Chaos Critters

By Our Group

Connor O Boyle

Eoin Clarke

Jack Neeson

Liam Murry

Chaos Critters: A web-based card game

Group Name: OUR Group

Project Members:

Liam Murray - 19373226

Jack Neeson - 19404476

Conor Boyle - 15350176

Eoin Clarke - 19378381

GitHub Link: https://github.com/nmq-hyt/OURGame

Web App link: https://ourgame-fdad7.web.app/

Table of Contents:

Introduction:

OUR Group was formed quickly after the project's announcement. Due to the impact of the pandemic, working and discussing became much more difficult. Thus a discord group was created. Here we collected all of our different ideas together. We had many ideas such as a Dungeons and Dragons character sheet creator. However, we all really liked one idea which was familiar to all of us: a video game, specifically a card game.

Chaos Critters:

Chaos critters is a web-based card game where two players can play locally as either a snowball-shooting penguin named Chilli or a trash-loving raccoon called Vector and fight over who gets the soup! This game was hugely inspired by other digital and tabletop card games we play ourselves, in particular Hearthstone by Blizzard Entertainment and Dungeon Mayhem by Wizards of the Coast.

The web app has a starting web page that functions as the main menu, where players choose who goes first and can read the rules of the game.

Once ready, the players are brought to a new web page with a playing board. The game is then prepared, with players being allocated decks from the external database and initial values for health, cards in hand and cards in the deck.

The players will then take turns playing cards that deal damage to the other player, heal themselves, draw more cards, gain energy in order to play more cards and gain shields to block damage. The players play on the system, with the gameboard changing values and cards to match whoever's turn it is currently. The battle continues until one player is reduced to 0 health, in which case the other player is the victor.

The current state of the web app is a demo, with the fundamental parts of the game and our own custom-made assets being highlighted. We sadly ran out of time to compile the full game but given more time we would have more to show. Regardless, we are proud of what we have completed.

Our Roles:

Jack:

I mainly worked on the front end, design, game balance and rules. I also made the home page and all of the art. The original idea for the game was also me but this has shifted over development as we all put forward changes and ideas.

The front page was my main contribution. It was made in CSS and Bootstrap. Along with this, I also helped in the creation of how the cards would act and work and the overall balance of the game.

The idea of the game was mainly from me. However, many times through the development of our app things would change. This could have been for many different reasons one of the main ones being to differentiate us from our inspiration.

Liam:

My primary role within the project was development of the frontend and the UI for the users during the game. This involved the use of HTML, CSS and JavaScript to ensure everything was visually implemented as was planned and that any frontend functionality required for the user was functioning.

The main two elements I designed were the character select screen (that we decided was no longer necessary) and the gameboard. I designed the layout to ensure it was fully functional and ready to be integrated with the backend when the backend had all the requirements for it prepared. I initially drew up rough plans for how the screens that I needed to design would be laid out.

In the character select screen the main functionality required was having a character portrait for each character and upon clicking these images it would prompt the user for confirmation and then communicate with the backend what character the user wished to play.

On the gameboard I had to design the layout and make sure all the layouts were given the appropriate space and location. The main functionality requirements were that the values for a player's health points, armour and remaining cards in their deck were all displayed correctly and in the correct location. A function was also implemented to enable the values to swap between each side of the gameboard at the end of each turn as control of the game switched between the players.

Conor:

As I had less interest, and even less experience, in front-end design, I elected to design and implement the backend, database and core game logic parts of the project. I was responsible for providing an initial high-level view of the overall project, writing schemas for the database design, for designing and implementing the majority of the game logic, including the classes representing components of the game, and writing the cloud functions which helped integrate game logic and database functionality, like checking if the game was over or if a player had run out of health. I developed suitable abstractions of various components of the game's internal logic, mainly modelling game agents using object-orientated programming.

I used Alloy, a declarative specification language, to construct high-level views of how the overall project should work, in terms of connecting varying components together. I used MySQL workbench to initially model the database objects, their fields, and how they connect. We didn't end up using a SQL database, but the initial model informed the data model I used for the NoSQL database I implemented. In trying to write object orientated JavaScript, I came across TypeScript, a language that greatly helped me in writing classes and generally saved time with JavaScript, a language I really, really don't like.

Eoin:

My main role with this project was to assist with the backend of the project, creating the fundamental game logic and applying this logic to JavaScript. I first constructed basic JavaScript and HTML code to show how the basic mechanics would work and how the values would change, which assisted later on when constructing the final JavaScript functions.

I would also test these functions and optimise them where necessary. An example of this was when I reworked our structure for decks. Initially, we were going to implement them as their own object, but a simple array of card objects within each player object would work just fine and so we saved ourselves from making extra unnecessary code.

I am also responsible for the writing up and management of this report, summarising all of the details of this project and congregating the other project members' own experiences and inputs into a neat, accessible format.

Project Description:

Inspirations: (Jack)

There was more than one inspiration for this game. However the main one that made the idea comes from a real-life card game called Dungeon Mayhem, a card-based battle game made by Wizards of the Coast.

With heavy inspiration from this game we made our cards work similar but we wanted them to have their own feel. For example, we also used 5 different symbols to be played. This being Attack, Defence, Heal, Draw and Energy.

The characters in this game also had unique attacks. This was also something we planned on adding later into development but time ran out on us and we decided to instead focus on making the core mechanics feel good to use.

We didn't do everything the same though. For example, we didn't want to do the same Defence system they had. We worked as a team on what we would do instead. The first plan was to try the same method, but instead we chose to implement an Armor system.

This leads to our other inspiration, Hearthstone.

Hearthstone, is a mobile and desktop card game simulator, developed by Blizzard Entertainment. This game does not have many similarities with our game, but it has one key feature that we thought as a team would work much better in our card game.

With our already implemented features we worked out how we would implement the armour mechanic into our game. It works very similar to Hearthstone, where when an armour symbol is played a counter beside the player's HP (hit points) would increase. This would then take damage for the player first as so the player would have some kind of way to protect themselves.

Teamwork and Communication: (Eoin)

We were fortunate enough to have known each other before this project, so we had some advantage in that. As stated above, we subdivided the project and assigned roles to ourselves. Liam and Jack took on the frontend work while Conor and I handled the backend.

Due to our constraints with the pandemic, we had the difficulty of working far from each other. This meant that communication and sharing of thoughts and completed work became much trickier. I admit that this project would have progressed further had we had the opportunity to meet and work together in person, but we still continued. To combat the

distance gap, we turned to online resources to help keep the project running as smoothly as possible.

Discord:

We used discord as our primary means of communication; holding weekly to biweekly meetings to discuss what we had worked on and what we needed to work on. We were also able to quickly share any small snippets of code that we worked on and get feedback quickly. The screen-share feature was very useful in allowing us to show our work too.

GitHub:

We used GitHub as our primary means of source control, and although we often tested locally, we would upload code that we were satisfied with to Github once it had finished being tested.

The GitHub features two branches: main and testing.
Main is where the final source code that we deployed to firebase is kept.
Testing contains all of the code we have tested, used and referenced in the duration of the project.

Goals: (Conor)

We concede that the software is ultimately entertainment software and should be simple and easy to run. The game's goals should therefore be to be:

- fun
- relatively easy to learn/not overly complex
- playable on low-end systems
- visually clear and easy to understand

How to Achieve These Goals?:

In order for the software to be non-resource intensive and playable on low-end systems, we would implement software that is

- efficient with its resources
- non-demanding (e.g. choosing front-end elements that demand little CPU usage)

In order for the software to be visually clear and easy to use, we would :

- follow accepted standards for accessibility in web content, like WCAG 2.0
- design any UI elements to be clear and easily readable

In order for the software to be easy to play and learn, we should deliberately have a minimalist design behind the game logic and the website.

