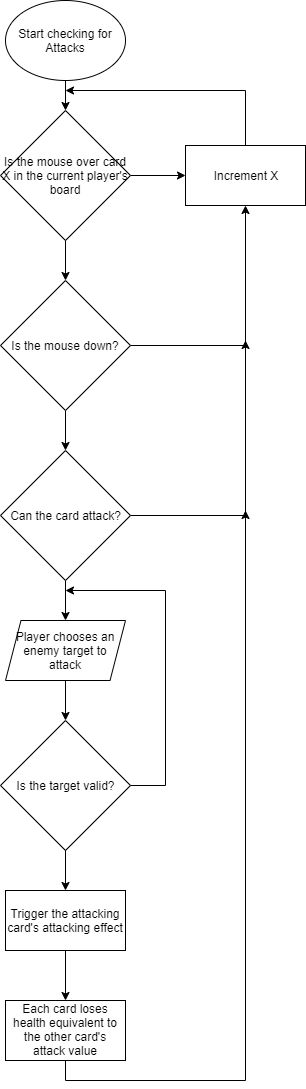
card.play()

card.playedEffect()

##### Test Plan

|  |  |  |  |
| --- | --- | --- | --- |
| Number | Item | Purpose | Expected Result |
| 1 | Attempting to play a card with insufficient mana | Make sure that the mana cost of the card is enforced | The card will not be played and no mana will be deducted. |
| 2 | Playing a card with sufficient mana | Make sure that the correct number of mana is deducted and the card enters the battlefield while triggering its played effect | The corresponding amount of mana to the card’s cost is deducted from the current player, the card is moved from the current player’s hand to the battlefield and its played effect is triggered. |
| 3 | Playing a card with a played effect that affects the board when other cards are populating the board | Make sure the played effect interacts as intended with the other cards on the board | The played effect will activate on all of the cards on the board as intended (i.e. one that deals damage to all cards on the board will deduct X amount of health from all the other cards once) |
| 4 | Playing a card with a played effect that affects the board when no other cards are populating the board | Make sure that the played effect does not break when there is nothing to interact with | The played effect will change nothing (i.e. one that deals damage to all other cards will not change anything or crash the program as there are no other cards to deal damage to) |
| 5 | Attempting to play a card when the current player’s board is full | Make sure nothing changed when this is attempted | The card will remain in the current player’s hand, will not trigger its played effect and will not end up on the current player’s board. |

#### 3)On Attack

Checks whether a card can attack and whether its target is valid and if it can it triggers its attacking effect.

For card X in playerBoard[currentPlayer]:

If mouse touching X and X can attack and mouse down: // Checks if the player is attempting to attack with the card

Attacker = X

For card Y in playerBoard[opposingPlayer]:

If mouse touching Y and Y.isAlive() and mouse down: // Checks whether the targeted card is valid to attack

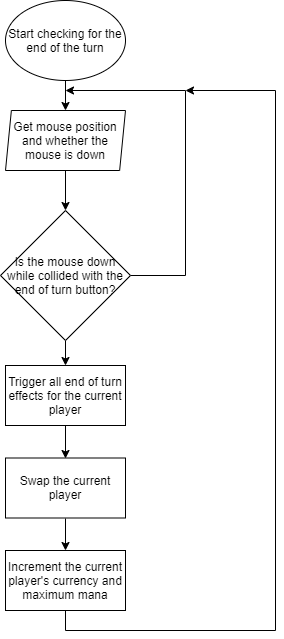
Attacker.attackEffect() // Triggers the attacking effect of the attacking card

Player.attack(Attacker,Y) //Deducts the attack of each card from each other’s healths and inverts X’s canAttack value to false

##### Test Plan

|  |  |  |  |
| --- | --- | --- | --- |
| Number | Item | Purpose | Expected Result |
| 1 | Attempt to attack with a card which cannot attack | Make sure that the condition that the card be able to attack for it to be able to trigger its attacking effect be enforced | The card does not attack any other card, the attack effect does not trigger and no health is deducted from either card |
| 2 | Attempt to attack a friendly card with a card that can attack | Make sure the card cannot attack friendly cards | The card does not attack the friendly card, the attack effect does not trigger and no health is deducted from either card |
| 3 | Attack an opposing card with a card that can attack | Make sure that the health values are correctly reduced for each card and that the correct effects trigger | The health values for each card are reduced by each other’s attack value and the attacking card triggers its on attack effect |
| 4 | Attack an opposing card with a card that has an attacking effect that affects the board | Make sure the effect of the attacking card triggers before the attack occurs and it does not crash if the attacked card is removed pre-attack | The attacking card’s board affecting effect triggers before the actual attack itself, If the opposing card Is destroyed by the effect the attacking card does not have its health deducted. |

#### 4)On Turn End

Checks whether the player has ended the turn and, if so, triggers all end of turn effects for that player before swapping to the other player and incrementing some of their stats.

mouse.getPos()

If mouse down and mouse touching endTurnButton: //Checks if the endTurn button is pressed

For minion in currentPlayerBoard:

minion.endEffect() // Triggers the end of turn effect for each minion in the current player’s board before the player is swapped

currentPlayer.swap() // Swaps the current player

currentPlayerMana += 1

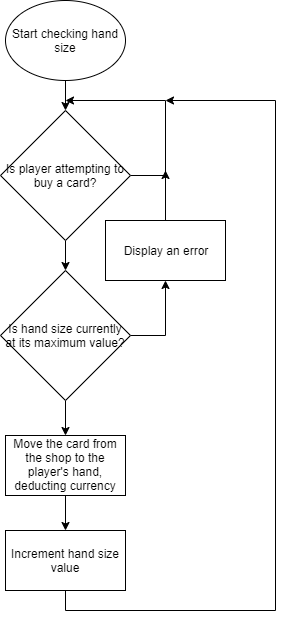
currentPlayerCurrency += 4

##### Test Plan

|  |  |  |  |
| --- | --- | --- | --- |
| Number | Item | Purpose | Expected Result |
| 1 | End the turn when the opposing player has the maximum amount of mana and currency | Make sure ending the turn, which would normally grant currency and mana, does not when these values are full | The opposing player, when the player is swapped, remains on the same amount of mana and currency and does not exceed the set maximum for each. |
| 2 | End the turn with a card in play which has an end of turn effect on the ending player’s board | Make sure the end of turn effect correctly triggers | The end of turn effect on the card would trigger for the ending player, subsequently swapping players |
| 3 | End the turn with multiple different cards in play with end of turn effects | Make sure each end of turn effect triggers as opposed to only one | The end of turn effect of each card in turn triggers once for each card |
| 4 | End the turn with multiple of the same card with end of turn effects | Make sure that the program can differentiate between multiple instances of the same card and still triggers the end of turn effect for each | The end of turn effect triggers for each instance of the card, not missing any out and not triggering more times than the number of instances of the card |
| 5 | End the turn when the opposing player has a card with an end of turn effect | Make sure that the opposing player’s end of turn effects do not trigger when the turn is ended | The end of turn effects of the opposing cards possessing end of turn effects do not trigger |

### 4) Placing limitations on the size of the players’ boards, hands, mana and currency

#### 1) Hand Size

Limits the player’s hand size so that it cannot reach values which would cause the cards to move off the screen display when playing and so that the player cannot infinitely hoard cards.

If player buying card and playerCurrency > cardCost: // Checks if the player Is attempting to buy a specific card and whether the player can actually afford it

If not handSize >= maxHandSize:

buyCard(card) // Removes the card from the shop and adds it to the player’s hand

handSize += 1 // Increments hand size so that buying more cards might be prevented if the hand size is too large

playerCurrency -= card.cost // Deducts the cost of the card from the current player’s currency balance

else:

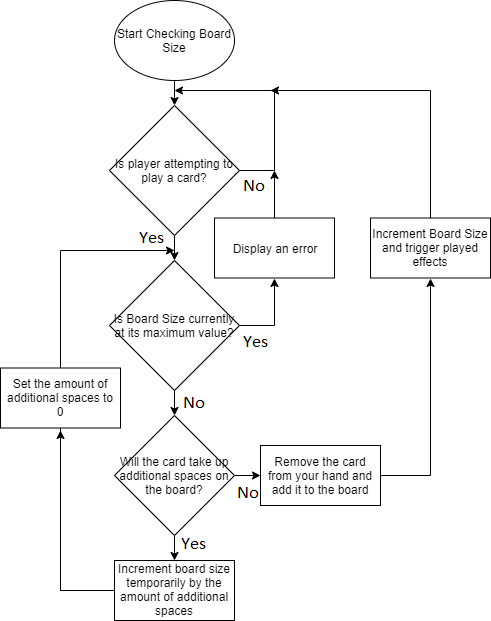
errorDisplay(Your hand is too big for that right now)

##### Test Plan

|  |  |  |  |
| --- | --- | --- | --- |
| Number | Item | Purpose | Expected Result |
| 1 | Try to buy a card with a hand size less than maximum | Check if buying cards functions as normal when the hand size is in a normal value range | The card is removed from the shop and added to the current player’s hand, deducting the correct amount of currency from their balance |
| 2 | Try to buy a card with the maximum hand size | Check that the imposed hand size limit functions for cards in the hand | The card remains in the shop and is not added to the current player’s hand, no currency is deducted and an error is displayed |

#### 2) Board Size

Limits the players’ board sizes so that they cannot reach values that would cause cards to be displayed off the screen.



If player playing Card:

If boardSize >= maxBoardSize:

errorDisplay(“Oops looks like you cant do that right now”)

else:

if additionalSpaces > 0:

boardSize += additionalSpaces // Temporarily increments board size to check against maximum if the card would take up additional spaces

additionalSpaces = 0

if boardSize >= maxBoardSize:

errorDisplay(“Oops looks like you cant do that right now”)

boardSize = length(board) // Resets board length to the correct value from the temporary value assigned by the increase in board length

player not playing Card

else:

play(Card)

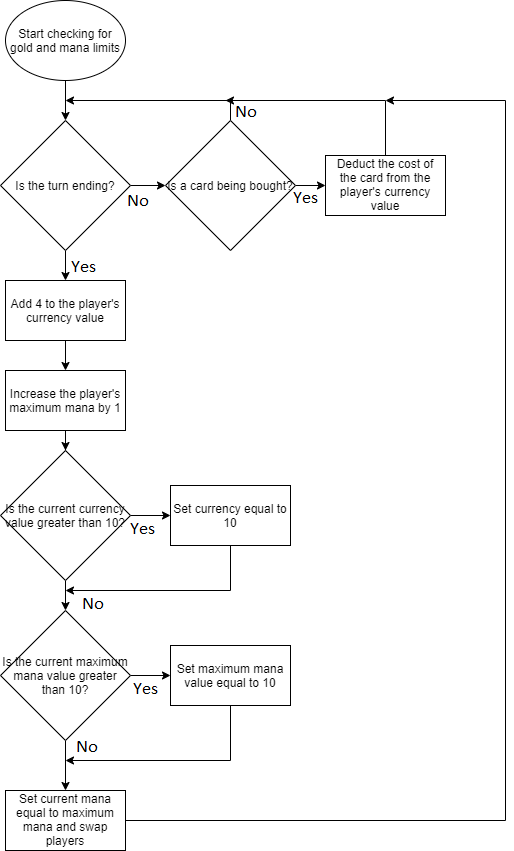
boardSize = length(board) // Updates the length of the board to the correct value after the card has been played

Card.playedEffect()

##### Test Plan

|  |  |  |  |
| --- | --- | --- | --- |
| Number | Item | Purpose | Expected Result |
| 1 | Attempt to play a card which does not take up more than one space with some remaining board space | Make sure standard playing of cards functions correctly. | The card is removed from the player’s hand and added to the board, the board size is incremented |
| 2 | Attempt to play a card which takes up more than one space when you have sufficient board space to hold all copies | Make sure copying cards can function as intended | The card is removed from the player’s hand and added to the board alongside the copies of the card |
| 3 | Attempt to play a card which takes up more than one space when you have exactly one board space open | Make sure that the card which takes up more than one space cannot enter the battlefield if the battlefield without summoning its duplicates | The card is not removed from the player’s hand, no mana is deducted and no on play effects are enacted. An error should be displayed |
| 4 | Attempt to play a card which does not take up more than one space when there is insufficient space on the board to play it | Make sure that the limitations of board space are still enforced for cards which do not take up more than one board space | The card is not removed from the player’s hand, no mana is deducted and no on play effects are enacted. An error should be displayed |

#### 3) Currency and Mana

Limits players’ currency and mana to 10 each so that the entire shop cannot be consistently bought out and the player cannot easily play an entire board worth of cards each turn.