Fun is ultimately a subjective term, and there are no standards for it, at least, none that are wholly objective. However, in designing the software, we took inspiration from several, long-running card games, to better learn from their mistakes and successes in creating an enjoyable card game, and in doing so we hope to make enjoyable entertainment software.

Benefits of The Goals:

Fun:
Any good game (any good piece of entertainment software) should try to be fun.

Ease of learning:
In our collective, personal experience, modern card games tend to be complicated, and sometimes difficult to pick up. For a concrete example, the complete rules reference for Magic the Gathering, a widely popular game, is 247 pages long. This can act as a barrier for users to adopt the game and, by extension, complicated software that is difficult to understand. Even if it is “good”, the game can ultimately not end up being used due to these difficulties, perceived or otherwise.

Playable on low-end systems:
Modern high-end games often demand expensive gaming machines to play them. We intend to make software that can run comfortable on machines with smaller amounts of RAM, or on slower connections.

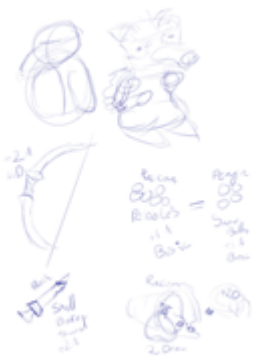
Another factor of this goal was to reduce the amount of time spent on networking operations; it turned out that performance would be harshly affected by time spent querying a database or retrieving assets.

Visual clarity:
Any good UI/UX should communicate to a person what is happening, what interactions are possible, what errors have occurred etc. clearly and concisely. Where possible, software should be made accessible.

Frontend:

Character Design: (Jack)

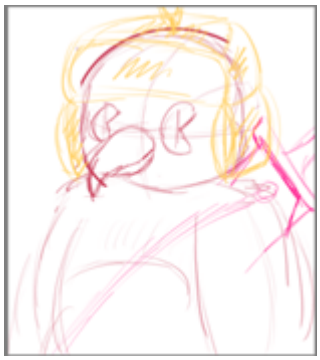
The characters were very straight forward but they were not always going to be what we ended on. Originally, we had the idea to come up with a concept each and implant them in but this got far too complicated and led to characters that just didn't fit with each other.



When the game went to 2 players, a new idea was hatched: using small animals as if to make it cute and more funny than serious and complicated. The idea for the animals was mostly just decided by myself for the purpose of making something recognizable and kind of outlandish. So the idea of a penguin and a raccoon having a confrontation was a funny idea, with a lot of support behind it for gags. The idea was to have the characters be cute and have 2 very different personalities. This would be shown through their card art in the end.

The idea of this penguin being a cute, innocent fluff ball would make sense and the raccoon being this cunning, dirty fighter would not just make sense within their animal stereotypes but also as blatant contrasts to one another as a way to give character to the voiceless animals.

Next was fleshing out their designs and getting portraits to reference all the rest of the art from.



I started and finished the penguin first. I mainly designed the penguin to be cartoony and over simplified. This worked really well but he needed something to identify him from just some penguin. This is where the idea of giving them all a piece of clothing and some kind of unusable weapon would be a funny and cute idea. So to keep with the theming of him trying to be cute a fluffy hat seemed appropriate. The weapon was another story, it mainly was just picked for no reason but the 2 animals' weapons were picked at the same time.

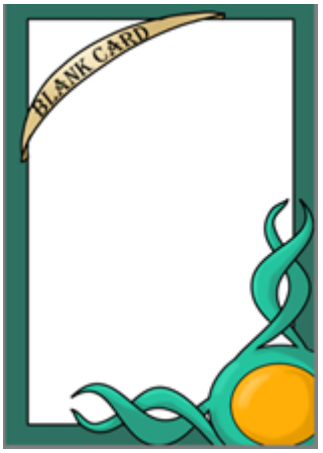
The raccoon was next. Unlike the simplicity of the penguin and his soft smooth edges, I wanted to make the raccoon have a bit of sharpness to him. This turned into him getting fluffy cheeks and fur. His whole personality was to try to look cute but still have that cunning side that raccoons are portrayed to have. The bandana around his neck was chosen to keep with the stereotype of a raccoon being a thief and also helped to separate his head from the rest of his body. This in turn slim-lined the figure. Finally, a bow was chosen again for no real reason than it would be funny to see him try to shoot a bow the size of him.

The final outcomes were the final versions of the above sketches. The last thing to do was to then give them names as "The Penguin" and "The Raccoon" wasn't right. We tried going for cute names like Fluff, Bo and Sneak.

The final names were: Chilli the Penguin and Vector the Raccoon.

Card Design: (Jack)

With the original idea and for testing purposes, a demo deck was made. However, before that could be made into a deck, a form factor needed to be chosen. This included getting dimensions that I could stick to.



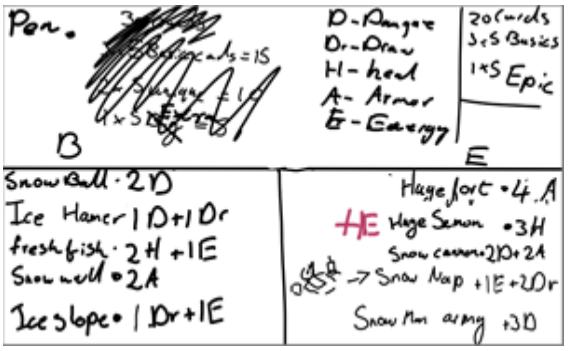
This included me making a custom format for my drawing software and making the card template in that. This would be things like the border running around the card that would stick with all the arts and then the placement for where the name of the card to go. I also needed to decide here where the actions on the card would go, so this ended up going in the top right corner.

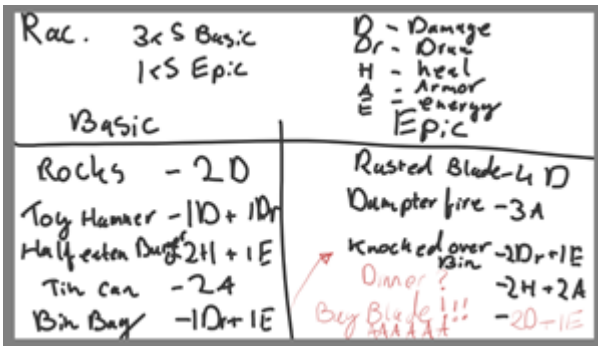
The card then needed to be coloured. This was just a placeholder for the demo deck as I wanted to have 2 separate colours for the animals. This was not perfect but for the purpose of this card it was what we needed to build all the cards off of.

The demo deck only consisted of 10 cards but for the real deck there was going to be 20 cards minimum. So 10 individual demo cards were made and 10 unique cards for each character. This is only an example of the cards for the demo. They just had funny looking art or had a name that was funny. But this would help with game balance later on.

First off when I started designing the individual cards for each player, they needed a colour each, In the end I chose blue for Chilli and grey for Vector.