If turn ending: // Checks if the player is attempting to tend the turn

playerCurrency += 4

playerMaxMana += 1

if playerCurrency > 10: // Checks that the updated currency value has not exceeded the set maximum of 10, and if it has it is then reduced to 10

playerCurrency = 10

if playerMaxMana > 10: // Checks that the updated mana value has not exceeded the set maximum of 10, and if it has it is then reduced to 10

playerMaxMana = 10

playerMana = playerMaxMana

player.swap() // Switches the current player as the turn has ended

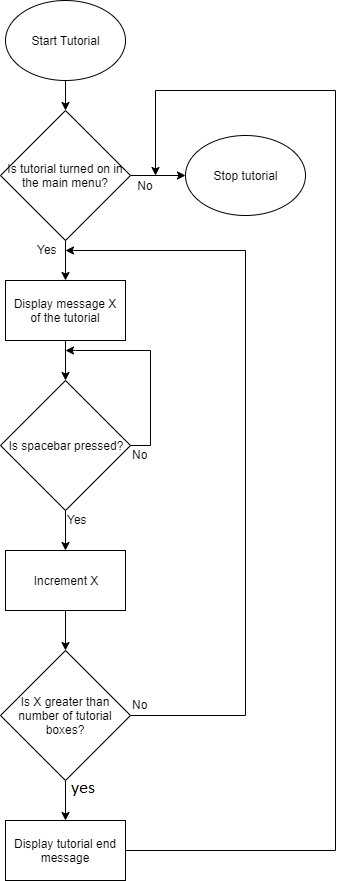
elif card being bought:

playerCurrency -= card.cost

##### Test Plan

|  |  |  |  |
| --- | --- | --- | --- |
| Number | Item | Purpose | Expected Result |
| 1 | End the turn with 0 maximum mana and 0 currency | Make sure that standard incrementing of currency and maximum mana functions correctly | The currency of the player increases by 4, the maximum mana increases by 1 and the current mana is set equal to the maximum mana (i.e. 1) |
| 2 | End the turn on 10 maximum mana and 0 currency | Make sure that the mana limitation functions separately to the currency limitation | The currency of the player is increased by 4 but the maximum mana remains the same at 10, the current mana is set equal to this value, becoming 10 |
| 3 | End the turn on 10 currency and 0 maximum mana | Make sure that the mana limitation functions separately from the currency | The currency of the player remains at 10 and the maximum mana of the player is increased to 1, as is the current mana |
| 4 | End the turn on 10 currency and 10 maximum mana | Make sure that the mana limitation and currency limitation can work simultaneously | The currency of the player remains stagnant at 10 as does the maximum mana and current mana values |
| 5 | Buy a card with sufficient currency | Make sure that buying a card will deduct the correct amount of currency from your current amount | The currency of the player has the cost of the card deducted from it and the card is moved from the shop into the player’s hand |
| 6 | Buy a card with insufficient currency | Make sure that nothing happens if the player tries to buy a card without the sufficient currency | The currency of the player remains at its current value and does not go negative, the card remains in the shop and is not added to the current player’s hand |

### 5) Tutorial Section



While tutorial:  
 display(TutorialMsg[x]) // Displays the current tutorial message

While spacebar not pressed:

Wait // Waits for the spacebar to be pressed to display the next tutorial message

x+=1

if x>length of TutorialMsg -1:

tutorial = False

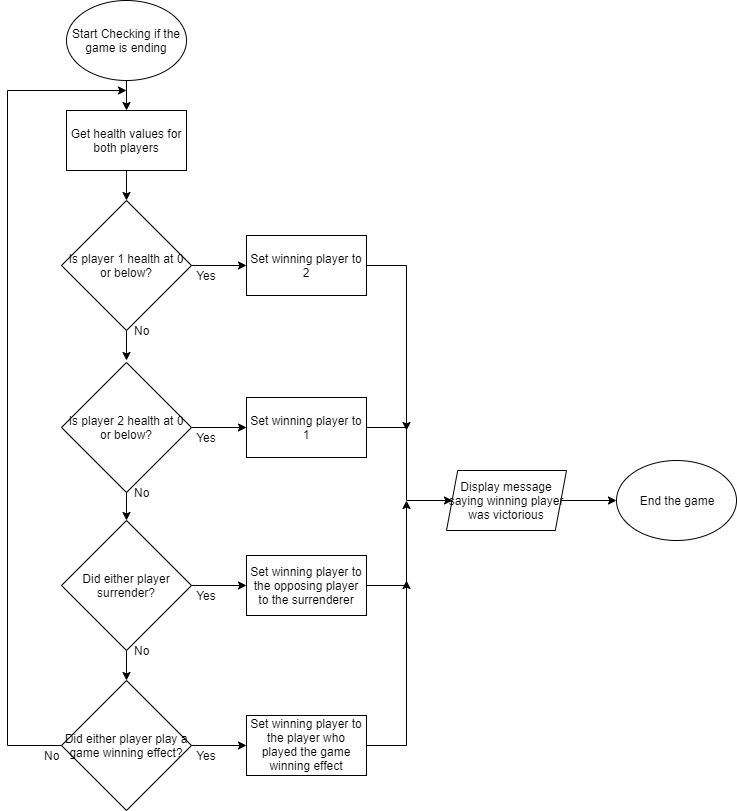
display(endmessage) // Displays a message indicating that the tutorial has ended

end tutorial

##### Test Plan

|  |  |  |  |
| --- | --- | --- | --- |
| Number | Item | Purpose | Expected Result |
| 1 | Activate the tutorial in the main menu and hold down the spacebar | Make sure holding down the spacebar correctly moves through the tutorial messages as if it were pressed repeatedly | The tutorial displays all messages in the correct order in quick succession before closing and displaying the endmessage |
| 2 | Activate the tutorial in the main menu and attempt to play the game while it is open | Make sure the tutorial messageboxes do not impair the playing of the game and It can still fully function while the windows are open | The tutorial displays messages over the screen but the game still functions fully (i.e. you can buy, play and attack cards correctly regardless of the tutorial) |
| 3 | Activate the tutorial in the main menu and iterate through it by pressing the spacebar repeatedly | Make sure that the tutorial functions as intended when used regularly | The tutorial displays messages in the correct order, updating whenever the spacebar is pressed and finishing with an end message before closing |
| 4 | Disable the tutorial in the main menu and press the spacebar repeatedly | Make sure the tutorial does not have any functionality when disabled | No tutorial messages are displayed and when the spacebar is pressed enough times that the end message would normally be displayed it does not display |
| 5 | Disable the tutorial | Make sure that when the tutorial is disabled it does not display | No tutorial message is displayed |

### 6) Ability to end the game when an end condition is reached

Allows the game to be completed when either player is reduced below 0 health or potentially if any other end condition has been met (e.g. a player surrenders or a card which ends the game is played)

playerHealths = getPlayerHealths() // Gets the health values for each player

if playerHealths[0] < 1:

gameWon = True

winner = 1

if playerHealths[1] < 1:

gameWon = True

winner = 0

if surrendered[0]: // Checks if that player has surrendered

gameWon = True

winner = 1

if surrendered[1]:

gameWon = True

winner = 0

if winningEffect in playerBoard[0]: // Checks if that player has played a winning effect

gameWon = True

winner = 0

if winningEffect in playerBoard[1]:

gameWon = True

winner = 1

if gameWon: // Checks if either player has won the game to end it and display a winner message

display(“player ” + winner + “ won”)

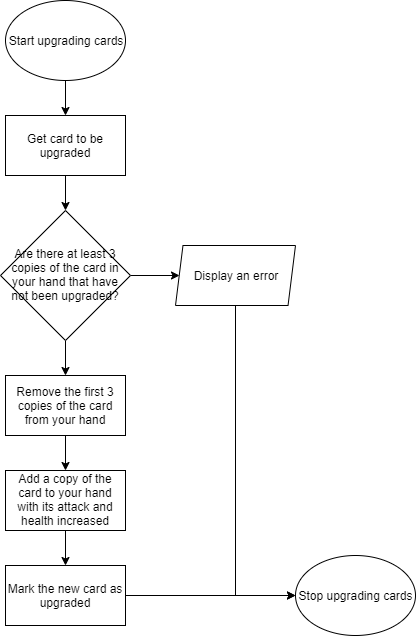
game.end()

##### Test Plan

|  |  |  |  |
| --- | --- | --- | --- |
| Number | Item | Purpose | Expected Result |
| 1 | Reduce each player in turn below 1 health | Test that both situations will end the game and give the opposing player the victory | When player 1 is reduced below 1 health a message will appear stating that player 2 has won the game and the game will end, the same will happen for player 2 except reversed |
| 2 | Reduce both players simultaneously below 1 health | Test that the player who loses when reduced below 1 health is always consistent when they happen simultaneously | When both players are simultaneously reduced below 1 health a message will appear stating that player 1 has won the game regardless of whose turn it currently is and the game will end |
| 3 | Surrender the game with each player in turn | Test that both situations will end the game and give the opposing player the victory | When player 1 surrenders the game a message will appear stating that player 2 has won the game and the game will end, the opposite will happen if player 2 surrendered |
| 4 | Play a winning card with each player in turn | Test that both situations will end the game and give the player who played the card victory | When player 1 plays a winning effect a message will appear that they have won the game and the game will close, the same will happen but in reverse if player 2 were to play one of these cards |
| 5 | Attempt to interact with the game during the brief window in which the winning message is displayed over the board | Test that the game will not function in this window of time | When the user tries to click on cards behind the winning message nothing will happen, the cards will not be played, bought or combined, no effects will trigger nor will any animations. |

### 7) Ability to combine cards to make them more powerful

Allows the player to use extra copies of the cards that they have bought to turn them into much more powerful versions of them to affect the game in a larger way



combineCard = getUpgrade()

For each card in playerHand:

If card == combineCard and card not upgraded: // Loops through each card in the combining player’s hand to check if the card is the same as the card to be upgraded

Count += 1 // Increments the number of combined cards by 1

removeList += card // Adds the card found to the removal list so it can be removed before being re-added in its upgraded form

if Count == 3:

break loop // Exits the loop if count is 3 so that no additional cards get removed past 3

If Count == 3: // Checks that a sufficient amount of cards have been removed

removeCards(removeList)

for each stat in removeList[0] // Doubles the attack and health of the card originally removed from the hand for it to be re-added as more powerful

stat = stat \* 2

removeList[0].upgraded = True // Sets the card re-added as “upgraded” so that it cannot be used in combining cards in future as it has already been combined

playerHand += removeList[0]

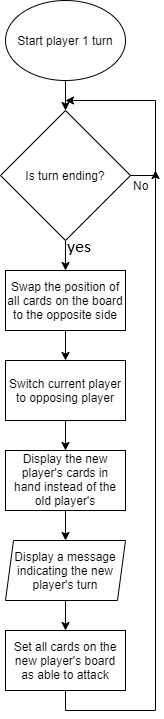
else:

errorDisplay(“Oops, looks like you cant do that right now, you don’t have enough cards”)

##### Test Plan

|  |  |  |  |
| --- | --- | --- | --- |
| Number | Item | Purpose | Expected Outcome |
| 1 | Combine 3 cards of the same type | Test that the basic functionality of combining cards works as intended | The 3 cards are removed from the hand and are replaced by one copy of the card with double the attack and health which is marked as upgraded |
| 2 | Attempt to combine cards with only one of that card in your hand | Test that the combining does not function if you do not have sufficient cards to combine | The card is not removed from your hand, its stats are not doubled and it is not marked as upgraded. |
| 3 | Attempt to combine cards with 2 regular copies of that card and 1 upgraded copy in your hand | Test that combining normal copies of a card and upgraded copies does not function | The cards are not removed from your hand, none of their stats are doubled and the upgraded card remains upgraded whilst the other two remain not upgraded |
| 4 | Attempt to combine 3 upgraded copies of a card | Test that combining 3 identical upgraded copies of a card does not allow you to upgrade them again | The cards are not removed from your hand, no new cards are added to your hand, their stats are not modified, they all remain upgraded |
| 5 | Attempt to combine 3 non upgraded cards with differing stats (i.e. they are the same type but one costs 1 mana less or has 1 more attack from various ingame effects) | Test that combining cards works regardless of the stat changes that have occurred on the card throughout the course of the game | The cards are removed from your hand, the leftmost card in your hand is used as the card from which the stats are doubled. The card is re-added to your hand with the attack and health of the leftmost card in your hand doubled and it is marked as upgraded. |
| 6 | Attempt to combine a card when you have more than 3 copies of it in your hand | Make sure that only 3 copies of the card are consumed when upgrading cards | 3 of the cards are removed from your hand and one is re-added with its stats doubled and marked as upgraded. The rest of the cards in your hand remain untouched (i.e. they are not marked as upgraded, no stats on them are changed and they are not removed from your hand) |

### 8) Implementation of local multiplayer against another human player

Allows the game to switch between players and for the user to play as both player 1 and 2 so that local multiplayer can be achieved

If turn.ending:

For each card in playerBoard[currentPlayer]: // Loops through each card in the current player’s board and swaps it with the card in that position on the opposing player’s board, meaning that their positions visually swap when displayed on turn change

Card.swap(playerBoard[opposingPlayer])

currentPlayer = opposingPlayer // Switches the current player to the opposing player

display(playerHand[currentPlayer]) // Displays the new hand of the current player in place of the hand of the old player

display(“It is now player” + currentPlayer + “’s turn”) // Indicates that the current player has changed over

for each card in playerBoard[currentPlayer]:

card.canAttack = True // Makes each card in the new current player’s board able to attack

##### Test Plan

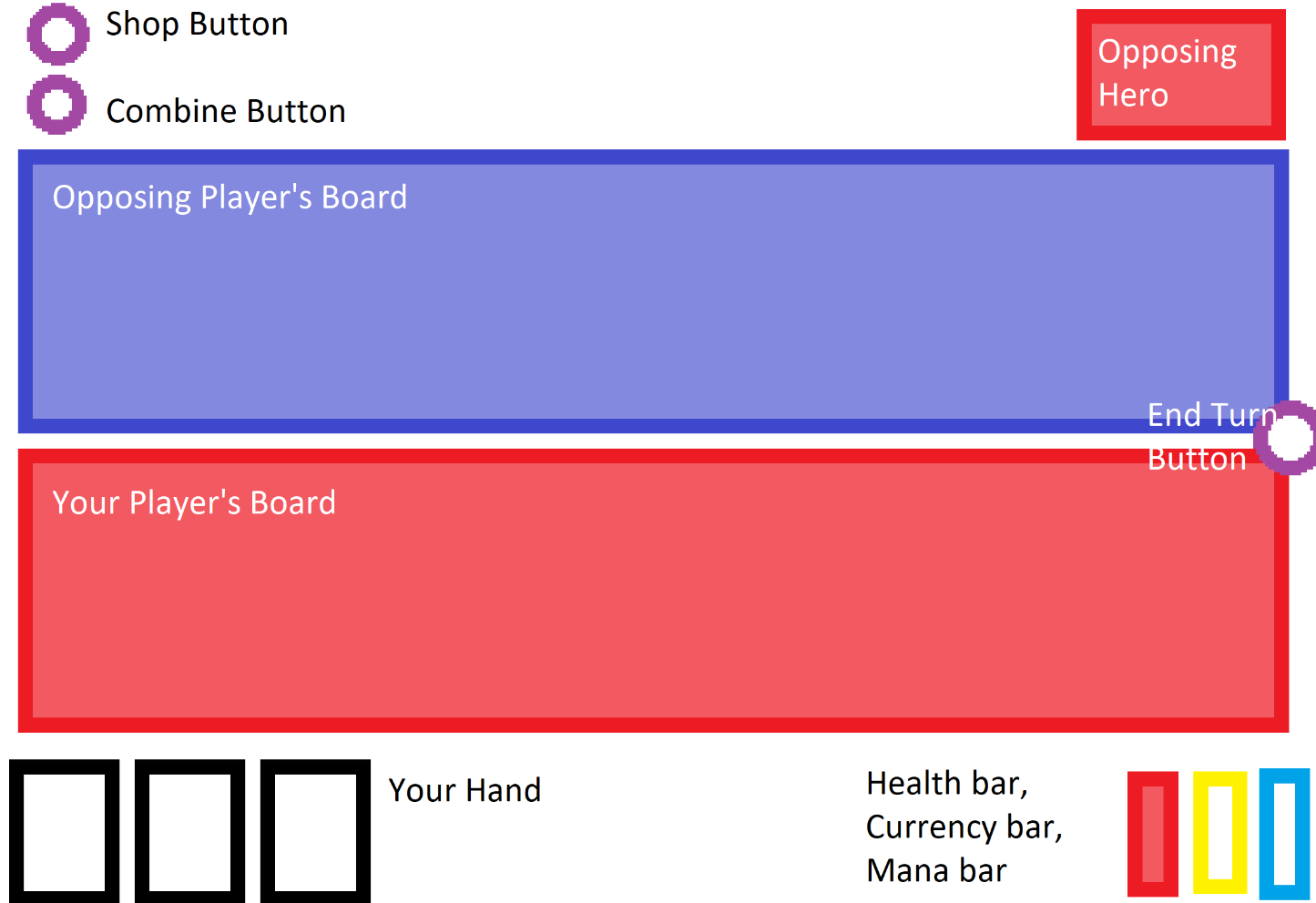
|  |  |  |  |
| --- | --- | --- | --- |
| Number | Item | Purpose | Expected Result |
| 1 | End the turn with no cards on the board | Test that ending the turn works correctly when there’re no items on the board | No cards will swap sides as there are no cards to swap, the rest of the ending of the turn functions as intended |
| 2 | End the turn with the full amount of cards in each player’s board | Test that no board size limits are exceeded when swapping full boards of cards | All cards will swap sides keeping their position on the board without exceeding board length limits or otherwise crashing the program. All other parts of the turn ending function as intended. |
| 3 | End the turn with a different amount of cards on each player’s board | Test that swapping a card with a blank space works as intended | All cards swap with the opposite position on the board, whether they be populated by a card or not. All other parts of the turn ending function as intended. |

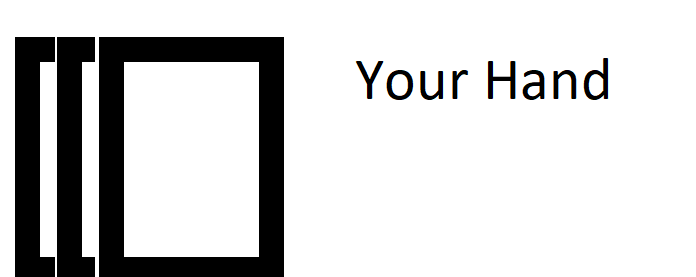
## Design of Screen layouts

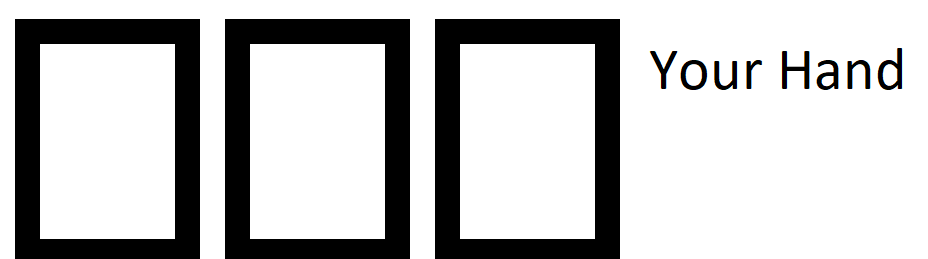
### Main Board

The basic screen of the game should look as follows, with a clear colour difference indicated on the board so players can tell themselves apart.

The colours red and blue were chosen as they contrast the best and are very clearly recognisable when they change positions.

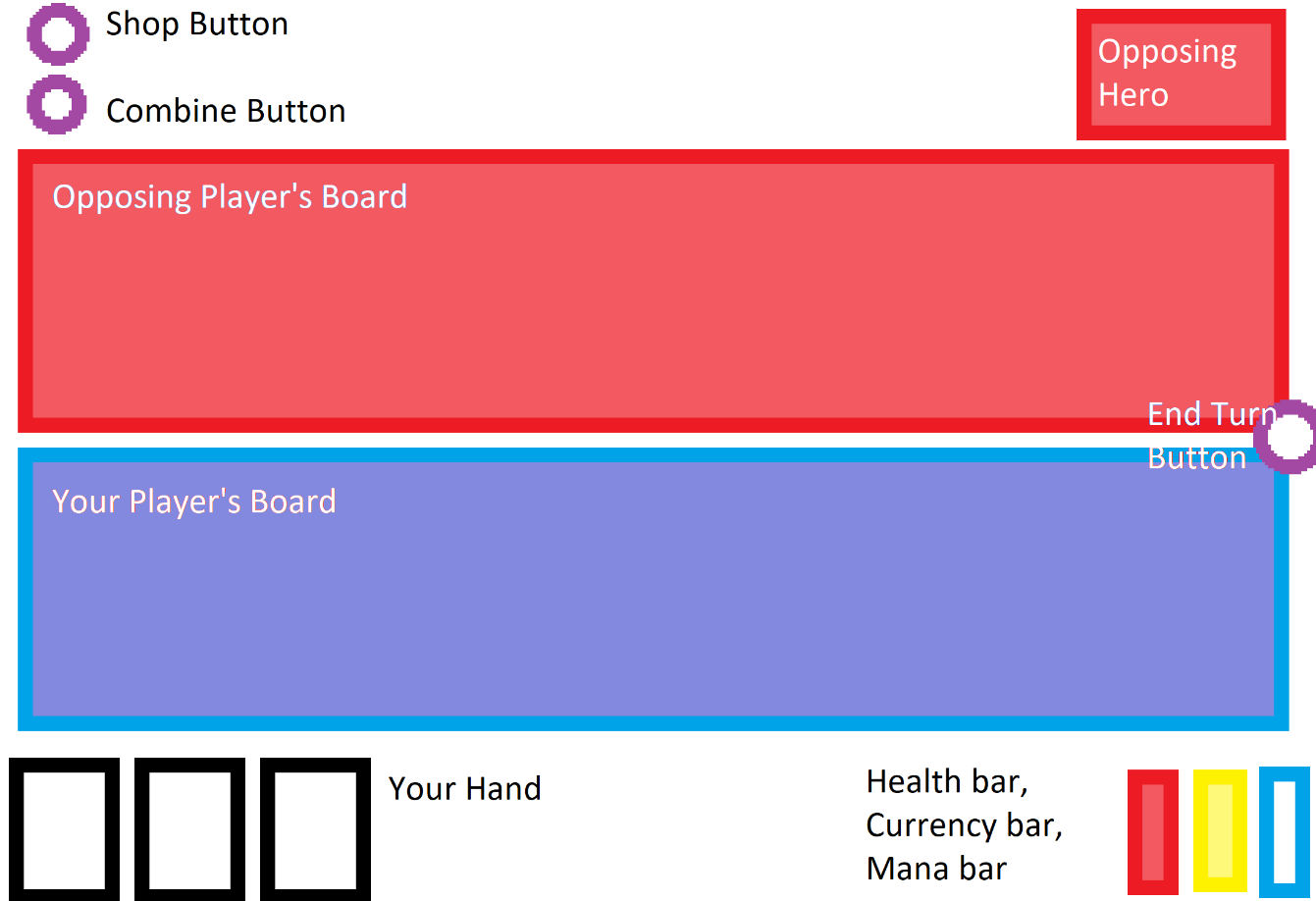


The current player’s cards should be displayed at the bottom right of the screen and should fan out so they are easier to interact with when the mouse moves near, moving from close together as you would naturally hold them in your hand as shown below:

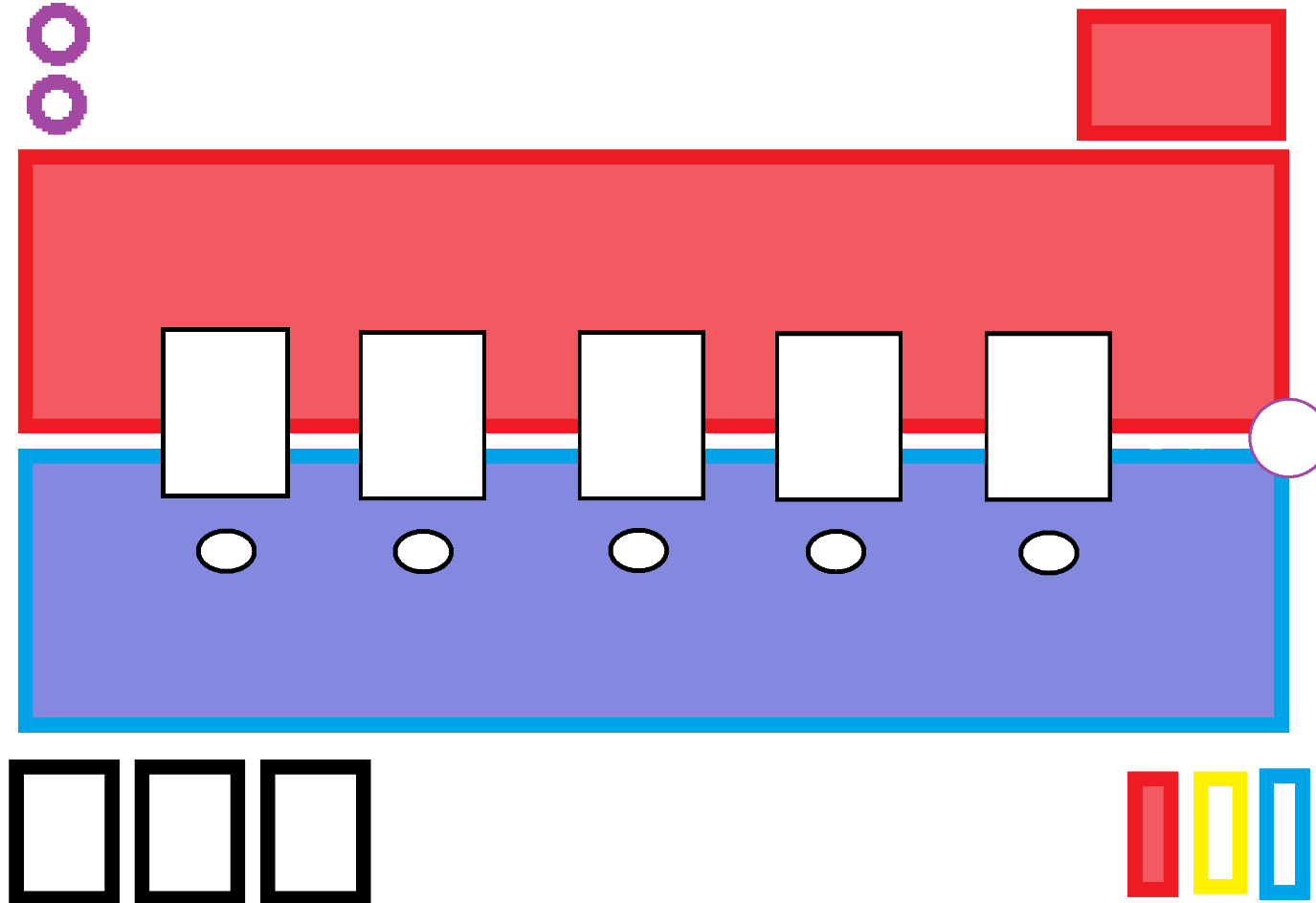
To spread apart so that they can be more easily interacted with in a digital medium as such:

This is done so that cards will not initially clutter the screen and will feel more real as they respond to your mouse’s movement.