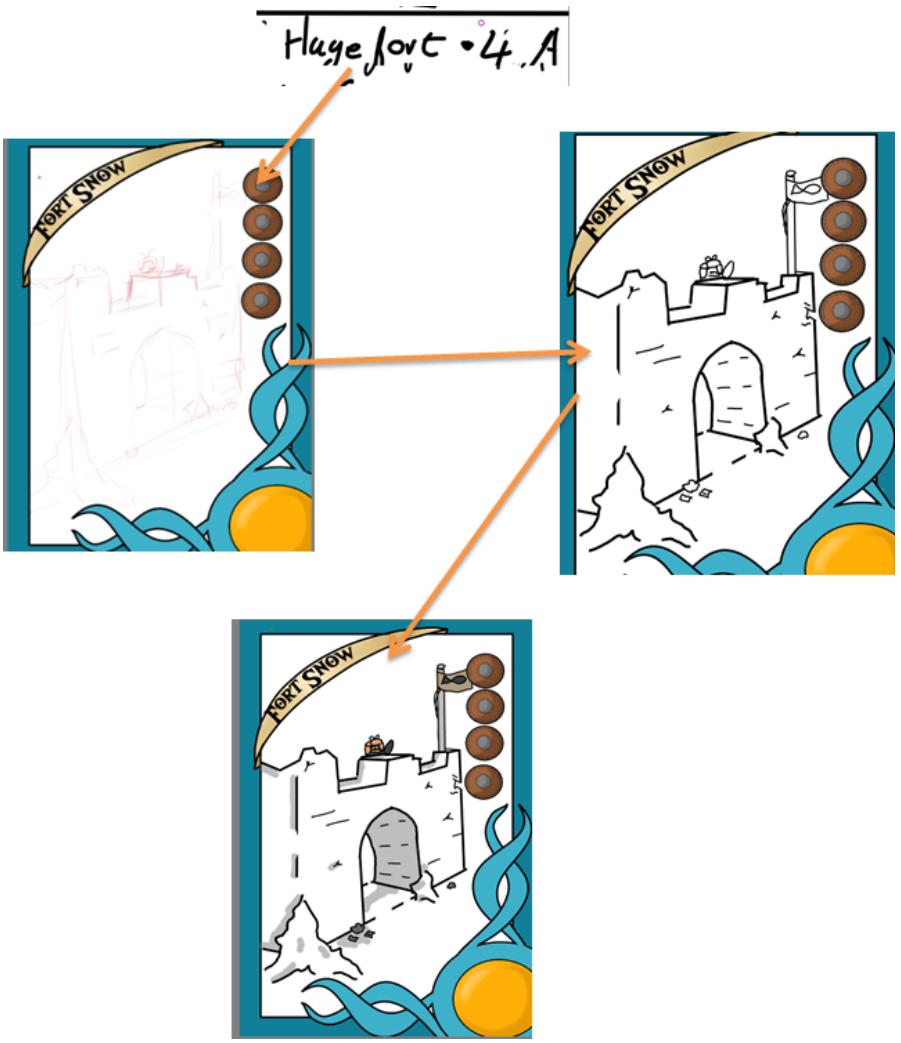
Then next was making up what would go on the cards. I made rough sheets for each and just came up with funny names or ideas and wrote them down and tried to match them to the symbols.





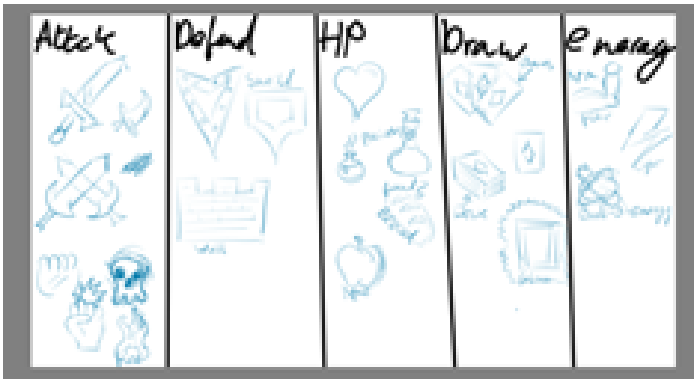
This kind of style allowed me to plan the decks on how they would play as well. This will be talked about further in the game balance section.

The next image shows my process for creating a card: Having an idea, creating the rough sketch, then the final piece was made. Finally, colour and shading were added.



This was then repeated for each card; the values were added and then sent to the back end for implementation.

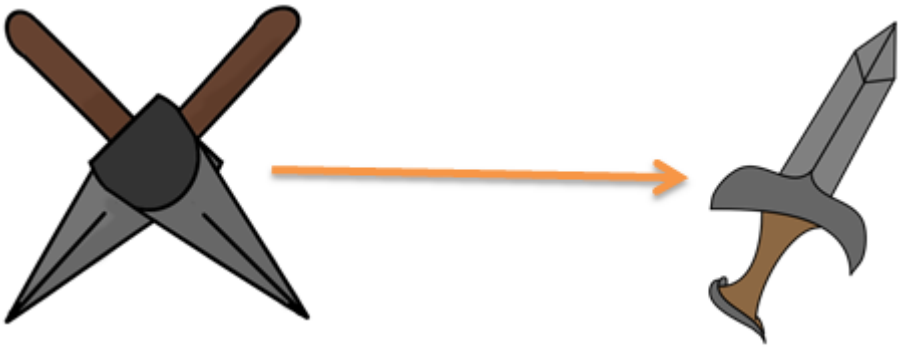
The final piece of art I want to talk about is the symbols on the cards. This was one of the first pieces of art made and even though they look fairly simple by design, they took many different forms throughout development.



Most of the final designs were obtained from this page but many of them would change through the process of either not liking the final product or just a vote.

The main symbol that saw a later change was the damage icon.

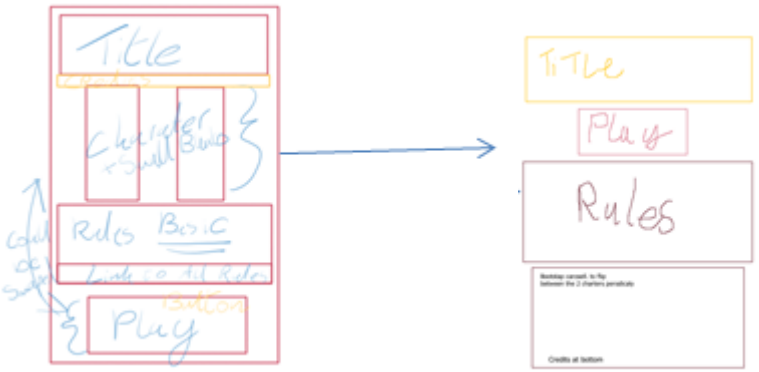
The original idea was to keep to the similar icons of one of our inspirations but during development the old icon looked sloppy and not well done. It was then chosen out of the rest of the designs sheet to which a more basic sword was chosen.



Main Page: (Jack)

The main page went through tons of iterations. Many things were planned then scrapped, or just didn't work out the intended way.
The first main page was 3 different pages: with the first being the play button and a brief explanation, the second being the game board and the third being a full rules page. To make the game easier to understand, the rules were drastically shortened and placed onto the front page with the button. The game board stayed the same.

Next was working out the layout of how the page would come together. This was a basic interpretation of how the page was going to look. This was quite early into development. It then evolved overtime to include other ideas.

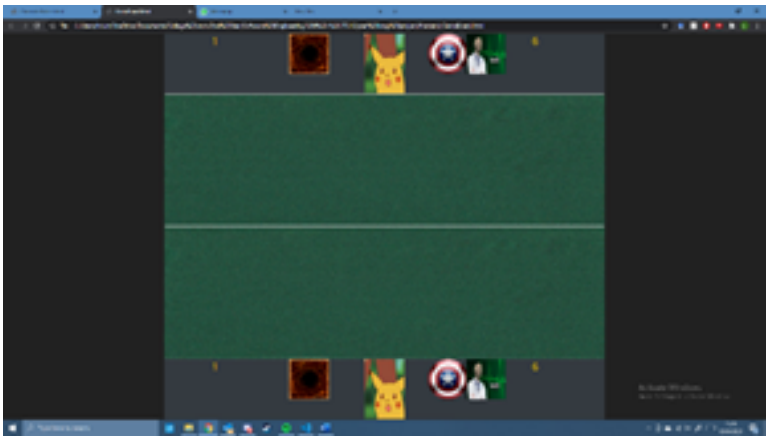


However this would also not stay. The idea of the Bootstrap carousel element also got tossed about a few times. This was implemented quite late and seems to be more confusing for the end user to what it was worth. In the end, this idea was scrapped and replaced by a much simpler option.