The current player’s health should be displayed at the bottom right alongside that player’s currency and mana values. This will switch whenever the current player switches. The colours of each side of the board will swap on turn end so that the current player can be known more easily as the image below shows:

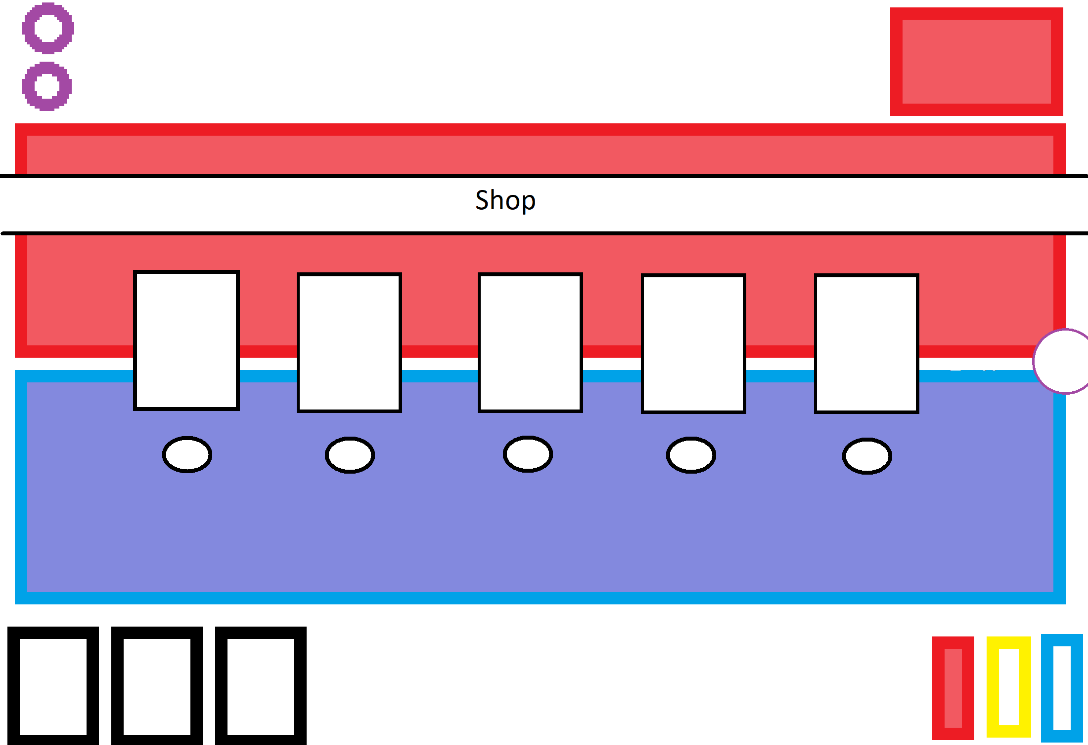


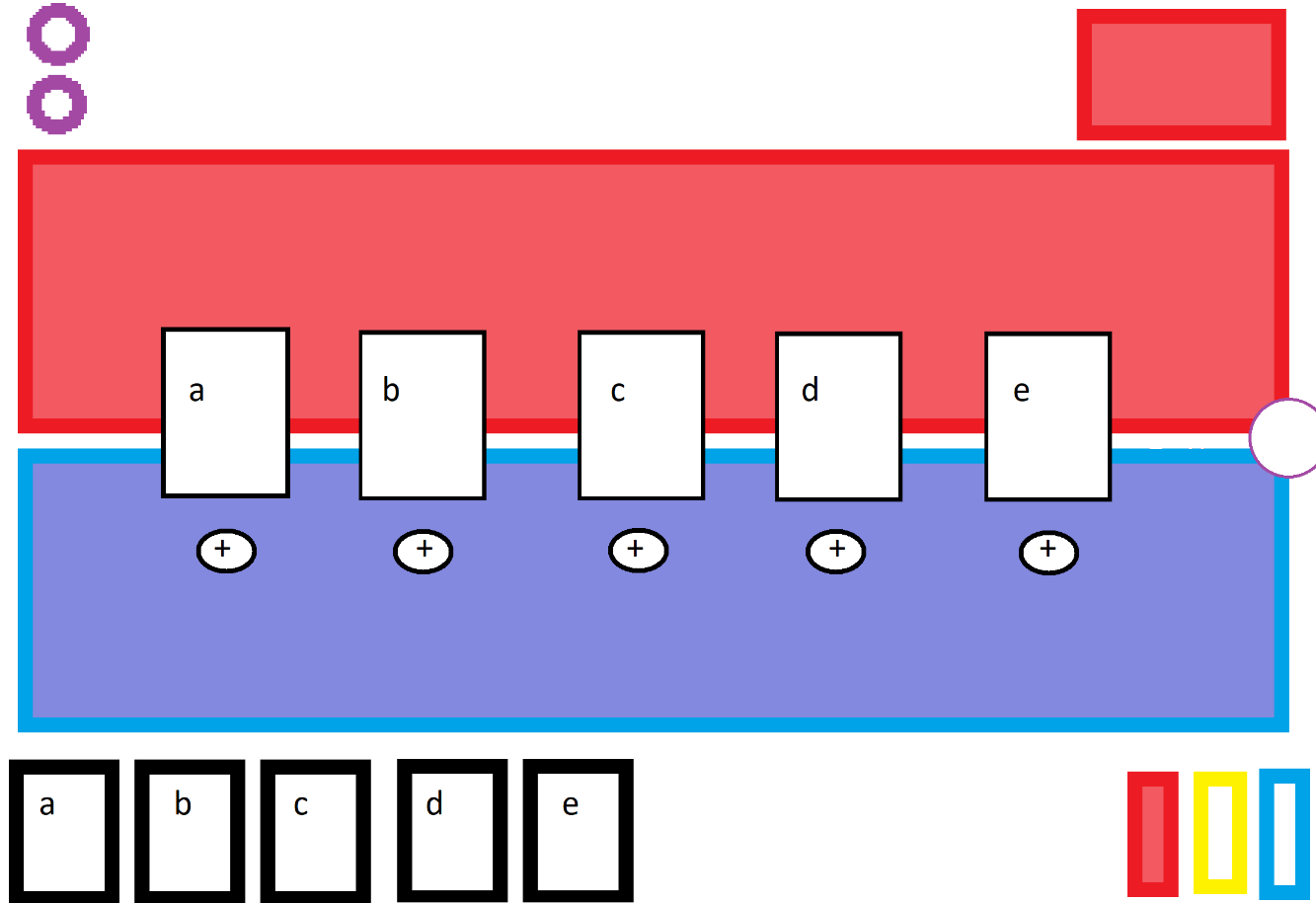
When the shop button (the top right button on the screen) is clicked the shop will be opened as follows

Where the white rectangles in the middle of the board represent the cards that can be bought and the ovals underneath them represent buttons that can be used to buy them.

These cards that can be bought vary each turn for each player or a new button which appears next to the shop button can be clicked to refresh the items within the shop. This button is initially not there so as to prevent confusion for players as it will not display any visible change if it were to be pressed when the shop was closed as its only function is to change the cards currently available in the shop.

These buttons are separate from the cards themselves to make it clear to the player what the buttons will do instead of just having unlabelled cards hovering in the centre of the screen.

The shop will be closeable with the same shop button in which case the board will look as it originally did before the shop was rendered. Another possible design is to have a banner stating that the shop is open above the shop as follows in order to be more obvious for the player:

When the combine button is pressed the combine screen will open. If the shop is currently open at that time the shop will close and the combine screen will open in its place. When opened it looks as follows:

Where the cards a, b, c, d and e in the player’s hand correspond to the cards a, b, c, d and e in the shop. The buttons labelled + underneath the cards are the buttons to combine their corresponding card (i.e. the card directly above them) and are done this way so that clicking on the cards in the combine screen does not have any unintended effects such as playing the copy of that card present in your hand. This also serves to make the buttons more distinct from the shop buttons and prevents unnecessary confusion on the side of the player.

### Main Menu

For the main menu only a few items are required so the screen should be fairly simple, with bright buttons to stand out from the background and no complex sub-menus to navigate within it so the user has no room for confusion.



The tutorial button toggles the tutorial messages on and off (these are displayed when the game is launched on the top of the screen). When pressed its colour changes to indicate that it has been highlighted as follows:

The button starts un-highlighted with a black border and a dark blue background

The button’s background colour then brightens and its border becomes white instead of black



The reason that the background both brightens and the border changes colour is so that there is a clearer indication if it has been pressed on systems with lower brightness levels or to users with worse eyesight.

The “Vs Player” and “Vs AI” buttons are mutually exclusive (i.e. you cannot both be playing against another player and an artificial intelligence simultaneously) so when one is pressed the other is switched off and vice versa. The game cannot be played when neither has been highlighted so one button is selected as the default mode that will be considered pressed when the program is launched.

The “Play” button is the largest as it is the one that the user must press to launch the game itself, it is also a different colour from the other 3 buttons to allow it to stand out from them. When it is pressed a similar effect occurs, turning from orange with a black border like this:



To having a white border and text with a lighter background as shown below:

This is only displayed briefly as the game will launch immediately but it allows the user to rest assured they pressed the button and do not need to repeatedly press it in the event that it takes a second or so to launch.

Another possibility is to have, instead of a distinctive colour for the play button, a distinctive shape. This serves a similar purpose and makes for a more unique menu experience.

For example a hexagonal play button would be ideal for differentiating it from the other buttons due to its distinctly different shape. This type of button may end up looking good in isolation but when combined with other shapes not end up looking as good as I would like.

# Implementation

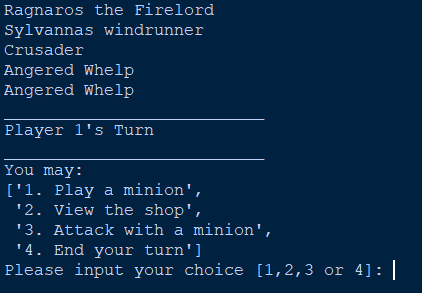
When starting the implementation I initially made the game text based in order to test out functionality instead of jumping straight into a pygame based iteration.

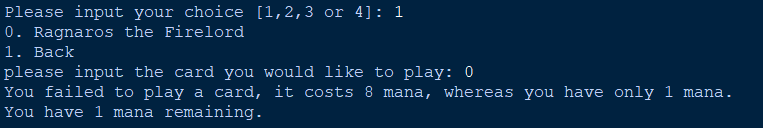
## Game Loop

### First Iteration

1. **while** **not** done:
2. **print**("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\nPlayer " + str(player.currentPlayer) + "'s Turn\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\nYou may:")
3. pprint.pprint(["1. Play a minion","2. View the shop", "3. Attack with a minion", "4. End your turn"])
4. choice = 0
5. **while** choice **not** **in** [1,2,3,4]:
6. choice = int(input("Please input your choice [1,2,3 or 4]: "))
7. **if** choice == 1:
8. counter = 0
9. **for** i **in** player.playerHand[player.currentPlayer-1]:
10. **print**(str(counter) + ". " + i.name)
11. counter += 1
12. **print**(str(counter) + ". Back")
13. playChoice = 999
14. **while** playChoice **not** **in** range(0,len(player.playerHand[player.currentPlayer-1])+1):
15. **try**:
16. playChoice = int(input("please input the card you would like to play: "))
17. **except**:
18. **print**("invalid choice")
19. #print(len(player.playerHand[player.currentPlayer-1]))
20. **if** playChoice != len(player.playerHand[player.currentPlayer-1]):
21. player.play(player.playerMana[player.currentPlayer-1],playChoice)

In the first iteration I created a basic game loop with 4 options, I used a heavily class based implementation, with variables and functions in the player class allowing cards to be played. The variable player.playerHand is a list which stores all the cards that are currently in each player’s hand which I used to choose a card to be played using the player.play() function. If the player chose “1” as their option the names of each card in the player’s hand would be displayed alongside an index and the player would be allowed to choose one of the cards displayed to attempt to play (or choose to go back and not play anything). Once the user selects a valid card to play the game will call the player.play() function which will verify whether the player has enough mana to play the card needed and if so then it will be moved from the player’s hand to the board.





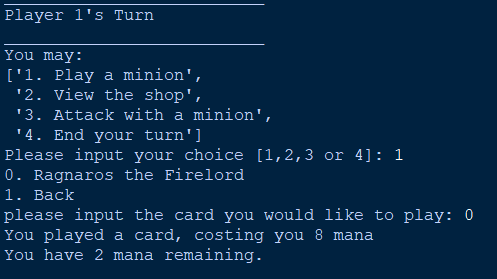
Upon testing out the code for playing cards like this I quickly realized that while in theory it should work correctly I had no way of gaining enough mana to play the cards I needed to play as my endTurn function in the player class had not yet had mana gain implemented.

1. **def** endTurn(self):
2. #Executes the end of turn functions of the cards in play
3. **for** card **in** playerBoard[0]:
4. card.executeFunction(card.endFunc, 1)
5. **for** card **in** playerBoard[1]:
6. card.executeFunction(card.endFunc, 2)
7. #Changes the current player
8. self.playerCurrency[0] += 3
9. self.playerCurrency[1] += 3
10. self.currentPlayer = (self.currentPlayer % 2)+1
11. self.genCards(5)

this was easily changed by simply adding in the lines

1. self.playerMana[0] += 1
2. self.playerMana[1] += 1

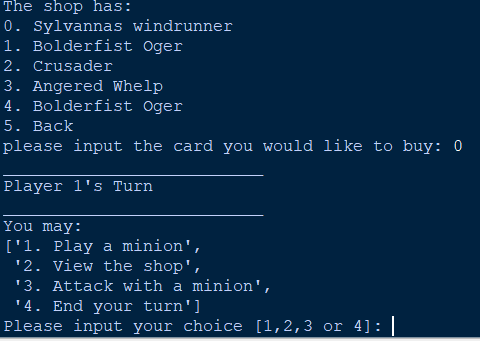
which, while not the desired end result for my mana gain mechanic allowed me to test the playing function thoroughly and confirm that it did in fact function correctly.



For the second section of the main game loop used very similar methods to the first, looping through each card in the shop to display them next to an index from and allowing you to choose which one you would like to purchase by typing in the index of the selection you would like. The try: except: statement in the validation is to make sure the program will not crash if the user inputs a string as it tries to convert the user’s input into an int, which cannot happen if it is not a number.

1. **elif** choice == 2:
2. counter = 0
3. **print**("The shop has:")
4. **for** i **in** player.forSale:
5. **print**(str(counter)+".",i.name)
6. counter += 1
7. **print**(str(counter) + ". Back")
8. purchaseChoice = 255
9. **while** purchaseChoice **not** **in** range(0,len(player.forSale)+1):
10. **try**:
11. purchaseChoice = int(input("please input the card you would like to buy: "))
12. **except**:
13. **print**("invalid choice")
14. **if** purchaseChoice != len(player.forSale):
15. player.buyCard(purchaseChoice)

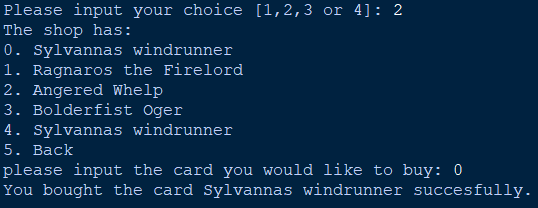
This called a function player.buyCard which validated that the player had sufficient currency to buy the card rather than this validation occurring within the game loop in order to improve readability.



There was, however, an issue. Even though the card was bought there was no indication that anything at all had happened and so I updated the buyCard() function in the player class to include a method of showing what happened.

1. **def** buyCard(self, cardPos):
2. **if** self.playerCurrency[self.currentPlayer-1] >= self.forSale[cardPos].shopCost:
3. #Adds the purchased card to the purchasinc player's hand and removes it from the shop if they have enough currency to buy it
4. self.playerHand[self.currentPlayer-1].append(self.forSale.pop(cardPos))
5. **print**("You bought the card " + self.playerHand[player.currentPlayer-1][len(self.playerHand[player.currentPlayer-1])-1].name + " succesfully.")
6. **else**:
7. **print**("Oops, looks like you dont have enough gold to purchase that card right now!")

The added print statement displaying the name of the latest card in the player’s hand successfully managed to function in the way I intended, displaying the card purchased by the player.



This addition, while not important to the fully pygame-based iteration later was extremely useful in debugging and the function of the first iteration as a useable product.

The third section of the initial game loop is slightly more complicated than the first two

1. **elif** choice == 3:
2. counter = 0
3. **print**("You have " + str(len(player.playerBoard[player.currentPlayer-1])) + " minions, of which:")
4. **for** i **in** player.playerBoard[player.currentPlayer-1]:
5. **if** i.canAttack:
6. **print**(i.name)
7. **print**("Can attack")
8. **for** i **in** player.playerBoard[player.currentPlayer-1]:
9. **print**(str(counter) + ". " + i.name)
10. counter += 1
11. **print**(str(counter) + ". Back")
12. attackChoice = 255
13. **while** attackChoice **not** **in** range(0,len(player.playerBoard[player.currentPlayer-1])+1):
14. **try**:
15. attackChoice = int(input("please input the card you would like to attack with: "))
16. **except**:
17. **print**("invalid choice")
18. enemyChoice = 255
19. **if** player.currentPlayer == 2:
20. playerSwap = 1
21. **else**:
22. playerSwap = 2
23. counter = 0
24. **for** i **in** player.playerBoard[playerSwap-1]:
25. **print**(str(counter) + ". " + i.name)
26. **while** enemyChoice **not** **in** range(0,len(player.playerBoard[playerSwap-1])+1):
27. **try**:
28. enemyChoice = int(input("please input the card you would like to attack: "))
29. **except**:
30. **print**("invalid choice")
31. **if** attackChoice != len(player.playerBoard[player.currentPlayer-1]):
32. player.attack(player.playerBoard[player.currentPlayer-1][attackChoice],player.playerBoard[playerSwap-1][enemyChoice])

This time almost all of the validation was done before the attack function was called as in this text based version higher numbered choices did different things, such as being the back button instead of a valid attack target. This meant that much more validation was needed and as two separate inputs were required, one for the attacking card and one for the card to be attacked, this simply cannot be done later in an efficient manner.

Firstly it loops through all cards for which “canAttack” is True in the current player’s board, displaying the name of each, the player then chooses which of the cards they want to attack with (or alternatively they can select “back” to cancel). When a valid selection has been made here the names of all possible enemy targets are printed and the player can then choose which of them they want to attack. Once this has been selected the player.attack() function is called to make the first selected card attack the second.

[\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Add](file:///\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Add) testing here\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

### Intermediate Iteration (Iteration 1.5)

1. cards = [
2. Ragnaros(),
3. Sylvannas(),
4. Thaurissan(),
5. Crusader(),
6. Whelp(),
7. Ogre()
8. ]
9. done = False
10. **while** **not** done:
11. #Start testing loop
12. mousepos = pygame.mouse.get\_pos()
13. player.screen.fill((0,0,0))
14. #Fill background with black ^
15. player.drawCard(mousepos,cards[0])
16. player.drawCard((10,10),cards[1])
17. #Test drawing multiple cards ^
18. pygame.display.update()
19. #Update screen ^
20. pygame.event.pump()
21. #Handle events ^

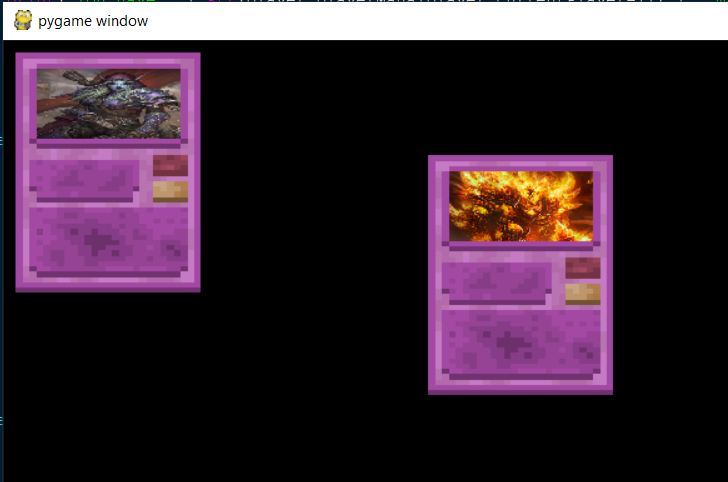
In this iteration I transitioned between text based and visual gameplay so while doing so I arrived at an iteration in which I tested all the aspects of the visual medium without the restrictions of the game’s ruleset which is why I am not calling this iteration 2. The game was not functional during this iteration but rather this was paving the way so that making the game functional in the long term would be easier.

As such this iteration of the game loop is very short and simplistic, simply filling the background with black and getting the player’s mouse position followed by rendering 2 cards. The first card is rendered statically in the top left corner of the screen and the other is rendered at your mouse’s position on the screen to test out the movement of cards and see whether it operated smoothly.

The “pygame.event.pump()” line is necessary as I have no need for pygame’s events but the module operates in a way in which if your events are not handled the window will crash as it thinks that nothing is happening. This line informs pygame that the game is still running but the events are not needed.

This new testing iteration introduces many new functions to allow for the most efficient drawing of cards possible, paving the way for drawing the many that will be needed for the actual game itself.The first iteration of the drawCard function is extremely simplistic, only drawing a static image of the card in the desired location with a picture to distinguish them on top.

1. **def** \_\_init\_\_(self, cardList):
2. self.playerHealth[0] = 25
3. self.playerHealth[1] = 25
4. self.playerHand[0] = []
5. self.playerBoard[0] = []
6. self.playerBoard[1] = []
7. self.playerHand[1] = []
8. self.playerMaxMana[0] = 1
9. self.playerMaxMana[1] = 1
10. self.playerMana[0] = 1
11. self.playerMana[1] = 1
12. self.globalCardList = cardList
13. self.currentPlayer = 1
14. self.forSale = []
15. self.playerCurrency[0] = 0
16. self.playerCurrency[1] = 0
17. self.cardImage = pygame.image.load("C:\\Users\\Gabriel\\Documents\\GitHub\\Donnan-Project\\Images\\Card.png")
18. self.cardImage = pygame.transform.scale(self.cardImage, (148, 192))
19. #Player class function for drawing a specific card in a desired location
20. #Needs ability to draw text and stats on
21. **def** drawCard(self, location, card):
22. self.screen.blit(self.cardImage,location)
23. self.screen.blit(card.picture,(location[0]+17,location[1]+13))



The cardImage variable stores the background card image used for all cards and as such it is stored In the player class but each card has their own picture to distinguish them which is stored within their subclass so that they can be easily separated out and new images can be added easily when a new card subclass is added.

The images for each card are stretched to fit the correct size regardless of the size or shape of the original image, meaning it will always fit but incorrectly shaped images (i.e. images with width to length ratios significantly different to the width to length ratio of the image section of the cards) will end up looking stretched, which is the compromise that has to be made to ensure consistency.

1. **class** CardBase:
2. **def** \_\_init\_\_(self,shopCost,name, mana, attack, health, picture):
3. #for all the func variables the input is a block of text which is passed into generic functions containing only an exec block, this saves me from having to write hundreds of new functions and allows for creations of new cards extremely quickly
4. #The function text defaults to a function that does nothing
5. self.canAttack = False
6. self.name = name
7. self.mana = mana
8. self.health = health
9. self.attack = attack
10. self.shopCost = shopCost
11. #Declare picture and transform it to the correct size so that it overlaps the correct section of the card background image
12. self.picture = pygame.image.load(picture)
13. self.picture = pygame.transform.scale(self.picture, (115, 56))

When images are the incorrect shape for cards they will look like this



The second image is a perfect square and the first is taller than it is wide, leading to pixilation and horizontal stretching in each. It is possible to get vertical stretching as well if the card was significantly wider than it was tall however this is an unusual image shape.

## Player Class

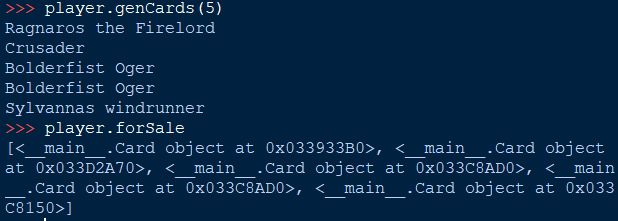
### First Iteration

1. **class** Player:
2. playerHealth = [None,None]
3. playerHand = [None,None]
4. playerBoard = [None,None]
5. playerMana = [None,None]
6. playerCurrency = [None,None]
7. **def** \_\_init\_\_(self, cardList):
8. self.playerHealth[0] = 25
9. self.playerHealth[1] = 25
10. self.playerHand[0] = []
11. self.playerBoard[0] = []
12. self.playerBoard[1] = []
13. self.playerHand[1] = []
14. self.playerMana[0] = 10
15. self.playerMana[1] = 10
16. self.globalCardList = cardList
17. self.currentPlayer = 1
18. self.forSale = []
19. self.playerCurrency[0] = 20
20. self.playerCurrency[1] = 20

The player class is what manages most of the gameplay and as such is used many times in the game loop. It is initialized with a health value for each player, what is in each player’s hand and board, each player’s starting mana value and currency value alongside other things. The list “cardList” contained declared versions of each card object so they could be more easily generated by the genCards() function

1. **def** genCards(self, amount):
2. displaylist = []
3. #randomly generates (amount) cards from the globalCardList (list of all possible cards) and prints their name, will display them for purchase later and replace card drawing
4. **for** i **in** range(amount):
5. displaylist.append(random.choice(self.globalCardList))
6. self.forSale = displaylist
7. **for** i **in** self.forSale:
8. **print**(i.name)

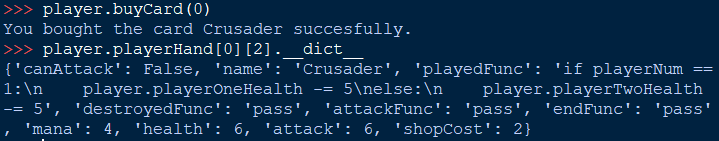
This function randomly chooses (amount) times from the globalCardList class variable and appends them to a list “displayList” which replaces the items in the shop and then prints each item in it once completed.



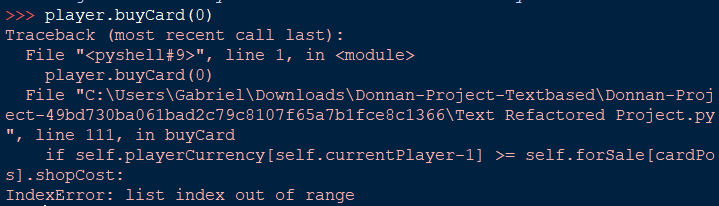
The items in the shop are Card objects as this makes moving them from the shop to your hand and from the hand to the board while acting as they should (i.e. calling played functions) extremely easy.