Mainly wanting to keep the code that was already there, the webpage was revised for a 3rd time. This would combine the 2 ideas to make one webpage that would have aspects of both.

The button was moved below the characters. This was then followed by the rules sheet. This was then fully developed into the webpage today. The page now used simple but effective cards that matched the whole dark and gold theming that was present on the card back and other parts of the design. The 2 characters just needed to be created and set into the world. The rules sheet also needed to be developed and finalised.
Finally when the last pieces were in place the whole page came together; the name was finalised, the charter designs were complete and the rules were ordered in a way that made them as simple as possible.

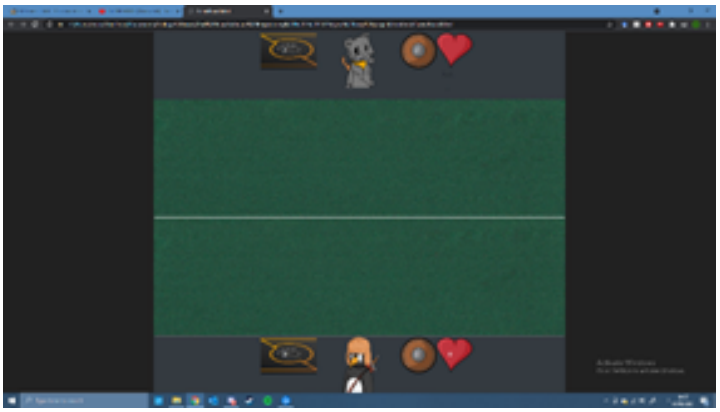
Gameboard Page: (Liam)



The gameboard page was a very important part of the project as this was where the players would be spending their entire time playing the game. They would be looking at it the entire time so it had to be visually appealing while also being adequately laid out to support all the in-game elements and providing space for the actual gameplay to occur.

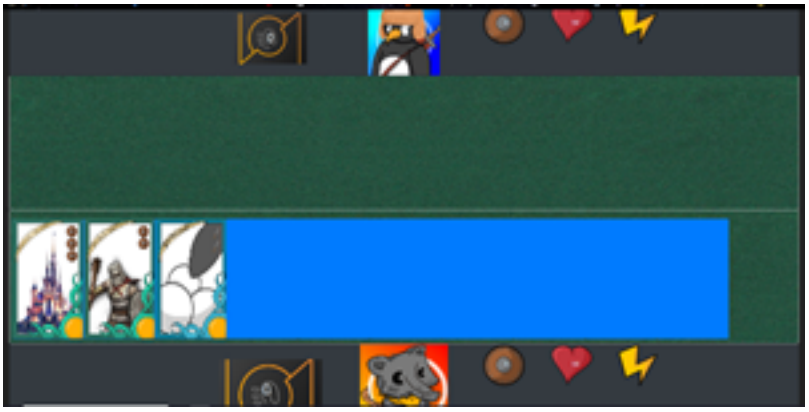
The main idea for the board was to have it be identical for both players and be symmetrical as each player has identical elements aside

from the cards handled dynamically and the player profile. With this in mind the player board was designed where each player had a player portrait and the four elements we needed to keep track of: number of cards left in their deck, hit points left, armour counter and energy counter.



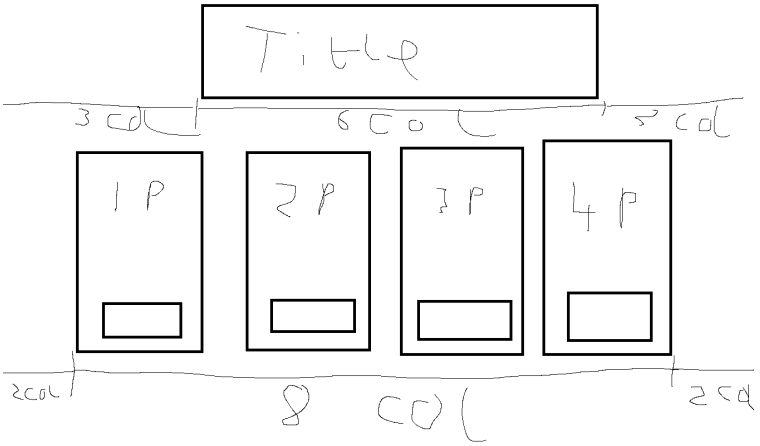
Aesthetically for the actual playing area we went for a green felt look to mimic a card table you might find in a casino. This was split in two to provide an area for each player to display their hand and choose their cards.

A big challenge with the gameboard was implementing a way using JavaScript that each player got control of the game for their turn but couldn't see the enemy's hand or hidden elements which was difficult as the game is on a shared device. Our solution was to only have the player in control's hand displayed at any given time with a blank card back displaying the opponent's hand size during their turn. At the end of the player's turn a JavaScript function would run and swap all tracked elements and the player's portrait to opposite sides of the board using temporary variables to hold the values of one player while they were being swapped.

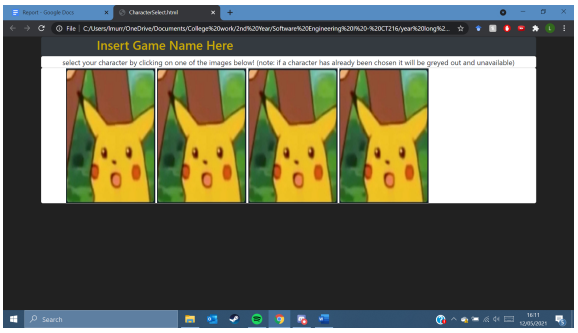


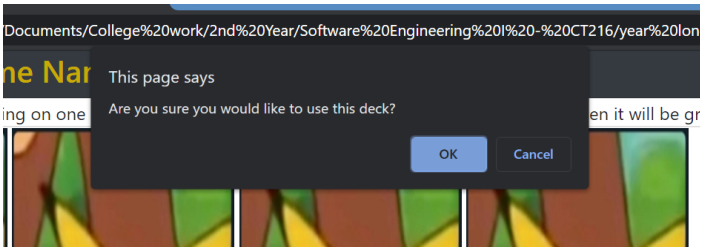
Character Select Page: (Liam)

Initially we had plans to create 4-playable characters and to give the players more options if we had sufficient time to implement these features. Before the design overhaul due to time constraints we had a base design for a character select page created and ready to be modified and integrated into the game.



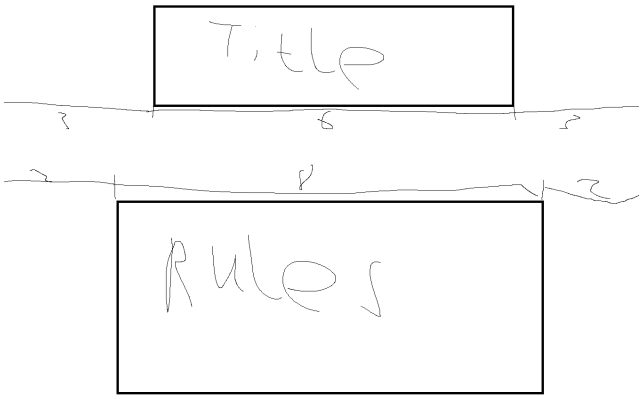
The base idea was to have 4 character portraits that upon clicking them would give the user a prompt to confirm that is the character they wish to select using simple JavaScript alerts and would then grey the image out and move over to the next player to select their character

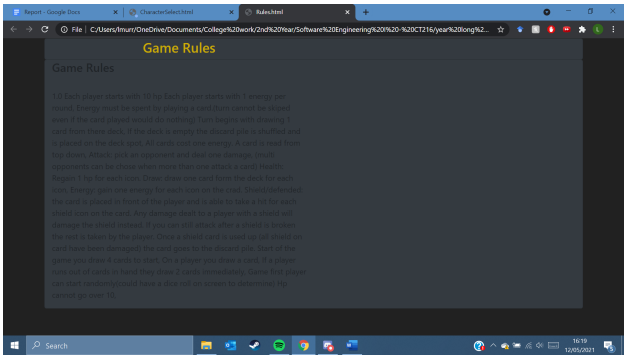




Rules Page: (Liam)

Initially we had planned to have the rules of the game listed on a separate page that would be linked to from the main page. We created a page for the rules initially but after further development we decided it would be more user friendly to display a simplified version of the rules on the main page rather than having it somewhere the users might not see and may miss.





This decision was made fairly early in the design process so the page was not fully styled and was more of a rough draft based off of early planning sketches

Integration with Backend: (Liam)

Integration of the frontend and backend was a challenge that we did not get functioning 100%. While the design of the frontend and the gameboard was a fairly simple task on its own and the backend was fully functioning and usable on a base level, having the elements display correctly and interact as they were intended was very challenging.
Some challenges were ensuring the variables were displaying the correct values, having each player's hand swap out as being visible at any given time and the combination of visual elements, backend object tracking and player control handling. Small inconsistencies between Internet Browsers also caused issues such as frequent crashing which led to the use of Firefox exclusively as our testing environment.

Rules: (Jack)

The rules of the game were a delicate subject and took lots of consideration to how they should work. I worked out the original rules and set boundaries. This would not stand for long as the rules would be shortened or changed to make the game better or a little simpler. However most key rules were set from the beginning and is what we used to build this game. Then the rules were broken into 3 separate pieces: player rules, card rules and development rules.

Player Rules:

These are all rules that the players are told so they can understand the game. This was telling the user about their HP (hit points) and how damage would affect other players. The player also needed to know about what energy did and how they could use it. All these key rules were placed on the front page, for the player to easily access.

Card Rules:

This aspect would cover the design and functionality of the cards. This rule was only really followed by me but all of us had to know an understanding on how this worked, such as the order on the card of symbols. This was key to the workings of the card as each symbol would have its action completed before the next action on the card. For example, the card draw symbol always had to be above the energy symbol, as it wouldn't make much sense to play another card then come back and draw the cards from the card before.

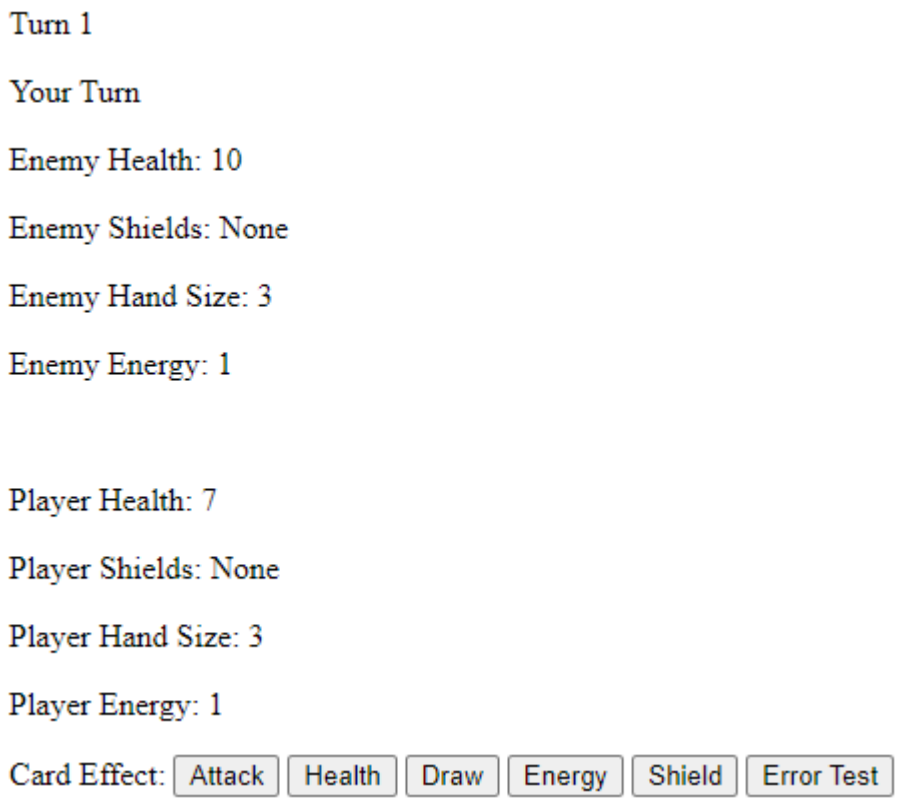
Development rules:

These are the rules we would have to follow as a team to make the game work. These would be things that the player may never see but could happen. Such as if the deck were to run out of cards what would happen? Other plans under this were the schedule we set up for ourselves and sharing it with the others. Also keeping things like what other assignments we had to get done was also tracked so we wouldn't fall behind from working on this or the other way around.

Backend:

Early Development: (Eoin)

With the aforementioned rules set, we needed to implement them properly into our backend. My first step with this was to create a basic client only web app called “turnhelp” and a complimentary JavaScript file named “turn.js” that accurately displayed how these rules would play out.

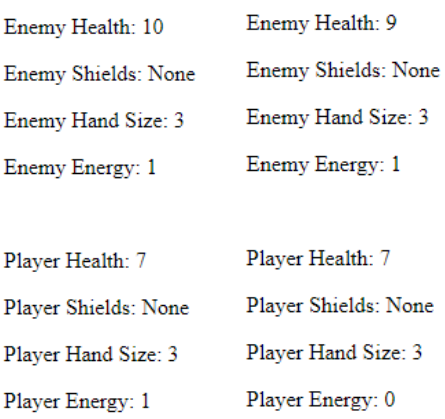


This web app would display the main variables for both players such as health, shields and hand size. This was made before we made some important design changes with the project like the different defence system, the focus on 1 vs 1 local play and the scrapping of the unique cards, so some elements in this were either unused or altered.

The different buttons allowed me to simulate how playing a card with a certain symbol would affect the game. Whenever one is pressed, a prompt asks for a value between 1 and 3. This value will be taken in and used with the corresponding function. There are 5 functions for the 5 different actions. Each action affects the other variables differently .

For example, playing a card with 1 Attack would do the following:

- The Player loses 1 Energy
- The Player Hand Size reduces by 1
- The Enemy Health reduces by 1



I also included buttons for cards with 2 actions. This was to show which actions get priority over other actions, which are outlined in our card rules. The two actions will occur during the one playing of the card, but not at the exact same instance.

There are many other elements I could have added, like a swapper between the players (which we later implemented in the final build) but I felt this was enough of a groundwork to begin working on the game objects and their functions.