1. **def** buyCard(self, cardPos):
2. **if** self.playerCurrency[self.currentPlayer-1] >= self.forSale[cardPos].shopCost:
3. #Adds the purchased card to the purchasinc player's hand and removes it from the shop if they have enough currency to buy it
4. self.playerHand[self.currentPlayer-1].append(self.forSale.pop(cardPos))
5. **print**("You bought the card " + self.playerHand[player.currentPlayer-1][len(self.playerHand[player.currentPlayer-1])-1].name + " succesfully.")
6. **else**:
7. **print**("Oops, looks like you dont have enough gold to purchase that card right now!")

The buyCard() function follows on logically from the genCards() function, taking in a card position of the item you want to buy in the shop and checking if you have sufficient currency to buy it, if you do then the card is moved from the player.forSale[] list to the player.playerHand[] list for the current player, printing a message depending on whether you are able to successfully do it.



However upon testing I found that if you tried to buy a card larger than the maximum index of the hand an exception would occur (this most often happened when trying to buy the first card after everything had already been bought)



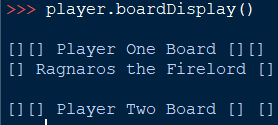
As such I edited the code to include the statement

**if** player.forSale[cardPos]:

at the beginning of the function to check whether the index exists in the forSale list to prevent crashes such as these

1. **def** boardDisplay(self):
2. #Update this with pygame stuff later, simple visualiser for logic for now
3. **print**("\n[][] Player One Board [][]")
4. **for** i **in** self.playerBoard[0]:
5. **print**("[] " + i.name + " []")
6. **print**("\n[][] Player Two Board [] []")
7. **for** i **in** self.playerBoard[1]:
8. **print**("[] " + i.name + " []")

The player.boardDisplay() function was relatively simple and was called after every change in the board state, displaying the names of the cards on each player’s board. It takes no inputs as it’s function is always the same



The attack function is significantly more complicated

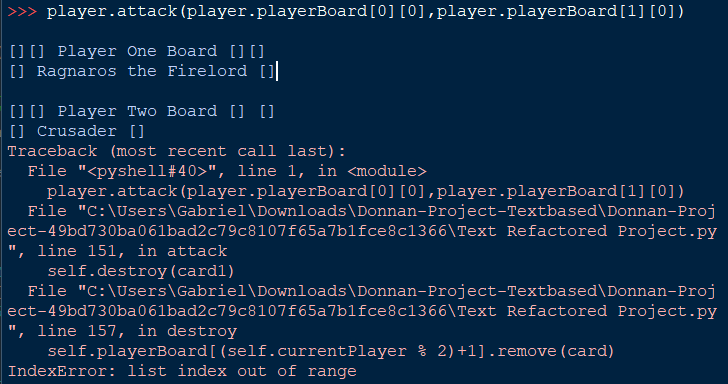
1. **def** attack(self, card1, card2):
2. self.boardDisplay()
3. card1.executeFunction(card1.attackFunc, self.currentPlayer)
4. card2.health -= card1.attack
5. card1.health -= card2.attack
6. **if** self.currentPlayer == 1:
7. **if** card1.health <= 0:
8. self.destroy(card1, 1)
9. **if** card1.health <= 0:
10. self.destroy(card2, 2)
11. **else**:
12. **if** card1.health <= 0:
13. self.destroy(card1, 2)
14. **if** card1.health <= 0:
15. self.destroy(card2, 1)

It takes in the arguments “card1” and “card2”, where “card1” is the attacker and “card2” is defending. For the attacking card it calls the card.executeFunction() function on its attack function, effectively calling a function specific to each card that occurs when they attack affecting the board in different ways depending on their attackFunc variable.

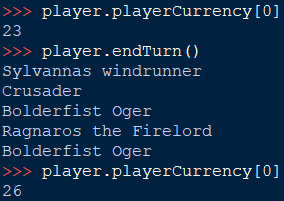
The health of each card is reduced by the attack of the other card and each card’s health is checked after this to call player.destroy() on them if they have less than 1 health.

1. **def** destroy(self,card, player):
2. card.executeFunction(card.destroyedFunc, player)
3. **if** player == 1:
4. (self.playerBoard[0]).remove(card)
5. **else**:
6. (self.playerBoard[1]).remove(card)

Player.destroy() takes a card and player input to remove the card from a specified player’s board and calls card.executeFunction on the card’s destroyedFunc to perform a specific action to each card that happens when they die. The player input is taken so that the program knows which player’s board to remove the card from, in an earlier implementation I did not have this and as such it crashed whenever a friendly card was destroyed on your turn.



The player.endTurn() function is called at the end of each player’s turn to enact the changeover from one player to the other

1. **def** endTurn(self):
2. #Executes the end of turn functions of the cards in play
3. **for** card **in** self.playerBoard[0]:
4. card.executeFunction(card.endFunc, 1)
5. **for** card **in** self.playerBoard[1]:
6. card.executeFunction(card.endFunc, 2)
7. #Changes the current player
8. self.playerCurrency[0] += 3
9. self.playerCurrency[1] += 3
10. self.currentPlayer = (self.currentPlayer % 2)+1
11.     self.genCards(5)

This is done relatively simply by performing the modulus 2 on the self.currentPlayer variable and incrementing by one. It also triggers the endFunc for each card in each player’s board and increases the currency of each player by 3 and calls player.genCards() so that new cards may be bought each round. The function requires no inputs as it is the same in all situations in its functionality.

The final function in the first implementation of the player class was player.play()

1. **def** play(self, mana, cardPos):
2. card\_played = self.playerHand[self.currentPlayer-1][cardPos]
3. #Checks if the card you're trying to play costs too much
4. **if** card\_played.mana <= mana:
5. self.playerBoard[self.currentPlayer-1].append((self.playerHand[self.currentPlayer-1]).pop(cardPos))
6. card\_played.executeFunction(card\_played.playedFunc,1)
7. mana -= card\_played.mana
8. **print**("You played a card, costing you " + str(card\_played.mana) + " mana")
9. **else**:
10. **print**("You failed to play a card, it costs " + str(card\_played.mana) + " mana, whereas you have only " + str(mana) + " mana.")
11. **print**("You have " + str(mana) + " mana remaining.")
12. **return** mana

This takes a mana and cardPos value as inputs and returns a mana value at the end after the card has either been played or the player has failed to play the card. The function uses the cardPos variable to find the position of the card the player wants to play within their hand, this makes playing cards easier when using text inputs as the user can simply input an index to play.

The function then checks the mana input against the mana cost of the card, if the mana input is higher than or equal to the cost of the card the card is popped from the player’s hand and appended to their board, triggering their playedFunc and producing a text output. The removal of the card from the player’s hand and adding it to the board are done in the same line, first popping the card from the player’s hand then immediately adding it to the board so that it is effectively instantaneous and to make sure that no logic can be programmed between these points as it would be one action in the real world.

### Intermediate Iteration (Iteration 1.5)

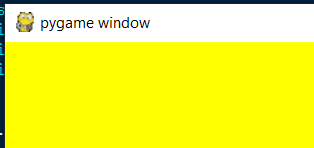
There are relatively few changes to the player class in this iteration, only adding a function to draw cards on the screen and a variable containing the background image of cards.

These have been covered in the game loop section as they tied in directly to the limited function of the testing as there is no more complicated logic required for the loop itself in this iteration.

Archaic functions in which text was output were removed so that the game can be completely visually based, these were replaced by placeholder functions which will later be developed into pygame based implementations of their previous functions.

One example of this which will later be made to draw all the cards on both player’s side of the board in bulk is:

1. #Placeholder, will be made to draw cards on board later
2. **def** drawScreen(self):
3. #Fills screen with solid colour and updates screen, cards should be done between these later
4. screen.fill((255,255,0))
5. pygame.display.update()



Currently all the function does is make the screen solid yellow.

The card drawing function in this iteration started as below, only drawing the card background and card image.

1. **def** drawCard(self, location, card):
2. self.screen.blit(self.cardImage,location)
3. self.screen.blit(card.picture,(location[0]+17,location[1]+13))

It was later modified to draw the attack and health values of the cards, the size of these does not need to be automatically scaled as they are never practically large enough numbers to fall off the edge of the card. The names of the cards are also drawn here but they cannot be too long or they will extend too far off the card

1. **def** drawCard(self, location, card):
2. #centres the image of the card instead of using the top left
3. location = (location[0]-74,location[1]-96)
4. self.screen.blit(self.cardImage,location)
5. self.screen.blit(card.picture,(location[0]+17,location[1]+13))
6. self.screen.blit(card.nameText,(location[0]+17,location[1]+86))
7. self.screen.blit(card.hpText,(location[0]+112,location[1]+79))
8. self.screen.blit(card.atkText,(location[0]+112,location[1]+100))

on line 3 the location value is changed so that it represents the centre of the card rather than the corner, meaning that placing the card on the screen in the correct location is easier and more intuitive. The

## Card class

### First iteration

1. **class** Card:
2. **def** \_\_init\_\_(self,shopCost,name, mana, attack, health, playedFunc = "pass", destroyedFunc = "pass", attackFunc = "pass", endFunc = "pass"):
3. #for all the func variables the input is a block of text which is passed into generic functions containing only an exec block, this saves me from having to write hundreds of new functions and allows for creations of new cards extremely quickly
4. #The function text defaults to a function that does nothing
5. self.canAttack = False
6. self.name = name
7. self.playedFunc = playedFunc
8. self.destroyedFunc = destroyedFunc
9. self.attackFunc = attackFunc
10. self.endFunc = endFunc
11. self.mana = mana
12. self.health = health
13. self.attack = attack
14. self.shopCost = shopCost

The Card class is for creating objects out of each card so they can be played and interacted with meaningfully with outside of simply being variables. They are initially defined relatively simply, taking the inputs shopCost, name, mana, attack, health, playedFunc, destroyedFunc, attackFunc and endFunc and assigning each to a variable for the object. Each variable ending in Func was made so that no subclasses would have to be created for the cards, instead allowing specific functions to be passed in as text which was passed to an exec() block whenever they need to be called. Each of these was defaulted to “pass” as if they were called without a value the program would crash as it would try and execute an empty variable.

The function to execute the “func” variables is simply one line

1. **def** executeFunction(self,text, playerNum):
2. **exec**(text)

It takes in a variable “playerNum” alongside the text to be executed. This function may appear to do nothing but it actually is used in the cards’ “func” variables when run to determine what side of the board they are on.

The cards were declared as such

1. cards.append(Card(4,"Ragnaros the Firelord" ,8,2,8,playedFunc = """
2. for i in player.playerBoard[player.currentPlayer-1]:
3. i.health -= 8"""))
4. cards.append(Card(3,"Sylvannas windrunner",6,5,5, destroyedFunc = """if playerNum == 1 and player.playerBoard[1]:
5. (player.playerBoard[0]).append((player.playerBoard[1]).pop(random.randint(0,len(player.playerBoard[1])-1)))
6. elif player.playerBoard[0] and playerNum == 2:
7. player.playerBoard[1].append(random.choice(player.playerBoard[0]).pop)"""))

With the “func” variables written as if they were code within multi line strings so that they can be executed. This unfortunately made debugging much harder as if a bug were discovered within one of these variables the error text from the traceback would not be as informative as usual and as such during testing this proved to be a less efficient method of going about the card class.

### Second Iteration

In the second iteration the cards are implemented each as their own subclass of the overarching “CardBase” class and as such the initialization statements are much simpler

1. **class** CardBase:
2. **def** \_\_init\_\_(self,shopCost,name, mana, attack, health):
3. self.canAttack = False
4. self.name = name
5. self.mana = mana
6. self.health = health
7. self.attack = attack
8. self.shopCost = shopCost

This implementation eliminated all the “func” variables, meaning that significantly fewer variables had to be initialized and that all the functions would be much more easily debugged, however this did come with some side effects in terms of the length of my code, requiring significantly more lines per card declaration.

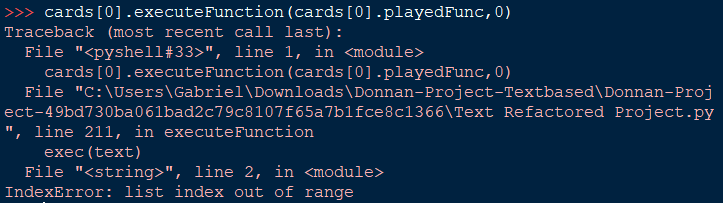
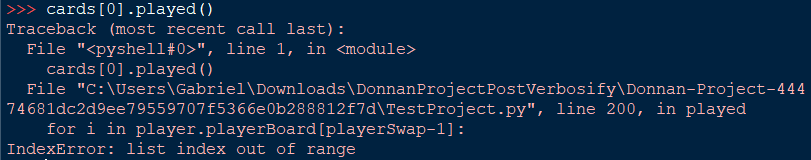
1. **def** played(self):
2. **pass**
4. **def** destroyed(self):
5. **pass**
7. **def** attacking(self):
8. **pass**
10. **def** end(self):
11. **pass**

In the “CardBase” superclass each possible function that the individual cards could have are declared with only “pass” as their code and through polymorphism the subclasses change this when initialized. The purpose of this is so that if a card were not to have a function that triggers when it is destroyed but the code tries to call it it could run the empty function with pass in it instead of crashing the program. For example for the card “Ragnaros” it has no “destroyed” function but the program will continue to run without crashing if this is called, not changing anything.