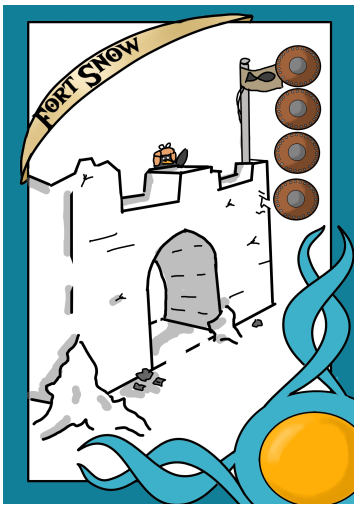
Card balance (Jack):

With the game being based around 5 different actions, balancing the game was going to be a long and hard road. Many iterations of how cards would work, or how they could be played were thought of but in the end, the game turned out to be fair because of only having to balance 2 characters. The game could have gotten much more complicated if we implemented 3, 4 or maybe even more characters and decks. A simple answer to game balance was just make both decks the same and while the game could have been balanced, more than likely the player just going first would be the person who would probably win. We also wanted diversity to make the game more enjoyable.

So the first point that came to mind is how the characters will play. Just for simplicity I took two very basic archetypes or in other words play styles. This would include the “Tank” and the “Rouge”. These terms were used loosely but it made a standard they would work around. The stats would be assigned to cards before a name or concept.

To allow for some simplicity the cards were divided into basic and epic cards. Even though there is no major visual difference, there is a power difference. Most of these “Epic” cards would be just drastically better than their basic counterparts. However, as to not make basic cards obsolete, there is a skewed number of cards in the deck. This would mean there is 3 of each basic card while there is only 1 of each epic per deck.

Next was balancing the basic cards to make sense and while trying to work this out and looking at the inspiration games, I decided that the 5 basic cards would be uniform between all classes. So even though the art may change, the cards themselves mechanically would all work the same. This meant that all the diversity would come from the epic cards.



To make things as fair as possible I decided on making flipped roles. I wanted to make 2 evenly matched opponents so for the 4 armor card the penguin had, the raccoon got a 4-damage card to deal with it. This was done with 2 cards.

Now for simplicity I made two of the cards work the same because it seemed both decks needed these effects to work. The last card however was unique for both of them. This was all because of the fact there were only 2 characters. If there were more characters in the game this would have probably changed, and balancing would have been done differently.

Game Objects (Eoin and Conor):

We decided to take an object-orientated approach to making our game. Many games are designed based on this principle as it is conceptually easier to create the various elements of a game like sprites or entities. It also means these elements can be extended to include more properties or be easily duplicated many times. In our case, having a card object that stores the actions and the corresponding card art is very useful.

JavaScript is not a class-based language inherently, it's a prototype-based language. This means instead of "traditional" OOP, with inheritance and composition of objects from classes, prototypes of objects can be made from which objects themselves can be made. This could be quite difficult to understand and could lead to many errors and bugs.

TypeScript:

While researching various technologies to use in this project, I came across an up-and-coming programming language: TypeScript. The essence of TypeScript is it's JavaScript, but with a type system, transpiled to standards-compliant JavaScript.

This was the ideal language for my purposes; JavaScript, the native language of the web, is a logical choice for web development. But having used Python in personal projects and finding frustration with that language's lack of a type system, TypeScript was an ideal compromise. While arguments about the effectiveness of type systems could be an entirely separate report, TypeScript's syntax allows for writing classes rather similar to how they might be written in Java, or C#, and given my experience with writing Java for OOP, it seemed like a great quality-of-life improvement, as the development of this project would necessitate the creation of custom "types" (i.e., the game objects).

TypeScript source code can be written and then checked using tsc, the TypeScript compiler; even if the source code produces errors, it is still compiled to legal JavaScript. This was also strongly motivated by my past experience from writing JavaScript that would "look" fine but not work when loaded; thus, TypeScript's compiler giving me errors would end up saving us a lot of time. TypeScript was used as a baseline to generate the final JavaScript code, and the transpile code was edited further to flesh out functionality.

An early example of a typescript class is displayed below:

```
class player{

    player_number : number;
    player_skin : number;
    player_health : number;
    player_shield: number;
    player_energy : number;
    //new arrays for hand and deck
    player_hand : Array<card> = [];
    player_deck : any;
    neighbours = new Set();
    // NEIGHBOURS represent a graph view of the game
    // the standard format is that
    // the zeroth member of the array is the direct opposite of given player for two players
    // in three players, it is 0 to represent the left player relative to the player, and 1 for the right
    // in four players, 0 is the left , 1 is the middle, 2 is the right
    constructor(num : number,  health: number, neighbours: number){
        this.player_number = num;
        this.player_health = health;
        this.player_shield = 0;
    }

    set_player_skin (player : player, skin: number) : void {
        player.player_skin = skin;
        return null;
    }

    //still unsure how to fully implement this
    add_card_to_deck(c : card){
       player_deck.push(c);
    }

    add_deck(deck_num : string) {
        handout = firebase.functions().httpsCallable("handout_deck");
        handout({deck_num})
            .then((result) =>{
                // so this part is actually just
                // for loop of size 20,
                // takes i , addes it space i in the deck index
                this.player_deck = result;
            })
    }

    add_neighbour(player : player) {
        //neighbours.add(player);
    }

    damage (target: player , amount : number) {
        target.take_damage(amount);
    }

    energy (amount : number) {
        if (((this.player_energy + amount % 30)) == 0) {
            this.player_energy = 30;
        } else {
            this.player_energy += amount;
        }
```

After some initial refactoring, reflection on the game objects and discussion with Eoin, it was made clear that a deck class was redundant and this eventually led me to removing it completely from both the game objects I'd written, as well as from the database. We instead used a cloud function that "hands out" a JSON object (our deck) with all of the values of a given card represented as a plain, JavaScript object, key-value pair. These are later sorted client-side (e.g. the Fisher-Yates shuffle is used to guarantee a random permutation of the cards in the deck).

Card Class (Conor):

The card class is a fairly static class, whose main function is to deserialize the plain JavaScript objects into card objects and to map individual cards to their respective individual card art. These card objects are then handled by the player objects to execute the game moves corresponding to the values displayed on the card.

Cards hold values for specific actions as indicated on their card art and a string containing a reference to the correct image for the card map. This allows the display of the card art dynamically on the frontend as every time a card is put into the player's hand it is rendered as a button styled with the card art's image.

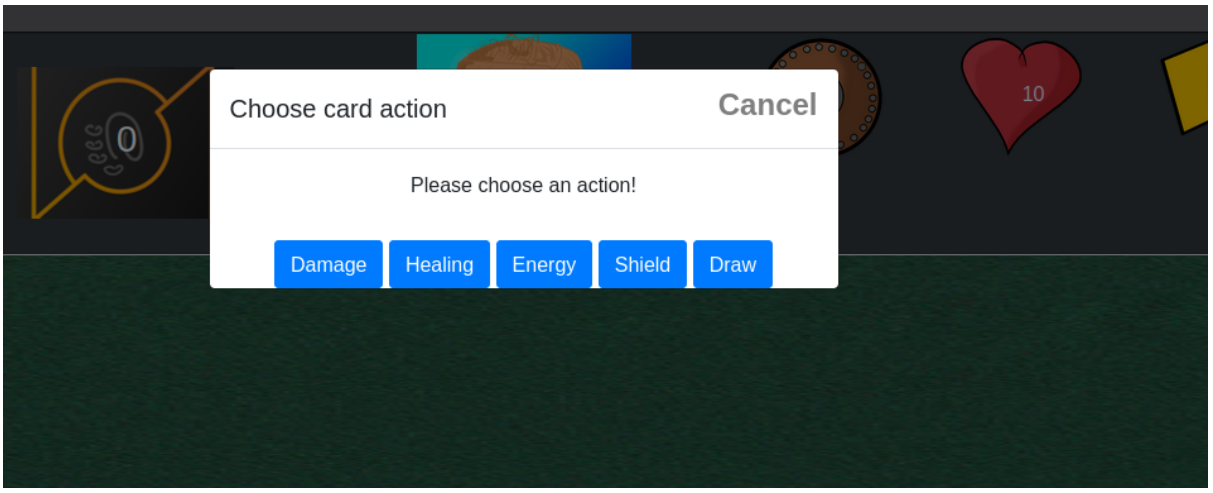
```
var card = /** @class */ (function () {
    function card(card_object,card_art_str) {
        keys = Object.keys(card_object);
        values = Object.values(card_object);
        length = keys.length;
        for (i = 0; i < length; i++){
            if (keys[i] === "attack" ) {
                this.card_dmg_val = values[i];
             } else if (keys [i] === "shield") {
                this.card_shield_val = values[i];
             } else if (keys [i] === "energy") {
                this.card_energy_val = values[i];
             } else if (keys [i] === "draw") {
                this.card_draw_val = values[i];
             } else if (keys [i] === "heal") {
                 this.card_heal_val = values[i];
             }
            this.card_art = card_art_str;
        }
    }
    return card;
}());
```