For the subclasses they each have one simple line in the \_\_init\_\_ function and the subclass nature mostly serves to allow the cards to have polymorphed versions of the CardBase class’s functions.

1. **class** Ragnaros(CardBase):
2. #All init statements for CardBase subclasses are extremely similar, merely passing in the values needed
3. **def** \_\_init\_\_(self):
4. CardBase.\_\_init\_\_(self, 5, "Ragnaros", 8, 2, 8)
6. **def** played(self):
7. #Deals 8 damage to all cards on the opposing side of the board
8. playerSwap = (player.currentPlayer % 2)+1
9. destroyed = []
10. **for** i **in** player.playerBoard[playerSwap-1]:
11. i.health -= 8

It is now much simpler to call the functions of the cards and most importantly proper tracebacks can be created if a bug is discovered in testing. 

Whilst the exec implementation recognises that a list index is out of range it cannot point out where within the string the error ocurred, making debugging significantly more tedious.

Using the new class-subclass implementation also makes the initialisation statements for the cards significantly simpler.

1. cards = [
2. Ragnaros(),
3. Sylvannas(),
4. Thaurissan(),
5. Crusader(),
6. Whelp(),
7. Ogre()
8. ]

All the cards can be initialized to a list in a single statement with no parameters needing to be passed in to any of them whereas the func variable implementation required much messier, longer statements such as

1. cards.append(Card(2,"Emperor Thaurissan",6,5,5, endFunc = """if playerNum == 1:
2. for i in player.playerOneHand:
3. i.mana -= 1
4. else:
5. for i in player.playerTwoHand:
6. i.mana -= 1"""))

which were extremely difficult to read and understand.

### Intermediate Iteration (Iteration 2.5)

This iteration’s changes are very minor, only adding images for each card

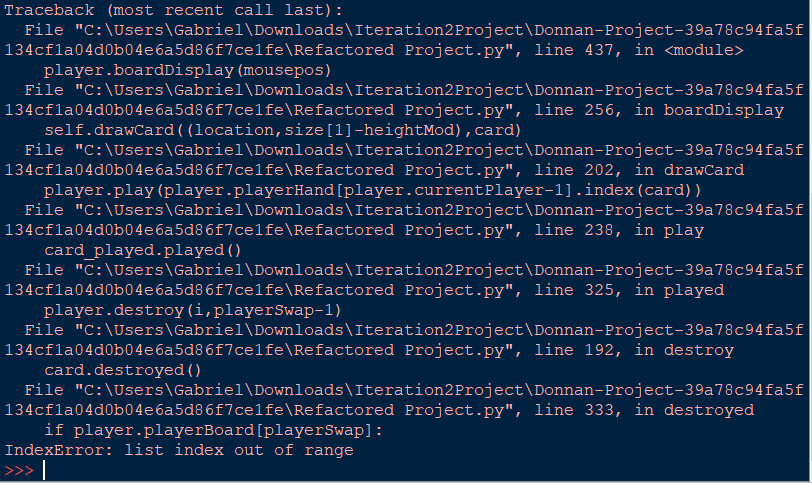
1. self.picture = pygame.image.load(picture)
2. self.picture = pygame.transform.scale(self.picture, (115, 56))

These images are all scaled regardless of size to 115 pixels by 56 pixels so that they fit correctly with the card background image stored in the player class.

The card text whilst still stored in the card subclasses is only displayed later in this iteration as the player class’s drawing function only includes this later.

# STUFF TO ORGANISE



In certain situations with in which the card “Ragnaros” was played while sufficient amounts of the card “sylvannas” populated each side of the board the game would be caused to crash due to the “on play” effect of ragnaros no longer being able to find the cards it is looking for as they had already swapped sides of the board and would result in a lengthy error message of: 

Which proved extremely difficult to debug as the issue is with the trigger order of the death effect and played effect coinciding meaning that both could not activate correctly and either it would be impossible to steal a card from the opposing side or it would be impossible to deal damage to a card that has already been stolen, effectively leading to the layering of two possible errors.

I debated removing the problematic cards from the game so that the error could not occur but I decided against it as the underlying issues causing the errors would still remain, meaning that the cards I wanted to add in the future may encounter similar game breaking bugs which would mean I had to resolve this regardless.

I added a relatively primitive game loop, enabling the purchasing and playing of cards as well as ending the turn.

1. **while** **not** done:
2. #player.screen.blit(cards[0].picture,(10,110))
3. mousepos = pygame.mouse.get\_pos()
4. player.screen.blit(boardPicture,(0,0))
5. shopButton.draw()
6. endButton.draw()
7. closeButton.draw()
8. #player.drawCard(mousepos,declaredCards[0])
9. **if** endButton.clickRect.collidepoint(mousepos) **and** pygame.mouse.get\_pressed()[0]:
10. endButton.press()
11. time.sleep(0.1)
12. **if** shopButton.clickRect.collidepoint(mousepos) **and** pygame.mouse.get\_pressed()[0]:
13. shopButton.press()
14. time.sleep(0.1)
15. **if** closeButton.clickRect.collidepoint(mousepos) **and** pygame.mouse.get\_pressed()[0]:
16. closeButton.press()
17. time.sleep(0.1)
18. **if** shopButton.pressed:
19. shopButton.displayCards()
20. **for** i **in** shopButton.buttons:
21. temprect = pygame.Rect(i.coords,(60,27))
22. **if** temprect.collidepoint(mousepos) **and** pygame.mouse.get\_pressed()[0]:
23. i.press()
24. time.sleep(0.1)
25. #player.drawCard((10,10),cards[1])
26. player.boardDisplay(mousepos)
27. pygame.display.update()
28. **for** event **in** pygame.event.get():
29. **if** event.type == pygame.QUIT:
30. done = True

The loop simply draws the board background image on the screen at the beginning of every loop, meaning that all other drawn effects will be on top of it. It then draws each button which the player can interact with in its respective spot, followed by checking for each button whether the player is clicking on it.

Here I use “pygame.mouse.get\_pressed()[0]” for each individual if statement, which is very inefficient as it means that the function will have to be called multiple times with each loop through the game loop even though it only has to be called once as the player’s mouse position will not change that significantly in between checking each button and as such it can be stored as a variable and used that way. This is used in a later implementation.

This game loop supports the opening of the shop which draws additional buttons onto the screen. As such the if statement which checks each of the individual shop buttons is nested within the if statement checking if the shop is opened, meaning that they cannot be accidentally clicked whilst the shop is closed. The rects for each of these buttons is assigned based on the amount of cards in the shop every loop rather than being set on the opening of the shop meaning that the shop will automatically take up as little space as possible as purchased cards will no longer take up any space and their button will be removed rather than staying in the same place and having to re close and open the shop to update the positions.

1. #This list stores the references to the classes in order that new objects can be created instead of duplicating old ones, meaning that specific instances of objects can be changed
2. cards = [
3. Ragnaros,
4. Sylvannas,
5. Thaurissan,
6. Crusader,
7. Whelp,
8. Ogre
9. ]
11. #This list is to help speed up graphical things as it will not have to repeatedly declare the objects to render
12. declaredCards = [
13. Ragnaros(),
14. Sylvannas(),
15. Thaurissan(),
16. Crusader(),
17. Whelp(),
18. Ogre()
19. ]
21. player = Player(cards)
22. #Gives the second player a small headstart as they are naturally at a disadvantage due to how turn based games work
23. player.playerCurrency[1] += 2

26. #For any variable "player" or "playerNum" within a function this refers to the player \*in control\* of the thing making the effect, not necessarily the player being affected
28. ##\_\_\_\_\_\_\_\_\_\_\_\_\_\_ MAIN GAME LOOP \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_##
29. player.playerHand[0].append(cards[0]())
30. player.playerHand[0].append(cards[1]())
31. player.genCards(5)
32. player.playerMana[0] = 100
33. player.playerMana[1] = 100
34. player.playerMaxMana[0] = 100
35. player.playerMaxMana[1] = 100
36. player.playerCurrency[0] += 100000000
37. player.playerCurrency[1] += 10
38. done = False
39. shopButton = ShowShop()
40. closeButton = CloseGame()
41. endButton = EndTurn()
42. boardPicture = pygame.image.load("C:\\Users\\Gabriel\\Documents\\GitHub\\Donnan-Project\\Images\\Board.png")
43. boardPicture = pygame.transform.scale(boardPicture, (player.screen.get\_width(), player.screen.get\_height()))

In order for the game loop to function many variables have to be set before it, the lists “cards” and “declaredCards” for example may appear to be identical in function but they are declared this way in order to fix a bug in which all cards purchased from the shop were treated as the same instance and so whenever one was destroyed all other copies of it were also. This is circumvented by the shops using the “declaredCards” list to display them and then if a card were bought the regular “cards” list counterpart was subsequently declared and added to the player’s hand so a brand new instance is created. The “declaredCards” list is used instead of the “cards” list for all displaying though as creating new instances of cards for each card loop becomes extremely quickly memory inefficient.

If these lists were not declared none of the main loop’s displaying would function for any card as the cards would only exist as classes but would never be instantiated.

1. **def** boardDisplay(self,mousepos):
2. size = self.screen.get\_size()
3. #For drawing contents of hand
4. location = 115
5. **if** mousepos[1] > 550 **and** **not** shopButton.pressed:
6. locFactor = 180
7. heightMod = 115
8. **else**:
9. locFactor = 56
10. heightMod = 25
11. **for** card **in** self.playerHand[self.currentPlayer-1]:
12. self.drawCard((location,size[1]-heightMod),card)
13. location += locFactor
14. #For drawing contents of board
15. **if** **not** shopButton.pressed:
16. location = 180
17. **for** card **in** self.playerBoard[self.currentPlayer-1]:
18. self.drawCard((location,size[1]-330),card)
19. location += 180
20. location = 180
21. playerSwap = (player.currentPlayer % 2)+1
22. **for** card **in** self.playerBoard[playerSwap-1]:
23. self.drawCard((location,size[1]-575),card)
24. location += 180

In order to replace the old text based display formatting a new function had to be created in order to draw all of the items on the board at once. It does this by getting the size of the screen on which the game is being displayed then scaling all positions to a percentage of this size so that they remain in the correct locations if the game were scaled to a screen of a different size (such as 1440p vs standard 1080p). The first if statement detects whether the mouse is in the lower half of the screen so that if so the cards can be spread out while displayed so that they are more easily interacted with, this is done by increasing the “locFactor” which scales the distance between the cards.

This will now only happen if the shop is closed so that there is no overlap between the cards in the shop and the cards in your hand (which happened when the shop being closed was not a condition) which is confusing for the user and provides no additional functionality as playing cards while the shop is open is not possible so additional spacing between them is useless.

1. **def** play(self, cardPos):
2. card\_played = self.playerHand[self.currentPlayer-1][cardPos]
3. #Checks if the card you're trying to play costs too much
4. **if** card\_played.mana <= player.playerMana[player.currentPlayer-1] **and** len(player.playerBoard[player.currentPlayer-1])<8:
5. self.playerBoard[self.currentPlayer-1].append((self.playerHand[self.currentPlayer-1]).pop(cardPos))
6. card\_played.played()
7. player.playerMana[player.currentPlayer-1] -= card\_played.mana

The played function remains largely similar to its previous iteration with the key difference that the input for mana is no longer required, the function will instead use the class variable player.playerMana[current player] as the initial value to check whether the card is playable. If it is then this value can be updated directly by the function instead of returning a mana value which could then be used to update the player’s total. This makes the program’s logic easier to follow as the mana will always be updated within this function rather than have to have extra statements written outside of it whenever it is called for the mana to be updated.

1. **def** attack(self, card1, card2):
2. **if** card1.canAttack:
3. card1.attacking()
4. card2.health -= card1.attack
5. card1.health -= card2.attack
6. card1.canAttack = False
7. **if** self.currentPlayer == 1:
8. **if** card1.health <= 0:
9. self.destroy(card1, 1)
10. **if** card1.health <= 0:
11. self.destroy(card2, 2)
12. **else**:
13. **if** card1.health <= 0:
14. self.destroy(card1, 2)
15. **if** card1.health <= 0:
16. self.destroy(card2, 1)
17. **else**:
18. **print**("That minion cannot attack right now")

In this iteration attacking is programmatically implemented such that cards can be made to attack each other via code inputs, however this is not yet functional ingame. The reason for this is that in order for the game to be completely visually based new functions will need to be written in order to indicate whether a card can attack, if the card is currently attacking and to display a new cursor to show that the player can target the card’s attack to whatever card the mouse is currently pointing at. This is necessary as visuals are not as instantaneous as text and as such there is a time between the player selecting the first card to attack with and the second card which they desire to attack so this cannot directly be input to one function such as this with a “card1” and “card2” input all at once.

1. **def** combineCards(self, card):
2. combinationCounter = 0
3. removed = []
4. **for** i **in** player.playerHand[player.currentPlayer-1]:
5. **if** i.name == card.name **and** i.mana == card.mana **and** i.attack == card.attack **and** i.health == card.health:
6. removed.append(i)
7. combinationCounter += 1
8. **if** len(removed) < 3:
9. **print**("Error, you do not have enough of these cards to combine")
10. **else**:
11. **for** i **in** removed:
12. player.playerHand[player.currentPlayer-1].remove(i)
13. player.playerHand[player.currentPlayer-1].append(player.upgradeCard(removed))

The “combineCards” function is similarly programmatically functional to the attack function in this iteration, however it does not yet have it’s visual implementation as that requires some infrastructure to be built around it. It will require a section with a corresponding button similar to the shop in which all the cards in your hand are displayed centrally to the screen to indicate that they can be upgraded. These will need buttons beneath them to be pressed to confirm that the user wants to upgrade that specific card and as such it is once again not as simple as the text based implementation as the function will need to be called by a button tied to a card rather than the user typing in the command themselves. As such the function has been updated to accommodate for this, now taking a card object as an input rather than an index so that once the infrastructure has been created it will be easy and intuitive to call the function.

######## ITERATION 1. 5 ########

1. **class** Button:
2. **def** \_\_init\_\_(self, coords, size, picture):
3. self.coords = coords
4. self.picture = pygame.image.load(picture)
5. self.picture = pygame.transform.scale(self.picture, size)
7. **def** press(self):
8. **pass**

The button class is relatively simple as a base class, taking in a coordinate, size and picture. It loads the image passed into it into a format representable by pygame and transforms it so it is the size passed in. The class has an empty “press()” function which is changed by the subclasses via polymorphism to do what each subclass of Button needs for their purposes. In this iteration there is no implementation of actually drawing the buttons as I planned to implement other functionality of the game before putting in pressable buttons that the user can see.

1. **class** EndTurn(Button):
2. **def** \_\_init\_\_(self):
3. Button.\_\_init\_\_(self, (10,10),(10,15),"C:\\Users\\Gabriel\\Documents\\GitHub\\Donnan-Project\\Images\\EndTurn.png")
5. **def** press(self):
6. player.endTurn()

An example of a subclass of button is EndTurn, this subclass will always be in the same place, be of the same size and have the same image so the constructor simply initialises the superclass with set values for each of these. The press function is changed so that instead of doing nothing it calls the function “player.endTurn()” to swap the current player so that the other player can have their turn.

##### ITERATION 2 #####

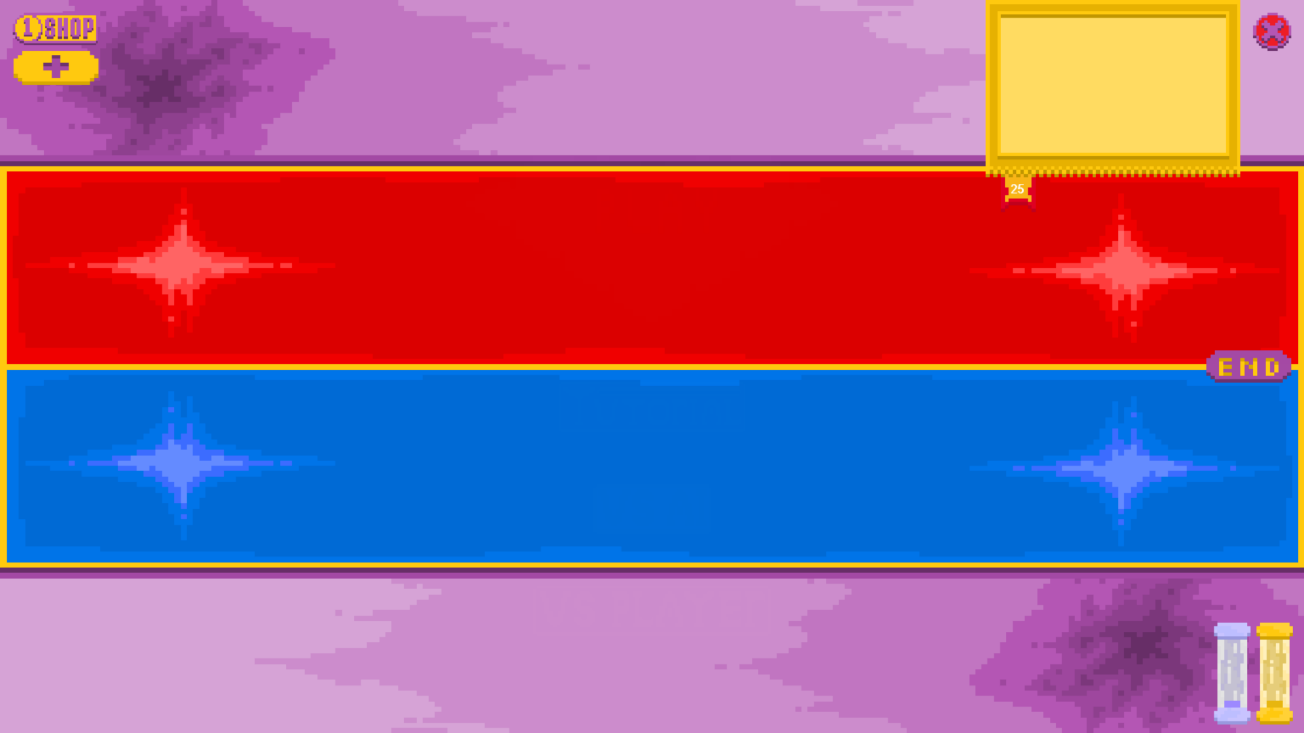
1. **class** Button:
2. **def** \_\_init\_\_(self, coords, size, picture, hoverPicture = ""):
3. **if** hoverPicture == "":
4. hoverPicture = picture
5. self.clickRect = pygame.Rect(coords,size)
6. self.coords = coords
7. self.hoverPicture = pygame.image.load(hoverPicture)
8. self.hoverPicture = pygame.transform.scale(self.hoverPicture, size)
9. self.picture = pygame.image.load(picture)
10. self.picture = pygame.transform.scale(self.picture, size)
12. **def** press(self):
13. **pass**
15. **def** draw(self):
16. mousepos = pygame.mouse.get\_pos()
17. **if** self.clickRect.collidepoint(mousepos):
18. player.screen.blit(self.hoverPicture,self.coords)
19. **else**:
20. player.screen.blit(self.picture,self.coords)

This iteration of the Button class is a little more complex than the initial iteration. It takes in coordinates, a size, a picture and optionally a “hoverPicture”. A hoverPicture in this case is supposed to be a slightly lighter version of the initial picture to indicate to the user that they are hovering over it but this is optional as it is not integral to the button’s function, it is simply a visual quality of life feature to make the game look more interactive. If no hoverPicture is passed in the function will then default it so that the hoverPicture variable is the same as the picture variable so no difference will be seen whether the button has been hovered over or not though it will technically be loading the same image from a different location when hovered.

This iteration of the button class has also added a “clickRect” class variable which defines the area in which the player’s mouse has to be located for the button to be pressed. This is checked against the mouse position within the game loop as having a function to check this would be mostly redundant.

# Testing

## Shop Testing

1. Check that the shop can be opened by pressing the shop button
2. Check that clicking on the card itself will not buy the card when the shop is open



#ADD RED CIRCLE AROUND CARD#

1. Check that you can buy a card when you have enough currency

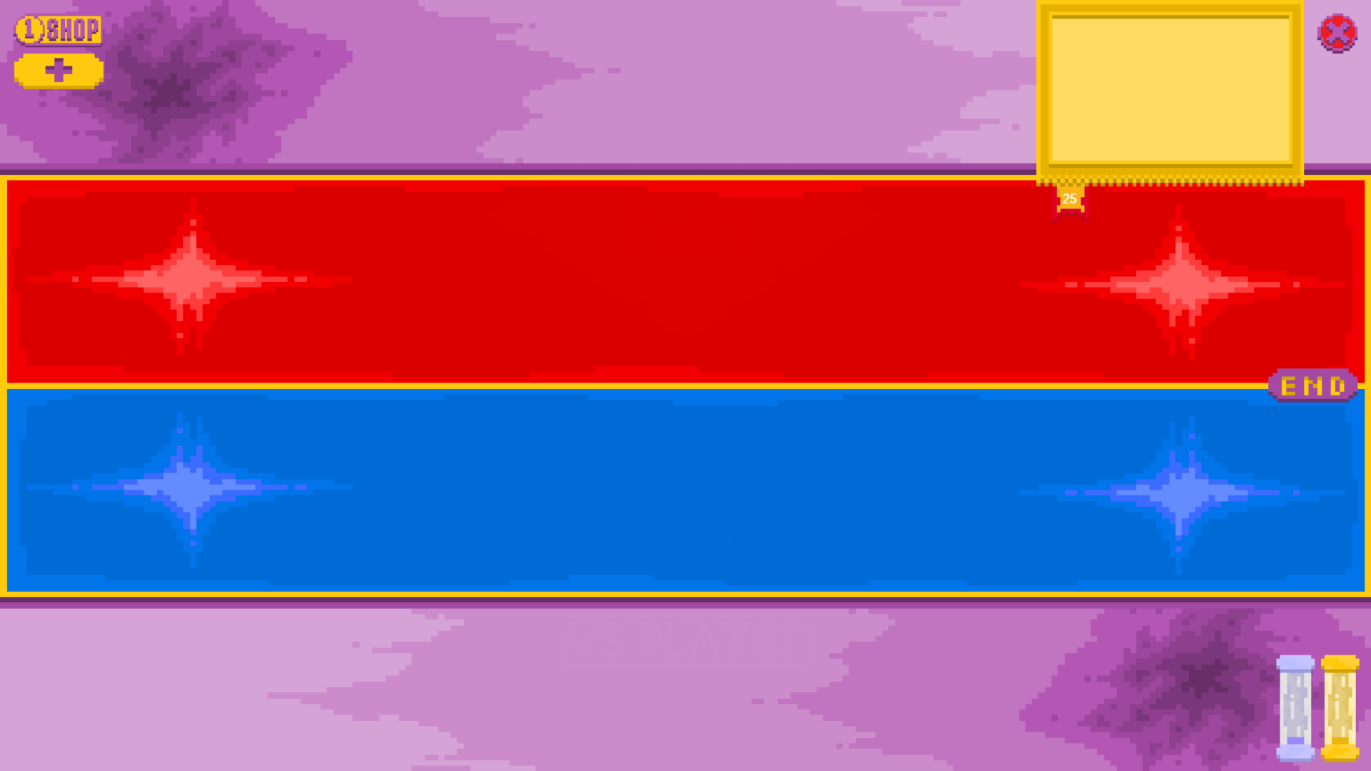




1. Check that you cannot buy cards with insufficient currency



# ADD RED CIRCLE AROUND CARD #

1. Check that you cannot buy a card by clicking the location in which the shop button would be

#ADD CIRCLE AROUND POINT WHERE BUY BUTTON SHOULD BE#

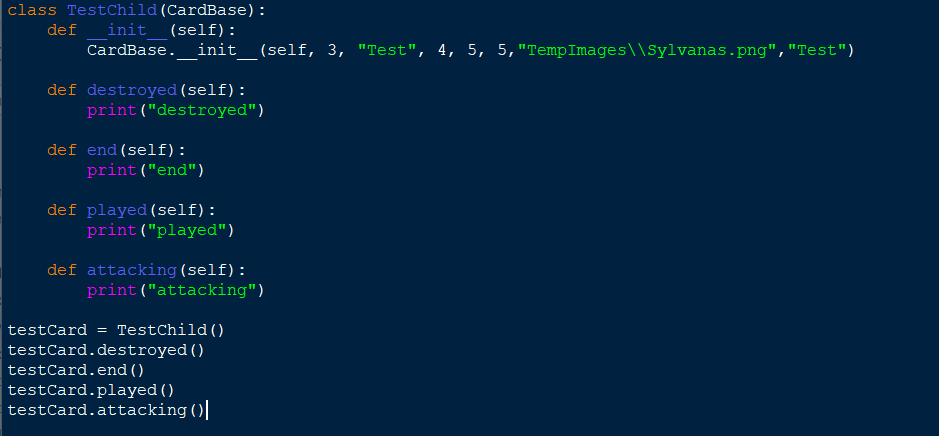
## Card Class Testing

1. Create a test card object from the base card class



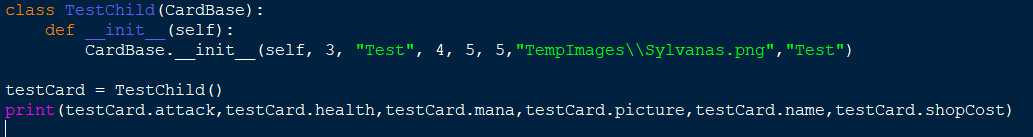


1. Create a card subclass with all methods changed and call them all



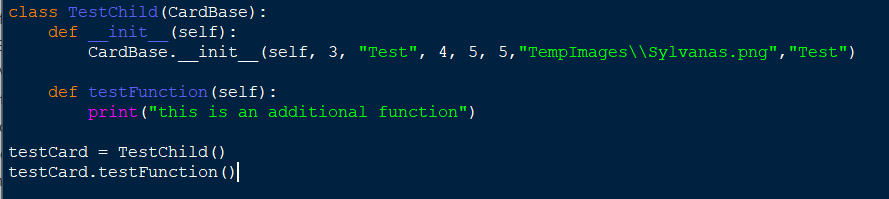


1. Create a card subclass with all attributes changed





1. Create an object with an additional function as a child class of the CardBase class





## Card Effect Testing

### On Death Effect and Card Destruction Testing

* 1. Check if on death effects can cause the program to try and remove cards from the opponent’s empty board
  2. Check full boards
  3. Check full boards of dead minions
  4. Check boards with mixed dead and living minions
  5. Check board with living and dead cards of the same name
  6. Check board where one player has a full board and the other has an empty board

# Evaluation

Analysis

Design

Implementation

Testing

Evaluation