Player Class (Conor):

The player class is a very lively one; it keeps track of a player's essential characteristics and also performs the basic rules of the game, specifically by consuming card objects to perform basic actions like drawing more cards and damaging the other player while keeping track of the player's hand using an array of cards. Since the player class is entirely too large to display neatly in this document, portions of the class will be shown.

The player class holds other players, in an array called neighbours, which was originally intended to make it easier for cards to target other players. This was obsoleted by the removal of player counts more than two. The player class contains fields representing essentially player characteristics, like energy, shield, health etc., and the deck, an array of card objects. An important function is play_card, which consumes a card object and a choice (from the front-end input, a modal box showing an array of buttons, which corresponds to a

particular choice of value on a card), which then calls another function corresponding to the action selected - this performs said action, like healing the player.



The handout_deck calls the cloud function of the same name, filling the deck array with card objects that are correctly mapped to the appropriate card (this all being done by hand), then shuffled using the classic Fisher-Yates shuffling algorithm.

```
};
//still unsure how to fully implement this
player.prototype.add_deck = function (deck_num) {
    var _this = this;
    handout = firebase.functions().httpsCallable('handout_deck');
    handout(deck_num)
        .then(function (result) {
            temp = Object.values(result)[0];
            tempArray = new Array(20);
            tempArray[0] = new card(temp.c1,"url('wooloo.png')");
            tempArray[1] = new card(temp.c2, "url('bonk.png')");
            tempArray[2] = new card(temp.c3,"url('candy.png')");
            tempArray[3] = new card(temp.c4,"url('Castel.png')");
            tempArray[4] = new card(temp.c5,"url('free.png')");
            tempArray[5] = new card(temp.c6,"url('Mate.png')");
            tempArray[6] = new card(temp.c7,"url('monstar.png')");
            tempArray[7] = new card(temp.c8,"url('pain.png')");
            tempArray[8] = new card(temp.c9,"url('potion.png')");
            tempArray[9] = new card(temp.c10,"url('rest.png')");
            tempArray[10] = new card(temp.c11,"url('wooloo.png')");
            tempArray[11] = new card(temp.c12,"url('bonk.png')");
            tempArray[12] = new card(temp.c13,"url('candy.png')");
            tempArray[13] = new card(temp.c14,"url('Castel.png')");
            tempArray[14] = new card(temp.c15,"url('free.png')");
            tempArray[15] = new card(temp.c16,"url('Mate.png')");
            tempArray[16] = new card(temp.c17,"url('monstar.png')");
            tempArray[17] = new card(temp.c18,"url('pain.png')");
            tempArray[18] = new card(temp.c19,"url('potion.png')");
            tempArray[19] = new card(temp.c20,"url('rest.png')");
            j = 0;
```

```
    j = 0;
    // the fisher-yates shuffle
    for (i = 0; i < 20; i++) {
        j = Math.floor(Math.random() * 20);
        temp2 = tempArray[i];
        tempArray[i] = tempArray[j];
        tempArray[j] = temp2;
```

Gamestate Class (Conor):

The gamestate class is generally very static, mainly used to hold other objects - namely the players of the game, to handle who goes first and what the current turn is and whose turn it is to play the game. This handles the meta-functions of the game, instead of the actual game logic (drawing card, damaging a player etc.). Below is the final gamestate class.

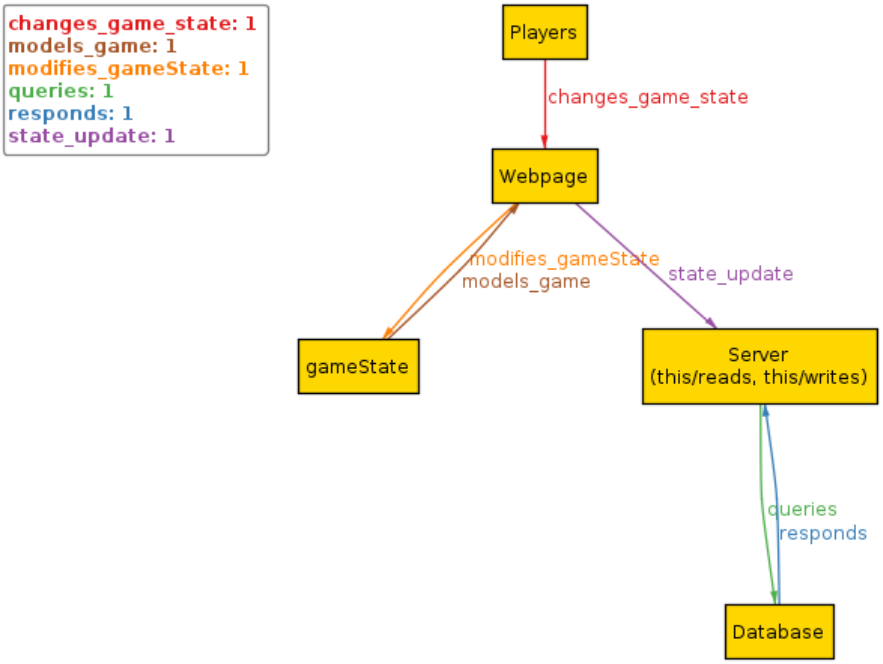
```
//below lists the functions used to generate game objects, like cards, the gamestate, decks and so on
var gamestate = /** @class */ (function () {
    function gamestate(game_string, num_players) {
        this.players = [];
        this.turn_num = 0;
        this.num_players = num_players;
        this.game = game_string;
        this.current_player = null;
    }

    gamestate.prototype.player_is_kill = function (player) {
       this.players["delete"](player);
    };

    gamestate.prototype.add_player_to_gamestate = function (add_player) {
        this.players.push(add_player);
    };

    gamestate.prototype.get_current_player = function () {
        return this.current_player;
    };

    gamestate.prototype.set_current_player = function (player) {
        this.current_player = player;
    };

    gamestate.prototype.flip_coin = function () {
        let temp = Math.floor(Math.random() * 2);
        return temp;
    };
    gamestate.prototype.det_current_player = function() {
        let temp = this.turn_num % this.num_players;
        return temp;
    };

    return gamestate;
}());
```

Gamestate holds all players in the game in a simple array, and thus allows for access to an individual player through the array. There is a getter and setter for the current player field and there was originally a function to determine the first player amongst four players, but that was cut for time.

Database and Cloud Functions:

Modelling: (Conor)

Alloy is a declarative specification language, designed to express structural relations based on first-order logic. I've used it before to model objects in assignments for object-orientated programming assignments, modelling class inheritance hierarchies and how objects interact with each other to create systems. I went through several prototypes before presenting this model to the rest of the team, to help illustrate how the systems should fit together.



Database Inspiration: (Conor)

The use of the database was primarily inspired by a talk given by Blizzard Entertainment developers on how they used a database to enforce game state in one of their more recent products, Overwatch. Although our project and Overwatch have radically different demands - Overwatch hosts up to twelve players in a game requiring multiple real time physics simulations, for one thing - it gave me an initial idea to represent the overall gamestate as one big object. This, in turn, led me to realising my recently learnt object-orientated programming paradigm would be well suited for this task.

Database Design and Implementation: (Conor)

I explored a variety of different technologies to handle the backend functionality, including writing an SQL schema, testing its functionality and testing a Linux server on my personal machine to handle backend duties using Node.js, like passing card values to the main webpage. In the end, as I could not find a reliable server hosting service (as in, I was too cheap to pay for one), as well as for time reasons, to use the NoSQL realtime database from

firebase (which was introduced to us as part of the software engineering course), which we also had educational grants for. However, writing the schema was a useful reference when I modelled game objects to simulate game logic, as well as for the gamestate database representation. While the schema was ultimately not used in the final product, it was a very useful prototype.

The schema was written while keeping Jack and Liam's completed ruleset in mind, which necessarily dictated what kind of data needs to be stored. Most of them are fairly obvious if you've played a card game of any kind before; the number of cards left in a players' deck, the values of certain cards and so on.

I originally thought I would need to use UUIDs as unique identifiers for the root of each game tree, in order to uniquely identify a given game, but it turns out firebase does something similar automatically with each node created manually through their API, and, for identification purposes, the string is "good enough", so I used that as any given game's unique reference.

Each game is identified by a unique ID and has two child nodes: gameState and players. gameState simply tells the state of the game, while the players display all the fields for a given player.

After some time experimenting with the NoSQL database, I realised it could be loosely described as a JSON-syntax tree-structure storage, so I wrote code that would roughly

represent the gamestate (using the schema notes for reference) and write it to the database, to generate a unique game instance, based on the number of players, and other parameters.

Inspired by reading about tef's post on REST, I thought about how to divide the work done by the game logic between the client and the server. Cloud functions seemed suitable to handle some of the work of the gameboard, namely handling whether the game is over and whether a player is dead. Since these cloud functions can be triggered on a write to a database, my idea was that the gamestate would be written to a particular database entry, and cloud functions would examine the gamestate, and see if any players were alive, or if the game was over.

"handout_deck" is a cloud function; in this case a callable cloud function, one that only ever activates when it is called explicitly by client-side code. See below for "handout_deck", and an example of how cards are structured.

```
exports.handout_deck = functions.https.onCall((data,context) =>{
    let return_object = null;

    let d1 =  {c1 : {attack:1, shield:1},c2:{attack:2},c3 : {draw:2,energy:1},c4:{shield:3},c5:{draw:2
energy:1},c14:{shield:3},c15:{draw:2},c16:{shield:2},c17:{energy:2},c18:{attack:3},c19:{heal:2},c20:{c
    let d2 =  {c1 : {attack:1, shield:1},c2:{attack:2},c3 : {draw:2,energy:1},c4:{shield:3},c5:{draw:2
energy:1},c14:{shield:3},c15:{draw:2},c16:{shield:2},c17:{energy:2},c18:{attack:3},c19:{heal:2},c20:{c

    // add more cards as necessary

    if (data == "d1"){
        return d1;
    } else if ( data == "d2") {
        return d2;
    } else if ( data == "d3") {
        return'{}';
    } else if ( data == "d4") {
        return '{}';
    } else {
        return "nothing";
    }

});
```

Difficulties and Challenges:

Jack:

- Communication was tough without being able to see each other. We did use online alternatives such as discord which did help. However to understand areas it would have been much easier for it to be explained in person.
- Bootstrap's dodgy implementation with certain browsers. The fact that some browsers will just not work with parts of code.
- One real struggle was finding a style for the cards to take. I would say it took over a dozen sketches to finally get an output I and the rest of the team could be happy with.

Liam :

- Communication and collaboration were a huge challenge with the project this year as we could not work together in person. While we made use of the alternatives that were blackboard and discord there are limitations as to the effectiveness of communication when screen sharing and emailing code are the best options for collaboration
- A lot of the work was not easily parallelised and situations would arise where something on the frontend was designed and ready but could not progress further as the backend may not have been fully functioning leading to periods where only the frontend or backend were being worked on one at a time and could not be worked on simultaneously as in our project, they were heavily dependent on each other

Conor:

- Far and away the hardest part was time management. It was hard to properly estimate how long something would take to implement, often something that I thought would be tough to implement taking maybe two hours, and something seemingly very simple taking far longer. This admittedly might be because this was the first time I had done something resembling this kind of time management.
- Coming in a close second was scope; we had deliberately set out with a harshly restricted scope of our project and even still that wasn't enough in terms of timesaving.
- Without trying to be hyperbolic, not having in-person communication was a fairly large barrier; we didn't have tools that filled that gap. Sometimes I wish I just had a big whiteboard to draw for everyone in.

Eoin :

- I found learning new things while simultaneously working on the project very difficult. TypeScript was a completely different format for me and while it was beneficial to the project, I found it challenging to learn while also coding in JavaScript. With some in person teaching, I feel this would have been less of an issue.
- Time discipline was very hard to manage in my opinion. While we are expected to be able to adhere to deadlines and manage our own work in the workspace, for many of us this was our first time having this responsibility fully thrusted onto. At least with set college times, we could work during those times strictly with extra time if necessary. However, I also feel managing our own timetables and workloads was a great learning experience.

Further Development:

Due to time constraints and the difficulties of working primarily online, we completed as much as we could and are satisfied with the result. However, if we are to continue this project in the future, we have a countless number of ideas. Here are a few examples of what we would like to add:

Online Play:

We want to get proper online play functioning, where two players from different clients can play against each other. This was one of our initial goals that we could not achieve within the given time. This would require us to use a server, which we could ask from the University or just buy ourselves.

More than 2 Players:

We want to have more than 2 players, to a possible maximum of 4. Our game objects have some functions and variables that could assist with this. For example, the attack function requests a player as a target. In the future we could make a function on the frontend that asks the player to select from more than one possible enemy.

More than 2 Characters:

In conjunction with having more than 2 players, we would want to make more decks and more playable characters other than a penguin and racoon. While making the cards mechanically would be very simple (all that needs to be done is assigning values and calling a firebase function), the artwork needed for all of the cards would take much more time. In terms of design, we had other ideas during the development of Chilli and Vector:

- A badger, who acted as a balanced class
- A baby panda who was all about healing but had really damaging epic cards
- A meerkat, who actually stole cards
- A time traveling pharaoh cat

Security:

Due to the open-source nature of our project, it is quite easy for a player to cheat in our game. Players could easily call functions within the client to gain an upper hand. With time, we could revise our code and make it more secure and keep the integrity of the game.

Sources:

- Dungeon mayhem img: https://boardgamegeek.com/boardgame/260300/dungeon-mayhem
- Hearthstone logo: https://logos-world.net/hearthstone-logo/
- Overwatch Gameplay Architecture and Netcode: https://www.youtube.com/watch?v=W3aieHjyNvw
- https://programmingisterrible.com/post/181841346708/what-the-hell-is-rest-anyway - tef - What the hell is REST, Anyway?